NOTE 204
Updating VLA Observe

W. Young
NRAO

1997 March 26

1 Background

VLA Observe is an important legacy-program for the NRAO. It provides most VLA users with the only practical way to schedule their VLA observations. By most accounts it does a reasonable job of scheduling the VLA. The current version of Observe was conceived in 1988 and completed in March of 1991. Observe has been fairly stable since 1991. A number of bug fixes and small enhancements have been made (on the order of 30 minor revisions have been recorded). The program was ported to C++ in the summer of 1996 and no minor releases have been made since its official release on Dec 31, 1996. It has approximately 44000 lines of code including comments (comments are fairly sparse). Over the years several problems have surfaced with Observe:

1. It is key-context driven with a "custom" windowing scheme (rather than event driven),
2. Navigation is tricky for novice and occasional users,
3. User documentation is out-of-date,
4. There is no design/implementation documentation
5. Newer features are poorly documented,
6. It is difficult to add new features,
7. Maintenance is done by one person with no backup, and
8. Most common, **keypad problems!**

In addition to the above problems, when Observe was originally written most users had terminals connected to a central computer. The general computing environment at NRAO (and for NRAO's users) has changed. We now

- have PC/Workstations,

- use GUI based applications,
- have AIPS++,
- use the World Wide Web, and
- have Java available.

2 Future Direction of Observe

2.1 Goals & Options

The changed nature of computing since Observe's introduction have caused us to review Observe's current status and reflect on how we might "modernize" it. Modernization might include:

- Remove the keypad dependence,
- Improve the user interface,
- Improve portability,
- Take advantage of AIPS++ Measures (coordinates, time, Doppler tracking),
- Easier maintenance,
- Easier modification, and
- Better access to documentation.

There are two options:

1. Revise user and programmer documentation.
2. Change to an event driven based program.

2.2 Discussion

Option 1 leaves things much as they are. Current documentation would be updated and there would be some programmer documentation. This option doesn't improve the interface for the observer or improve the maintainability of Observe. Screen real-estate in the current version is very limited, it is difficult to add new features. Information is hidden and navigation is not always straight forward.

Option 2 requires rewriting and to some extent redesigning Observe. A redesign would allow a more natural interface (and avoid the keypad altogether). The current version closely couples the user interface with program functionality. A redesign would allow loosening the coupling, so the user interface and scheduling/file-checking functionality are more logically separated.

There are several choices for an event-driven program:

1. "Classic X11",
2. Tcl/Tk, or
3. GlishTk.
4. Java,

In all choices, **almost none** of the existing code will be preserved. Translating many of the concepts and algorithms from VLA Observe to the new version should be fairly straight forward.. The AIPS++ Measures code will be used for Doppler corrections, coordinate and time conversions (probably as a server object). All AIPS++ tasks/objects use the Measures code, so Observe's numbers will be consistent with those of AIPS++.

A "Classic X11" based program would require compiling and linking for each platform. This is very much a classical approach for providing software. Users down-load Observe and ancillary files to their local machines, configure for their local environment, and run Observe on the local machine. Often we see files submitted using an out-of-date version of Observe. Porting to a non-supported architecture while not difficult does require time and some effort. Often when a port is needed, a computer of that architecture is not available locally.

Tcl/Tk and GlishTk are a Tk-widget-based X11-toolkit called from a command interpreter. Tcl/Tk uses the wish shell and GlishTk is the AIPS++ command interpreter. Writing Observe in GlishTk provides a straight-forward interface with existing AIPS++ code. While being quick-to-prototype, both have their drawbacks. Some problems with Tcl/Tk include:

- AIPS++ measures code may not run on all platforms,
- The code would have to be distributed (latest version not always guaranteed),
- Requires the users to have the proper Tcl/Tk configuration, and
- It can be difficult to debug Tcl/Tk code.

Problems with GlishTk include:

- Requires AIPS++ to be installed,
- Current version GlishTk doesn't support all of Tk features/options,
- It can be difficult to debug GlishTk code, and
- Latest version not always guaranteed.

A Java based program offers the hope of "write-once run-anywhere" (no porting issues). A user connected to the net could be guaranteed the latest version (including the latest calibrator positions and NRAO defaults). Observe seems particularly well suited to "net". Links are possible between Observe and the most-current (best?) advice on observing strategies, interference problems, and other timely observing information. User could also down-load the Java byte-code and run Observe locally. The biggest drawback to Java is, "it is cutting edge".

2.3 Redesign/Reimplementation

There are three phases in redesigning and reimplementing Observe:

First, Modify the design to remove the interface from the scheduling/verification routines;

Second, Code/Test the foundation; and

Third, Code/Test the user interface.

Considering the nature of observe, the language of choice is Java. Why Java?

- Portability. Java will run on most platforms as is, i.e. no recompilation/relinking.
- Can be run effectively over the network.
- Provides the most seamless distribution. A VLA observer only has to connect to the observe web page to run the latest version.

The last point provides the most compelling reason.

The following list is a "back-of-the-envelope" targets list that would be refined after the design work. Time estimates indicate the relative amount of work (rather than the actual time). The time required to implement the GUI is likely to be independent of implementation (Java, X11, Tcl/Tk, or GlishTk).

Testing individual components and programs will follow the AIPS++ model. As we finish parts of the project, we prepare test-cases with known results. At regular intervals the test-cases are run and compared with expected results. If a test fails we can track it down before it becomes a bug report. Also, we can run the test-cases after any changes to ensure no unforeseen problems have occurred. Testing the GUI is more labor intensive and will require interactive testing by NRAO staff members.

VLA Observe – Targets
Phase 1 – Design - deliverable (2 weeks)

---

  rudimentary design of current Observe
  redesign Observe

Phase 2 – Foundation (17 weeks)

ASCII Text to Observe file - deliverable (9 weeks)
  conversion into Java
  parser
    user defined

        - observe file
    - interface with AIPS++ Measures
    - scheduling routines
    - frequency checking
    - output routines
    - unit testing
- Reports - deliverables(4 weeks)
    - analysts
    - scheduling
    - frequency
    - time-on-source
    - summary
    - unit testing
- Databases - NRAO/user (3 weeks)
    - frequency setups - deliverable
    - calibrator - deliverable
    - source - deliverable
        - position
        - orbital elements
        - interpolated from table

Integration into a scheduler - deliverable(1 week)

Phase 3 – Interactive Interface (20 weeks)

- Scheduler GUI (10 weeks)
    - screen layouts - deliverable
    - source catalog - deliverable
    - frequency/correlator setups - deliverable
    - scan editor - deliverable
    - visual loser - deliverable
        - LO calculator
        - band/interference plots
        - Doppler tracking
    - observe file viewer - deliverable
    - scan contraction/expansion
    - starting conditions - deliverable
    - editing schemes

User Testing (4 weeks)

- Planning Tools - deliverables (2 weeks)
    - uptime
    - uv-coverage

- az-el plots
- calibrator flux history

Distribution - deliverable (1 week)

Documentation (4 week)

- programmer - deliverable
- user - deliverable
  - tutorial
  - on-line help

Phase 4 – Futures (?)

- suggested schedule
  - time range, source, frequency, pick calibrators, etc...
- optimal scheduling
- incorporate VLA plan