Sneezy Note #6: The Parallel Debugger *Sneaky*

Lars T. Hansen

June 12, 1995; August 22, 1995; February 15, 1996

Abstract

I outline the design of a debugger for large parallel programs. The most important features of the debugger are its programmability and the way in which the parallel program is exposed to the programming language.

This note is a bit of a hodgepodge because it mixes ideology, design, and implementation into one unholy mess. As it matures, it will be split into multiple notes. For now, consider it a snapshot.

1 Introduction

The *debugging problem* is the problem of providing the programmer with a flexible and powerful debugging interface to her program. Traditional sequential debuggers like `adb`, `dbx`, and `gdb` attempt to solve this problem by providing the programmer with a rich command set which is more or less on the level of the source language of the program being debugged. Apart from macro definitions allowed by some of these debuggers, and an ability to read commands from a configuration file so as to set up a debugging context, the command sets are static, and the capabilities of the debuggers are fixed. Parallel debuggers have a more difficult problem to solve, that of debugging multiple, often distributed, threads of control and their interactions. Providing the user with a comprehensive command set for parallel debugging which will be adequate for every task seems like a truly hard problem. This paper suggests that there is a different technique, tending toward minimalism rather than maximalism, which may make the lives of both the debugger writer and the programmer simpler than they would be with a mega-debugger.

Inspired initially by the ACID debugger of Plan 9 [1], where the debugging problem is viewed as that of providing abstractions on top of primitive interfaces to the program being debugged,[1] I propose *Sneezy*, a portable low-level interface to the HPC++ runtime, and *Sneaky*, a debugger infrastructure based on a high-level multi-threaded interactive programming language with access to the Sneezy API.[2] Sneezy has already been desribed in numerous notes (see for example note #1).

The general idea is for Sneaky to connect to the parallel program by way of Sneezy, and, using facilities in Sneezy, hang *event handlers* from hooks provided by the RTS and the program. When an event occurs, the handler is triggered, causing a message to be sent to the debugger, and causing the triggering thread to suspend execution, awaiting further instruction from the debugger. The debugger can then instruct the debuggee to go into a command loop which allows the debugger to inspect the program. This is no more than a normal debugger can do. We propose, however, that the interactions between the debugger and the debuggee be exposed in a full-fledged programming language, and that the debugger user (the programmer) can write programs in this language which utilize the debugger infrastructure to accomplish novel debugging and data extraction tasks. The added power becomes important when the programmer is debugging a concurrent program with a large number of address spaces and threads: the programmer's

[1] An informal literature search reveals that the idea is by no means new. Many research projects have worked on programmable debuggers for various languages, one of the earliest seems to be David Hanson's debugger for SNOBOL4 [2], and the most sophisticated so far being the Dalek debugger from Ronald Olsson's group at UC Irvine[3, 4, 5]. Undoubtedly there must exist such debuggers in the Lisp community as well, but I am not yet aware of any published work.

[2] *Sneezy* is an acronym for "Scheme Nerd's brEEZY"; *Sneaky* is an acronym for "Scheme Nerd's Extensible Analysis frameworK and displaY" (for example).

debugging needs will become very complex, and by allowing her to capture these interactions in the form of an executable program both lends power to the debugging and promotes early specification and reuse of debugging structure.

The parallel program is exposed to the debugger programming language as a *model* of the parallel program. The model is a programming language object with operations defined on it, and it can be extended using subtyping mechanisms available in the debugger language (subclassing in object-oriented languages or type extensions in Oberon, for example). The base model provides a slew of operations for manipulating the program, its symbol table, its run-time state, and so on.

The choice of extension language is in my view important. There are conflicting opinions here: some advocate a debugging language which is close or identical to the language being debugged, reasoning that the programmer's familiarity with the language will aid debugging. For example, the Dalek group has a debugging language (called GDL) which is a dynamically typed C with minor extensions to cope with the dual scopes present (one in the debugger, one in the debuggee). High-level constructs are implemented as GDL library procedures. Others advocate using a common higher-level language, for example Tcl, on the basis that a high-level language is important but that just as important is programmer familiarity.

Personally, I sympathize with the latter view but find Tcl's semantics repulsive and have chosen to use Scheme[6] for Sneaky's debugger language. Scheme is small, well understood, and have clean semantics, and its emphasis on lexical closures and list processing give the programmer considerable power to write both one-shot programs and reusable code.

2 The Model-View-Controller Paradigm

The model-view-controller (MVC) paradigm was pioneered by Smalltalk. The idea is that a *model* (a programming language object) represents a system and its state at all times, and that different and multiple *view-controller* pairs provide users with interaction capabilities to the model and therefore the system.

The model accomodates installation of views which will be notified when aspects of the model changes. When the view receives a notification of a change, it will query the model for its new state and update itself. The controller interacts with the model by changing the model's state, which causes views to be notified of the changes.

In an object-oriented setting, models, views, and controllers can be subclassed, allowing existing views to be used with extended models (as well as extended views to be constructed for (extended) models).

3 Debugger Features

In this section I try to outline some features which Sneaky might have. Some of these have been tested in practice in the previous Sneaky prototypes; others are just off the top of my (or someone else's) head.

3.1 Some Displays

- An *event trace display* is a running listing of all the events that have occurred so far. The view can be implemented as a scrollable text window; it would be reasonable to have some logic behind the view which would let the user restrict the focus of the event trace to a subset of contexts or threads. It might also be reasonable to restrict the number of events remembered, and to store the event trace in an object which can be manipulated by a user program.

  It should be possible to have multiple views; it's OK to only allow disjoint sets of threads for now.

  A simple prototype which only has one global trace already exists.

- A *control panel* is a visual interface to the control of individual threads or groups of threads. It lets the user enable and disable events by checking boxes. Methods for selecting the threads to which to apply the selected events is outlined in section ??.

  There will also be buttons which pertain to all currently selected threads and which let the user perform common operations: continue a thread, stop a thread (not currently supported by Sneezy), others?

It should be possible to have multiple control panels.

The control panel should probably also have buttons on it to control global behaviour: continue all, quit, restart, and so on. Perhaps it should be possible to start an event trace display from the control panel after selecting the threads to apply the trace to.

A simple prototype already exists.

- A *command window* is a text window where the user can enter debugger commands which are not easily expressed by GUI means; some command loop needs to be running here. The command set should be (a subset of) that of gdb, for compatibility with something reasonable ("accept no dbx!"). The command window should not be the xterm from which the debugger was started, but an actual window.
- A *state display* displays the state of each thread (it must be scannable in some reasonable way; see section ??).

3.2 Some Commands

Here are some ideas for functionality which has not yet been tried out.

- Correlating events in different threads or contexts: when one thread does a begin-rpc, and another thread in response does a begin-service-rpc, it would be possible, on a task display, to note the dependence by drawing an arc between the two points; ditto for the end of the rpc.
- Bernd's idea: a display which shows the dynamic position (on a procedure basis) of each thread, perhaps condensed in some way by summarizing several procedures in the same procedure by a number on the arc.
- Run-time profiling: dynamic utilization feature on a per-context (or per-processor) basis. This is very related to profiling; utilization time is essentially time spent executing, not waiting in barriers or for remote get/put/rpc.
- Communications volume: number of messages sent to other nodes, their relative type frequencies, times to complete, topologies.

3.3 Interactive Thread Selection

Here are some idea for how to select which threads to apply a command to.

Textual expressions: In the control panel there will be a text (with a pull-down list attached to it). The users enters expressions in this box to select contexts and threads. For example,

- `1-5` = all threads on contexts 1 through 5.
- `1,3-5` = all threads on contexts 1, 3, 4, and 5.
- `1(1,3),4` = threads 1 and 3 on context 1, all threads on context 4.

It is also possible to name a group using the syntax `mygroup=1-5`. Other operations might be desirable (group difference, "except") but it's not important right now.

When a group is defined (with name or without) it is remembered and can be accessed from the list box. If the group is named, it can be recalled by typing its name only. When a group is selected, the display shows data (events, states) related to that group only.

Note that the list of group definitions should be available as an object so that user code can get at the data.

It's very unclear what partially overlapping groups mean, but at the very least it's reasonably meaningful for a group to include another group (even though manipulating the subgroup may cause trouble).

Graphical processor display: This is a display which lets the user select contexts and threads by GUI actions. The exact layout depends on the structure of the parallel machine, but assuming that we have an SPMD system (Paragon) then a 2-d layout might work OK. Operations should be select, lasso-select, shift-unselect, shift-select. This display should also interact with some object which maintains the resulting group, and there should be a name box in the display so that a group can be named.

4 The Sneaky Model

A model in Sneaky is an object which hides the Sneezy interface to the program and which defines a number of common and useful operations on the program state. For example:

- There will be operations which correspond to primitive Sneezy commands (`continue`, `install-handler`, `read`, `write`) but they may be higher level; for example, Sneezy does not decode data read or encode data written, but the read and write interfaces to the model would provide these services.
- There will be extended services based on primitive Sneezy commands; for example, the operation `continue*` sends a continue to all the threads named by its argument.
- There will be higher-level services to cope with symbol tables, scopes, stacks, variables, and so on in the running program. These will constitute the greater part of the model and will provide significant services to the model's user. Most likely, this part of the model will be specific to the programming language of the model.

For concreteness, I will discuss how the model will be defined and used in STk, which is the Scheme system in which the prototype Sneaky will be written. STk provides a CLOS-like object system called STklos. STklos is a slot-based object system with meta-classes, multiple inheritance, generic functions, multi-methods, and the kitchen sink.[3] In my opinion it is a goal to stick to a subset of STklos which requires only single inheritance and can be re-implemented in reasonably portable Scheme, and uses no meta-classes. The restrictions impose a modicum of portability and comprehensibility on the resulting code.

A Sneaky model will be realized in the `<model>` class.[4] There are no public slots in `<model>`; all model access is by calling generic functions.

Some of the generic functions which are[5] defined on the model are described in the following sections. Note that these definitions represent a first cut and are subject to change.

4.1 Creating and manipulating models

You create a model by calling the `make` procedure with the class name as a parameter:

`(make <model>)` ⇒ *instance of* `<model>`

There is a generic procedure `initialize` defined on the model?

```
(let ((model (initialize (make <model>))))
  (run-program model "fire.cs.uoregon.edu" "shallow")
  ...)
```

run-program takes host, path of executable on that host, and starts the program. It returns `#t` if the connection succeeded and all the `context-ready` messages were received; `#f` otherwise.

`(run-program <model>` *host path*`)` ⇒ *boolean*
`(contexts <model>)` ⇒ *integer*
`(context-list <model>)` ⇒ *(integer ...)*
`(threads <model>` *cid*`)` ⇒ *integer*
`(state <model>` *cid tid*`)` ⇒ *one of* `waiting`, `running`, `terminating`, `terminated`

[3] The version of STklos distributed with STk 2.1.7 will be used. See Appendix C of the STk User's Manual for a discussion.
[4] It is generally accepted practice to name all STklos classes with enclosing angle brackets.
[5] Will be...

4.2 Controlling the program

The program is controlled with simple extensions to the Sneezy model:

`(continue <model>` *cid tid*`)` ⇒ *unspecified*
`(continue* <model>` *((cid . tid) ...)*`)` ⇒ *unspecified*
`(continue-all <model>)` ⇒ *unspecified*
`(handle-event <model>` *cid tid procedure*`)` ⇒ *unspecified*
`(handle-event* <model>` *((cid . tid) ...) procedure*`)` ⇒ *unspecified*
`(handle-event-all <model>` *procedure*`)` ⇒ *unspecified*

4.3 Manipulating symbol tables and program state

Let's define a *name* as a unique token which identifies a live variable in the context of a currently stopped thread. Furthermore, a *scope* a token which identifies some lexical block in the dynamic call chain of a currently stopped thread. Then a string can be mapped to a name with the call:

`(lookup <model>` *scope string*`)` ⇒ *name*

The scope can be manipulated with these functions:

`(current-scope <model>` *cid tid*`)` ⇒ *scope*
`(enclosing-lexical-scope <model>` *cid tid scope*`)` ⇒ *scope*
`(enclosed-lexical-scope <model>` *cid tid scope*`)` ⇒ *scope*
`(enclosing-dynamic-scope <model>` *cid tid scope*`)` ⇒ *scope*
`(enclosed-dynamic-scope <model>` *cid tid scope*`)` ⇒ *scope*
`(current-location <model>` *cid tid*`)` ⇒ *integer*

(In C and C++, nested lexical and dynamic scopes are equivalent within a function, but not at function barriers due to the way the class functions as an enclosing lexical scope for methods and nested classes.)

Other possible interfaces to the symbol table might be:

`(type-of <model>` *name*`)` ⇒ *type-representation*
`(get-value <model>` *name*`)` ⇒ *value-representation*
`(set-value <model>` *name value-representation*`)` ⇒ *unspecified*
`(file-and-line <model>` *cid tid location*`)` ⇒ *(string . integer)*

Notice how the `read` and `write` interfaces can be hidden behind higher-level interfaces that know about types, names, and so on.

5 A Hierarchy of Models

As noted earlier, many of the operations defined on a model are language specific. Therefore, we can contemplate a hierarchy or set of models rather than a single model which serves all needs.

For example, a model for C++ programming can be based on a more basic model which does not support any specific language, but which (perhaps) provides generic symbol table access and so on. The decoding of elements of the symbol table is up to the extended model. The extended model can either subclass the basic model, or it can use an instance of the basic model. The latter is perhaps more useful, since it supports multi-language models well: Imagine that there exists a Fortran model and a C++ model. We can build a Fortran-and-C++ model which incorporates both of these, where the higher-level model is the model the debugger relates to and the underlying models are used in separate parts of the program, as appropriate. But the underlying model of the Fortran and C++ models (the most basic model) must be the same *physical* model so that the Fortran and C++ models are synchronized. Therefore, using model instances in extended models seems more appropriate than using inheritance for these cases.

5.1 Problems and Open Questions

5.2 Problems

Here are some problems which are important and which depend fundamentally on the programming language being used.

- Not using inheritance means that the higher-level models do not share the data of the underlying models, which again means that the operations defined on the underlying models can only be used on the higher-level models if all field access is expressed in terms of generic function calls on models, *and* the extendend models override all field access functions for the underlying classes. This is *extremely* ugly.

- If there are multiple views and controllers all accessing the model at the same time, then access to internal data must be controlled by some synchronization primitive. Assume that each model instance has a mutex which can be used to control access to the object. We can clearly define the default operations on the model to synchronize on the mutex:

  ```
  (define-method continue ((m <model>) cid tid)
    (call-with-mutex (slot-ref m 'mutex)
      (lambda ()
        ...)))
  ```

  and this works fine. However, there are three issues: new methods on existing generics, new generics on existing models, and new generics/methods on derived models. All of these must also perform the locking, and eventually, someone is bound to forget.

  The CLOS `:before` and `:after` methods seem almost perfect for the job: the before method locks the lock, the after method unlocks it, and the requirement for synchronization is reduced to remembering to install before and after methods for new generics, everything else being taken care of by the default mechanisms.

5.3 Open Questions

Some problems are of less of a language nature:

- How to make the model extensible in terms of dealing with multiple clients? This seems truly hard, and we may have to use different models for these situations.

- Using locking can lead to deadlock in scenarios like the following: The view is calling the model and maintains a lock on the view. At the same time, an arriving event in the model causes a callback to the view; the callback maintains a lock on the model. Deadlock ensues.

References

[1] Winterbottom's Usenix paper on Acid.

[2] Hanson's SPE paper on the Snobol 4 debugger.

[3] Olsson et al's SPE paper on Dalek.

[4] Olsson et al's IEEE paper on Dalek.

[5] Olsson et al's Usenix paper on Dalek.

[6] IEEE Standard for Scheme.

[7] Gallesio's VHLL Usenix paper on STk.