Using Scheme to Develop Control Systems for a Large Telescope

Richard A. Cleis

Theme:

When the essence of a language is permitted to access hardware, unexpected functionality can appear.

Engineers tend to do the opposite,
accomplishing too much with low-level code ...

... limiting functionality to the defined behaviors.

This presentation describes using Scheme

at the lowest levels (servos),

at the highest level (executive application), and

for the testing environment.

This presentation will show:

- how telescopes are used at the Starfire Optical Range
- how Scheme was used as an interactive environment for developing servos for the gimbals and secondary mirror
- how (and why) initialization, configuration, and operation are implemented with Scheme.
- how s-expressions are used for communication between executive applications and servo controllers.
- how the Small Fast S-Expression library is used to evaluate Scheme messages in the gimbals
- how scheme was used to interface HTML Forms to the executive application of the telescope

Starfire Optical Range

develops and demonstrates optical wavefront control technologies.

3.5 meter telescope: one of the largest in the world that is both equipped with adaptive optics and is capable of tracking fast objects

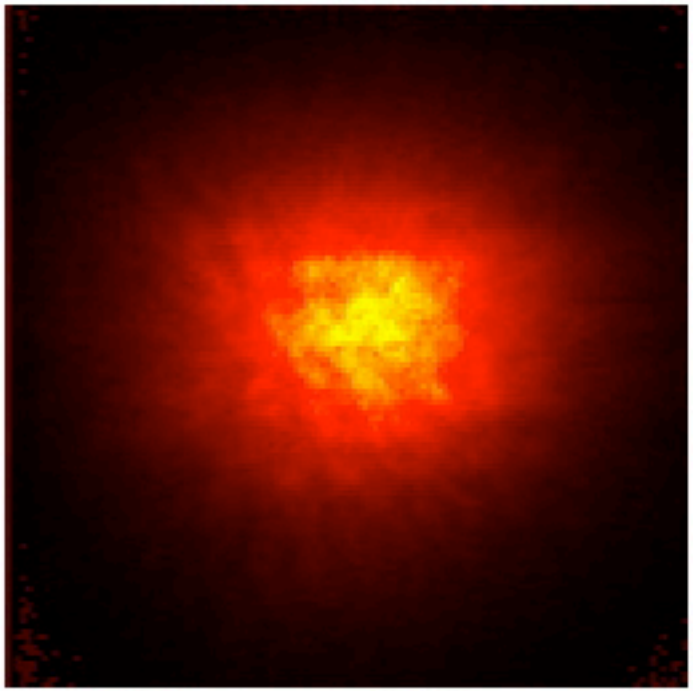
1.5 meter telescope

1.0 meter beam director

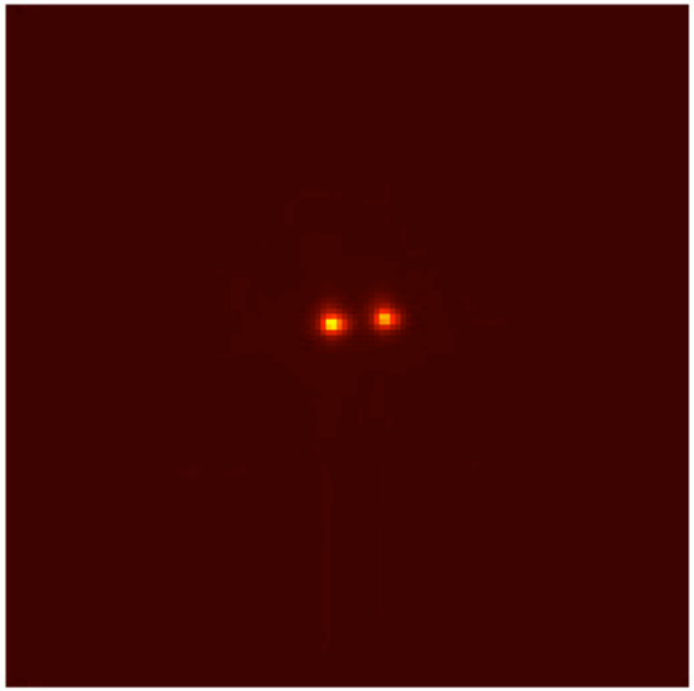
supports field experiments by others within the research community

k-Peg

First light for the adaptive optics system on the 3.5-m telescope at the Starfire Optical Range occurred in September, 1997. This astronomical I Band compensated image of the binary star k-Peg was generated using the 756 active actuator adaptive optics system.



Uncompensated Image



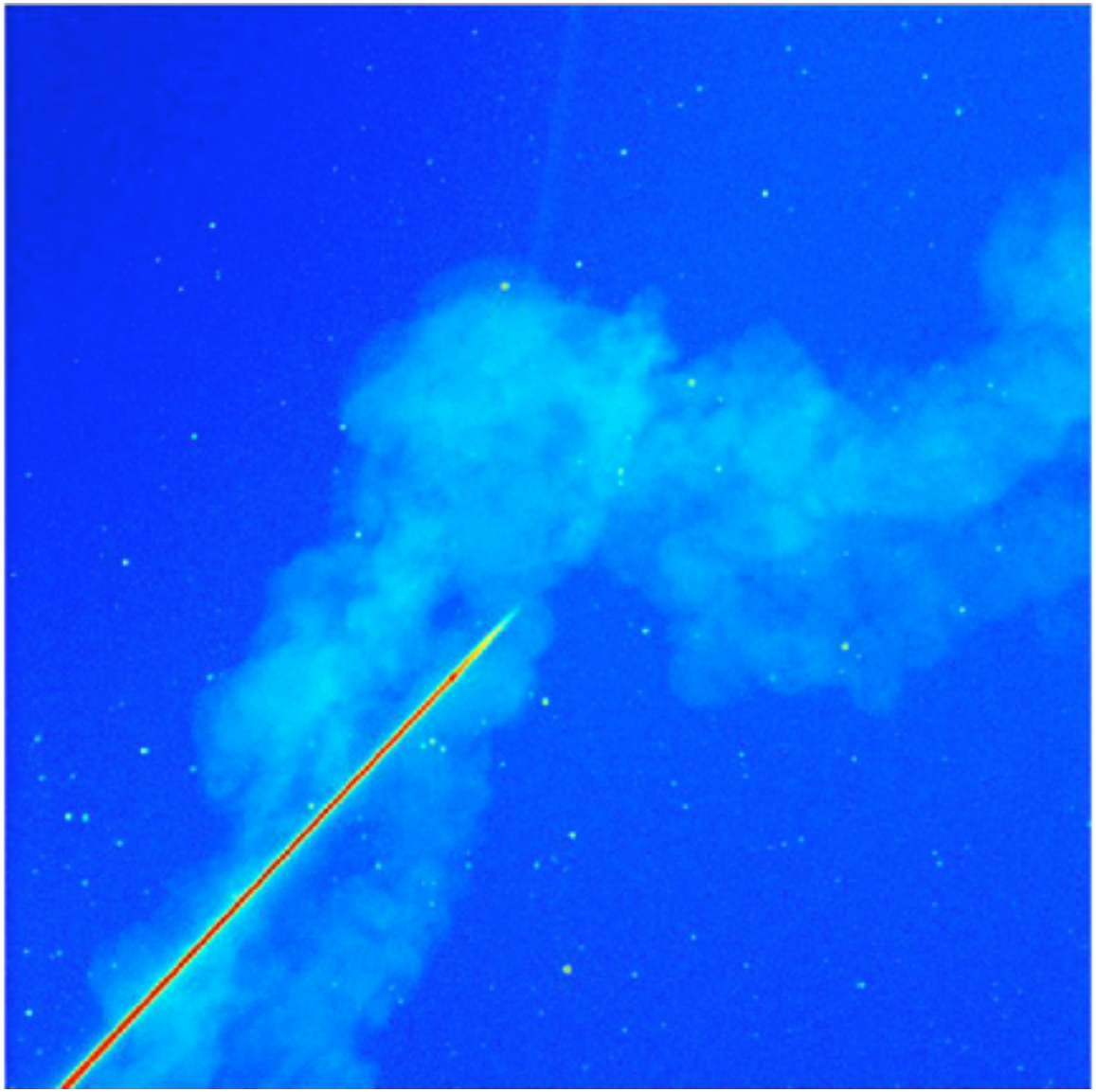
Compensated Image. 0.3 arcsec separation

What SOR tracks

- artificial satellites
- aircraft
- balloons
- interplanetary vehicles
- lunar retro-reflectors
- STS entering the atmosphere
- Leonids meteor trails
- SOR diagnostic sites
- planets, stars, cars

Many experiments require more than one telescope

For example: LIDAR measurements of sodium in lingering Leonids meteor trails



It seems simple enough, at first:
Point three telescopes at a meteor trail.
    (One transmitted the bright laser, another transmitted the
    dim laser in the top of the image, and one made the image)

But where is the trail?
Which telescope will find it first?
Which telescope follows another?
Which telescopes will drift with the trail?
Which telescopes will chase a new trail?
What if the LIDAR telescope adjusts its position...
... but during a following telescope's exposure?
What about parallax?
    The LIDAR measures the range, but what if it has no signal?
What about reference frames...
    Should the telescopes be fixed with the earth
    or the stars? All of them the same?
etc.

This is just one example...
Most experiments are more complicated than it seems.

After years of edit-build-debug cycles, mostly in C...

Scheme was chosen to allow software to be rapidly changed with the requirements of experiments.

Why Scheme?

- It is useful in its most fundamental form: the s-expression
- Many tools are available
- Many tutorials are available
- It is a functional language, appropriate for human interaction
- It is appropriate for automation

Why not include Scheme at lower levels, like the servos?

And why not use Scheme statements for communication?

Configuration, control, automation, and communication would be based on a single language, even if it accesses others.

The 3.5m Telescope was upgraded first...

The C executive app was extended with MzScheme

The 3-axis secondary mirror controller was written in MzScheme

A new servo for the gimbals (a C program) was linked to the Small Fast S-Expressions Library and tested with DrScheme.

A communications paradigm was developed to configure and link everything entirely with Scheme statements

DrScheme was used for prototyping, testing, and learning

SOR 3.5 meter Telescope

Gimbals: 2-axis, el over az

Secondary mirror: 3-axis

Weight: 275,000 pounds

Azimuth torque:
10,000 foot-pounds

Adaptive Optics equipped

Precise control along trajectories having rates up to 1.7 degrees per second

Environments

development

DrScheme, OS X, G4
DrScheme, Red Hat Linux, Pentium
DrScheme, Windows 2000/XP, Pentium

executive application

C/MzScheme, OS X, G4

gimbals servo (2 axes)

C/sfsexp, White Dwarf Linux, 300 MHz X86 PC104

secondary mirror controller (3 axes)

MzScheme, White Dwarf Linux, 300 MHz X86 PC104

DrScheme was used to ease the development of the gimbals, secondary mirror, and the executive application.

The gimbals, and secondary mirror need to communicate to the executive,

so DrScheme was used to interactively prototype and simulate each, as it interacts with the others.

After all were debugged while communicating with DrScheme, they functioned properly with each other.

DrScheme emerged as a permanent validation station.

Example:

The executive must send to the gimbals servo

time, every minute

motion vectors, every second

sun vectors, every 20 seconds

The servo was debugged first by creating threads in DrScheme to send the required messages.

The functions in DrScheme then served as the prototype for the functions added to the executive app.

In the past, the servo and executive had to be debugged simultaneously... a much more intricate procedure since neither are interactive environments; they are just apps.

While the above threads were running, servo parameters were changed to optimize the performance of the gimbals.

friction coefficients

torque constants

disturbance-estimator coefficients

etc.

Functions were interactively created to adjust the parameters.

Motion functions were interactively created to simulate requirements.

difficult portions of typical satellite passes

aircraft trajectories

etc.

Code snip: constant speed, back and forth

```
(define make-speed-of-ms
  (lambda (ms0 ig og deg-per-sec )
    (let* ((full-ms 60000)
           (half-ms (/ full-ms 2))
           (deg-per-ms (* .001 deg-per-sec))
           (og-max (+ og (* half-ms deg-per-ms)))
           (round-ms0 (round ms0)))
      (lambda (ms)
        (let ((rem (remainder (- ms round-ms0)
                              full-ms)))
          (if (< rem half-ms)
              (list ig (+ og (* rem deg-per-ms)))
              (list ig (- og-max
                          (* (- rem half-ms)
                             deg-per-ms)))))))))
```

“ms” is derived from time provided by DrScheme.
“ig” and “og” refer to the inner and outer axes of the gimbals.

Three classes of motion functions were used to “tune” the servo.

The software mechanisms required for hardware configuration and control often dominate specifications and documentation...

By using Scheme, the only mechanism to document is:
(function parameters)

The gimbals servo only understands Scheme;
there is no other way to operate or initialize it.

operations via ethernet,
results returned via ethernet

initialization via a file specified at launch,
results to stdout (a useful diagnostic tool)

The executive-app is mostly configured with Scheme.

The secondary mirror controller is written in MzScheme, so it is naturally configured via Scheme.

Example: Executive needs constants.

Why bury them in a ".h" file? Let users see the "inputs".

This technique also allows for sensitivity analyses;
e.g. try *22/7* for pi... see if Capella still shows up in the telescope.

```
(let ((angular '(24.0 15.0 60.0 3.141592653589793))
                ; hrs/day, deg/hour, seg unit, pi

      (metric '(299792458.0  0.3048 1852.0))
              ; c (m/s), met/foot, meters/nm

      (julian '(2451545.0 2400000.0 365.0 36525.0))
              ; JD of J2K, JD minus MJD,
              ; days/norm-yr, days/julian-century

      (other '(4294967296.0)))
             ; 32 bits

  (set-defined-constants angular metric julian other))
```

set-defined-constants is a primitive function in embedded MzScheme.

Another example: Configuring telescopes

```
(let ((sor3-id '("3.5m Telescope" "gimbals-3"))
                ; long name, short name (no spaces)

      (sor3-location (list wgs84-semimajor-axis
                           wgs84-inv-flat-factor
                           34.9876543 ; geodetic lat
                         -106.1234567 ; geodetic lon
                         1812.34      ; elliptic h (m)
                                    7 ; utc-local (hr)
                     )))
  (display (set-system-gimbals sor3   ; index for C
                               sor3-id
                               sor3-location ))
  (newline))
```

The data is handled freely in Scheme w/o rebuilding the C app;
this is particularly useful when configuring multiple telescopes.

e.g. Scheme might retrieve the data from a database.

Lists of data are typically delivered to C;
functions provided by embedded MzScheme are used to extract them.

The communication paradigm is based on s-expressions.

Scheme environments evaluate the messages.

Non-Scheme environments are not excluded.

Messages merely ask for a list of the results of functions provided by the responding application.

Properties of the responder can be set by supplying parameters.

Example message from executive to gimbals:

```
(list 'do-t-p (do-time 123456789.1)(get-axes))
```

The gimbals controller will respond with something like:

```
(do-t-p 123456789.1 '(123.45678 12.3456))
```

The executive evaluates the above; it already defined do-t-p.

The controller sets its time with the parameter.

The Good

No libraries or tools are required; it's just a technique.

The technique is easy to describe, so
systems that use it are easy to specify.

All incoming messages are simply evaluated;
the destination function is contained in the message.

It can be implemented with an s-expression parser.

The Bad

Reliable exception handling or validation is required.

Functional concept is abandoned:

> The result of sending a message is not the response;
> the technique depends entirely on side effects.

The Rationale

MzScheme does have reliable exception handling.

The motion of a 275,000 pound telescope IS a side effect.

Message evaluator implemented in MzScheme:

```
(define make-function-to-eval-then-reply
  (lambda (socket n-chars)
    (let ((buffer (make-string n-chars)))
      (lambda()
        (let-values (((n ip port)
                      (udp-receive!* socket buffer)))
          (if n (udp-send-to
                 socket ip port
                 (let ((o (open-output-string)))
                   (write
                    (eval
                     (read
                      (open-input-string
                       (substring buffer 0 n))))
                    o)
                   (get-output-string o)))))))))
```

This is fully operational, yet it is so compact that anyone who wants to modify it (or object to it) can understand it.

MzScheme was embedded in the gimbals controller.

- It was easy to do.
- It worked.

However

- Periodic garbage collections (~ every 10s) used about 10 ms.
- The gimbals servo tolerates the issue, but a better solution was desired for other controllers.

The Small Fast S-Expressions Library was used instead.

- No gc... memory is reused, but never recollected
- Elementary Scheme behavior was added to select C-functions.
- C-interface was written to isolate developers from s-expressions

Small, Fast S-Expression Library sexpr.sourceforge.net

Written for supermon, a high-speed cluster monitoring system, at the LANL Advanced Computing Laboratory

Goals: efficiency and simplicity

It reads, parses, modifies, and creates s-expressions.

- reads from I/O
- parses them into an AST equivalent
- modifies them (with functions like ‘cons’)
- converts ASTs into strings of s-expressions

Some other features:

- functions for atom-finding
- Parser returns continuation-state for partially received expressions.

How sfsexp is used

- the AST is traversed to find the first atom in a list
- a string search selects a function and passes the AST
- the function traverses the rest of its list to recover the parameters
- the function creates a result to be returned to the sender

Very simple, very limited

- It only knows ‘list’.
- No nested functions are permitted.
- Servos are endpoints; the real programs are in the sender.

Functions were written to isolate development from the s-expressions

- Arrays of numbers
- Upper/lower limits
- Error messages are generated and returned.

3-axis secondary mirror controller:

Mostly MzScheme; C accesses:

the position inputs (5 voltages)
outputs to motors (3 voltages)
sockets (not necessary, Mz socket-code is fine)

The servo algorithms were developed remotely from DrScheme.

MzScheme (on the controller) handled messages
voltages to the motors
returned positions (voltages) of the axes

Then the algorithms were easily modified to run on the controller.

This was a convincing example of the utility of 3 concepts:

An interactive environment: DrScheme
An embedded Scheme: MzScheme in the controller
S-expressions for communication

Simple Remote Control of Telescope for an Experiment

Convert submission of HTML Form into an s-expression.

Evaluate it.

Return result in new web page.

Implementation

Field-names are functions.

Field-contents are parameters.

Multiple fields/expressions are surrounded by (begin ...).

This technique may be elementary, but
it accesses Scheme functions simply by using a web page...

Web page of telescope controls, implemented with Forms:

```
<html><head><title>Scope Control</title></head>
<body>
<form action='the.cgi' method='post'>
    <input type ='text'
           name ='focus'
           value='7000'>Move the secondary

    <... plus other needed controls ...
```

If 7200 is entered into this form, then

focus=7200

is sent to the executive which converts it into

(focus 7200)

If (* 6 1200) is entered into this form, then

focus=%28*+6+1200%29

is sent to the executive which converts it into

(focus (* 6 1200))

A Web page of functional instructions implemented with Forms:

```
<html>
   <head>
      <title> S-expression Manual </title>
   </head>
   <body>
      <B>S-Expressions</B>
      <form action= 'the.cgi' method='post'>
         <input type='text'
                name='begin'
                value='(focus 7000)'>
         moves the secondary mirror<br>
      </form>

      <... more forms with more instructions ...
```

Makes a form containing (focus 7000) followed by the explanation:
“moves the secondary mirror”

The evaluator eventually gets: (begin (focus 7000))
since the filename is ‘begin’... so the instructions actually work.

CGI implemented in Applescript (before Mz was ‘discovered’):

```
property crlf : (ASCII character 13) &
                (ASCII character 10)
property Redirect : “HTTP/1.0 301 REDIRECT” &
                    crlf & “Location”
property page : “http:experiment.html”
property top :
“<html><head><title>Reply</title></head>
<body><form><textarea ROWS=32 COLS=80>”

property bot: “</textarea></form></html>”

property theReply : “Nothing returned.”

on handle CGI request
   theVar with posted data theData
   tell application “executive”
        set theReply to <<event LISPDOIT>> theData
   end tell

   return top & theReply & bot
end handle CGI request
```

What now? Possibilities:

Automate the servo calibration

Fundamental functions in DrScheme already exist, but

they were used interactively from DrScheme.

Other telescopes could be upgraded quickly with automation.

Immediate optimization possible when load is changed.

Develop rigorous messaging

Apps could register patterns of messages, then

the pattern of each message would be validated. Next,

the functions and parameters would be validated.

The message would finally be evaluated or an error returned.

Final thoughts:

Using Scheme statements at every level enhanced the benefits of using an extension language

The testing station, DrScheme, naturally accessed everything.

Algorithms written and tested in DrScheme were easily moved to applications embedded with MzScheme.

Initialization and configuration at runtime were achieved by merely evaluating Scheme files.

The sfsexp library was easily extended to evaluate messages in a servo that requires efficient software.

Automation efforts are not hindered by hardware inaccessibility.

Inputs: Scheme expressions

Outputs: Scheme expressions

There are no distracting exceptions.