Experiences with Scheme in an Electro-Optics Laboratory

Richard A. Cleis
Air Force Research Laboratory

Keith B. Wilson
Air Force Research Laboratory

Abstract

The Starfire Optical Range is an Air Force Research Laboratory engaged in Atmospheric Research near Albuquerque, New Mexico. Since the late 1980's it has developed numerous telescope systems and auxiliary devices. Nearly all are controlled by C programs that became difficult to manage due to the large number of configurations required to support the experiments. To alleviate the problem, Scheme has been introduced in at least six distinct ways. This paper describes the uses of Scheme, emerging programming techniques, and general experiences of the past several years.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming — Scheme ; D.2.3 [*Programming Techniques*]: Coding Tools and Techniques — DrScheme, MzScheme ; J.2 [*Computer Applications*]: Physical Sciences and Engineering — aerospace, astronomy, engineering

General Terms Algorithms, Design

Keywords embedding, scripting, extension language, motion control, servo, communication, remote development

1. Introduction

Laboratory Mission The primary mission of the Starfire Optical Range (SOR) is to develop and demonstrate optical wavefront control technologies. In other words, it builds systems that remove the distortion caused by air turbulence when light propagates through the atmosphere. [1] The site also supports field experiments by others within the research community.

Experiments are conducted on five permanent precision telescope systems. Many experiments involve multiple telescopes and all telescopes are used for multiple experiments. Some telescopes have components that must move in concert with the main gimbals and most have at least one tracking system. The variety of systems is significant; the largest telescope has a 3.5 meter diameter primary mirror and weighs nearly 150 tons, while the smallest has an aperture of about a quarter meter and weighs about one half ton.

Tracking requirements include celestial objects, artificial satellites, aircraft, balloons, lunar retro-reflectors, interplanetary spacecraft, Space Shuttles returning from space, Leonids meteor trails, terrestrial vehicles, and diagnostic sites downrange of the SOR.

Proceedings of the 2006 Scheme and Functional Programming Workshop
University of Chicago Technical Report TR-2006-06

Laboratory Environment The following lists highlight, from the perspective of developing and operating telescope systems, the nature of the laboratory and strategic improvements that could increase the effectiveness of laboratory software.

- Five telescope systems are run with about a dozen computers.
- As many as three telescopes are needed for a single experiment.
- Two developers do most telescope related programming.
- About eight developers are needed to program all systems.
- Most telescopes are used for several long term experiments.
- At least several operators are needed for most experiments.
- Operators have only casual knowledge of the technology.
- Principle Investigators normally have only casual knowledge of the software.
- New experiments sometimes require new computations and configurations.
- Debugging is sometimes only possible during experiments.
- Subsytems need to communicate during experiments.
- Extensive legacy C software is essential to real-time operations.
- Planning requires significant computations.
- Software developers must consider safety.

At least several of us believe that software could be significantly more effective if the following issues were addressed:

- A non-expert programmer needs to be able to maintain the computational and control systems. Currently, only one developer can maintain or improve them.
- Software is needed to reliably repeat operations as much as several years apart. Currently, if specific operations are to be repeated, the original crew is normally required; this is often difficult for a variety of reasons.
- Principle Investigators need to be provided with clear scripting interfaces for their experiment, even if they only use them to understand and record the procedures.
- Non-expert programmers need to be able to program the creations of the expert computer engineers; most programs can only be maintained by the original engineer.
- Maintenance and performance verification needs to be automated so that they are accomplished more often and trends can be identified.
- Experiments need to be more automated. Most events are sequenced by operators pushing virtual buttons on user interfaces.

Why Scheme? We chose Scheme to address the above issues for a number of reasons. Its elegant syntax appropriately matches the

modest programming requirements, and macros are used to provide simpler syntaxes for scripting. S-expressions are suitable for ethernet communications between dozens of our platforms, including those hosting scripting environments. The same s-expressions may be created and read by C++ and Java programs; this allows Scheme to be used in the lab without requiring all programmers to adopt it. Garbage collection, the absence of assignment statements, and the ability to redefine functions while programs are running are capabilities essential to the rapid development that is occasionally needed while experiments are in progress.

As a Functional Programming Language, Scheme is suitable even in scripting environments that are mainly used to sequence the movement of equipment. Beneath the procedural scripting, functional programming elegantly access values on remote systems, apply filters, and delivers the results to other remote systems. Scheme is also appropriate for eventually extending our C language planning and scheduling software; functional programming in Scheme could more sensibly manipulate lists which are normally built from the results of functions related to celestial and orbital mechanics.

MzScheme, in particular, is embeddable in C and extendable by C. Both techniques are used to access hardware. MzScheme is also a reliable environment for subsystems that run continuously.

Aside from the useful tools that are provided, we chose PLT Scheme because it is supported by numerous teaching aids: web tutorials, textbooks, and an active online forum. The efforts expressed in this paper represent a shift in the implementation of telescope system computers at the SOR; the changes would reverse if learning materials were not available to the programmers who choose to see the value of Scheme.

2. Six Uses for Scheme

The SOR is an electro-optics laboratory that depends on numerous devices which were nearly all programmed in C over the past two decades; C is the natural choice because most programs require access to the hardware bus. Experiments require the specific configuration of up to ten telescope subsystems, a problem which becomes more difficult to manage as the laboratory grows, more devices are built, and more experiments are attempted. To alleviate these problems, extension languages have been implemented in many of the motion control devices such as the telescope gimbals and large optical components. These telescope gimbals consist of precision bearing structures, accurate position transducers, and embedded motors which form closed-loop servos that precisely move the telescopes.

We are using Scheme in six different ways: embedding in C, Scheme motion controllers, C-language gimbals servos interfaced with s-expression parsers, the remote command and status paradigm for all telescope systems, scripting environments for experiments, and the remote development environment for gimbals servos. Highlights are shown in this section.

Embedding in C We embedded MzScheme in the site's ubiquitous motion control application, the Starfire Optical Range Acquisition and Pointing System (SORAPS). This C application had become unwieldy after numerous capabilities were gradually added, since 1987, to accommodate all experiments conducted on the principle telescopes. We also embedded MzScheme in several programs required for the operation of subsystems. Configuring and controlling these programs is significantly more efficient using Scheme because behavior can be changed without rebuilding C.

Motion Control Systems We developed Scheme motion control systems for two optical rotators and the 3-axis secondary mirror controller on the largest telescope. Using DrScheme, we prototyped these relatively low bandwidth controllers then installed the final program on a single board computer that continuously runs MzScheme. We could have used one of the commercial servo development tools acquired for other projects, but they impose restrictions on hardware and software. MzScheme enabled us to build unique controllers with open communications; they can be ported to a variety of ordinary hardware capable of running MzScheme.

Interface to Servos for Large Gimbals The gimbals servos for the three largest telescopes are C language applications linked to the Small Fast S-expression library. [2] We extended the library to provide rudimentary single-depth evaluation of s-expressions that are either received via ethernet or read from a configuration file. This approach allows the servos to be configured and accessed as if they had Scheme embedded, maintaining consistency with the Scheme systems yet requiring only a very light weight library.

Paradigm for Remote Commands and Status S-expressions are the only messages used for remote commands and status of the telescope systems. Other devices (e.g. optical image stabilizers and telemetry servers) also communicate to the system with Scheme commands, even if formed with a printf statement in C. The typical "bit-speak" often found in hardware interface documents is replaced with elegant and self-documenting s-expressions.

Scripting Environment for Experiments We use DrScheme as a scripting environment to automate the telescope and subsystems during experiments. The scripts consists of macros that access support functions which maintain communications with remote systems and control their activity. DrScheme enables our most complex telescope experiments to be conducted using simple keywords that may be manipulated without interrupting operations.

Remote Development Environment We wrote tools in DrScheme for remotely running and testing the gimbals servos of the three largest telescopes. The "viewport graphics" displayed performance while we used the REPL and threads to move the gimbals and optimize properties contained in the servo. After developing the tools for a new controller for the largest telescope, we used them to complete similar controllers on two other telescopes. Only a few days were required for each of the second two because DrScheme had enabled us to easily write and reuse tools specific to large telescopes.

3. Programs and Tools

The first author wrote most of the following software, the second author has been writing the programs for the newest subsystems and has been configuring the diskless linux platforms that host most of the controllers.

Major Programs The major programs are SORAPS and NMCS. Each of five telescopes needs to be connected via ethernet to one of many installations of SORAPS, a full featured C program that handles operations, planning, and real-time computation. MzScheme was embedded to aid configuration and communication.

NMCS is a gimbals servo written in C and linked to the Small Fast S-Expression Library. The gimbals for the three largest telescopes require a separate NMCS; each runs continuously on a single board computer. SORAPS and NMCS are linked via ethernet and communicate exclusively with s-expressions. The remaining two gimbals are controlled via serial ports connected to SORAPS.

Minor Programs Embedded MzScheme programs are used to control a secondary telescope mirror and bridge new s-expressions to legacy subsystems that require proprietary communication formats. These programs communicate to both SORAPS and the scripting environment for purposes such as dome control, optical configuration and alignment, and telescope focus.

Two optical rotators are controlled by programs we wrote for MzScheme, using a few extensions to access the hardware. These

rotators communicate with SORAPS to report their position and receive commands.

Applications we created in DrScheme include simulators for gimbals, telemetry, an image-stabilizer, a rotator, and a communication bridge.

Developmental Software We developed a suite of functions and a module to script the telescope systems for complex experiments. We also developed servo development tools that were used to optimize NMCS for each of the three telescope gimbals.

Libraries, Extensions, and Development Tools Programs that embed MzScheme are linked to libmzgc.a and libmzscheme.a. SORAPS and all of its supporting Scheme files are contained in a volume which can be mounted by any OS X platform. Because the libraries are linked to the application, platforms can run SORAPS even if no Scheme environments are installed.

NMCS programs are linked to libsexp.a, built from the source files of the Small Fast S-Expressions project. Two telescopes and one rotator are connected via serial ports implemented in a Scheme extension that we wrote. For all of the development described in this paper, we used DrScheme, KDevelop, and Xcode.

4. Using Scheme to Script a Telescope System

We developed Scheme functions and scripts for a long term series of experiments that began in early 2006. The scripts are designed to be readable by non-programmers and modifiable by personnel who understand the operations. They serve as an executable operational summary as well as an essential tool for operations. A principle script is used to guarantee operations; it assumes no state and performs all known initializations. Other scripts are used to suspend and resume the principle script without performing initializations that would be disruptive.

The scripts are supported by a suite of functions that communicate with the subsystems and perform experiment-specific computations. A module of macros defines key-phrases that can be sequenced by a macro that defines each script.

4.1 The Principle Script

The system needs to point to a moving balloon-borne platform containing scientific instruments. Telemetry messages are requested, filtered, and routed to SORAPS which computes dynamic vectors for the telescope, the dome, the elevation window contained by the dome, and the optical rotator. All subsystems are set into motion, then the script waits until they mechanically converge on the platform. The script also positions several mirrors in the optical path and sets the position of the mirror used for focus. A macro, `define-sor-script`, creates a script that is named with the first parameter:

```
(define-sor-script find-platform
 prepare-subsystem-connections
 verify-subsystem-connections
 prepare-soraps-for-wgs84-propagation
 feed-telemetry-to-soraps
 tell-subsystems-to-follow-soraps
 wait-for-subsystems-to-converge)
```

`Define-sor-script` wraps the script elements in an exception handler that will call a function that stops the system if an error occurs. It then executes the sequence in a thread that can be killed by other scripts, such as `stop`.

```
(define-syntax define-sor-script
  (syntax-rules ()
    ((_ proc-name f1 ...)
     (define-syntax proc-name
       (syntax-id-rules
        ()
(_ (begin
     (keep-thread
      (thread
       (lambda ()
         (with-handlers
             ((exn?
               (lambda (exn)
                 (stop-system
                  (exn-message exn)))))
           (f1) ...
           (display-final-message 'proc-name)))))
     (display-initial-message 'proc-name))))))))
```

The module containing `define-sor-script` manages the thread (saved with `keep-thread`) and defines functions that display initial and final messages in the REPL. The script is defined with `syntax-id-rules` to enable operators to type the name, without parentheses, to execute the script. This elementary macro threads the sequence, stops the system when an exception occurs, and provides diagnostics messages; the script writer is merely required to use `define-sor-script`.

`Prepare-subsystem-connections` is a macro that sets the addresses and ports of connections to all subsystems and starts a thread that evaluates incoming messages. It also sends a message to a a common displayer used to show the occurrence of significant events:

```
(define-syntax prepare-subsystem-connections
    (syntax-id-rules
     ()
     (_ (begin (set-senders-to-subsytems!)
               (start:evaluate-network-inputs)
               (send-to-displayer
                "Connections were created.")))))
```

This macro is typical of those in the rest of the script. It contains functions that provide utility to a script writer, and it has alternatives which make it necessary; macros for simulators could have been used instead. Some macros could have been implemented as functions, but we wanted all elements of the scripting environment to be executable without parentheses. This allows the script writer to test individual macros in the REPL.

More complex scripts include search algorithms and the use of automatic tracking systems, but we have not yet used them.

We developed this elementary environment and started using it for all balloon experiments. However, it is merely a first attempt at scripting, a software experiment inside of a physics experiment. We plan to thoroughly investigate other possibilities, such as creating specific languages for the experiments and subsystems.

4.2 Strategy

Writing scripts and their support functions “from the top, down” biased scripts toward what users want, rather than what programmers want to write. To minimize the use of resources, we developed as much software as possible before requiring the operation of any remote subsystems. Skeleton functions in each layer were debugged with the entire program before the skeletons were “filled in”, creating the need for another layer. The lowest layer provided the network connections; they simulated messages from the remote systems which were not yet available.

This minimal executable script served as executable documentation suitable for preparing for the Critical Design Review. Each network simulation also provided an operating specification for the

development of incomplete subsystems. Eventually, we debugged the connections, one at a time, as we replaced each simulator with the corresponding subsystem.

We used an outliner program to specify the scripts. Simple conditionals and looping can be represented in an outline; anything more complicated was not considered for the scripting layer. After many iterations, the top layer of the outlines evolved into simple lists of procedures that were eventually implemented as macros. The second layer was implemented as the top layer of Scheme functions; so no conditionals or looping was required in the script.

The outline formed what is suggestive of "wish lists" in the text *How to Design Programs*. [4] Functions were written to simulate requirements of the wish lists so that a working simulation could be completed before expensive engineering commenced. In the future, we intend to employ such textbook ideas in documentation for writing scripts and support functions.

We developed the software with the intent that it could be maintained and extended in three levels. The highest level, scripting, is usable by anyone who has familiarity with programming. The lowest level functions are to be maintained by the experts who created the optical-mechanical subsystems. The middle level is for programmers who can understand the needs of the script writers and the operation of the systems. We expect that this strategy will help us effectively manage the software for telescope systems and experiments.

5. Details of Scheme Use

5.1 Embedding in C

We embedded MzScheme in C programs for three reasons. First, it serves as the extension language for the legacy C program, SORAPS. Second, it runs Scheme programs that access hardware in C. Third, it provides a bridge between Scheme programs and proprietary network libraries. Embedding Scheme, here, refers to linking MzScheme with C and creating a single Scheme environment when the C program is started. This environment contains definitions of scheme primitives that access functions previously written in C. Functions in the environment are invoked by the evaluation of Scheme files, the evaluation of expressions received via ethernet, or by calls from the C side of the program.

5.1.1 Extension Language for SORAPS

Embedding MzScheme in SORAPS allows other programs to configure it, enables networked operation, and provides a means for other programs to access essential calculations.

Configuring SORAPS Configuring SORAPS is a difficult problem. Individual SORAPS installations control the five largest telescopes, and multiple installations may control each telescope. Also, temporary subsystems (including telescopes) occasionally need access to the calculations. These configurations cannot be handled elegantly with simple configuration files partly because the telescopes have different command and status requirements. Furthermore, many configuration items will likely be moved from legacy files to an on-site database.

The differences between the telescopes are not trivial abstraction issues. The NMCS control systems are a service provided to SORAPS via ethernet, as SORAPS initiates all transactions with little time restriction. A commercial system behaves as a client to SORAPS, pushing status over a serial port in a precisely synchronous manner. Another commercial system is a combination of the two. Two recently replaced systems had no communication protocol at all; they required a hardware connection to their bus. Furthermore, telescope systems often have components that affect the optical perspective of other components in the system. The Scheme environment of SORAPS contains functions to handle these problems so that computations can be modified for affected devices, yet SORAPS doesn't need to be rebuilt or restarted.

Configuration, command, and status functions are contained in Scheme files that may be manipulated in an external environment like DrScheme. Primitives provide a means for setting properties of the system with relatively unrestricted programs instead of fixed-format files that were required by the original C program.

Serial ports are sometimes needed to interface dynamic equipment such as gimbals controllers. Scheme programs, embedded or independent, load a serial extension and interface functions specific to the application. This is an advantage over earlier solutions that depended on configuration-specific libraries; the Scheme files are available and editable on the operations computer, only a text editor is required to change the behavior of the program significantly.

The following fragment represents the kind of function used for configuring fundamental properties of a telescope:

```
(let ((id '("3.5m Telescope" "cetus")
          ; long name, short name (no spaces)
          )
      (loc (list wgs84-semimajor-axis
                 wgs84-inverse-flattening-factor
                 ;any ellipsoid may be used
                   34.9876543 ; deg latitude
                 -106.456789  ; deg longitude
                 1812.34      ; met height
                 6 ; hours local to utc in summmer
                 )))
  (display (set-system-gimbals cetus id loc ))
  (newline))
```

`Set-system-gimbals` is a primitive in C, it returns readable messages that indicate success or failure. "Wgs84..." are numbers that are defined in Scheme, but any reference ellipsoid may be used; on rare occasions, researchers want to use their own.

An example for configuring an evaluator of network messages:

```
(define eval-socket (udp-open-socket))
(define eval-the-socket
        (make-function-to-eval eval-socket 4096))
```

`Make-function-to-eval` was written to support Scheme communications (it is explained later.) Its product, `eval-the-socket`, is either used in a thread or it can be called by scheduling functions invoked by C. Both the behavior of the communications and their implementation are entirely handled in Scheme, a more effective environment than C for maintaining the communications.

Operating SORAPS We installed primitives in SORAPS to allow the selection of computations and the control of its associated telescopes. For telescope movement, a mode primitive is installed to evaluate expressions like:

```
(do-mode telescope-index 'stop)
```

Changing between `stop` and `vect` makes a telescope stop or follow vectors that are supplied by other expressions. Following is the C function needed for the primitive.

```
static Scheme_Object
*do_mode( int nArgs, Scheme_Object* args[] )
{
  enum { SYSTEM = 0, MODE };
  char text[32] = "''"; //For quoted symbol

  int iSystem  = 0;
  char *pChars = 0L; // 0L will retrieve mode
  const char *pMode  = "error";
```

```
  if( int_from_num( &iSystem, args[ SYSTEM ] )) {
    if( nArgs == 2 ) {
      if( !char_ptr_from_string_or_symbol(
          &pChars, args[ MODE ] ) ) {
          pChars = 0L; // paranoia
    } }
    pMode = DoSorapsMode( iSystem, pChars );
    // returns a string from legacy C
  }

  strcpy( &text[2], pMode );
  return scheme_eval_string( text, sEnv );
  // e.g. "''stop" evals to a quoted symbol
}
```

The first object in `args` is an index to a telescope and the second is the desired control mode: `stop`, `vect`, etc. If the second argument is not provided, no attempt is made to change the mode. The primitive returns the final mode, in either case, as a quoted symbol suitable in expressions evaluated by the client. Internally, `DoSorapsMode` returns a string that represents the mode contained in the original C code. Typical primitives are more complex.

This simple primitive allows the control mode to be changed by any embedded Scheme program or any remote program that sends a `do-mode` expression. (A few more lines of code are needed to add `do-mode` to Scheme and bind it to the primitive.)

Serving Calculations A telescope system may include several subsystems that need information calculated by SORAPS. Rotators sometimes maintain orientation in a telescope's naturally rotating optical path, so they typically send an expression that uses the primitive `path-deg`; it contains arguments that specify the telescope and the location in the optical path where the rotation is needed. Other primitives perform time conversions or supply values such as range to satellites. These primitives are a work in progress. As requirements are added, primitives are written so that the capabilities are available to any s-expression that any client sends; this is more effective than writing specific messages for specific clients.

5.1.2 Bridge to Proprietary Network Libraries

Some systems are accessible only through nonstandard network libraries, so we embedded MzScheme in C programs that access those libraries. Primitives were then written to complete the bridge. This gives the other Scheme applications on the network a consistent way to access the bridged systems. These systems include a telescope focus controller, a dome controller, electro-pneumatic mirror actuators, and temperature sensors.

5.1.3 Hardware Access

We embedded MzScheme in several C programs that need to read and write electronic hardware in servos. Hardware includes analog output voltages, analog input voltages, parallel input ports, and bidirectional serial ports. Simple C programs were first written and debugged, then Scheme was embedded and furnished with primitive access to the hardware functions. The main software was then written in Scheme.

5.2 Motion Control Systems

We wrote servo software for a three-axis secondary mirror in Scheme. The program runs in MzScheme embedded in a small C program that accesses the hardware. A few primitives provide access to the input voltages, which indicate position, and the output voltages that drive the axes. Scheme reads the position voltages of the axes, calculates command voltages based on the position and the desired state, then sets the output voltages.

The program may access SORAPS to determine the range to the object and the elevation angle of the telescope. These values are used to adjust the focus and to maintain alignment between the primary and secondary mirrors. The servos also receive commands from user interfaces that are connected to the network.

The development process was remarkably efficient. On a suitable workstation, the servo program was developed remotely from DrScheme by using sockets to access the hardware primitives (i.e., the input and output voltages) on the servo computer (which is inconvenient for development.) When completed, the program was transferred to the servo computer and run in the embedded Scheme.

We also developed two optical-mechanical rotators, both prototyped in DrScheme. One interfaces custom hardware, while the other uses a serial port to interface a commercial motor driver.

5.3 S-expression Interface to Gimbals Servos

The gimbals for each of the three largest telescopes are controlled by a Networked Motion Control System, a C-program we developed for single board linux computers. An S-expression interface was developed for configuration, command, and networked control. NMCS periodically reads the encoders, computes the desired state vectors, computes the desired torques, drives the two axes, then processes s-expressions if any arrived over ethernet. These systems are markedly different than typical servos which are implemented with real time operating systems, digital signal processors, and rigid command and status interfaces. In NMCS, we use excess processor speed to run elegant interface software that can be accessed with anything that can read and write text over ethernet.

5.3.1 How S-Expressions are Used

The s-expression interface is used to configure the program, accept commands from SORAPS, and return servo status to SORAPS. Typical configuration values are position limits, rate limits, friction coefficients, and torque constants. Commands are required to periodically set the time, report the position of the sun, and provide state-vectors for the gimbals. A few examples demonstrate the flexibility of using s-expressions to specify one or more axes of the gimbals and to specify trajectories of variable order.

Commands Trajectories are specified as lists, and lists may contain lists for each axis:

```
(do-axes 0 '(t0 position velocity acceleration))
```

where 0 indicates axis-0, and the quantities represent numbers that describe a trajectory with a reference time of `t0`. Multiple axes are specified with a list of lists:

```
(do-axes '((t0 p0 v0 a0)(t1 p1 v1 a1)))
```

where all elements (`t0` etc.) represent numbers. For higher clarity, it is not necessary to list zeros in trajectories of lower order; the program integrates the available terms to calculate the position propagated from `t0`. For example, the following two expressions evaluate to the same fixed position for axis-1; the rate and acceleration are assumed to be zero in the second case:

```
(do-axes 1 '(12345678.01 45.0 0 0))
(do-axes 1 '(12345678.01 45.0))
```

The reference time, `t0`, is not needed for calculating fixed positions, but it is used to validate the command by testing if the trajectory time is within one second of the servo's time.

Status Lists of variable length are used to return status values that are ring-buffered each time the servo is serviced (typically every 20 milliseconds.) It is assumed that only one client is operating any telescope, so the primitives only return data that was buffered since

the previous request. This method allows the client program to casually query the servo for groups of information rather than forcing it to accept data as it is produced. An upper limit is established for the size of the replies in case the status was not recently requested.

For example, `(get-diffs)` returns a list of two lists. Each list contains the differences between the calculated and sensed positions for an axis. SORAPS queries about every quarter second and receives lists of about 12 values for each axis of each diagnostic value that was requested. These are used to compute statistics, display diagnostics on user interfaces, and to optimize the servo.

5.3.2 How NMCS Processes S-Expressions

The Small Fast S-Expression Library (SFSExp) is used to parse the s-expressions that NMCS receives over ethernet or reads from its configuration file. SFSExp was developed for a high speed cluster monitor at Los Alamos National Laboratories, so it easily handles the relatively low speed requirements for the motion control systems at the SOR.

We originally embedded MzScheme in NMCS, but garbage collections consumed around 10ms every 10 seconds or so; that was too marginal for the 20ms time slices needed by the servo. Rather than pursue a solution involving high priority interrupts or a real time operating system, the SFSExp library was employed to parse incoming expressions that essentially select C functions and call them with the remaining parameters.

Form for Data Modification and/or Access The only form implemented or needed in NMCS is

```
(list ['callback] (function1 parameter1...) ...)
```

where the optional callback function is intended to be defined on the client and the parameters of the functions cannot contain functions. All functions return s-expressions.

Functions return a value or a list; if parameters are supplied then the function attempts to set the values before returning the result. The gimbals have multiple axes, so lists-of-lists are converted into C arrays for each axis.

Destination of Replies NMCS supports a single socket that evaluates the incoming expressions and returns the result. The above message returns an expression that the client may evaluate:

```
(callback result1 ...)
```

The clients nearly always have a single receive thread that evaluates these responses. In other words, NMCS allows the client to call its own function with the results of NMCS functions. This behavior is compatible with the communications paradigm, described elsewhere in this document, that is more thoroughly implemented in MzScheme.

Tools for the Form We developed an API in C to provide a consistent way to set upper and lower limits on values and values in arrays. It also returns errors for illegal values or bad indices, for example. These features proved to be invaluable during the development of the servos because nearly all of the expressions involved passing numbers. Even these tiny subsets of language behavior are more useful than methods typically found in engineering: passing cryptic bits with rigid procedure calls, many without descriptive error messages.

5.4 Paradigm for Remote Command and Status

All communications between telescopes and their subsystems are conducted with s-expressions that can be evaluated with MzScheme or the s-expression interface of NMCS. The outgoing expressions produce incoming expressions that, when evaluated, cause a local function to be called with parameters that consist of the results of functions that were called on the remote system. This style allows concurrent access of multiple remote systems without requiring input decision trees or the management of multiple connections. When a programmer is working from a REPL, the callback mechanism is occasionally not used; in those cases the requesting function sends for a list of results by using a local function that blocks until it receives the result.

Communication takes place over UDP sockets whenever possible. Blocked TCP connections, whether due to network problems or software, are difficult for operators to solve because they often have no programming experience and little technical experience. A single unresolved timeout often leads to wholesale rebooting of computer systems if the connection can not be reestablished. Beginning with the implementation of PLT Scheme, nearly all connections are UDP, and all programs are written to be tolerant of incorrect or missing messages. We originally intended to write error detection functions for the adopted paradigm, but all of our activity takes place on a quality private network that is either working perfectly or not at all. This author observes that the connection problems caused by TCP far outweigh the packet error problems that they solve.

Typical programs use two sockets that may communicate with all of the remote systems. One socket sends all requests and evaluates any replies. The other socket evaluates any incoming requests then sends the replies. A catch-all exception handler was implemented after debugging was completed.

The functions shown below were used with version 209; slight changes are required for later releases of PLT Scheme, mainly due to unicode implementation.

Form for Requests Requests merely ask for a list of results of functions called on the remote application. The message form is described in Form for Data Modification and/or Access (for NMCS), but is much more generally useful in Scheme environments because no restrictions are placed on the expressions.

Send-Receive-Evaluate Requests can be sent with `udp-send` or `udp-send-to`, then incoming replies on the same socket are discovered and evaluated with a function that is either polled or looped in a blocking thread. The polled version is used in SORAPS because it is designed to wake up, check the socket, perform tasks, then sleep. On the other hand, scripts might use a blocking version if an operator is using commands in a REPL. The following function is intended to be polled, while a blocking version can be made by eliminating '*' from `udp-receive!*`.

```
(define make-function-to-eval ; accept udp socket
  (lambda( socket buffer-size )
    (define buffer (make-string buffer-size))
    (lambda()
      ; messages must fit in a single packet
      ; perhaps udp? should verify socket type
      (if (udp-bound? socket)
          (let ((n-rxed (call-with-values
                         (lambda()
                           (udp-receive!* socket
                                          buffer))
                         (lambda(n ip port) n))))
            (if n-rxed
                (with-handlers((exn? exn-message))
                  (eval
                   (read
                    (open-input-string
                     (substring
                      buffer 0 n-rxed)))))
                #f)) ; false instead of void
          #f)))) ; ditto
```

Receive-Evaluate-Reply The following form evaluates any request, then sends the reply. A simpler function uses MzScheme's `format` instead of the output string, but this function was developed first and has been used for several years.

```
(define make-function-to-eval-then-reply
  (lambda (socket buffer-size)
    (define buffer (make-string buffer-size))
    (lambda()
      (if (udp-bound? socket)
          (let-values
              (((n-rxed ip port)
                (udp-receive!* socket
                               buffer)))
            (if n-rxed
                (udp-send-to
                 socket ip port
                 (let ((o (open-output-string)))
                   (write
                    (with-handlers
                        ((exn? exn-message))
                      (eval
                       (read
                        (open-input-string
                         (substring
                          buffer 0 n-rxed))))) o)
                   (get-output-string o)))
                #f))
          #f))))
```

Multiple messages may be sent over the same socket because the replies may arrive in any order.

5.5 Scripting Environment for Experiments

DrScheme served as a scripting environment during development and operations of an experiment that required numerous motion control systems. This is described in Using Scheme to Script a Telescope System.

During operations required for the experiment, we were able to modify programs that were in use. For example, a telemetry thread in the DrScheme environment requests data via ethernet socket, processes the positions in the reply, then sends the results to SORAPS. When marginal reception was encountered, we developed and debugged a filter in a separate workspace. Genuine packets were taken from the running workspace to test the filter. When the filter was complete, we installed it and restarted the thread in a matter of seconds. Telescope operations were not interrupted.

We also over-wrote a measurement function while it was in use. Measurements from video tracking subsystems are scaled and rotated in Scheme before they are applied to SORAPS. The operation is complicated by the dynamic rotator that affects the apparent orientation of the tracker. A new tracker and rotator had unknown orientation, so we debugged the measurement function simply by editing and loading a file that overwrote the previous version. In the past, we rebuilt and restarted C code in the tracker for every iteration. Using Scheme, we were able to accomplish in one hour what previously required many.

5.6 Remote Development Environment

We used DrScheme for remote control and diagnostics while developing NMCS. Normal debugging techniques could not be used because such programs can not be arbitrarily suspended; the gimbals would "run away" if a breakpoint were encountered while torque was being applied. The environment consisted of threads which sent simulations of essential data that is normally sent from SORAPS. In the meantime, servo properties were changed by sending expressions from the REPL. The "viewport graphics" in MzScheme were used to display servo parameters while test functions moved the telescope along trajectories designed to be sensitive to parameters being adjusted.

We could have used one of several commercial development environments that we maintain, but they restrict both the hardware selection and the software techniques. On the other hand, NMCS is designed to be portable to anything that can support sockets and run programs built from C. The commercial environments are intended to run very fast, so they sacrifice software flexibility. Large telescopes cannot benefit from such speeds, so we do not believe performance could be gained by accepting the aforementioned restrictions. Furthermore, writing specific tools in DrScheme is arguably as fast as learning and using the commercial tools.

Threads and REPL Tests To prevent unsupervised telescope motion and prevent expensive damage due to slewing through the sun, three essential messages are periodically delivered to NMCS: the time, the desired trajectories of the axes, and the position of the sun. NMCS stops the gimbals if any of the messages are missed for more than a specified period; it uses the information to predict where the gimbals are going as well as the current location. Human action (on the SORAPS GUI) is then required to restart the gimbals to avoid an accident after a network dropout is resolved, for example. DrScheme was used to test these behaviors as KDevelop was used to debug NMCS.

To simulate SORAPS, three threads were started. One sent the position of the sun every 20 seconds, another sent the time every 10 seconds, and one sent gimbals vectors every second. From the REPL, the threads were individually suspended (while the gimbals were running) to ensure that NMCS stopped the gimbals. They were then restarted to ensure that NMCS did not start the gimbals without receiving other required commands that were also tested from the REPL.

Adjustment of Servo Properties We adjusted properties like torque constants, friction coefficients, and coefficients for parameter estimation while the telescope was following trajectories commanded by a thread that sent gimbals-vectors. Performance was optimized by viewing diagnostics displayed in the viewport window. These diagnostics included position, velocity, servo error, command difference, integrated torque, and timing errors.

Instead of creating large sets of functions and possibly GUI's to access them, we interactively wrote functions to change subsets of arguments during the servo optimization activities. For example, three acceleration constants are required by NMCS, but sometimes only one of them is adjusted:

```
(define (aa1 torque-per-bit) ;; aa1: adjust axis-1
 (send-to-nmcs
  (format
   "(list 'show (do-acceleration 1 '(1 2046 ~s)))"
   torque-per-bit)))
```

A receive thread evaluates the reply, causing the local function `show` to be called with the results of `do-acceleration` when it was called on NMCS. We intend to automate many of these optimization procedures, so this REPL approach forms a more appropriate foundation than a GUI.

The above expressions, especially ones that have variables, are sometimes assembled from lists rather than using `format`. It is arguably more sensible to do so, but some sort of formatting must eventually occur before the message is sent. We tend to form the strings as shown because they are blatantly readable.

Test Functions Test functions included sine-waves, constant rate dithering, and "racetracks". Trajectories were closed so that they

could be run indefinitely. This functional method contrasts typical servo development where files are "followed" after the gimbals are positioned before each diagnostic run. This technique was motivated by the use of functional programming: Commands are created from nested calls of functions whose only root variable is represented by a function, MzScheme's `current-milliseconds`.

6. Programming Techniques

This section describes many of the reusable techniques that were developed while working with MzScheme version 209.

6.1 Senders and Displayer

We wrote a simple displayer to show, in sequence, incoming and outgoing messages that access remote subsystems. `Make-sender` labels the diagnostics (to indicate their origin) and sends them to a program running in a separate DrScheme workspace (i.e., another window with a REPL.) Using a separate workspace prevents cluttering the operations REPL. A socket is implemented so that the displayer may also be hosted on a separate computer.

The displayer is normally a file that evaluates the following:

```
(letrec
    ((uos (udp-open-socket))
     (buffer (make-string 256))
     (lupe (lambda()
             (let-values
                 (((n ipa port)
                   (udp-receive! uos buffer)))
               (display (substring buffer 0 n))
               (newline))
             (lupe))))
  (udp-bind! uos #f 9999)
  (lupe))
```

The sender to each remote system is created with the function:

```
(define (make-sender soc ipa port prefix)
  (if (equal? prefix "")
      (lambda (text)
       (udp-send-to soc ipa port text))
      (lambda (text)
       (udp-send-to soc ipa port text)
       (udp-send-to
        soc "127.0.0.1" 9999
        (format "~a ~a" prefix text)))))
```

When the sender is created, non-empty text in `prefix` will cause all expressions passing through that sender to be displayed with the contents of `prefix`. The scripting environment also sends received expressions to the displayer, so that a clear ordering of messages is indicated in the window.

6.2 Simulating Transactions

We wrote a general transaction simulator before implementing ethernet communications. This simulator was used often:

```
(define (sim-transaction sleep-sec text)
  (thread
   (lambda()
     (sleep sleep-sec) ; simulate round-trip
     (eval
      (eval (read (open-input-string text)))))))
```

It first sleeps to simulate the expected round-trip delay, then evaluates the outgoing expression, and finally evaluates the result which is returned from a local simulation of the remote function. For example, the following will simulate stopping the gimbals:

```
(sim-transaction 1.5 "(list 'do-gimbals-mode
                            (do-mode corvus
                                     'stop))")
```

The simulation requires a local definition (a simulation) of the function `do-mode` and the definition of the telescope designator `corvus`. The callback function `do-gimbals-mode` is at the core of the local software that is being tested. Simulating the remote definitions also guided the creation of a clear, executable specification for the interface. For long delays, such as waiting several minutes for a telescope to move to a new location, the reference returned from `sim-transaction` was available for manipulating the thread.

6.3 Exception Handlers

We added exception handlers to the evaluator functions when we started using the new software. During development, it was better to let Scheme handle the exception and generate an error message. The exception handler is mainly used to prevent programs from halting due to misspelled or improperly formed expressions that are generated by new clients.

6.4 Connection Protocol

The communication paradigm relies on application level error control to compensate for the lack of detection and recovery provided by TCP-like protocols. To prevent the applications from halting, the message evaluators are wrapped in exception handlers which return any error message to the client. The motion control systems check messages by content; e.g., a rate of a thousand degrees per second is not accepted even if delivered without error. Most systems are designed to tolerate missed messages; e.g., a second order trajectory vector can be missed without noticeable errors in tracking. The clients nearly always use the response from a server to verify correct delivery; e.g., if a client sends a message to start a computational process in SORAPS, the response is used for verification.

We started to develop general techniques for error detection, such as echoing requests along with the responses, but we stopped for the lack of ways to cause or at least experience network errors.

6.5 Casual Polling of Remote Systems

Because TCP connections are avoided for practical reasons, we use an efficient technique for for getting uninterrupted repetitive data like telemetry. About every 8 seconds, the remote telemetry system is sent a message that requests data for ten seconds. This keeps the data flowing, it doesn't require the client to terminate the messages, yet it does not require a transaction for every message. A data message is not repeated after an overlapping request. Generally, the requests have the form:

```
(get-data 'callback-function seconds-to-send)
```

The server is required to send expressions of the form:

```
(callback-function the-data)
```

A related form is:

```
(get-data-if-available 'callback-function
                       within-the-next-n-seconds)
```

The server is required to send new data only if it is available within the next `n` seconds. This can be used when a telescope system is scanning for an object and it needs a camera to report when it firsts detects the object. The time limit prevents the camera from unexpectedly sending information at a much later time.

6.6 Message Timing

The arrival time of some expressions are stored in global variables. Typical functions that use these variables determine lateness and

provide re-triggering of events. For example, a task may keep the latest arrival time of a message, then compare it later to determine if a new one has arrived. Boolean techniques are not much simpler, and they do not contain enough information to determine elapsed time.

6.7 Making Specifications

A Scheme program needed access to a remote system written in C++, so we agreed that it must communicate via s-expressions. We wrote a Scheme simulation of the remote system and tested it with the Scheme client, then gave the simulation to the C++ programmer; no other specification was required because it was a working program written in an elegant language. We integrated and tested the complete system in about an hour, yet most of the hour was needed for the C++ programmer to debug socket software that he would not have needed to write had he used Scheme.

7. General Experience

How Scheme was Adopted I (the first author) was introduced to Scheme running on an Astronomer's Palm Pilot. Incapable of seeing usefulness, I dismissed Scheme until an email from the same person contained Scheme in one Day (SIOD), a pioneering Scheme environment. I obliged his suggestion to embed it in SORAPS at the same time that we were preparing to run the largest telescope remotely as it interacted with another telescope. Concurrently, another colleague showed me a paper on the Small Fast S-Expression Library because we were in search of a consistent way for our subsystems to communicate. By then, the potential of Scheme was obvious. I embedded MzScheme in SORAPS along with SIOD and gradually converted the few functions accessed by SIOD to MzScheme primitives. SIOD was finally removed, and I began writing a full suite of MzScheme primitives.

Reliability MzScheme v209 has been running continuously on 4 subsystems which are mostly single board computers running a diskless linux. One of them has been running for over a year, the rest are newer. Power failures and lightning strikes make it difficult to determine a "mean time before failure" that can be assigned to the subsystems; they normally are interrupted by such external events before they have a chance to fail.

The Small Fast S-expression Library has proven to be perfectly reliable as three have been running continuously for several years, only to be interrupted by power failures and lightning strikes. Ironically, the only maintenance needed in the first few years was a single-line fix of a memory leak caused by the first author... a leak that couldn't have occurred in a Scheme environment.

Language Issues Scheme statements have proven to be an effective basis for operating the systems. We have written thirty two primitives for SORAPS and a few dozen for the gimbals servos. Other subsystems have about a dozen. Error messages are a significant benefit of using Scheme. When a programmer incorrectly requests a message (especially from a REPL), an error message is returned which often reveals the problem. Typically, a parameter is forgotten, a Scheme error message is returned to the REPL, and the user solves the problem by reforming the message. When the serving program is implemented with MzScheme, the error messages are produced by the environment; the programmer is not required to develop them.

Elementary scripts have been written and used extensively for one experiment, they are planned for two more. Four scripts are needed for the current experiment, about four more will be required when all of the subsystems are complete. The scripting strategy has been unquestionably successful, but we will not improve it until we thoroughly study the possibilities... which include abandoning the approach and requiring users to learn basic Scheme.

We used closures as an effective way to realize object like behavior. The language requirements for any of the efforts in this paper are not overly complex, so adopting a much more complicated object oriented programming environment is probably not a good trade for the elegance of Scheme. The object system provided by PLT Scheme was not used in any of this work, mainly because of a lack of time to learn any of it.

Simulations We wrote simulations of nearly all subsystems; DrScheme was used to create executables for them. The simulations are used for developing scripts and to verify system operation before an experiment session begins. The balloon experiment has never been delayed by telescope systems because operations are tested with appropriate simulators before the experiment begins; electrical and mechanical problems are sometimes discovered and fixed. For these reasons, the combination of scripting and simulations are planned for all future experiments.

Foreign Function Interface vs Embedding Foreign Function Interfaces were not used in any of this work, mostly because the largest efforts were concentrated on SORAPS. Its C functions are so tightly tied to an event loop and graphical user interface that they are not appropriate for a Scheme interface. SORAPS functions are are being rewritten as primitives are added, so FFI's will be eventually be a viable option.

8. Conclusions

Scheme has significantly improved the efficiency of the two programmers who maintain telescope systems and prepare them for experiments. By using Scheme for configuration, communication, control, and at least elementary scripting, we are able to maintain the software for all systems and experiments in almost as little time as we previously needed for each. We modify SORAPS much less and the majority of configuration files have been eliminated because most configuration and all communication are contained in a few files of Scheme programs. We were able to create a single volume with directories containing software for all experiments and telescope systems; small Scheme programs load Scheme residing in the directories needed for given experiments.

Scripting the balloon experiment was valuable for two reasons. We successfully used the system at least once per month, yet another reason is perhaps more significant: Making scripts leads to better programs because the development process demands an accurate description of the problem. We were forced to answer two questions: What will be the sequence of events? What functions are needed to support them? The final few scripts for our experiment were so simple that they hardly seem worthy of discussion, but many iterations were required because problem definition is more difficult than many of us want to admit. This scripting effort directly contrasts our previous programs which rely entirely on graphical user interfaces. In those cases, we asked different questions: What GUI features are needed for fundamental operations? How can they be arranged so that operators can figure out how to conduct different experiments? Developing programs using the second approach is easier, but those programs depend on human expertise.

The s-expression communication paradigm allowed programmers to avoid using Scheme rather than encourage them to use it. SORAPS services messages sent from tracking systems, video displays, and timing devices which are written in C++, Java, and proprietary embedding languages. The lack of embedded Scheme in these systems significantly reduced development efficiency because each required the debugging of new communication software and new parsers; neither effort would have been required had Scheme been used. Most of our programmers (about 6) did not mind these difficulties, so they did not choose to adopt Scheme. Perhaps this is due to the fact that the basics of Scheme are easy to learn, then

programmers reject Scheme without realizing how much more it can offer.

We can gain much more utility from Scheme, even though the basics have contributed so positively. However, deciding where to spend development time is becoming more difficult. Significant wisdom is needed to understand the relationships between Scheme staples like modules, units, macros, and languages. Such knowledge is essential to the development of formal laboratory tools that could safely be used by people with diverse capabilities. An experienced computer scientist could contribute significantly to these efforts, but personnel in laboratories like ours need to be convinced that computer science can provide more than just programmers and compilers.

9. Future Effort

Future effort will include developing formal language layers for the controllers and experiments. Common functions have already been adopted, so a few layers of modules should guarantee common behavior.

Automatic optimization of servos and automatic calibration of gimbals pointing are also planned. While tracking stars and satellites, a Scheme program could observe control loop behavior and pointing corrections. From these observations, it could then update servo parameters and pointing models. These tasks are currently done manually.

References

[1] R. Q. Fugate, Air Force Phillips Lab; et. al. Two Generations of Laser Guidestar Adaptive Optics at the Starfire Optical Range. Journal of the Optical Society of America A II, 310-314 1994

[2] Matt Sottile, sexpr.sourceforge.net

[3] Matthew Flatt, Inside PLT Scheme 206.1

[4] Felleisen, Findler, Flatt, Krishnamurthi, How to Design Programs, MIT Press, Section 12.1