June 26, 1998

SRC Research Report 153

Continuous Monitoring and Performance Specification

Sharon E. Perl, William E. Weihl, and Brian Noble

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Continuous Monitoring and Performance Specification

Sharon E. Perl, William E. Weihl, and Brian Noble

June 26, 1998

Author Affiliation

Brian Noble is currently an Assistant Professor in the Electrical Engineering and Computer Science Department at the University of Michigan. He can be reached as bnoble@eecs.umich.edu.

Abstract

CMon is a general-purpose performance monitoring system. It enables monitoring of long-running programs in a setting where the experimenters who are interested in the performance data are different from the users who run the programs that generate the data. Among other things, this permits programs to be monitored under real workload conditions.

PSpec is a language and set of tools for *performance assertion checking*, an approach to automating the testing of performance properties of complex systems. It can be used in conjunction with the CMon system to produce a performance monitor that runs continuously, over the lifetime of a program, and automatically detects performance anomalies.

In this report we describe the design and implementation of CMon and the continuous monitoring features of PSpec, and report on our experience using the systems.

Contents

1 Introduction

In this report we describe a performance monitoring system with several key properties. The system allows monitoring of long-running programs, such as servers, editors, drawing programs, and browsers, where the experimenter would like continuous or periodic analysis of performance. It automates the analysis of the performance data so that the experimenter need not constantly attend to the monitoring system in order to detect problems. The system also permits the users of the program—who can generate data under real workload conditions but who don't want to be bothered with performance data gathering—to be different from the experimenters who want to process the data generated by the users as it becomes available.

The system we built consists of two separate tools. The first is called CMon (Continuous Monitoring). It provides the ability to capture logs produced by appropriately instrumented programs while the programs are being run by users, and direct the logs to experimenters who are interested in processing them. The experimenters can set up tools that process the logs and provide notification when performance problems occur. CMon is designed to work with a wide variety of log processing tools.

The second tool is a log processing tool designed specifically for use in the Continuous Monitoring setting. The tool is part of a system called PSpec (Performance Specification), which embodies an approach to automating the testing of performance properties of complex systems. System designers write assertions that capture expectations for performance, which can then be checked automatically against monitoring data to detect potential performance bugs. Automatically checking expectations allows a designer to test a wide range of performance properties as a system evolves: data that meets expectations can be discarded automatically, focusing attention on data indicating potential problems.

The PSpec system consists of a language for writing performance assertions together with tools for testing assertions and estimating values for constants in assertions. The language is small and efficiently checkable, yet capable of expressing a variety of performance properties. The PSpec tools are designed to be useful both online in a continuous-monitoring setting as well as offline in settings where logs are not processed until after the programs that generated them have exited (e.g., during performance regression testing).

There are hundreds of published papers on software performance monitoring tools, covering topics such as data collection techniques, visualization, data analysis systems, tuning, and debugging. (See the citations describing Pablo [10], monitoring based on relational algebras [13], IPS-2 [6], Paradyn [5], XPVM [15], the Windows NT Performance Monitor [14], and SNMP [11] for a sampling of the work that has been done.) In addition, most (perhaps all) computer systems include support for performance monitoring of some kind, and additional tools are available from third-party vendors. It is beyond the scope of this paper to include a survey of this vast body of work. We believe, however, that the combination of PSpec and CMon is interesting in the following ways:

- CMon provides a generic mechanism that can be used by programmers and developers of any system or application to acquire and process monitoring logs.

- CMon permits a monitoring log to be processed online by an arbitrary program, supporting the construction of a flexible suite of data analysis and notification mechanisms.

- The continuous monitoring features of PSpec provide a language that is efficient, expressive, and easy to use for analyzing monitoring logs to detect performance anomalies and compute other useful data from logs.

Together, PSpec and CMon provide a powerful and general mechanism for automatically processing performance data obtained online under production workloads.

The initial PSpec work was done as part of the first author's Ph.D. thesis [8] and reported on in SOSP'93 [9]. The idea of continuous monitoring arose in the context of that work. Preliminary design discussions for the CMon system took place in 1993, and detailed design and implementation were done from mid-1994 through mid-1995. Changes to PSpec to support continuous checking were implemented in early-to-mid 1995.

In Section 2 we describe the design of the CMon system and discuss some implementation details. Section 3 gives an overview of PSpec, with a description of the extensions specifically designed for continuous monitoring. Section 4 describes several experiments in using the systems to do continuous monitoring and performance assertion checking. Section 5 discusses what we accomplished and what we learned from this work. Section 6 concludes. The Appendices contain a reference manual for the PSpec language, as well as the manual pages for the PSpec and CMon tools.

2 Continuous Monitoring

We had several goals in designing CMon. We wanted to be able to monitor programs continuously or periodically, so that performance bugs would be detected as soon as possible. This goal necessitates a certain degree of “hands-free” operation, since it is unreasonable to expect experimenters to be attending to performance monitoring at all times. We wanted to monitor programs remotely, so experimenters could be different from users of the programs. We wanted the monitoring to have a minimal impact on the programmer (in terms of setting up a program for continuous monitoring) and on the program (in terms of performance and robustness). Finally, we wanted to allow the use of a wide variety of tools for processing the performance data gathered during monitoring, so that a single continuous monitoring mechanism could support many kinds of performance data processing.

The remainder of this section describes the system that we designed and built to meet these goals. The system runs on MIPS-based DECstation workstations running the Ultrix operating system and on Alpha-based workstations [12] running Digital Unix [2] (formerly OSF-1). Appendix C contains the manual pages for the CMon tools.

2.1 Design

The CMon system has three major kinds of components, illustrated in Figure 1. *Snarflog* is the interface between the program being monitored and the rest of the CMon system. A *telemonitor* is a kind of control panel through which an experimenter can direct continuous monitoring experiments. A *registry* connects snarflog instances with telemonitor instances. Typically there are multiple snarflogs, one for each instance of a monitored program, and multiple telemonitors, at least one for each experimenter. There is a single registry for any collection of snarflogs and telemonitors that need to communicate with one another. A snarflog communicates with at most one telemonitor at a time. A telemonitor may be communicating with several different snarflogs at once.

A snarflog has two jobs: to direct its associated log to interested telemonitors, and to make the monitored program robust to failures of telemonitors or registries. Our philosophy is that if a piece of the CMon system fails, the monitored program should continue to work, though perhaps some of the log it produces will be lost. This supports our goal of having minimal impact on the performance, robustness, and usability of the monitored program.

Snarflog runs in a separate process from the monitored program, taking the log either on the standard input or via a named pipe. We assume that a log is composed of a (possibly empty) *log header* followed by sequence of *log records* containing the performance data. Snarflog has a fixed-size buffer for log records. If some records are not shipped to a telemonitor by the time the buffer fills up, those records are discarded. We guarantee to discard a whole number of log records, and to save the log header so that it is always available when a new telemonitor connects to the

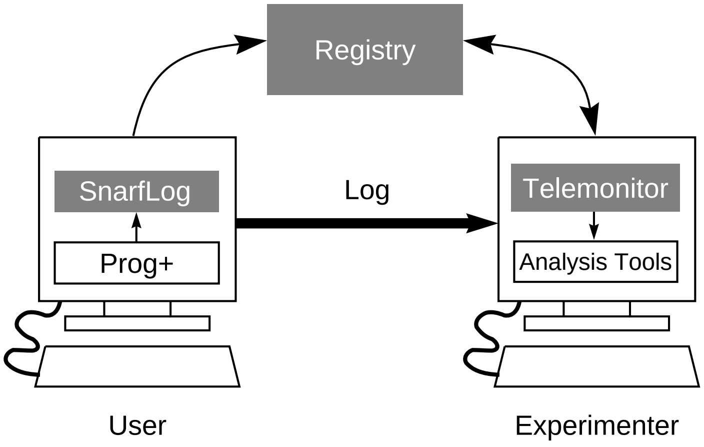

Figure 1: The CMon System. CMon components are in black.

log stream (since the log header may contain information necessary for interpreting the log records). Snarflog communicates with the registry when it starts up, so as to make its presence known to the rest of the CMon system. Thereafter, it can receive requests from telemonitors to start or stop sending log data. Once one telemonitor stops receiving log data another can start. Snarflog runs as long as the monitored program is producing a log.

A telemonitor provides a "control panel" through which an experimenter can control monitoring experiments. The experimenter indicates which classes of monitored programs are of interest by giving a pattern that matches a program name, machine name, and process id (possibly including wild cards). This information is passed to the registry. The experimenter also specifies a tool for processing the log of a monitored program, and says whether monitoring is to start automatically whenever an instance of that program starts. Through the telemonitor interface, the experimenter can see which monitored program instances are running or have completed, and can look at the results of processing the logs as they are computed.

Finally, the registry ties together snarflogs and telemonitors. Snarflog notifies the registry when it starts up, providing a handle through which telemonitors can make direct requests for log data. The registry keeps track of which telemonitors are interested in which classes of monitored programs, and notifies the interested telemonitors whenever an instance of a class starts up.

The mechanism for causing programs to produce logs is outside of the CMon system proper. The system that we built uses two different mechanisms. One is an in-house tool called *etp* (“elapsed time profiler”), which modifies a program binary to produce log records of procedure calls and returns. Another is the *trace* facility, available in many varieties of Unix systems, which produces log records of system calls and returns.

A programmer who wants to arrange for a program to be monitored would do the following. First, run a telemonitor and set up a class that specifies the program to be monitored (probably by name) and the desired tool for processing the output. This information can be saved in a telemonitor configuration file. Next, arrange that the program is always run in conjunction with snarflog—there are automatic tools provided with the CMon system that generate scripts to do this. Finally, make sure that a registry and a telemonitor are running while the log data from running programs is to be collected.

2.2 Implementation Issues

CMon’s implementation is fairly straightfoward. We chose to implement all of the programs (snarflog, telemonitor, and the registry) in the Modula-3 programming language [7], using Network Objects [1] (Modula-3’s object-oriented remote procedure call mechanism) for communication among the programs. The implementation was done in 1994, and there was no clearly better choice of programming technology at the time for our prototype system. If we were starting on the implementation today, in 1998, we would probably make different choices. In particular, we would probably implement the telemonitor UI as a Java applet, thereby making it accessible through a web browser.

The implementation of the telemonitor required careful attention to concurrency and locking, to handle all of the asynchronous events correctly while keeping the UI responsive. Other than this, the more interesting implementation issues involved the handling of logs.

The snarflog program has the responsibility for forwarding an application’s log to interested telemonitors. By design, the log may be directed to different telemonitors over the course of the application’s run, or any given telemonitor may intermittently stop receiving the log. Snarflog has to do two things to ensure reasonable behavior when a telemonitor begins receiving a log in midstream. First, if the log has a header, snarflog must save the header and prepend it to the log stream each time the log stream is picked up anew by a telemonitor. Second, snarflog must ensure that a telemonitor picks up the log stream at a log record boundary. These actions ensure that the log processing tools invoked by the telemonitor receive what appears to be a complete log, with a header and a whole number of log records.

Two other implementation issues arose specifically in the context of handling logs produced by etp-instrumented programs. First, in order for snarflog to recognize log headers and log record boundaries it is necessary to disable etp's normal log compression. This was easily accomplished by an existing switch to etp. Second, etp puts timestamps on all of its log records, recording a relatively small time counter (a few seconds worth) and reconstructing higher order bits of timestamps by inferring when the counter wraps. Initially, etp ensured that at least one log record was generated within each counter wrap interval, so that all wraps could be inferred. But this meant that during a long idle period for a program, a steady stream of log records was still being produced just for the purpose of detecting timer wraps. This seemed wasteful of resources when continuously monitoring a long-running program. At our request, etp's designer modified etp so that it instead counts the number of counter wraps during an idle period and emits just a single log record counting the number of wraps rather than one log record per wrap interval. This reduces logging and continuous monitoring overhead significantly while a program is idle.

3 Performance Specification

We now turn to the second portion of this work: the PSpec tools. The PSpec system embodies an approach that we call *performance assertion checking*. In this approach, system designers write assertions to capture their expectations about performance. These assertions are then checked automatically, focusing the designers' attention on monitoring data that indicate potential performance bugs. Our intention is that once a good set of assertions for a system is in place, they should be monitored continuously, while the system is in actual use. This enables performance problems to be detected soon after they appear.

While the PSpec language and tools were designed before the CMon system, we had in mind that they would eventually be used for continuous monitoring. However, PSpec is useful in other contexts as well:

1. Performance regression testing: when a system is changed, performance assertions can be rechecked to ensure that the system still meets expectations.

2. Performance debugging: successively more detailed performance assertions may be helpful for pinpointing the location of performance problems in the system.

3. Clarifying expectations: writing precise performance assertions helps system designers understand what they can and cannot guarantee about their systems.

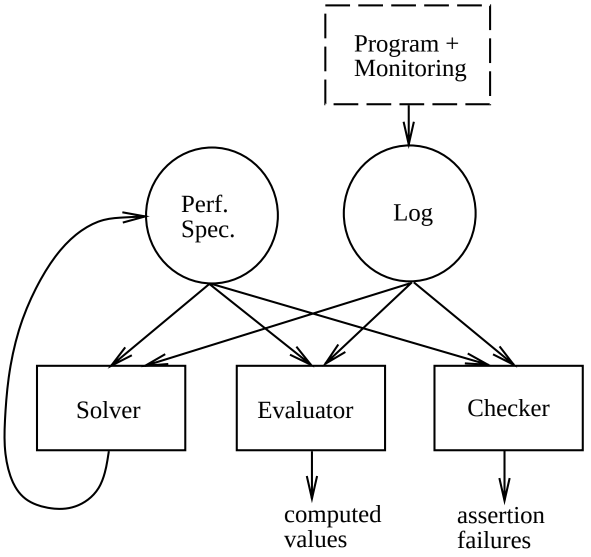

Figure 2: The PSpec approach

An earlier paper [9] gives a good overview of the PSpec language through a series of examples. That paper was written before the language and tools fully supported continuous monitoring. In this report, we briefly review the basic concepts of the language and then present the extensions for continuous monitoring. Appendix A gives a complete description of the current PSpec language, while Appendix B contains the manual pages for the PSpec tools.

3.1 Overview

The PSpec system, illustrated in Figure 2, has several components: performance specifications, monitoring logs, and the solver, evaluator, and checker tools.

A monitoring log represents an *event stream*, which is an abstraction of a program's execution that contains the information relevant to expressing performance assertions.

Performance specifications contain assertions about performance written in the PSpec language. The language is a notation for expressing predicates about event streams. Many common kinds of performance metrics, such as elapsed time, throughput, utilization, and workload characteristics, can be expressed.

The user supplies an augmented version of the program that generates an event stream in the form of a monitoring log for each run. The logging facility is not part of PSpec. Instead, the PSpec tools use a log interface that can be implemented on top of available logging facilities. PSpec is most useful with logging facilities that permit user-defined event types, but also works if only a fixed set of event types is available. In particular, PSpec works with the etp and trace monitoring facilities mentioned earlier.

The checker is the PSpec tool of interest for continuous monitoring. It takes as input a performance specification and a monitoring log; it produces as output a report of which assertions fail to hold for the run represented by the log. The checker was used in our continuous monitoring experiments. It is discussed further is Section 3.5 and its manual page appears in Appendix B.

The evaluator and solver are useful for performance debugging and for writing performance specifications. The evaluator provides a read-eval-print loop for evaluating expressions against data in a log. The solver uses logged data to help a specification writer determine values for numeric constants in assertions. Manual pages for these tools appear in Appendix B and a sample evaluator session appears in Appendix D, but we do not discuss them further in this paper.

3.2 Concepts

An *event stream* is a sequence of *typed events*. These events have named, numeric *attributes*. An event contains information recorded at a single point in a program's execution. Depending upon the available monitoring facilities, a specification writer may or may not have control over what types of events appear in event streams.

An *interval* consists of all events in a log between a designated *start event* and an *end event*. Intervals have named types and named *metrics*. An interval's metric values are based on the events in that interval. Interval types are defined by a specification writer; they are the primary abstraction used in writing PSpec assertions. Typically, a specification writer decides what to assert about a programs's performance, defines interval types that capture the necessary metrics, and then writes predicates that apply to a set of intervals.

3.3 Language Features

The PSpec language provides constructs for declaring event types, declaring interval types, and expressing assertions. A performance specification is comprised of a set of performance assertions with their accompanying declarations. Figure 3 shows a sample performance specification containing event type declarations, an interval type declaration, and assertions; we will use it to illustrate the main PSpec language features. The specification contains assertions about Read operations in a file system.

```
perfspec  FSRead
   timed event  StartRead(tid, size);
                EndRead (tid);
   event  CacheHit(tid);
   interval  Read =
         s: StartRead, e: EndRead where e.tid = s.tid
      metrics
         time = ts(e) – ts(s),
         size = s.size,
         hit = {count c : CacheHit where c.tid = s.tid} ≠ 0
      end Read;
   assert {& r : Read : r.time ≤ 10 ms};
          {mean r : Read : r.time} ≤ 5 ms;
          {& r : Read where r.size ≤ 4096 : r.time ≤ 8 ms};
          {count r : Read where r.hit} / {count r : Read} ≥ 0.75
end FSRead
```

Figure 3: An example performance specification.

An event type declaration introduces a type name and a list of attribute names. The example specification in Figure 3 includes declarations that introduce the event names *StartRead*, *EndRead*, and *CacheHit* (the first two events are declared to have timestamps, while the third does not). The names in parentheses, *tid* and *size*, are the attribute names for the event types. The rules for matching event types in a performance specification with events in event streams are specific to the particular monitoring log format.

An interval type declaration introduces a new interval type, which identifies a set of intervals resulting from an event stream. The declaration includes predicates for determining whether an event in the log is the start or end event for an interval of the type, and expressions for computing the metric values for an interval of the type.

The predicates for the start and end events in an interval type declaration may refer to the type of the event and to values of the event's attributes. The predicate for an end event may also refer to attribute values of the corresponding start event. In Figure 3 we see an interval type declaration where the start event has type *StartRead*. The end event has type *EndRead*, along with the restriction that its *tid* attribute be the same as the start event's *tid* attribute. The *tid* attribute records a thread identifier. This type of restriction is useful for matching start and end events of intervals in an event stream that contains events from multiple threads.

The example interval declaration contains three *metric definitions*, for metrics named *time*, *size*, and *hit*. Each of an interval's metrics is evaluated over the events spanned by the interval. The *time* metric is defined to be the difference of the timestamps of end and start events of the interval, i.e., the duration of the interval. The *size* metric is the value of the *size* attribute from the interval's start event (which records how many bytes are being read). The *hit* metric records a boolean value that indicates whether a *CacheHit* event with the appropriate thread identifier occurred between the start and end events for the interval. The intention is that a *CacheHit* event would be generated in the event stream if the Read operation could be serviced entirely from an in-memory cache, without having to go to disk.

An assertion in a performance specification is a predicate on the events and intervals in event streams. One simple kind of assertion is a predicate that applies to all events or intervals of a particular type in a log. For example, the first assertion following the *assert* keyword in Figure 3 can be read as: "for all intervals *r* of type *Read*, the value of *r*'s *time* metric is at most 10 milliseconds". This assertion is an example of an *aggregate expression*, which provides a way to generate a sequence of values and combine them with an operator. The remaining assertions in the example also use aggregate operators: to compute the mean value of a series of metrics, to express a "forall" type assertion over a restricted series of intervals, and to count the number of intervals in a restricted series.

This section has glossed over many details of the PSpec language in the interest of conveying the main ideas in a brief space. See Appendix A and the paper cited earlier for more extensive descriptions of the language.

3.4 Support for Continuous Monitoring

The basic interval types described above permit interval type declarations—and therefore, metric calculations—based on actual events occuring in monitoring logs. For continuous monitoring, we would also like to write assertions about what happens in a monitored program over fixed periods of time. For this purpose, we introduce time-based interval types, which effectively allow us to divide an event stream into chunks based on time, and then write assertions about each of those chunks.

In a time-based interval type, either or both of the start and end event declarations can be replaced by expressions denoting times. These declarations cause new *virtual time events* to be merged into an event stream from a monitoring log that contains timestamped events; the time-based intervals are then defined in terms of the virtual time events. These events are merged in while processing the log so they do not affect the actual monitoring.

```
interval  Chunk =
    s: every 1 day, e: after 1 day
  metrics
      hitratio =  {count r : Read where r.hit} / {count r : Read}
  end Chunk.
```

Figure 4: A time-based interval type declaration.

Figure 4 shows an example of a time-based interval declaration. It declares an interval of type *Chunk* to start every day and to last for a day. The start and end events are new virtual time events with type names *start$Chunk* and *end$Chunk*. The start events occur in the event stream at intervals of one day and each corresponding end event occurs one day after the start event, so each interval spans one day. As usual, an interval of type *Chunk* includes all events between its start and end events, including other virtual time events. The metric *hitratio* calculates the hit ratio over the course of a day for file system Read operations. Using this declaration we could write the assertion:

assert $\{\& \; c : Chunk : c.hitratio \geq 0.75\}$.

That is, for all intervals *c* of type *Chunk*, the value of *c*'s *hitratio* metric is at least 0.75. This is similar to the last assertion in Figure 3, except that it is more suitable to being checked by a continuous monitor because its truth can be evaluated each time a *Chunk* interval is generated (any interval for which the hit ratio is less than 0.75 falsifies the assertion). The earlier assertion is defined to apply over an entire log, and couldn't be checked until the program generating the log had terminated. We hope that the file system is a very long-running program, so we wouldn't want to have to wait until it exited to check a performance assertion about it. Also, the time-based assertion is more explicit about the range of Read operations over which the ratio should be computed.

Other variants of time-based intervals are possible. For example, to do periodic monitoring one could define an interval that starts every day and ends after one hour. To have a sliding window, an interval could start every hour and end after one day. Intervals that start with a particular real event type and end after a fixed amount of time, or start every given time interval and end with a real event type may also be useful. Note that in order to connect intervals to real time (for example, an interval that starts at 2 am every morning), there must be events in the log that record real timestamps in their attributes.

Writing useful performance specifications using time-based intervals does require some care. Arbitrarily dividing a log into chunks at specific time intervals may leave the chunks with partial event-based intervals, which will then not be recognized as sub-intervals of the chunks. So a specification writer must think carefully about the meaning of metrics for time-based intervals. This problem shows up in the example presented in Section 4.4.

3.5 The Checker

The job of the PSpec Checker is fairly straightforward: to report which assertions in a specification fail to hold for a monitoring log. It also attempts to provide some information (log context) that can help an experimenter determine why an assertion failed.

As the checker reads through a monitoring log, it incrementally evaluates all expressions in the specification that refer to events in the log. This includes aggregate expressions over intervals such as the ones in the examples above. As each interval's end event is encountered, the metrics for the interval are evaluated, and any aggregate expressions using the interval's metrics are evaluated incrementally. The "forall" (&) aggregate operator is treated specially when the checker is running in "continuous mode." In that mode, which is intended for use with continuous monitoring tools, whenever a conjunct of a "forall" aggregate expression evaluates to false, the checker reports an assertion failure and the value of the interval that caused it. It continues to evaluate further conjuncts of the aggregate expression as they are encountered, even though the ultimate value of the aggregate expression is known to be false from the first false conjunct. In this way, details of any further assertion failures are also reported.

All other types of assertions (those that are not aggregates with the "forall" operator) are evaluated after the end of the monitoring log is reached; this may be when the monitored program exits, or when an experimenter using a telemonitor disconnects from the monitored program.

4 Experience

We have applied the CMon and PSpec tools to four systems used at SRC:

- Argo, a teleconferencing system that provides real-time audio and video connections among multiple users; it has stringent performance requirements.

- The NFS automounter, which dynamically mounts and unmounts shared network file system volumes; network problems at SRC were being manifested as delays in automounting.

- Juno-2, a constraint-based drawing system [4]; it uses a numerical constraint solver whose performance is critical to the interactive nature of the application.

- Lectern, a document viewing system designed to provide fast high-quality display of documents; it also has stringent performance goals.

Argo, Juno-2, and Lectern were all developed at SRC. The NFS automounter is a standard system utility.

In the subsections below, we briefly summarize our goals in monitoring these systems and how we accomplished the monitoring.

4.1 Argo

The goal of the Argo system is to allow medium-sized groups of users to collaborate remotely from their desktops in a way that approaches as closely as possible the effectiveness of face-to-face meetings [3]. In support of this goal, Argo combines high quality multi-party digital video and full-duplex audio with telepointers, shared applications, and whiteboards in a uniform and familiar environment. As for other teleconferencing systems, jitter is an important issue for Argo; variations in the rate at which packets (both video and audio) are delivered can be quite annoying to users. Audio jitter is particularly noticeable, and there had been occasional problems with the audio, so our experiments focused on *argohear*, the subsystem responsible for delivering audio to the user. Our goal was to discover how much jitter was occuring, how frequently it occurred, and to isolate its causes.

The monitoring was accomplished by using etp to instrument the argohear program, then setting up argohear to run under the CMon system. No source code changes were required. Five people ran the instrumented argohear program for one week. A telemonitor was used to collect the data and to save it to disk; data from each separate run of argohear was saved in a separate log file. The performance of argohear itself was minimally affected by the monitoring.

There were two possible sources of jitter in the audio system: the network delays seen by separate packets of audio data, and the service time required for each packet. After all the log files had been collected, we ran the PSpec checker on each of the event logs to compute distributions of inter-arrival times and service times for audio packets. The data showed that the jitter estimation algorithm used to predict and compensate for network delays seemed to work; however, the service time became erratic under high load, and the MIPS-based DECstations (40-MHz R3000's) used in the experiments could saturate with as few as three audio streams. This information was quite useful to the designers of argohear in understanding and fixing the jitter problems.

Note that the PSpec checker was not invoked directly by the telemonitor; we experimented with argohear early in this work, and the checker did not support continuous checking at that point. Given the way we monitored argohear, we could have accomplished the same thing without the CMon infrastructure just by saving log files in the shared file system. However, our experiments with argohear helped us debug the CMon system and provided useful insights into how continuous monitoring could be done better.

4.2 Automounter

We decided to try monitoring the automounter for several reasons: it was not a "home-grown" application; it required different monitoring (the system "trace" utility for tracing system calls rather than etp for tracing procedure calls); and at the time SRC had been having a series of serious network problems that we thought might have been detected by watching for unusual delays in the automounter. Our hope was that early detection would have allowed a problem to be fixed before the network died altogether. The latter was the primary motivation; since the network was having problems that manifested themselves as serious performance problems (to the point of making many machines unusable), we wanted to see if our tools could both provide an early warning as problems were developing and also help track them down.

We monitored the automounter by running "trace", an Ultrix utility that traces the system calls from a specified process. By tracing the activity of the automounter process, we were able to identify intervals corresponding to the automounter responding to a request to mount a particular volume. One common symptom observed during periods of network difficulties was a very long delay in the automounter responding to requests to mount a volume, so watching for this seemed like a good way of detecting intermittent network outages.

Our automounter experiments were not very successful, for several reasons. First, it was difficult to set up automated tracing of the automounter on machines throughout our facility. Using etp to set up monitoring allows one to install an instrumented executable, which then generates an etp log whenever it is run. With the automounter, however, we had to run the trace utility on every machine on which we wanted to collect a log, we had to set up snarflog to acquire the output from trace (something that we had automated for etp but not for trace), and we had to restart trace on a machine whenever the machine crashed or was rebooted. Second, the PSpec checker did not support continuous checking at that point, so manual intervention was required to feed the logs into the checker. Third, the overhead of shipping trace logs around was quite high, which made using trace less attractive than using etp for some other application. (The trace log format uses

ASCII and is much more verbose than etp's format.) Finally, the underlying cause of our network problems was found and fixed while we were working on setting up monitoring for the automounter. We monitored the automounter for a few days on a small number of machines, but no problems were detected. Since we were not yet in a position to set up continuous monitoring and to use the system to monitor the network on a regular basis, we decided to focus on completing the CMon system.

4.3 Juno-2

A key research question about Juno-2 is whether the constraint solver will scale. The goal of monitoring was to understand the performance of the constraint solver better by collecting data from program runs involving real data with real users, not just from a small suite of test cases used by the developers. The developers wanted to obtain data about the sizes of the constraint systems that arise in real use as well as the elapsed time required to solve those systems.

As with argohear, the PSpec checker did not support continuous checking when we started monitoring Juno-2, so the logs were simply saved by the telemonitor in the file system so they could be analyzed later. The Juno-2 designers set things up to monitor two procedures: one was the main constraint solver and the other was introduced simply so that its arguments (the number of constraints and variables in a constraint system, and the number of Newton iterations required to solve the system) could be recorded in the log. The main goal of the monitoring was to see if the data obtained previously from experiments with a small set of constraint systems were representative of real-world constraint systems. As a minor side benefit, the telemonitor was configured to send mail to one of the Juno-2 designers whenever someone started up Juno-2, so he could keep track of how much people were using it.

In the end, the designers of Juno-2 did not do much with the data collected. There were two main problems. First, whenever the telemonitor crashed or the machine on which it was running was rebooted, it had to be manually restarted. Second, due to the nature of etp, shipping a new binary executable would render useless any etp logs collected up to that point. A facility to save the symbol table information from a binary made it possible to interpret old logs, but the process required too much manual intervention and was too error-prone to be very workable. Since the designers were actively changing Juno-2 while they were monitoring it, this made most of the collected logs useless.

4.4 Lectern

One of the main performance goals for Lectern was to display a page in less than one second. We decided to monitor Lectern to test if the goal was being met during actual use of the system by real users. We also wanted to collect data on people's use of the program. For example, do they tend to browse documents, flipping quickly from one page to the next, or do they read them in depth, spending significant amounts of time on each page? How do they use the user interface? And so on.

4.4.1 Monitoring Setup

The monitoring was accomplished by using etp to instrument the Lectern executable, tracing four procedures. One of the procedures corresponds to virtually all the time involved in displaying a page. Another corresponds to the user inputting a command. The other two correspond to particular commands for moving among pages and documents. The arguments of the procedures recorded by etp included enough information to be able to tell what command was requested and whether two successive display requests were for the same page (e.g., to scroll within a page) or for different pages.

We replaced the normal Lectern executable with a shell script that ran the instrumented executable together with snarflog, making the log available to any interested telemonitors. We set up a telemonitor to pass the Lectern logs to the PSpec checker, running the checker in continuous mode to check that the display time for each page was less than a second. We also used the checker to produce a report once per day with a summary of the usage during that day, including histograms and averages of the time spent on each page and the number of pages accessed per document, as well as a histogram showing the usage frequency of each of the user-level commands in the user interface.

4.4.2 Assertion Checking

The performance specification used to check the assertions and compute the histograms is shown in Figures 5 and 6. Four procedures are monitored: *ApplyOp*, which does the work of dispatching a user-level command; *lgm*, which does virtually all of the work of reading a page image from a file and rendering it to the screen; *ReadDoc*, which is invoked when a new document is loaded; and *GotoPage*, which is invoked whenever the user either scrolls on the current page or moves to a new page.

The Lectern performance specification uses several constructs not discussed above. First, *procedures* (***proc***) are declared. A declaration of the form ***proc*** *P*

```
perfspec lectern
  proc Lectern__ApplyOp(lect, op, time, event);
  proc ImageRd__lgm(t, width, height, hasMap) returns pixmap;
  proc Lectern__ReadDoc(lect, path, time, from);
  proc Lectern__GotoPage(lect, page, class, noisy) returns okay;

  interval Apply = intv@Lectern__ApplyOp
    metrics
      op = s.op,
      % msTime is the time in milliseconds, truncated to the nearest multiple of 10.
      msTime = trunc((timestamp(e) − timestamp(s))/10ms)∗10
    end Apply;

  interval lgm = intv@ImageRd__lgm
    metrics
      width = s.width,
      height = s.height,
      area = s.width ∗ s.height,
      msTime = trunc((timestamp(e) − timestamp(s))/10ms)∗10
    end lgm;

  interval Page =
      s: call@Lectern__GotoPage, e: call@Lectern__GotoPage where e.page != s.page
    metrics
      time = trunc((timestamp(e) − timestamp(s))/1sec)
    end Page;

  interval Doc =
      s: call@Lectern__ReadDoc, e: call@Lectern__ReadDoc
    metrics
      pages = {count p : Page} + 1
    end Doc;
```

Figure 5: Pspec input for Lectern monitoring, part 1.

```
  interval Chunk =
        s: every 1 day, e: after 1 day
     metrics
        % histogram of opcodes:
        applyOps = {+ a : Apply : a.op ⟶1},

        % histogram of times for lgm:
        lgmTimes = {+ l : lgm : l.msTime ⟶1},

        % histogram for each image size of times for lgm:
        lgmTimeByArea = {+ l : lgm : l.area ⟶(l.msTime ⟶1)},

        % average pages read per document:
        pagesPerDoc = {count p : Page} / ({count d : Doc} + 1),

        % distribution of pages per document:
        pageDist = {+ d : Doc : d.pages ⟶1},

        % distribution of time per page:
        pageTimes = {+ l : Page : l.Time ⟶1}
     end Chunk;

  assert "1-day stats": {& c : Chunk : false};
  assert "1-sec max for display": {& l : lgm : l.msTime <= 1000};
end lectern;
```

Figure 6: Pspec input for Lectern monitoring, part 2.

implicitly declares two event types, one of the form *call@P* (corresponding to a call of *P*) and the other of the form *ret@P* (corresponding to a return of *P*). The event type *call@P* has attributes as listed in the procedure declaration; the event type *ret@P* has an attribute if the optional ***returns*** clause is provided. These event types can be used in interval declarations and in defining metrics or writing assertions. Declaring a procedure *P* also declares an interval type *intv@P*, with start and end events named *s* and *e*, that corresponds to a single invocation of that procedure (from a call to a return in the same thread); the intervals *Apply* and *lgm* use this predeclared interval type to define a new interval subtype with additional metrics (e.g., *Apply = intv@Lectern__ApplyOp*).

The call and return events and interval types resulting from the four declared procedures are used to define a number of additional interval types: *Apply*, which corresponds to a single invocation of *ApplyOp*; *lgm*, which corresponds to a single invocation of the *lgm* operation; *Page*, which corresponds to the interval from one call of *GotoPage* to the next; and *Doc*, which corresponds to the interval from one call of *ReadDoc* to the next. Informally, *Apply* is the interval during which a single user command is processed; *lgm* is the interval during which a page is read from a file and rendered to the screen; *Page* is the interval during which a single page is displayed on the screen; and *Doc* is the interval during which a single document is being examined by the user. The metrics for each of these intervals record or compute various useful pieces of information; for example, *Apply*, *lgm*, and *Page* all record the duration of the interval, *lgm* computes the size of the displayed image, and *Doc* computes the number of pages examined for the document.

An additional interval, *Chunk*, is defined to capture all of the events during a one-day period. It has a number of metrics that compute statistics about the use and behavior during that period. For example, the metric *applyOps* provides a histogram of the different opcodes, mapping each opcode to a count of the number of times it was used during the one-day period. This metric definition uses a feature of the PSpec language called a *mapping* (defined in the language reference in Appendix A). Briefly, a mapping is a partial function from integers to values; operators are provided to create single-element mappings and to combine mappings. Mappings can be used to construct histograms by making the domain of the mapping identify the thing to be counted, by making the range of the mapping be 1 for each thing, and by combining the mappings with the $+$ operator.

Notice that some *Doc* and *Page* intervals may overlap the boundaries of *Chunk* intervals and, hence, would not be contained in any *Chunk* interval. As a result, some pages and documents would not be accounted for in the statistics of any *Chunk* interval. For this application it doesn't really matter, but we point it out as an example of the subtleties of writing useful time-based interval definitions.

Two assertions are used to print data periodically while a monitored instance of Lectern is running. The first, called *1-day stats*, is a hack for printing each day's *Chunk*. It asserts *false* for every *Chunk* interval. Since *false* always fails, this forces the system to report an assertion failure for every *Chunk* interval. When an assertion fails for an interval, the checker prints out the interval's metrics; thus, all of the statistics computed for each one-day period are printed out at the end of the period.

The second assertion, called *1-sec max for display*, asserts that the time for an *lgm* interval is no more than one second. Whenever an *lgm* interval fails this assertion, the checker prints out the metrics for the interval.

4.4.3 Results of Monitoring

We found no significant performance bugs in several months of monitoring Lectern. All instances of Lectern started by users in our laboratory were monitored during this period. Multi-second display times did occur, but rarely, and then they were only slightly greater than one second and never greater than two seconds. The information about people's use of the program turned out to be less useful than we had hoped. Lectern was quite mature when we started monitoring it, and by the time we had accumulated enough data for it to be interesting, the designers of Lectern were no longer interested in tuning its user interface.

5 Evaluation and Lessons

In this section we evaluate our tools based on our experience using them as described in the previous section and discuss the lessons we learned about how continuous monitoring needs to be done for it to be effective.

5.1 Successes

CMon was successful at monitoring long-running programs continuously. Many users would start Lectern and leave it running for days or weeks. The CMon tools handled long runs such as this without any problems. The ability to monitor remotely also turned out to be quite useful. All instances of instrumented applications could be easily (and mostly automatically) monitored from a single place. In the process, the CMon tools made it possible to gather data from runs of programs under actual workloads. This allowed the developers of argohear and Juno-2 to learn things about their systems that they could not learn by running test suites. It also avoided any need for the users of the monitored programs to know about details of the monitoring.

From the programmer's perspective, using CMon in combination with etp was easy. For most applications, properties of interest could be monitored and tested by using etp to trace appropriate procedure calls; the application source did not need to be changed. (Running etp is straightforward.) Occasionally it was necessary to add "dummy" procedure calls to mark the occurrence of particular events.

The impact on programs was also small. Except for the overhead of logging events and sending them to a telemonitor (about 20 microseconds per log record on a 40-MHz MIPS R3000-based DECstation), CMon caused no significant performance impact. As long as the granularity of logging was reasonably coarse, overhead was undetectable by users. In all the applications we studied, the properties of interest could be tested using very coarse-grained logging. As for robustness, there were no reports of the CMon tools causing crashes or other problems with monitored applications. The system was designed so that snarflog would insulate applications from problems arising in the rest of the CMon system. We found this property to be quite useful while developing the system, since we frequently had cases where a telemonitor would crash but monitored applications would continue running. CMon also tolerates failures of the registry very well; both snarflog and the telemonitor reconnect when the registry restarts.

CMon permits experimenters to use a variety of log processing tools. The telemonitor was designed to allow any program to be used to process logs, as long as it could take the log on standard input. The PSpec checker was set up to permit this, but we also used Unix shell scripts to save logs in files (named with distinct names based on the application name, machine name on which the application ran, and the process id of the application). In some cases we used shell scripts to save the log in files and at the same time process it continuously with the checker. It would have been useful to have more tools that could perform interesting "continuous" processing of etp logs. For example, online graphical display of data extracted from a set of logs might be very useful for building a "watch" tool for monitoring systems.

5.2 Lessons Learned

Perhaps the most important lesson we learned is that the telemonitor needs to be more robust to system crashes and restarts. During network problems the telemonitor would often crash because of bad interactions between the Modula-3 system and the NFS file system; when this happened it had to be manually restarted. A better design would separate the monitoring functionality from the control functionality (user interface). The monitoring functionality should be in a reliable and automatically restarted server. The control functionality should be in a program (perhaps a Java applet) that an experimenter runs in order to examine or reconfigure the state of a telemonitor.

It would also be useful to have built-in support in the telemonitor for notifying someone when a problem is detected. This could be cobbled together using shell scripts to look for particular patterns in the output of the PSpec checker and then send email to an appropriate user, but built-in support would be easier to use.

A related problem is the need to save relevant log data when a problem is detected. CMon makes it easy to save the entire log (e.g., using the Unix utility “tee”), but we believe it would be useful to save a chunk of the log containing events from a relatively short time period in which a problem is detected in order to save disk space. The problem here is determining what chunk to save; should it be based on time, on event-based intervals, on total log size, or something else? We discussed several options, but never implemented any of them. This issue would need to be resolved to allow people to understand what caused a problem after it was reported without saving all of the log data generated by a monitored program.

The CMon system processes the logs from separate instances of the same application independently. It would be very useful to be able to process all the logs together with a single tool, particularly for collecting data about the usage of a program. The same is true (but less important) for using logs to detect performance problems. Allowing a single tool invocation to process logs from multiple instances would also reduce the load on the system running the telemonitor by not creating a separate tool process for each instance of a monitored application.

In a similar vein, CMon supports monitoring of multi-threaded programs with a single log from the entire program, but provides no way to merge the logs from distributed pieces of a single distributed program. This was not a problem for the systems we monitored, but it could be for other systems.

Finally, one needs to ensure that logs can be interpreted even if the program that generated them has changed or been deleted. The developers of Juno-2 were actively changing the program while they were running performance experiments. An executable’s symbol table is needed to correctly process a corresponding etp log. Hence, changing the executable causes old logs to become meaningless. Better automatic support for saving the appropriate symbol table information (and occasionally, the old executables) and correlating it with the logs would have made processing the logs much easier.

6 Conclusions

The CMon system demonstrates the feasibility and utility of continuously monitoring long-running programs, with support for remote monitoring and minimal impact on programmers and monitored programs. Based on our experience building and using CMon, we are convinced that a monitoring system like CMon with more complete hands-free operation could be invaluable in making it easy to monitor a wide range of systems.

7 Acknowledgments

We are grateful to the following people for help with this work. Mike Burrows, the author of etp, graciously accommodated our requests for modifications that would allow etp to deliver logs to CMon easily and efficiently. Greg Nelson participated in the initial design of CMon as well as helping with the Juno monitoring experiments. Allan Heydon has been our most active PSpec and CMon user and has provided many helpful bug reports and suggestions. Lance Berc assisted with the Argo experiments. Andrew Birrell and Paul McJones assisted with the Lectern experiments.

We would also like to thank our proof readers, Allan Heydon, Greg Nelson, Shun-Tak Leung, and Cormac Flannagan, for many helpful comments on this report.

A PSpec Language Specification

This appendix gives a complete description of the PSpec language. See Appendix D for an extended example of using PSpec.

A.1 Definitions

First, some definitions. A PSpec *event stream* is a sequence of primitive components called *events*. Each event has a type and a sequence of named, numeric-valued attributes. An event stream is derived from a *monitoring log*, which is produced by executing an instrumented program. Most of the events in an event stream correspond directly to events in a monitoring log. (The details of this correspondence are implementation-dependent. See, for example, the section of Appendix B on monitoring using etp.)

Some of the events in an event stream are fabricated by the PSpec system. For example, events may be fabricated to mark the start and end of the log. In addition, a performance specification can introduce periodic events that are fabricated based on time; these are called *virtual time events*.

An *interval* corresponds to a contiguous subsequence of an event stream starting at some start event and ending at some end event. An interval has associated metrics, which are named and have values (not necessarily numeric). Intervals may be disjoint, may overlap, or may nest. An interval i is nested inside another interval j if i's start and end events are properly between j's start and end events in the event stream.

Values are mathematical entities with types. Some examples of value types are events, intervals, numbers, and booleans. PSpec also has a special value called *UNDEFINED*, which is a value of all types. *UNDEFINED* is used as the value of certain expressions that do not have sensible values for particular event streams.

An *expression* specifies a computation that produces a value.

An *identifier* is a symbol declared as a name for a value. The region of a specification over which a declaration applies is called the *scope* of the declaration. The outermost or top-level scope of a specification is called the *global scope*. Event and interval type names and declared constants are all in the global scope. In addition, interval declarations and aggregate expressions (to be described later) introduce local scopes. Scopes nest, with names in the global scope accessible from all enclosed scopes (though there are certain restrictions, described later). Identifiers must be declared before they are used in expressions.

A.2 Types

Every value in a specification has a type that dictates how the value may be interpreted. With the exception of event and interval types, types are never named explicitly in specifications. However, the type of any value or expression can always be inferred statically.

The base types are *number*, *boolean*, and *string*. From these base types, *triple* and *mapping types* can be constructed.

A *triple* consists of three numbers and represents a measurement (or combination of measurements) with associated error. The triple *[t,p,m]* represents a number in the range *[t-m,t+p]*, where *t* is the measured or *favored* value, and *p* and *m* are always non-negative. For example, a timestamp taken from a discrete clock measuring elapsed time represents a time value sometime between the time at which the clock last ticked and the next time it will tick. Such a timestamp is represented as a triple with a favored value as the clock value (the time of the last tick), a *p* value of one clock tick, and an *m* value of zero.

A *mapping* is a partial function from integers to values.

In addition to the above types, specification writers can declare *event* and *interval* types. The declarations for these types are described in the next section. Event and interval types have names that uniquely identify them (i.e., two types are equivalent if and only if they have the same name).

A.3 Declarations

A declaration introduces a name for a constant, event type, or interval type into the global scope. The name is available in all expressions that follow the declaration. An interval type name is not available in metric definitions for that type (thus, interval declarations are not recursive). A declaration may also introduce a name for an unknown number to be solved for later. It is an error to redeclare a name in the same scope (but a name may be redeclared in a nested scope).

A.3.1 Constants

If *id* is an identifier and *e* is an expression, then:

def *id = e*

declares *id* as a constant bound to the value of *e*. *e* is evaluated in the global scope. The declaration

def *id = ?*

declares *id* to be an unknown. Unknowns are recognized by psolve, the PSpec solver, and are different from *UNDEFINED* values.

A.3.2 Event Types

If *id* is an identifier and *alist* is a (possibly empty) comma-separated list of identifiers, then the two declarations

event *id (* [*alist*] *)*

and

timed event *id (* [*alist*] *)*

declare *id* as an event type.[1] Events of the type have attributes named in *alist*. The second form also indicates that events of type *id* have an implicit timestamp attribute, which is used by the *elapsed* function on intervals and the *timestamp* and *elapsed* functions on events. Note that the timed events in a log need not necessarily occur in timestamp order.

A.3.3 Interval Types

An interval type declaration introduces a new interval type, which identifies a set of intervals in an event stream. An interval type declaration provides predicates for determining whether an event in the event stream is the start or end event for an interval of the type, and expressions for computing the metric values for an interval of the type.

An interval declaration may also introduce virtual time events into the event stream to mark the start or end of an interval based on time. These are described in the next section.

The basic interval type definition identifies an interval by the types of its start and end events in an event stream. If *id*, *s*, and *e* are identifiers, *stype* and *etype* are event type names, and *spred* and *epred* are boolean-valued expressions, then:

[***nested***] ***interval*** *id =*
    s: stype [**where** *spred*] ,
    e: etype [**where** *epred*]
  [***metrics***
    mlist]
  end *id*

[1] The square brackets here and in the rest of the appendix indicate optional elements and are not themselves part of the syntax.

declares an interval with type name *id*. *mlist* is a comma-separated list of metric definitions of the form *m* = *expr* where *m* is an identifier and *expr* is an expression.

The declaration defines an interval of type *id* to be one that has a start event named *s* of type *stype* for which *spred* is true, and an end event named *e* that is the next event in the event stream of type *etype* following the start event for which *epred* is true.

An interval declaration may be modified slightly by inserting the keyword ***nested*** before ***interval.*** A nested interval type has the further condition that the end event *e* is not an end event for any other interval of type *id* that starts after the event *s*. In this case, *e* is the end event for the interval started by the nearest preceding *s* that does not already have an end event. Thus, without the nested restriction, multiple intervals of the type may share the same end event. With the restriction, all intervals of the type will have different end events and will nest like parentheses. It is always the case that multiple intervals of different types may share end events.

For each interval of type *id* in the event stream, its metrics are computed as follows. A new scope is introduced with *s* and *e* bound to the start and end events for the interval. The metric expressions are evaluated in this scope over the portion of the event stream enclosed by the interval, and their values are bound to the interval's metric names. This binding is, in effect, simultaneous for all of an interval's metric definitions; hence metric definitions may not reference each other. Metric expressions that aggregate over events or intervals may not refer to the end event *e* (because of implementation efficiency considerations).

The where-clause for the start or end event may be omitted, in which case it defaults to *true.* The *metrics mlist* clause may be omitted, in which case the interval has no metrics (but the *elapsed* function may still be applicable for the interval).

The identifier *s* is available in *spred* and *epred*. The identifier *e* is available in *epred*. Both identifiers are available in the metric expressions, as explained above. The identifier *id* cannot be referenced inside *mlist*. Moreover, expressions in *mlist* cannot reference constants that are defined in terms of aggregate expressions.

A.3.4 Time-based Interval Types

In an interval type declaration, the start event declaration

> *s: stype* [***where*** *spred*]

can be replaced by the declaration:

> *s:* [***from*** *oexpr*] ***every*** *texpr*

where *s* is an identifier, and *oexpr* and *texpr* are positive number-valued expressions. This declaration introduces new virtual time events of type *start$I* (where *I* is the interval type name) into the event stream. The events have timestamp values *startts* + *oexpr* + (*i* * *texpr*), for *i* = 0,1,. . . , where *startts* is the first event timestamp in the log. In other words, the virtual time events occur starting *oexpr* time units after the first event in the log, and every *texpr* time units thereafter. Each such virtual time event starts a new interval of the type. The *from* clause may be omitted, in which case *oexpr* defaults to 0.

If the events in the log are in timestamp order, the virtual time events are inserted in timestamp order, up to the last real event in the log. If a virtual event has the same timestamp as a real event, the virtual event is inserted before the real event. If two virtual events have the same timestamp, their relative order is undefined.

If the events in the log are not in timestamp order, then the sequencing is more complicated. For each declaration of an interval type *I*, virtual events ve_0, $ve_1, \ldots$, of type *start$I*, are inserted into the event stream as follows:

1. ve_0, which has timestamp $t_0 = \mathit{startts} + \mathit{oexpr}$, is inserted before the first real event whose timestamp is at least t_0.

2. ve_i, which has timestamp $t_i = \mathit{startts} + \mathit{oexpr} + (i * \mathit{texpr})$ (for $i = 1, 2, \ldots$), is inserted after ve_{i-1}, and before the first real event whose timestamp is at least t_i and that occurs after ve_{i-1}. If no such real event exists, ve_i is discarded (as are all subsequent virtual events resulting from this interval type declaration).

In addition, two virtual events that are adjacent in the event stream are in timestamp order.

Similarly, the end event declaration for an interval type,

> *e: etype* [***where*** *epred*]

may be replaced by the declaration:

> *e:* ***after*** *texpr*

where *e* is an identifier and *texpr* is a positive number-valued expression. This declaration introduces, for each start event *s* that matches the interval type's start declaration (*s* may be virtual or real), a new virtual time event *v* of type *end$I* where *I* is the interval type name, and with timestamp value *timestamp(s)* + *texpr*. *v* ends the interval that was started by *s*, and is inserted in the event stream before the first real event following *s* that has a timestamp value at least *timestamp(s)* + *texpr*.

A.3.5 Interval Subtypes

If *id* is an identifier, *inttype* is the name of a previously declared interval type, and *mlist* is a comma-separated list of metric definitions, then

```
interval  id = inttype
   [ metrics
      mlist ]
   end id
```

defines a new interval type, named *id*, that is a subtype of *inttype*. The new interval type is nested if *inttype* is nested, and has the same start and end event specification as *inttype*. It inherits all of *inttype*'s metric definitions, in addition to those newly defined in *mlist*. If *mlist* is supplied, the metric names in *mlist* must be different from *inttype*'s metric names.

A.3.6 Procedure Sugar

A ***proc*** statement provides a convenient syntax for declaring events and intervals corresponding to procedures in a program. If *p*, a_1, a_2,. . ., a_n, and *r* are identifiers, a statement of the form

proc *p ([* a_1, a_2, ..., a_n *]) [* ***returns*** *r]*

has the effect of declaring two new events types named *call@p* and *ret@p* and a new interval type named *intv@p*. (These identifiers are special since @ cannot appear in an identifier supplied by a PSpec user). The declarations for these new types are equivalent to:

```
timed event call@p( [ a1, a2, . . ., an ] ); ret@p( [ r, exact ] );
nested interval intv@p =
      s: call@p,
      e: ret@p where thread(s) = thread(e)
   end intv@p
```

The argument list and returns-clause are optional.

An event of type *call@p* corresponds to a call of procedure *p* with integer arguments a_1 through a_n, and an event of type *ret@p* corresponds to a return of *p* with result *r*. Any of a_1 through a_n may be replaced by the symbol *?*, if the corresponding procedure argument is not of interest. If the *returns r* clause is omitted, the equivalent return event is *ret@p(?, exact)*. The *exact* attribute on the return event has a value of 0 or 1, indicating whether the return event's timestamp is approximate or exact. If *exact*=0, then the timestamp was estimated from other events in the log and may be later than the actual time at which the procedure exited. (This is to accommodate some monitoring techniques; for example, inexact timestamps result when procedures exit with exceptions rather than normal returns.) Note that in the interval type *intv@p*, the start and end events are always named *s* and *e*, respectively.

A.4 Imports

An import statement allows names declared in one specification to be used in another specification. A statement of the form: ***import*** *idlist* makes available to the specification containing the import statement the event and interval types defined in the specifications listed in *idlist*. These types are referenced by prefixing the name of the declaring specification followed by "." to the name of the type. For example, if specification *S* declares event type *E*, a specification *T* that imports *S* may refer to *S.E*. The name *S.E* is in *T*'s global scope and is called a *qualified name.*

Event and interval types produced by the ***proc*** statement are a special case. These are never qualified by a specification name. If ***proc*** *p* appears in a specification *S*, the event types *call@p* and *ret@p*, and the interval type *intv@p* are referred to in any specification that imports *S* without qualification. If ***proc*** *p* appears multiple times, its last occurrance takes precedence. (Imports are processed in the order in which they occur in the ***import*** statement.)

A.5 Assertions

An assertion is a predicate (boolean-valued expression) that is expected to be true when a specification is checked against a log. If *e* is a predicate, then ***assert*** *e* is an assertion that *e* should evaluate to *true. e* is evaluated in the global scope. An error is reported by the checker if *e* has the value *UNDEFINED*.

An assertion may be labelled with a double-quoted string for easier identification in messages produced by the checker. A labelled assertion has the form

assert *"string" : e*

A.6 Solve Declarations

The psolve tool accepts specifications with constants declared as unknowns and estimates values for those constants. Solve declarations provide guidance to psolve.

If *id* is an identifier, *idtype* is an event or interval type, *v* and *c* are constants declared as unknowns, *pred* and *e* are expressions, and *m* is a mapping-valued expression, then the following three declarations are solve declarations:

solve *e*
solve data *id : idtype* [***where*** *pred : e*] [*,* ***var*** *v*] [*,* ***cor*** *c*]
solve data *id* ***in domain****(m)* [***where*** *pred : e*] [*,* ***var*** *v*] [*,* ***cor*** *c*]

See the psolve manual page in Appendix D for more information about how solve declarations are used.

A.7 Print Statements

If *e* is an expression (possibly string-valued), then ***print*** *e* evaluates *e* in the global scope and prints its value on the standard output.

A.8 Specifications

If *id* is an identifier and *stmts* is a semi-colon separated list of statements, which may be declarations, assertions, solve statements, imports, or print statements, then:

perfspec *id*
  stmts
end *id*

is a specification. *id* must be different from any top-level identifier in *stmts*.

A.9 Expressions

An expression specifies a computation that produces a value. Expressions are either operands (identifiers or literals), operators applied to arguments that are themselves expressions, triple constructors, mapping constructors, or aggregate expressions.

The operators that have special syntax are classified and listed in decreasing precedence in Figure 7. All infix operators are left associative. Parentheses can be used to override the default precedence rules.

Except as noted below, most operators deal with *UNDEFINED* values by propagating them. That is, if any of the arguments to an operator has the value *UNDEFINED*, the result of applying the operator to its arguments produces the value *UNDEFINED*.

A.9.1 Literals

There are three kinds of literals: numeric, boolean, and string.

The boolean literals are *true* and *false*.

f(x)	function or mapping application
i.f, *n us*,*n ms*, etc.	infix dot for interval or event field access, numeric literals with time units
$-$	unary minus
$*$ / *div mod*	infix arithmetics
$+$ $-$	infix arithmetics
$=$!= $<$ $<=$ $>=$ $>$	infix relations
!	prefix “not”
&	infix “and”
|	infix “or”
=>	infix “implies”
$\longrightarrow$	infix mapping constructor
?	infix “if” operator
˜	infix “else” operator

Figure 7: Operators in order of decreasing binding power.

Numeric literals denote non-negative numbers and use the Modula-3 [7] syntax for integer, real, and longreal literals. All numbers are converted into longreal format internally, but we can still check when necessary whether a number has an integral value.

A string literal is similar to a Modula-3 text literal (although extended characters are not currently supported). No operations are provided on string literals. (Thus, strings can only be printed, assigned to identifiers or metric names, and stored in mappings.)

A.9.2 Triple Constructors

If *v*, *p*, and *m*, are numeric-valued expressions, then *[v,p,m]* is the triple whose components are the values of *v*, *p*, and *m*, in that order. Both *p* and *m* must be non-negative.

A.9.3 Mappings

If *i* is an integral expression and *v* is any expression, then *i*$\longrightarrow$*v* is the single-element mapping with the value of *i* mapped to the value of *v*. The expression *i*$\longrightarrow$*v* is called a single-element mapping constructor. The value of *v* may be *UNDEFINED*, in which case the single-element mapping will contain the value of *i* mapped to *UNDEFINED*. If the value of *i* is *UNDEFINED* then the value of the mapping is *UNDEFINED*.

If $e_1, e_2, \ldots, e_n$ are constant single-element mapping constructor expressions, then the comma-separated, parenthesized expression $(e_1, e_2, \ldots, e_n)$ evaluates to a multi-element mapping. The set of domain values in the multi-element mapping is the union of the domain values of e_1 through e_n and their range values are the corresponding range values. All of the component single-element mappings must have the same type of range value and different domain values; if not, an error is reported.

If *m* is a mapping and *i* is a number-valued expression, then *m(i)* evaluates to the value to which *m* maps *i*. If *i* is not in *m*'s domain, *m(i)* evaluates to *UNDEFINED*. The expression *mapped(m,i)* evaluates to *true* if and only if *i* is in *m*'s domain.

Some of the arithmetic and logical operators are overloaded to work on mappings. These operators provide various ways of combining mappings. In particular, $+$, $*$, &, |, *min*, and *max* can take mappings as arguments. The result of combining a sequence of mappings $m_1, \ldots, m_n$ with one of the above operators *op* is a new mapping *r* whose domain is the union of the domains of m_1 through m_n. For any number *i* in *r*'s domain, $r(i)$ is the value obtained by applying *op* to the sequence of values $m_k(i)$ for all m_k such that $mapped(m_k, i) = true$.

A.9.4 Field Access

If *id* is an identifier bound to an event or interval and *f* is one of its field names (a metric or an attribute), then *id.f* evaluates to the value of the field.

A.9.5 Time Units

Numeric values are unitless. Times computed using the *elapsed* function are also unitless as values, but they represent a time value in some time unit specific to the implementation. In order that a specification writer may use these time values in a sensible way (e.g., to compare them to literals), operators are provided to convert literals in specified time units to their equivalent values in internal time units. If *n* is a numeric literal, then any of the following operators can follow *n* in an expression:

Operator	Meaning
us	microseconds
ms	milliseconds
sec	seconds
min	minutes
hour or *hours*	hours
day or *days*	days

week or *weeks*	weeks
cyc	cycles

For example, *10 ms* evaluates to the real number of internal time units equal to 10 milliseconds. Hence, if *n* is a time in internal time units, $n/(\mathit{1\ ms})$ is the same amount of time in milliseconds.

A.9.6 Arithmetic Operations

Some arithmetic operations are overloaded to work with triples and mappings as well as numbers. The operations on numbers and triples are described here. The section on mapping expressions describes operations on mappings.

A.9.7 Numeric Operations

The numeric operations are: - (unary), - (infix), +, *, /, *div*, *mod*, *min*, *max*, *log*, *power*, *abs*, and *trunc*. The first five of these are defined as in Modula-3.

div and *mod* are infix operations whose arguments must be integral values. They produce integral results and are defined as in Modula-3.

min and *max* are invoked as functions, each taking two numeric arguments and returning a number. *min* returns the minimum of its arguments and *max* returns the maximum.

If *a* and *b* are numeric expressions, then the logarithm to the base *a* of *b* is written *log(a,b)*, and *a* raised to the *b* power is written *power(a,b)*. Both of these operations return numbers.

If *n* is a numeric expression, then *abs(n)* is the absolute value of *n*. *trunc(n)* returns the greatest integral number that is at most *n* for *n* positive, and the smallest integral number that is at least *n* for *n* negative.

A.9.8 Relational Operations

The relational operations are: <, <=, >, >=, =, and != (not equal). These are defined both on numbers and on triples, and the result is a boolean. Their definitions for numbers are as expected. Their definitions on triples are discussed below. An expression of the form "*a op b op c*," where *op* is a relational operator, is equivalent to the expression "*a op b* & *b op c*."

A.9.9 Operations On Triples

In what follows let *t* and *u* be triples of the form *[v,p,m]*. The notations *t.v*, *t.p*, and *t.m* refer to the components of triple *t*. Note that a number *n* can be represented as the triple *[n,0,0]*. Arithmetic for mixed triples and numbers (except for the *log* and *power* operations) is defined first to convert the numbers into the corresponding triples, and then to use triple arithmetic.

The arithmetic operations on triples are defined in Figure 8. The definitions are derived using the notion of a triple as a representation of a range of values with a "preferred" value. The *v* component of the result is the operation applied to the *v* components of the operand triples. The *p* component of the result is defined so that *v+p* for the result is the maximum possible value that could result from applying the operation to values in the ranges of the operands. Similarly, the *m* component is defined so that *v-m* for the result is the minimum possible value for the operation, treating the operands as ranges.

The relational operations on triples are defined in Figure 9. Two triples are considered equal if their ranges overlap. Note that this means that equality on triples is not transitive. A triple *t* is less than a triple *u* if all values in the range represented by *t* are less than all values in the range represented by *u*, and similarly for "greater than."

A.9.10 Logical Operations

The logical operations are: & (and), | (or), ! (not), and => (implication). They have their usual meanings applied to boolean-valued arguments. & and | evaluate all of their arguments.

As described in Section A.9.3, & and | are overloaded to work for mappings as well.

A.9.11 Operations On Events

If *e* is a timed event, then *timestamp(e)* returns the value of *e*'s timestamp, which is a triple whose "preferred value" is the clock value read when the event was recorded. Typically the error associated with a timestamp is plus one clock unit (because the time when the clock is read is actually somewhere between the time value when the clock last ticked and the time of the next tick). However, inexact timestamps (as mentioned in the section on "Procedure Sugar" above) may have a different associated error, reflecting the known bounds on the time. If *e* is not a timed event, evaluating *timestamp(e)* is an error.

thread(e) returns the identifier of the thread that generated event *e*, if thread identifiers are available in the log; otherwise *thread(e)* returns 0.

If *e* and *s* are both timed events, and $ets = timestamp(e)$ and $sts = timestamp(s)$ are triples of the form $[v, p, m]$, then:

$$elapsed(e, s) = [ets.v - sts.v, ets.p + sts.m,$$

$$
\begin{aligned}
t + u \quad &= [t.v + u.v, t.p + u.p, t.m + u.m] \\
-t \quad &= [-t.v, t.m, t.p] \\
t - u \quad &= t + (-u) \\
t * u \quad &= [t.v * u.v, \\
&\qquad \max_{i \in t, j \in u}\{i * j\} - t.v * u.v, \\
&\qquad t.v * u.v - (\min_{i \in t, j \in u}\{i * j\})] \\
1/t \quad &= \text{if } 0 \in t \text{ then error} \\
&\quad \text{else } [1/t.v, \\
&\qquad\qquad \max\{1/(t.v + t.p), 1/(t.v - t.m)\} - 1/t.v, \\
&\qquad\qquad 1/t.v - \min\{1/(t.v + t.p), 1/(t.v - t.m)\}] \\
t/u \quad &= t * 1/u \\
\mathit{abs}(t) \quad &= \text{if } t.v - t.m \geq 0 \text{ then } t \\
&\quad \text{elseif } t.v + t.p \leq 0 \text{ then } -t \\
&\quad \text{else } [|t.v|, \max(t.v + t.p, t.m - t.v) - |t.v|, |t.v|] \\
\mathit{trunc}(t) \quad &= [\mathit{trunc}(t.v), \\
&\qquad \mathit{trunc}(t.v + t.p) - trunc(t.v), \\
&\qquad \mathit{trunc}(t.v) - \mathit{trunc}(t.v - t.m)] \\
\mathit{min}(t, u) \quad &= [\min(t.v, u.v), \\
&\qquad \min(t.v + t.p, u.v + u.p) - \min(t.v, u.v), \\
&\qquad \min(t.v, u.v) - \min(t.v - t.m, u.v - u.m)] \\
\mathit{max}(t, u) \quad &= [\max(t.v, u.v), \\
&\qquad \max(t.v + t.p, u.v + u.p) - \max(t.v, u.v), \\
&\qquad \max(t.v, u.v) - \max(t.v - t.m, u.v - u.m)] \\
\mathit{log}(b, t) \quad &= [\log_b t.v, \\
&\qquad \log_b(t.v + t.p) - \log_b t.v, \\
&\qquad \log_b t.v - \log_b(t.v - t.m)] \\
\mathit{power}(b, t) &= [b^{t.v}, b^{t.v + t.p} - b^{t.v}, b^{t.v} - b^{t.v - t.m}] \\
\mathit{power}(t, b) &= [t.v^b, (t.v + t.p)^b - t.v^b, t.v^b - (t.v - t.m)^b]
\end{aligned}
$$

Figure 8: Arithmetic operations on triples. t and u are triples. b is a number. For *log* and *power*, $(t.v - t.m)$ must be greater than 0; otherwise, the result is unspecified. The notation $i \in t$ means $t.v - t.m \leq i \leq t.v + t.p$.

$$
\begin{aligned}
&t = u \equiv (t.v - t.m \leq u.v + u.p) \wedge (t.v + t.p \geq u.v - u.m)\\
&t \mathrel{!=} u \equiv !(t = u)\\
&t > u \equiv t.v - t.m > u.v + u.p\\
&t < u \equiv t.v + t.p < u.v - u.m\\
&t \geq u \equiv t = u \vee t > u\\
&t \leq u \equiv t = u \vee t < u
\end{aligned}
$$

Figure 9: Relational operations on triples.

$$\min(ets.v - sts.v, ets.m + sts.p)]\ .$$

This definition of *elapsed* gives the elapsed time between the two events, under the assumption that $ets.v \geq sts.v$ (so all the values in the triple representing the elapsed time are non-negative).

A.9.12 Operations on Intervals

If *i* is an interval whose start and end events are both timed, then *elapsed(i)* returns a triple representing the elapsed time for interval *i*. If *e* is the end event of *i* and *s* is the start event, then:

$$elapsed(i) = elapsed(e, s)$$

Note that *elapsed* is overloaded; it can take a single interval-valued argument or two triple-valued arguments.

A.9.13 Aggregates

An aggregate expression describes a computation over a sequence of values. The sequence of values is produced by introducing a new identifier that gets bound to a series of values in a specified range and evaluating a specified expression (which may use the identifier) for each binding. The sequence resulting from evaluating the expression for each binding is then combined using a specified aggregate operator.

The range may be specified in two ways: it may be a sequence of events or intervals of a specified type and satisfying a specified predicate, or it may be the values in the domain of a mapping that satisfy a specified predicate.

We define an implicit component of a scope, called the "current event sequence," that is used in evaluating aggregate expressions that range over events and intervals. In the global scope, the current event sequence is all events and intervals in the event stream. Within the scope of an interval declaration, it is all events and intervals wholly contained between (and not including) the start and end events of the interval being declared. For the purpose of defining the current event sequence for intervals, intervals are ordered by their end events. If two intervals have the same end event, they can appear in either order in an interval sequence.

The first form of aggregate expression (ranging over the current event sequence) is written:

$\{$*op id : idtype* [***where*** *pred*] *: expr*$\}$

where *op* is an aggregate operator, *id* is an identifier, *idtype* is an event or interval type name, *pred* is a boolean-valued expression, and *expr* is an expression. *pred* and *expr* may use *id* but may not reference any identifiers bound by outer aggregate expressions. The where-clause may be omitted. If the where-clause is present and has the value *UNDEFINED* for any value of *id*, the value of the entire aggregate expression is *UNDEFINED*. The values bound to *id* are those events or intervals in the current event sequence that have type *idtype* and for which *pred* is *true*.

The second form of aggregate expression (ranging over the domain of a mapping) is written:

$\{$*op id* ***in*** *domain(m)* [***where*** *pred*] *: expr*$\}$

where *op*, *id*, *idtype*, *pred*, and *expr* are as above, and *m* is a mapping-valued expression. The where-clause may be omitted. If the where-clause is present and has the value *UNDEFINED* for any value of *id*, the value of the entire aggregate expression is *UNDEFINED*. The values bound to *id* are those values in the domain of *m* for which *pred* is *true*. The domain values are produced in an arbitrary order.

The aggregate operators are:

$+$ $*$ & | *min max mean stdev var the last first count*

The *count* operator is special—“*: expr*” is omitted from the aggregate expression, and the result is the number of different values to which *id* gets bound. (*count* is provided for convenience and readability. The same result is produced using the $+$ operator and letting *exp* be the constant 1.) The other aggregate operators are defined to work on mapping-valued expressions as well as on non-mapping values.

For non-mapping values, the operators are defined as follows. The definitions of the first six operators ($+$, $*$, &, |, *min*, *max*) are simply extensions of their definitions for two arguments. If $a_1, \ldots, a_n$ is a sequence of values and *op* is one of these six operators, then the result of combining the sequence values with the

op	empty seq.	single element v
+	0	v
*	1	v
&	*true*	v
|	*false*	v
min, max, mean, the, last, first	error	v
stdev, var	error	error
count	0	1

Figure 10: Boundary case results for aggregate operators

operator is $(((a_1\ op\ a_2)\ op\ \ldots)\ op\ a_n)$. (In other words, these six operators are reduction operators.) The result when the sequence is empty or has only one element is defined in Figure 10.

The operators *mean*, *stdev*, and *var* compute the arithmetic mean, standard deviation, and variance, respectively, of the sequence values. For a sequence $a_1, \ldots, a_n$ of numbers these are defined as:

$$mean(a_1, \ldots, a_n) = \tfrac{1}{n} \textstyle\sum_{i=1}^{n} a_i$$

$$var(a_1, \ldots, a_n) = \tfrac{1}{n-1} \textstyle\sum_{i=1}^{n} (a_i - \overline{a})^2$$
$$\quad \text{where } \overline{a} = mean(a_1, \ldots, a_n)$$

$$stdev(a_1, \ldots, a_n) = \sqrt{var(a_1, \ldots, a_n)}$$

(The variance formula is what statisticians would call a “modified sample variance” or “unbiased estimate” of the variance). The result of applying one of these operators to a sequence of triples is computed using just the first component of the triples (the “favored values”).

The expression $\{$*first id : idtype : expr*$\}$ evaluates to the value of *expr* with *id* bound to the first value of type *idtype* in the aggregate’s range. Similarly, $\{$*last id : idtype : expr*$\}$ uses the last value of type *idtype*.

The operator *the* applied to a single-element sequence returns the single value. Applying *the* to a multi-element sequence results in the value *UNDEFINED*.

Finally, when the sequence values are mappings, the aggregate operator defines how to combine the mappings. The result of combining a sequence of mapping values with an aggregate operator is a new mapping whose domain is the union of the domains of the mapping values. The value to which the new mapping maps a domain element *i* is the result of combining the values to which *i* is mapped by the mappings in the sequence using the aggregate operator.

A.9.14 Conditional Operations

The infix operators *if* (written *?*) and *else* (written ˜ (tilde)) provide for conditional evaluation of expressions.

Let *e1* be a boolean-valued expression and *e2* be any other expression. If *e1*'s value is equal to *true* then the expression *e1 ? e2* has the value of *e2*. Otherwise (if e1 is *false* or *UNDEFINED*), the value of *e1 ? e2* is *UNDEFINED*.

Let *e3* and *e4* be expressions that have the same type of value. If *e3* is not *UNDEFINED* then the expression *e3 ˜ e4* has the value of *e3*. Otherwise it has the value of *e4*.

The *?* operator binds more tightly than ˜. Combining the two gives the expected "if-then-else" semantics. The expression *e1 ? e2 ˜ e3* has the value of *e2* if *e1* is *true*, and otherwise has the value of *e3*. *e2* and *e3* must have the same types.

A predicate is provided for testing whether an expression's value is defined. The expression *defined(e)* where *e* is any expression, evaluates to *true* if *e*'s value is not *UNDEFINED* and *false* otherwise.

A.10 Grammar

In the grammar that follows, italicized words are non-terminals, words in typewriter font are literals, *term*,+ means one or more occurrences of *term*, separated by commas, and *term*,* means zero or more occurrences of *term*, separated by commas. (*term*;+ and *term*;* are defined similarly). Optional elements are enclosed in square brackets. Comments in PSpec are preceded by `%` and extend to the end of the line.

```
spec ::= perfspec id import;* stmt;+ end id
import ::= import id;+
stmt ::= def constdef;+
        | solve solvedecl;+
        | [ timed ] event eventdef;+
        | [ nested ] interval intervaldef;+
        | proc sig;+
        | assert [ string : ] expr;+
        | print expr;+
sig ::= id [ ( [ argid,+ ] ) ] [ returns id ]
argid ::= id | ?
```

```
constdef ::= id = expr | id = ?
solvedecl ::= [ data varrange : ] expr [ , var id [ , cor id ] ]
eventdef ::= id ( id,* )
intervaldef ::= id = intervalhead [ metrics metricdef,+ ] end id
intervalhead ::= intvstart , intvend
intvstart ::= varrange
                | id : [ from expr ] every expr
intvend ::= varrange | id : after expr
```

Note: *expr*s following `from`, `every`, and `after` must evaluate to a constant number, which is interpreted as a number of internal time units. In addition, they cannot contain *id*s.

```
metricdef ::= id = expr
expr ::= expr op expr
          | - expr
          | [ expr , expr , expr ]
          | expr -> expr
          | expr ( expr,* )
          | expr ? expr
          | expr ~ expr
          | aggrexpr
          | ( expr [ , expr ]* )
          | const
          | id
          | expr . id
aggrexpr ::= { aggrop varrange [ : expr ] }
varrange ::= id varset [ where expr ]
varset ::= : intevtname | in expr
aggrop ::= + | * | & | | | count | mean | stdev
             | var | max | min | the | last | first
op ::= arithop | relop | boolop
arithop ::= + | - | * | / | div | mod
relop ::= < | <= | > | >= | = | !=
boolop ::= & | | | ! | =>
const ::= num | num timeunit | true | false
timeunit ::= us | ms | sec | min | cyc | hour | hours
                | day | days | week | weeks
intevtname ::= [ id . ] unqualname
unqualname ::= id | call@id | ret@id | intv@id
id ::= alpha [ alphanum+ ]
alpha ::= _ | a | b | ... | z | A | B | ... | Z
```

alphanum ::= *alpha* | *digit*

num ::= *digit*+ [`.` *digit*+ [*exp* [`-`] *digit*+]]

exp ::= `E` | `e` | `D` | `d` | `X` | `x`

digit ::= `0` | `1` | `2` | `...` | `9`

string ::= `"` *char** `"`

char ::= *printingchar* | *escapedchar*

escapedchar ::= `\n` | `\t` | `\r` | `\f` | `\\` | `\"`
| `\` *octaldig octaldig octaldig*

octaldig ::= `0` | `1` | `...` | `7`

printingchar ::= any character with ascii code 0x20 to 0x7e,
excluding `\` (0x5c) and `"` (0x22)

A.11 Built-in Functions

The built-in functions names that can be used in expressions and their signature(s) (multiple if the function names are overloaded) are shown in Figure 11.

The type *map[t]* is the type of mappings with range type *t*.

Name	Signature(s)
max, min	num × num ⟶ num
	num × triple ⟶ triple
	triple × num ⟶ triple
	triple × triple ⟶ triple
	map[t] × map[t] ⟶ map[t]
	(where t is num, triple*, or* map[t’]*)*
power	num × num ⟶ num
	num × triple ⟶ triple
	triple × num ⟶ triple
log	num × num ⟶ num
	num × triple ⟶ triple
elapsed	interval ⟶ triple
	triple × triple ⟶ triple
abs	num ⟶ num
	triple ⟶ triple
trunc	num ⟶ num
	triple ⟶ triple
timestamp	event ⟶ triple
thread	event ⟶ num
defined	any ⟶ bool

Figure 11: Signatures of built-in functions.

B PSpec Tools - pcheck, peval, psolve

This appendix contains the Unix-style manual pages for the PSpec checker, solver, and evaluator.

Name

pcheck, peval, psolve - PSpec tools

Syntax

peval [**-s** *specfile*] [**-etp** *etplog exefile*] [**-etpsym** *etplog symfile*] [**-trace** *tracefile*]
    [**-r** *clockres*] [**-i** *specpath*] [**-v** *intervalfile*] [**-e** *eventfile*] [**-c** *cmdfile*]
pcheck **-s** *specfile* [**-etp** *etplog exefile*] [**-etpsym** *etplog symfile*] [**-trace** *tracefile*]
    [**-r** *clockres*] [**-i** *specpath*] [**-v** *intervalfile*] [**-e** *eventfile*] [**-f**] [**-cont**]
psolve **-s** *specfile* [**-etp** *etplog exefile*] [**-etpsym** *etplog symfile*]
    [**-trace** *tracefile*] [**-r** *clockres*] [**-i** *specpath*]
    [**-v** *intervalfile*] [**-e** *eventfile*] [**-d** *datafile*] [**-u** $name_1 \ldots name_n$]

Description

PSpec is a language together with a set of tools for testing the performance of software. This man page describes a number of the PSpec tools.

Checker

Pcheck takes as input a specification written in the PSpec language and a monitoring log in an appropriate format and reports which assertions in the specification fail to hold for the run represented by the log. A message is printed for each assertion failure, giving the line number of the assertion in the specification file. Where appropriate, the particular events or intervals in the log that caused the assertion to fail are listed. An interval is identified by a uid (the same as is printed in *intervalfile* when using the **-v** flag). An event is identified by a two part index giving its position in the event stream. (This index is printed in *eventfile* when using the **-e** switch. For real events, the first part of the index gives the event's position in the underlying log, and the second part is 0; for virtual events, the first part is equal to the first part for the closest preceding real event, and the second part is monotonically increasing for adjacent virtual events in the event stream.) A message is also printed for each true assertion, unless the **-f** flag is given. It is also possible to use the PSpec ***print*** statement in a specification to cause pcheck to print out values of arbitrary PSpec expressions. These will be printed after the messages about which assertions failed.

Use the **-cont** switch on the pcheck command line when it is to be run in continuous monitoring mode, that is, when the reports of assertion failures should be printed as they are encountered, rather than waiting until the entire log has been processed. All print statements will still be done after the end of log has been reached. (This may be changed in future versions to allow continuous printing.)

Evaluator

Peval provides an interactive read-eval-print loop for evaluating PSpec expressions relative to a specific monitoring log and PSpec performance specification. The evaluator repeatedly prompts the user to enter a command, evaluates the command, using the monitoring log if necessary, and reports the result.

The evaluator accepts many of the commands that can appear in a PSpec specification, with slightly modified syntax (all commands must be followed by a semicolon, and expressions are not preceeded by a keyword). In addition to being able to evaluate expressions, a peval user can define new named constants, define new interval types, and declare new procs. The interval, event, and constant names accessible in the evaluator include those defined or imported in *specfile* (if it is supplied) as well as those defined interactively. It is not possible to redefine names in a peval session.

In addition, the evaluator accepts two commands that cannot appear in a PSpec specification. The ***echo*** command, followed by a double-quoted string, writes the string (without quotes) to the standard output. This is particularly useful when the evaluator is used in a batch mode (with its input coming from a file) in order to record in the output what is being evaluated. The ***help*** command prints out help on using the evaluator, including the grammar of the input language.

Hint: The evaluator normally prints timestamp and elapsed time values in “ticks” (internal time) units. To examine such a value in more reasonable units, simply divide by the unit of choice. For example, if *t* is an elapsed time value, typing *t/1ms* to the evaluator will print the value of *t* in milliseconds. This works because the expression *1 ms* evaluates to the number of ticks equivalent to one millisecond.

Solver

Psolve helps to estimate values for unknown constants in PSpec performance specifications using data in a monitoring log. The specification must contain unknowns and solve declarations. The output, written to the standard output, is a revised specification with the unknowns bound to their estimated values. By default, psolve computes values for all unknowns appearing in the specification. To only solve for specific unknowns, use the **-u** switch described below. See the section More on Psolve below for more details.

Flags

-s *specfile* *specfile* is the name of the file containing a PSpec specification. The file must exist in some directory on the search path, which defaults to ".:../src" and may be changed with the **-i** switch

-etp *etplog exefile* For etp logs, *etplog* is the name of the etp log file and *exefile* is the name of the associated executable that produced the log. The executable can be either the original, or etp'd version. If *etplog* is "-" the log is read from the standard input. Note that for peval, if the log is on the standard input, the **-c** flag is required for specifying the source of expressions to be evaluated. Either the **-etp** switch or the **-trace** switch must be specified.

-etpsym *etplog symfile* Like the **-etp** switch above, however *symfile* is given as the source of symbols in the log instead of an executable. *symfile* is an ascii file, in the form generated by the command "nm -Bpd" run on the executable file. This switch is useful in two cases: *symfile* is probably smaller than the executable, so if you want to keep a log around for later evaluation but don't want to save the executable, you can just save the symbols. Also, it is useful when generating an etp-format log (probably by a program other than etp) on a machine where the executable does not contain the desired symbols, or where the "nm" program cannot be used to read the symbols.

The file *symfile* must have the following format. The symbol definitions are a sequence of ascii lines, one definition per line, where each line has:

```
<addr> <type> <symbol>
```

where `<addr>` is the decimal value of the symbol, `<type>` is a single letter, and `<symbol>` is the symbol name. Only lines with `<type> = 't'` or `<type> = 'T'` are used by PSpec. `<addr>`, `<type>`, and `<symbol>` must be separated by exactly one blank or tab and `<addr>` must not be preceded by whitespace.

-trace *tracefile* For MIPS trace logs, *tracefile* is the name of the ascii log file. If *tracefile* is "-" the log is read from the standard intput. Note that for peval, if the log is on the standard input, the **-c** flag is required for specifying the source of expressions to be evaluated. Either the **-etp** switch or the **-trace** switch must be specified.

-r *clockres* Optional flag to specify the resolution of cycle counter "tick" values used in the log in the case where the log does not specify the resolution itself. *clockres* can be the special symbol "myclock", in which case the cycle counter value is measured on the current machine and used to compute timestamp values in the log regardless of where the log was generated. Otherwise, *clockres* should be expressed as a floating point number of seconds per tick. For example, for a 40 ns clock the value would be `40e-9`. If the flag is not specified and the log does not contain the cycle counter resolution but the log was generated on the current machine, the cycle counter value of the current machine is used. Otherwise, the cycle counter resolution cannot be determined and an error message will be printed.

-i *specpath* Optional flag that changes the search path for *specfile* and any specifications it imports. *specpath* has the form of a colon-separated list of directory names.

-v *intervalfile* Optional flag which says to write to the file named *intervalfile* detailed information (in ascii) about the intervals in the log file (as interpreted relative to *specfile*). peval writes and closes the file on startup so that it can be examined while peval is in use.

-e *eventfile* Optional flag which says to write to the file named *eventfile* detailed information (in ascii) about the events relevant to *specfile*. peval writes and closes the file on startup so that it can be examined while peval is in use.

-c *cmdfile* Optional flag for peval which tells it to read the expressions to be evaluated from the file *cmdfile*. This flag is required when the log for peval is on the standard input.

-f Optional flag for pcheck which tells it to report failures only (normally, true assertions are also reported).

-cont Optional flag for pcheck which tells it to report failures as they occur, rather than waiting until the end of log.

-d *datafile* Optional flag for psolve which tells it to write to *datafile* all the datapoints used in linear regressions.

-u $\textbf{\textit{unknown}}_1 \ldots \textbf{\textit{unknown}}_n$ Optional flag for psolve which tells it to solve for just the unknowns named $\textit{unknown}_1$ through $\textit{unknown}_n$. These names must be fully qualified identifiers (of the form *specname.defname*). If this flag is absent, values for all unknowns are computed.

Error Messages

Parse and Evaluation errors

Errors in *specfile* or any file it imports are reported with the specification name (if available) and the character offset in the file near where the error occurred, followed by a message indicating the problem. Errors in expressions typed to the evaluator are reported with the character offset (admittedly, not the most useful form of message).

Log errors

For errors in log files, a message indicating the problem, along with the byte offset in the log file (where appropriate) are printed. The message "log format mismatch" is reported when the log format version number in the log header is one that pcheck does not recognize.

Notes

Shared libraries are not yet supported.

Monitoring

Monitoring logs are the only connection between program runs and the Pspec tools; therefore they must contain all information necessary to check performance assertions about program runs.

A monitoring log is logically a sequence of "events" corresponding to points in a program's execution. An event has a type and some number of numeric attribute values. It may also have a timestamp and a thread id. Events in monitoring logs correspond to events in PSpec specifications. The mechanism for matching event types in logs to event types in specifications depends on the monitoring mechanism being used.

Monitoring with Etp

etp (the "elapsed time profiler") provides a convenient mechanism for generating logs of events that can be processed by the PSpec tools. Most of the events in an etp log correspond to calls and returns of procedures. There are also events marking log buffer flushes and the end of the log. For monitoring logs produced by etp, log events are matched to PSpec events as follows:

- If the PSpec specification contains a ***proc*** declaration for procedure *P* the resulting PSpec events of types *call@P* and *ret@P* correspond to log events that are calls and returns, respectively, of the procedure named *P*. (For a log produced from a Modula-3 program, a procedure name is the fully qualified Modula-3 name with _ replacing "."). In addition, there will be fabricated events of type *ret@P* that don't appear explicitly in the etp log but can be inferred to have occurred based on stack information in the log.

  The attributes for call events correspond, in order, to at most the first four integer arguments to the procedure being called. The first named attribute for a return event is the first integer result returned by the procedure. The second named attribute for a return event indicates whether the return appeared in the log explicitly (and therefore the return event's timestamp is exact) or was inferred (and therefore the return event's timestamp is approximate). The attribute value is 1 if the return was explicit and 0 otherwise.

- An event type declared with name *E* in a PSpec specification named *S* corresponds to events in the log that are calls of a procedure named *S_E*. The attribute values of *E* come from up to the first four integer arguments to *S_E*.

- A PSpec event of type *logflush@* corresponds to an etp log flush event. It has a single attribute, *time*, whose value is the time spent flushing the log (in internal timestamp units).

- A PSpec event of type *logstart@* is a fabricated event with the same timestamp as the first event in the log.

- A PSpec event of type *logend@* corresponds to the end-of-log event in the etp log.

The timestamps and thread ids for all event types come from the timestamps and thread ids recorded in corresponding log events (timestamps are made relative to the first event in the log), except for inferred return events, where the timestamp and thread id comes from the explicit log event following the inferred event. For inferred return events, the error in the timestamp is computed using the event occurring immediately before the event that produces the inferred return. For all other events, the error in the timestamp is plus one clock unit. Timestamps are converted to be relative to the first event in the log, which is assigned a timestamp of 0.

Monitoring with Trace

Another way on MIPS/Ultrix systems to generate a log that the PSpec tools can understand is to use the trace(1) command. This command generates logs of system call and return events for a process, in ascii format, one line per event.

Trace log events are matched to PSpec events as follows. A trace event marking the invocation of the system call named *S* corresponds to the PSpec event *call@S*. Similarly, a trace event marking the return of the system call named *S* corresponds to the PSpec event *ret@S*. These PSpec events are declared in a PSpec specification using the procedure syntactic sugar, with the system call name as the name of the procedure. The timestamp of the PSpec event is the timestamp of its corresponding trace entry; the error in the timestamp is plus one clock tick. The thread id is the process id in the trace entry. For invocation events, the attributes are the numeric attributes in the trace entry. For return events indicating successful system call completion, the first attribute is the return value, and the second attribute is 1 (indicating that the return event is explicit). For return events indicating that an error occurred, the first named attribute is the negative of the error number returned and the second named attribute is 1 (indicating explicit). Additionally, there is an initial fabricated event of type *logstart@* with the same timestamp as the first event in the log, and a final fabricated event of type *logend@* with the same timestamp as the last event in the log.

More on Psolve

Unknowns

An unknown is a symbolic constant whose value is left undetermined. It is introduced into a specification using the ***def*** statement:

def *id* = ***?***

where *id* is the name of the unknown.

Solve Declarations

Solve declarations come in two forms. The simple form contains an equation that is linear in one unknown:

solve *eqn* .

eqn has to be an equation, where the expression on one side of the equal sign is constant (contains no unknowns), and the expression on the other side is “linear” in one unknown. In general, an equation is linear in *k* unknowns if it has the form of a sum or difference of terms, where each term is one of the following:

texpr1 $*$ *texpr2*, where one of the *texprs* is an unknown
and the other is a constant expression,
texpr1 / *texpr2*, where *texpr1* is an unknown and *texpr2* is a
constant expression, or
texpr, which is an unknown or a constant expression.

Exactly *k* terms contain unknowns. Each term must have a different unknown from any other term, and at most one term may be an unknown by itself. At least one term must contain an unknown.

The unknown in the simple form of solve declaration is solved for algebraically. The constant expressions may contain log-aggregate expressions, which will be evaluated relative to the supplied monitoring log.

The second form of solve declaration describes how to compute a set of data points and gives an equation for use in a linear regression analysis that produces estimates for the values of the unknowns. It is written:

solve data *id : idtype [* ***where*** *pred] : eqn [,* ***var*** *vid [,* ***cor*** *cid]]*

or

solve data *id* **in domain***(mid) [* ***where*** *pred] : eqn [,* ***var*** *vid [,* ***cor*** *cid]]*

where *id*, *idtype*, *mid*, *vid*, and *cid* are identifiers, *pred* is a boolean-valued expression, and *eqn* is an equation linear in some number of unknowns.

In the first case, the data values for the regression are computed using the log. *idtype* must name an event or interval type. The constant expressions in the equation are evaluated with *id* bound to each event or interval of type *idtype* in turn to produce one data point for the regression. The total number of data points will be equal to the number of matching events or intervals in the log.

In the second case, the data values for the regression are computed using the elements of a mapping. *mid* must name a mapping. The constant expressions in the equation are evaluated with *id* bound to each element of the mapping's domain in turn to produce one data point for the regression. The total number of data points will be equal to the size of the mapping's domain.

In both cases, where-clauses may be used to restrict the set of events or intervals or domain values in the iteration.

The ***var*** and ***cor*** clauses, if present, name identifiers to be bound to the computed variance and coefficient of correlation of regression. These identifiers must have been declared previously as unknowns.

Hints

It is a good idea to plot the data points for a linear regression along with the line estimated by psolve to see whether the estimate is reasonable (and whether the data is linear to start with). The output of psolve can be transformed without too much trouble to input for gnuplot(1). For example, suppose you use psolve to estimate the constants a and b in the equation $y = a * x + b$. Use the **-d** switch to psolve to get a dump of the data points for the regression. Suppose the data points are in the file “solve.data” and the estimates produced by the solver for a and b are 3.0 and 4.5 respectively. You could run gnuplot and issue the following commands to see a plot of the data and estimated line:

```
f(x) = 3.0 * x + 4.5
plot 'solve.data' using 2:1, f(x)
```

See Also

For topics related to psolve, see a textbook on linear regression or statistics. The following texts may be useful:

Robert V. Hogg and Elliot A. Tanis, Probability and Statistical Inference, Macmillan Publishing Company, 1988.

Murray R. Spiegel, Theory and Problems of Statistics, Schaums Outline Series in Mathematics, McGraw-Hill Book Company, 1988.

C CMon Tools - telemonitor, telemonreg, snarflog

This appendix includes the Unix-style manual pages for the telemonitor, registry, and snarflog tools.

C.1 telemonitor

Name

telemonitor - user interface for processing logs during continuous monitoring

Syntax

telemonitor [-**debug**] [-**regmachine** *machine*] [-**regname** *name*] [-**rc** *file*]

Description

The telemonitor is a part of the Continuous Monitoring system. It provides a user interface for selecting programs to monitor and for processing log data. The telemonitor can connect to programs already running and can discover when programs start up.

A telemonitor must be able to connect to an instance of a registry (see the telemonreg(1) manual page below) in order to find program instances that are available for monitoring. Unless explicitly stated, the registry is assumed to be running on the same machine as the telemonitor, and is assumed to be exported under the name “TelemonRegistry”. To cause the registry to be found under a different name or on a different machine, either use the command line switches **-regname** and **-regmachine**, or set the environment variables TELEMONREG_NAME and TELEMONREG_MACHINE to the desired values.

When the telemonitor starts up, it attempts to read its initial settings from a configuration file. If the **-rc** command line switch is given with a file name, it reads that file. If no **-rc** switch is given, it checks the environment variable TELEMONRC for a file name, and if that is undefined, it tries “$HOME/.telemonrc”. The format of configuration files is described below.

User Interface

The primary user interface of the telemonitor shows two pieces of information: the classes of programs for which the telemonitor is registered to receive notifications when an instance starts running, and the running instances of programs in those classes. Each class of programs has the following information associated with it:

- The program name, machine name, and process id of program instances in the class (any of which may be empty, indicating that any value is acceptable). The sets of instances represented by two different classes must not be equal. (They can overlap.) The “program name” is the name under which the program is registered with telemonreg.

- A csh command line specifying a tool or set of tools to run over the log data; the log data is passed to the processes created by the command line via standard input, and the standard output of the processes is captured and written to a typescript that can be displayed in the telemonitor. The csh variables M_MACHINE, M_PROG, and M_PID will be set to the machine name, program name, and pid, respectively, of the program instance providing the log. These variables can be used on the command line, preceded by `$`; an example command line might be: `cat >/tmp/log.$M_MACHINE.$M_PID`.

- Control flags, indicating: (1) whether it is acceptable to block the monitored program (to avoid losing log data when the monitoring tool cannot process data as fast as the monitored program generates it); and (2) whether monitoring should be started automatically when an instance of the class starts up, or whether the instance should simply be added to the list of running instances to be available for the user to start monitoring later.

Controls are provided to add, delete, and modify program classes.

When the telemonitor is informed of a new running instance for one of its registered classes, it adds the instance to the list of running instances, highlighting it to indicate that it is new. If the instance’s class is configured for automatic monitoring, the specified command will be started and log data will be passed to it. When a class is configured for automatic monitoring, the telemonitor synchronizes with the monitored program to ensure that all log data is captured from the beginning of the program’s execution.

A status/control window can be popped up for any instance in the list of running instances. This windows provides controls to start and stop monitoring for the instance and to view the output from the tool that is processing the log data, and to direct the tool’s output to a file. Summary status information is also displayed indicating the amount of data captured so far, and how long monitoring has been going on.

The telemonitor maintains a set of default settings for new classes; these can be modified using the “Config” menu in the user interface. The current state (consisting of the default settings and the list of classes) can be saved to a file (default ~/.telemonrc) or restored from a file. The file should contain a sequence of S-expressions, each describing either the defaults for new classes or a single class.

Exactly one default can be provided, and it must come first.

Configuration File Syntax

The S-expressions come from the following grammar:

```
description ::= ( (defaults | class) proplist )
proplist    ::= property*
property    ::= ( prop value )
prop        ::= prog | pid | machine | block | auto | tool
value       ::= text | boolean
boolean     ::= TRUE | FALSE
```

(*Text* is a double-quoted string.)

Prop names are case-sensitive. The type of a *prop*'s value is constrained as follows: `prog`, `pid`, `machine`, and `tool` require type *text*; `block` and `auto` require type *boolean*. If a given property appears multiple times in a description, the last value provided for it is used. Blank lines and other whitespace outside of *text*s are ignored. A comment starts with a semicolon and ends with a new line, and is ignored. (Note: for a value specified for `pid` to be useful, it must be a positive integer.)

If a text-valued *prop* is not specified for a class, its value is taken from the defaults specified; if there is no default, its value is empty. Similarly, for a boolean-valued prop, if there is no default, its value is `TRUE`.

Error Conditions

The telemonitor usually pops up an error log window when an error occurs. This window can be recalled at any time using the "Show Error Log" button in the upper right corner of the telemonitor UI. The "registry unavailable" message is reported specially, in the upper left corner of the UI, rather than in the error log window.

Here is a partial list of the possible error messages and their causes.

- "Registry Unavailable": the telemonitor lost contact (or could not initially make contact) with a registry of the particular name and on the particular machine indicated in the top left corner of the telemonitor window. While the registry is unavailable, the telemonitor will not be able to learn about new running instances of programs available for monitoring, but existing monitoring sessions will continue to work. To correct the situation, check to see whether the registry is running on the specified machine under the specified name. The telemonitor will repeatedly try to reconnect to the registry until it is successful (so if the registry dies, it can be restarted without restarting the telemonitor). For now, it is not possible to redirect a running telemonitor to use a different registry. It must be restarted with the new registry name and machine.

- “Could not create process for tool command”: The telemonitor encountered an error when trying to start the tool command process and connect it to a monitored program. Check the tool command to make sure it does not contain errors.

- “Failed writing to tool command”: An error occurred trying to write the monitoring log on the standard input of the tool command process. Check whether the tool command appears to still be running and whether it has an error.

- “Failed reading log from monitored program”: An error occurred trying to transfer the log from the monitored program to the telemonitor. Check whether the monitored program is still running. It is possible that the monitored program is still running but has dropped the connection to this telemonitor.

- “Failed closing pipe to tool command”: An error occurred trying to shut down the tool command process cleanly. You can probably ignore this error in most cases, but it is reported just in case.

- “Failed calling donerd on snarflog for ...”: The telemonitor received an error trying to shut down the connection to a monitored process. (This message should probably be reworded.) You can probably ignore this error in most cases.

- “Failed getting output from tool command”: An error occurred reading the standard output from the tool command process. Check whether the tool command still seems to be running.

- “Getrd raised NetObj.Error...”: There was an error trying to contact a monitored program. Check whether the monitored program is still running.

- “Instance already connected (couldn’t start monitoring)...”: Another telemonitor is already monitoring the named instance. If it later disconnects this telemonitor could have the opportunity to try connecting again, but it will not happen automatically.

C.2 telemonreg

Name

telemonreg - global registry for the Continuous Monitoring system

Syntax

telemonreg [**-debug**]

Description

telemonreg is part of the Continuous Monitoring system. It acts as a global, well-known registry for instances of the snarflog and telemonitor programs. Upon starting up, snarflog reports its existence, as well the program it represents, the machine on which it runs, and its process identifier. If there are any instances of telemonitor that have indicated an interest in that particular snarflog, then they are notified and given an opportunity to receive log records from the snarflog instance. Likewise, when a user of a telemonitor adds a new class of programs in which that user is interested, the telemonitor registers this interest with the telemonreg, and is notified of any already existing snarflog instances that are in that class. Only one instance of a telemonreg with a given TELEMONREG_NAME should be run on any machine. In general, a single instance of a telemonreg should be used for all snarflog and telemonitor instances that need to find each other.

Options

-debug Cause the telemonreg to print diagnostic messages when snarflog instances come and go, as well as when telemonitor instances register and unregister program classes.

Environment

TELEMONREG_NAME The name telemonreg should use to export a reference to itself. The default is “TelemonRegistry”.

C.3 snarflog

Name

snarflog - send log data extracted from an etp’d executable or traced program to a telemonitor

Syntax

snarflog *progname* <**-etp**|**-ascii**> [**-buffer** *nbytes*] [**-debug**]
snarfetp *progname* [**-buffer** *nbytes*] [**-debug**]
snarfascii *progname* [**-buffer** *nbytes*] [**-debug**]

Description

The snarflog program is part of the Continuous Monitoring system. It extracts log data from a running program and forwards the data to a telemonitor (which in turn provides a user interface for processing the data with a variety of tools). There are currently two versions of snarflog, for processing etp log data or ascii log data.

snarfetp (or equivalently, "snarflog -etp") is the version of snarflog that extracts log data from a running program that has previously been processed with etp(1) (the elapsed time profiler). For it to be possible to monitor a program continuously, snarfetp must be started together with the program. The program and snarfetp communicate via the named pipe that is the value of the environment variable ETPOUT.

snarfascii (or equivalently, "snarflog -ascii") is the version of snarflog that reads ascii log data from a program, parsing the log as a series of log records separated by newlines. It reads the log on its standard input. An example of where snarfascii can be useful is in processing logs generated by the trace(1) command.

When snarflog (or snarfetp or snarfascii) starts up, it registers itself with a telemonreg instance. It checks for the registry on the machine named by the environment variable TELEMONREG_MACHINE, or the local host if no such variable exists. It looks for a registry named by the environment variable TELEMONREG_NAME, or the registry named "TelemonRegistry" if no such variable exists. snarflog registers with program name *progname*, the machine name of the machine on which it is running, and its own process id. A telemonitor user supplies some or all of these names to the TelemonRegistry to locate the snarflog instance. *progname* should identify the program being monitored such that the telemonitor user can distinguish different programs and find any relevant data files required for processing log data from the program.

Options

-etp Run the version of snarflog that processes etp logs. This is equivalent to running snarfetp.

-ascii Run the version of snarflog that processes ascii logs. This is equivalent to running snarfascii.

-buffer ***nbytes*** Set the size of snarflog's internal buffer to *nbytes* (2 megabytes by default).

-debug Cause the snarflog instance to print diagnostic messages about whether data is being dropped or forwarded to a telemonitor.

Environment

TELEMONREG_MACHINE The name of the machine on which telemonreg is running. The default is the local host.

TELEMONREG_NAME The name under which telemonreg exports itself. The default is "TelemonRegistry".

ETPOUT The pathname of a named pipe to use to extract etp log data from a monitored program.

Bugs

snarflog should be able to tell the monitored program not to produce log data when no telemonitor is listening.

D Extended PSpec Example

This appendix provides an additional example of using the PSpec language and tools. It starts with a simple C program that can be compiled, linked, and instrumented with etp, and then shows some PSpec expressions that can be evaluated against the log resulting from running the instrumented program. (To run the commands described below, you must have etp and and peval installed.)

The following C program calls the `fwrite` function in a loop, writing a portion of a memory buffer to a temporary file repeatedly. On each iteration of the loop, one more character is written than on the previous iteration. The program reveals aspects of the performance of the `write` system call and `bcopy` library routine. (`fwrite` calls `bcopy` to copy the buffer contents into an I/O buffer, and it calls `write` periodically to flush the I/O buffer).

```
#include <stdio.h>

main ()
{
        int i;
        char buf[2000];
        FILE *fp = fopen("/tmp/foo", "w");

        for (i = 0; i != 1024; i++) {
                fwrite (buf, i+200, 1, fp);
        }
}
```

This program can be compiled and linked with a C compiler to produce an executable image, which can then be instrumented with etp. If the program is in a file named “x.c” then the commands to produce the executables “x” and “x.etp” are:

```
cc -o x x.c
etp x
```

“x.etp” is the instrumented version of “x”. Whenever “x.etp” is run, it will produce a file named “etp.out”, which is the log of timestamped procedure call and returns events for the execution. Run x.etp as:

```
x.etp
```

Now, given the “etp.out” file, the command

```
etplog x >x.etplog
```

produces a file “x.etplog” containing an ASCII version of the log.

We can explore the log “etp.out” using the PSpec language by running “peval” and evaluating expressions on the log. The command to start “peval” is:

```
peval -etp etp.out x
```

Here is an annotated sample “peval” session:

```
PSpec Evaluator. Type ”help” or ”?” for syntax.
-> proc fwrite;
-> proc bcopy;
-> proc write;
```

Declare events and intervals corresponding to the fwrite, bcopy, and write procedures, using the procedure sugar.

```
-> {count f : intv@fwrite};
1024
```

Aggregate expression to count the number of fwrite intervals in the event stream - there are 1024 of them.

```
-> {count b : intv@bcopy};
1112
```

Count the number of bcopy intervals in the event stream - there are 1112 of them. This must mean either that bcopy is called from other places than fwrite or that sometimes fwrite calls bcopy more than once.

```
-> interval fw = intv@fwrite
metrics bcnt = {count b : intv@bcopy},
btime = {+ b : intv@bcopy : elapsed(b)/1us}
end fw;
```

Declare an interval subtype of fwrite, with some additional metrics: bcnt counts the number of bcopy intervals in an fwrite interval and btime is the elapsed time (in microseconds) for all bcopy intervals in an fwrite interval. The new subtype is named fw.

```
-> {count f : fw where f.bcnt = 1};
936
```

There are 936 fw intervals that contain only 1 bcopy...

```
-> {count f : fw where f.bcnt = 2};
88
```

..and 88 that contain 2 bcopys. This accounts for all 1024 fwrites and 1112 bcopys.

```
-> {mean f : fw : elapsed(f)/1us};
191.4820313
```

The mean elapsed time for fwrite calls is about 190 microseconds.

```
-> {stdev f : fw : elapsed(f)/1us};
2563.017376
```

But the standard deviation is very large - over 2 milliseconds. Much of this is probably due to the write calls in some of the fwrite intervals.

```
-> interval fw2 = fw
metrics wcnt = {count w : intv@write},
wtime = {+ w : intv@write : elapsed(w)/1us}
end fw2;
```

Define a new subtype of fw with counts and elapsed times of the write sub-intervals to check this.

```
-> {mean f : fw2 where f.wcnt = 0 : elapsed(f)/1us};
32.85547009
-> {stdev f : fw2 where f.wcnt = 0 : elapsed(f)/1us};
31.56264679
-> {mean f : fw2 where f.wcnt != 0 : elapsed(f)/1us};
1878.691818
-> {stdev f : fw2 where f.wcnt != 0 : elapsed(f)/1us};
8607.104389
```

Indeed, the fwrite intervals with writes take much longer than those without, and they tend to have higher standard deviations.

```
-> {mean f : fw2 : elapsed(f)/1us - f.wtime};
35.21230469
-> {stdev f : fw2 : elapsed(f)/1us - f.wtime};
31.98921867
```

If we consider the elapsed times for all fwrite intervals subtracting out the time for any write system calls during the intervals, the mean and standard deviation are pretty close to those we computed when we excluded from the calculation those fwrite intervals that contained writes.

```
-> {count f : fw2 where f.wcnt = 0 & elapsed(f) > 60us};
```

22

Suppose we wanted to examine some of the long fwrite intervals that didn't contain writes. The following expressions show a useful way to build mappings using the aggregate operator called "the".

```
-> interval fw3 = fw2 metrics sts = timestamp(s) end fw3;
```

First we define yet another subtype of the fwrite interval that contains a metric that is the timestamp of the interval's start event. This timestamp is a unique identifier for all intervals of the same type.

```
-> {the f :  fw3 where f.wcnt = 0 & elapsed(f) > 60us :
f.sts->elapsed(f)/1us};
(3.641230271e9 -> [606.2,0.04,0.04],
3.641252357e9 -> [128.68,0.04,0.04],
3.641393206e9 -> [201,0.04,0.04],
3.643867565e9 -> [88.92,0.04,0.04],
3.644452903e9 -> [92.44,0.04,0.04],
3.644746097e9 -> [113.12,0.04,0.04],
3.644773098e9 -> [219.2,0.04,0.04],
3.644781774e9 -> [100.88,0.04,0.04],
3.645039358e9 -> [90.6,0.04,0.04],
3.645137258e9 -> [87.12,0.04,0.04],
3.645159924e9 -> [61.6,0.04,0.04],
3.645239792e9 -> [63.52,0.04,0.04],
3.645326065e9 -> [80.68,0.04,0.04],
3.645332368e9 -> [83,0.04,0.04],
3.645409843e9 -> [476.16,0.04,0.04],
3.645545851e9 -> [100.08,0.04,0.04],
3.645798962e9 -> [60.2,0.04,0.04],
3.64594163e9 -> [158.84,0.04,0.04],
3.645989011e9 -> [311.44,0.04,0.04],
3.646030861e9 -> [178.92,0.04,0.04],
3.64608297e9 -> [236.6,0.04,0.04],
3.646088959e9 -> [91.96,0.04,0.04])
```

Here we've used PSpec's mapping constructor, combined with the "the" operator to produce a multi-element mapping where each element maps an interval start timestamp (on the right of the `->`*) to the elapsed time for the interval in microseconds (on the left). The elapsed times are shown as triples. The result mapping only includes intervals that contained no writes and that took more than 60 microseconds. If instead of the elapsed time for each of those intervals we* *wanted to know how long their bcopys took, we could change the expression as follows:*

```
->{the f :  fw3 where f.wcnt = 0 & elapsed(f) > 60us :
f.sts -> f.btime};
(3.641230271e9 -> [23.2,0.04,0.04],
3.641252357e9 -> [123.68,0.04,0.04],
3.641393206e9 -> [10.6,0.04,0.04],
3.643867565e9 -> [83.72,0.04,0.04],
3.644452903e9 -> [30.88,0.04,0.04],
3.644746097e9 -> [107.28,0.04,0.04],
3.644773098e9 -> [214.28,0.04,0.04],
3.644781774e9 -> [34.56,0.04,0.04],
3.645039358e9 -> [85.92,0.04,0.04],
3.645137258e9 -> [82.12,0.04,0.04],
3.645159924e9 -> [56.92,0.04,0.04],
3.645239792e9 -> [57.48,0.04,0.04],
3.645326065e9 -> [76,0.04,0.04],
3.645332368e9 -> [78.24,0.04,0.04],
3.645409843e9 -> [46.68,0.04,0.04],
3.645545851e9 -> [95.4,0.04,0.04],
3.645798962e9 -> [55.52,0.04,0.04],
3.64594163e9 -> [147.52,0.04,0.04],
3.645989011e9 -> [306.36,0.04,0.04],
3.646030861e9 -> [174.04,0.04,0.04],
3.64608297e9 -> [231.92,0.04,0.04],
3.646088959e9 -> [26.12,0.04,0.04])
->
```

This example session gives a flavor of how the PSpec language can be used to explore etp logs quantitatively. Once the behavior is understood, the above kinds of expressions can be turned into assertions, such as:

assert { **_mean_** _f : fw2_ **_where_** _f.wcnt = 0 :_ **_elapsed(f)/1us_**} < _50_ **_us_**_;_

This asserts that the mean elapsed time for an `fwrite` call that does not include a `write` call is less than 50 microseconds.

References

[1] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230. ACM, December 1993.

[2] Digital unix web page. URL http://www.unix.digital.com.

[3] Hania Gajewska, James J. Kistler, Mark Manasse, and Dave Redell. Argo: A system for distributed collaboration. In *Proceedings of the ACM Multimedia '94 Conference*, October 1994.

[4] Allan Heydon and Greg Nelson. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-131a.html.

[5] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995.

[6] Barton P. Miller, Morgan Clark, Steven Kierstead, and Sekl-See Lim. IPS-2: The second generation of a parallel program measurement system. Computer Sciences Technical Report 783, University of Wisconsin—Madison, Madison, Wisconsin, August 1988.

[7] Greg Nelson, editor. *Systems Programming With Modula-3*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[8] Sharon E. Perl. Performance assertion checking. Technical Report MIT/LCS/TR-551, MIT Laboratory for Computer Science, Cambridge, MA 02139, September 1992.

[9] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 134–145. ACM, December 1993.

[10] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*, pages 104–113. IEEE Computer Society, October 1993.

[11] Marshall Rose. *The Simple Book: An Introduction to Management of TCP/IP-based Internets*. Prentice-Hall, 1991.

[12] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[13] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[14] David Solomon. *Inside Windows NT (2nd edition)*. Microsoft Press, 1998.

[15] Xpvm: A graphical console and monitor for pvm. http://www.netlib.org/utk/icl/xpvm/xpvm.html.