AGDB: A Debugger for Agent Tcl

Melissa Hirschl and David Kotz
Department of Computer Science
Dartmouth College
Hanover, NH 03755
E-mail: {hershey, dfk}@dartmouth.edu

Technical Report PCS-TR97-306

February 4, 1997

Abstract

The Agent Tcl language is an extension of Tcl/Tk that supports distributed programming in the form of transportable agents. AGDB is a debugger for the Agent Tcl language. AGDB mixes of traditional and distributed debugging facilities. Traditional debugging features include breakpoints (line-specific, conditional, and once-only), watch conditions and variables, and interrupts. Distributed-debugging features address issues inherent in distributed programming such as migration and communication. These capabilities make debugging distributed programs difficult because they add complexities like race conditions to the set of problems a program can encounter. This paper discusses how AGDB uses distributed debugging features to debug agents.

1 Introduction

A *transportable agent* is an autonomous program that can interrupt execution at any point, collect its state, migrate to different machines and resume execution at the point of interruption. They provide a shift in programming paradigm from the traditional client-server model in that they can move the computation to the location of the data, or vice-versa. This flexibility can often reduce network usage and improve overall performance. This often blends well with applications such as mobile computing and information retrieval where network bandwidth is limited.

The Agent Tcl language is an extension of Tcl/Tk that supports distributed programming in the form of transportable agents [Gra95a, Gra95c]. AGDB is a debugger for the Agent Tcl language. AGDB mixes traditional and distributed debugging facilities. Traditional debugging features include breakpoints (line-specific, conditional, and once-only), watch conditions and variables, and interrupts. Distributed-debugging features address issues inherent in distributed programming such as migration and communication. These capabilities make debugging distributed programs difficult because they add complexities like race conditions to the set of problems a program can encounter [MH89]. This paper discusses how AGDB uses distributed debugging features to debug agents.

1.1 Tcl/Tk Overview

Tcl is a scripting language similar to other UNIX shell languages [Ous94, Wel95]. Tk is an extension of Tcl that supports creation and manipulation of user-interface widgets. The Tcl/Tk interpreter contains built-in functions such as `source`, `catch`, `exit`, `main`, etc. This paper assumes the reader is familiar with (though not an expert on) Tcl.

The Tcl `source` procedure is analogous to the `source` command available in most UNIX shells. It allows the user to include a library or to divide code into multiple files to keep the code organized. The `main` procedure creates a widget on the display; it is built-in to Tk and not Tcl. The `catch` procedure takes a script as its argument, and the interpreter executes the script. If the script executes without errors, `catch` returns 0, otherwise `catch` traps the error and returns 1, thereby averting an `exit` call.

1.2 Agent Tcl Overview

The Agent Tcl language is an extension of Tcl/Tk in which built-in functions support implementation of transportable agents [Gra95a, Gra95b, Gra95c, Gra96][1]. In Agent Tcl, agents assume their identity, `(machine-name, machine-ip, numeric-id, symbolic-name)`, by executing the `agent_begin` and `agent_name` commands. After the execution of `agent_begin`, the agent is able to communicate with other agents, *migrate* to remote machines, and spawn child agents.

Agents migrate to remote machines by executing the `agent_jump` procedure. During *migration* of transportable agents written in Agent Tcl (caused when the agent executes the `agent_jump` command), the Agent Tcl interpreter suspends an agent's execution, sends the agent's state to the destination machine, and arranges for a new interpreter process to load that state and resume the agent's execution at its point of suspension. Since an agent executing on a remote machine cannot affect the display of its home machine, it may be difficult for a programmer to verify an agent's progress.

In Agent Tcl, agents spawn children (*non-root* agents) by executing either the `agent_fork` or the `agent_submit` command. Like UNIX fork, `agent_fork` creates a copy of the agent. After an

[1] For more information, see `http://www.cs.dartmouth.edu/~agent/`

`agent_fork` call, both parent and child agents begin running at the parent's point of suspension. The `agent_submit` command takes a script as its argument and spawns a child agent to execute that script.

Agent Tcl supports communication among agents through the `agent_send`, `agent_receive`, and `agent_meet` procedures. Knowledge of an agent's birth, whereabouts, and interaction with other agents may give the programmer a clearer picture of the agent's progress. We have implemented AGDB to aid in the programmer's struggle to monitor an agent's progress.

1.3 AGDB Overview

The AGDB implementation is split into two parts: a debugger agent and a library containing procedures an agent uses to communicate with the debugger agent. The debugger agent can monitor as many agents as the user chooses. The debugger monitors an agent by transforming it to include the library component of AGDB as well as a *password* and other information discussed in sections 2.1 and 3.1. The agent begins each communication with the debugger with its password. The debugger thus identifies and authenticates the agent's communication via the password.

The library component of the debugger contains wrappers for most built-in Agent Tcl procedures (such as `agent_jump` and `agent_receive`) as well as several built-in Tcl/Tk procedures. By *wrappers*, we mean that the procedure is redefined to execute extra code both before and after the actual procedure call. The wrappers include code to help the agent maintain contact with the debugger. Throughout this paper, we discuss details concerning the wrapper procedures.

2 File Preprocessing

Whenever the agent makes contact with the debugger, the agent tells the debugger which line of code it is about to execute, and the debugger highlights that line in the agent's window (see Figure 2 in Appendix C). For the agent to know which line it is about to execute, the debugger must infuse a knowledge of line numbers into the agent. The debugger does so by preprocessing a file when the user adds it to the debugger's domain.

The file is only preprocessed again at the user's direction, using the **reload** menu option. Also, if the user chooses the **auto reload** option and then presses **run**, the debugger checks the file's most recent modification time, and if the modification time is later than the time of the last preprocessing, the debugger preprocesses the file again, resulting in removal of all line-specific breakpoints.

2.1 How the Preprocessor works

The debugger reads the original file and saves the preprocessed version of the file in the `/tmp` directory. We call this new file the *p-file*. The *p-file* is a copy of the original file, except that it has calls to the AGDB library procedure `_agdb_dynamic_break` interspersed among the lines of code. For a sample *p-file*, see Appendix A. During execution of the `_agdb_dynamic_break` procedure, the agent decides whether to stop at a breakpoint.

2.2 Limitations of the Preprocessor

Since the preprocessor analyzes code on a line-by-line basis, the user cannot specify breakpoints between statements that appear on the same line in the file. We call same-line code *debug-atomic* because the debugger treats it as an atom for the purposes of breakpoints and watch conditions. The user can interrupt or kill the agent, however, during execution of a *debug-atom*. Although the *debug-atom* is interruptible, the user only sees the *debug-atom*'s entry-point. Figure 1 shows how the user can write code to minimize the number of *debug-atoms*.

```
# debug-atomic code:  ***
                      if {$x > 6} {set y 4} else {set y 0 ; set z 3}
                      ***

# debugger friendly code:  ***
                           if {$x > 6} {
                             ***
                             set y 4
                             ***
                           } else {
                             ***
                             set y 0
                             ***
                             set z 3
                             ***
                           }
                           ***
```

Figure 1: debug-atomic versus trackable code. *** denotes a possible breakpoint.

3 Running Agents

When the user presses **run**, AGDB creates another file in the `/tmp` directory. We call this file the *g-file* because it contains global variables that the agent needs to communicate with the debugger. After the debugger creates the *g-file*, it is executes the *g-file* in the background. After setting up the debugging environment, the *g-file* sources the *p-file*. For a sample *g-file*, see Appendix B.

3.1 G-File

In this section, we describe the *g-file*'s structure. The first line of the *g-file* must tell whether the agent is a Tcl agent or a Tk agent. We copy this line from the agent's original file. The second part of the *g-file* sources `agdb_lib.tcl`, which contains all the procedures the agent needs to communicate with the debugger. The next section of the *g-file* initializes global variables, including the agent's password, the debugger's hostname and agent id, information describing when and why to initiate breakpoints, and other options chosen by the user. The *g-file* sources the *p-file*. We enclose the `source` in a `catch` to preserve the return value of the *p-file* (or the error information therein). After sourcing (and thus executing) the *p-file*, the *g-file* calls the library procedure `_agdb_end_comm` with the `catch` and `source` results as its arguments. For Tcl agents, the `_agdb_end_comm` procedure performs a final communication with the debugger, and the *g-file* ends by returning the source result. We want Tk agents to enter the event loop, so after sourcing (and thus executing) the *p-file*, the Tk agent only makes a final communication with the debugger if an error was caught in the *p-file*. The Tk agent normally makes its final communication with the debugger when the agent calls the `exit` wrapper defined in AGDB's library.

4 Breakpoints

AGDB offers traditional debugging features such as line-specific breakpoints and watch variables and conditions. The user can add and remove these features via the graphical user interface (see Figure 2 in Appendix C). To add a line-specific breakpoint, the user selects a line of code and decides which type of breakpoint to add (via the agent window's breakpoint menu). A line-specific breakpoint may or may not be conditional, and may occur only once or each time the line of code is reached. If the user adds a watch condition (via the agent window's watch menu), the agent will break before each line of code for which the condition the user entered is true. If the user adds a watch variable (via the agent window's watch menu), the agent will break after every modification of the the variable the user entered.

At startup, the debugger modifies its mask to trap all meeting requests to the `_debug_trap` procedure, where it accepts the meeting and validates the identity of the agent being debugged via its password.

```
mask add [mask new] "ANY -handler _debug_trap"
```

When an agent hits a breakpoint (during either `_agdb_dynamic_break` or a wrapper procedure), it calls `agent_meet` to arrange a meeting with the debugger, and the request is trapped to the debugger's `_debug_trap` procedure. The agent blocks until the debugger accepts the meeting, at which time a TCP/IP connection is made. Thereafter, the TCP/IP connection is maintained until the user directs the agent to continue running.

AGDB allows the user to examine the agent's stack as well as to evaluate commands in the context of the agent and to print variable values during any break. The user's requests for these activities warrant immediate response from the agent. Since other agents may be trying to communicate with the debugger, we cannot allow the debugger to wait for a response from the agent. Instead, when the agent has formed its response, it calls `agent_send` to send the response to the debugger. The debugger traps all messages received to the `_answer_trap` procedure, which outputs the agent's response to the user.

5 Beginning Agents

All agents are required to call `agent_begin` to acquire a controlling server and receive an identity other than their process id. When an agent calls `agent_begin` successfully, it uses its password to initiate its first meeting with the debugger. For non-root agents, `agent_begin` is implicit. A non-root agent's first meeting with the debugger (through its parent's password) is similar to the meeting initiated during an `agent_begin` call. During the first meeting, the agent notifies the debugger of its identity `(machine-name, machine-ip, numeric-id, symbolic-name)`. After a successful `agent_begin` call, the agent modifies its mask variable to receive interrupts from the debugger (see Section 6). The agent also activates the `source` wrapper described in Section 11.

6 Interrupting Agents

The user can interrupt an agent between the time the agent calls `agent_begin` and `agent_end`, except during jumps and breakpoints (see Section 9.1 for details on jumps). After an agent calls `agent_jump` or `agent_begin` (either regularly or implicitly), the agent modifies its global mask variable `mask(message)` to trap messages from the debugger to the `_agdb_debug_trap` procedure, thereby enabling the agent to accept interrupts from the debugger.

```
mask add $mask(message) \
    "$_agdb_debugger_machine $_agdb_debugger_id -handler _agdb_debug_trap"
```

When the user presses **interrupt**, the debugger sends the agent a message. If the agent is blocking for any reason (such as an I/O call or a blocking receive), the interpreter queues the debugger's message until the agent returns from the blocking call. Unfortunately, this precludes the debugger from relieving the agent of indefinite blocking. When the interpreter dequeues the message, the agent initiates a non-interactive breakpoint with the debugger. By non-interactive, we mean that the user cannot access any information pertaining to the stack frame of the interrupted agent (other than climbing up and down to see each level's entry-point). This is because the interpreter executes the `_agdb_debug_trap` call at the global level, which conceptually is the same as executing the call on a separate stack. Future releases of Agent Tcl/Tk will offer a way for `_agdb_debug_trap` to access the original stack.

7 Killing Agents

The user can kill an agent by pressing **kill** in either the agent's window or the manager window. These buttons are only effective between the time the agent starts running and the time the agent calls `agent_end`, except during jumps (see Section 9.1 for details on jumps). Before an agent calls `agent_begin`, the debugger kills it by executing a kill on the agent's process id (obtained when the debugger executes the agent in the background). For as long as an agent is registered, the debugger can employ `agent_force` to kill it. Unfortunately, after an agent calls `agent_end`, the agent is no longer registered, and the debugger has no way of killing it. Future releases of Agent Tcl will allow a more reliable kill command.

8 Naming and Forcing

The debugger uses an agent's `machine-name` and `numeric-id` to interrupt or kill the agent. Although the debugger does not use symbolic names to communicate with agents, other agents may use the symbolic names as parameters to `agent_send` and `agent_meet` procedures. For this reason, the user may find it useful to keep track of an agent's symbolic name. After each successful `agent_name` call, an agent notifies the debugger of its new name. If `agent_name` fails, the agent notifies the debugger of the failure. In both cases, the debugger notifies the user of the result. The most likely cause for failure in an `agent_name` call is the existence of another agent with the desired name on the same machine.

Users may wish to avoid naming conflicts by calling `agent_force` before each `agent_name` call, thereby killing an agent with the desired name if one exists one the local machine. AGDB offers no support for the `agent_force` procedure. Due to security issues, the specification of `agent_force` will change significantly in future releases of Agent Tcl/Tk.

9 Jumping to Remote Machines

Agents use the `agent_jump` procedure to move to remote machines. Although agents can migrate at will, AGDB must always keep as close a contact as possible with the agents it is debugging. Whenever an agent calls `agent_jump`, it initiates a pre-jump meeting to tell the debugger that there will be a brief loss of contact. After the jump is completed, the agent initiates a post-jump meeting to notify the debugger of its new identity `(machine-name, machine-ip, numeric-id, symbolic-name)` or of its failure to jump.

9.1 How AGDB Handles Jumping

Between the pre-jump and post-jump meetings, we consider the agent to be in a transitional state. The debugger can neither kill nor interrupt a transitional agent. Even if the debugger tried to contact a transitional agent on both the original and the target machines, there is still a moment during which the agent is unreachable. Therefore, the debugger takes no action if the user tries to interrupt or kill a transitional agent. If the user quits AGDB while an agent is in transition, the agent hangs indefinitely while trying to initiate a post-jump meeting with the debugger. Section 14.3 discusses several possible solutions to this problem.

9.2 Jump Options

AGDB offers the user five options concerning jumping: ignore, track, break before, break after, and break before and after. The **track** option causes the debugger to notify the user of all `agent_jump` calls, regardless of their outcome. The **track** option is the default for root agents. The **ignore** option causes the debugger to obtain but suppress (from the user) information about `agent_jump` calls. Although the user is not informed of the agent's jumping activity, the debugger must still keep track of the agent's location. The **break** options are an extension of the **track** options in that they cause the agent to initiate a breakpoint just before or after (or both) it makes the actual `agent_jump` call. The user is able to switch jump options before running an agent or during any breakpoint.

10 Agent Communication

Agents can communicate with one another via `agent_send`, `agent_receive`, and `agent_meet` calls. We find that most Agent Tcl/Tk programmers have not used `agent_meet` to implement communication between their agents. This choice may be due to `agent_meet`'s additional complexity. We do not yet support `agent_meet` in AGDB, but expect to do so in the future. For now, AGDB supports both `agent_send` and `agent_receive`.

10.1 Send and Receive Options

AGDB offers four separate options for `agent_send` and `agent_receive`: **complete**, **1K**, **128 bytes**, and **ignore**. These options allow the user to choose how much of each message the debugger should display in the agent's window. If the user chooses the **ignore** option, the agent does not notify the debugger of any `agent_send` or `agent_receive` calls. The user is able to switch send and receive options before running an agent or during any breakpoint.

10.2 Implementation Details of Sending and Receiving

Before each `agent_send` call, an agent tells the debugger the contents and destination of its message. After an `agent_send` call, the agent tells the debugger whether or not the call was successful.

Before each `agent_receive` call, an agent tells the debugger the options with which it will attempt to receive a message. After a successful `agent_receive` call, the agent tells the debugger the contents and the sender of the message it received. If the `agent_receive` call fails, the agent reports the failure to the debugger. In the event that an agent calls `agent_receive` with the `-blocking` option, the agent hangs indefinitely until another agent sends it a message. Unfortunately, the debugger is not able to interrupt the blocked agent because messages sent by the debugger are automatically trapped to a `_agdb_debug_trap` event, which is then queued. Section 14.2 describes two possible solutions to this problem.

11 Sourcing Files

For AGDB to debug all activities of an agent, it must be able to debug the code in files sourced by an agent. Hence, AGDB must preprocess sourced files as it does the main file. AGDB supports the sourcing of files, but it does so on a more limited basis than does Tcl.

11.1 Implementation Details of File Sourcing

AGDB is capable of debugging any files sourced between the `agent_begin` call and the first `agent_jump` attempt or the `agent_end` call. We call the state of execution between these calls *p-safe* because the debugger can preprocess any files the agent sources. For the debugger to carry out the preprocessing, the agent must have called `agent_begin`, so the agent can contact the debugger. Also, the agent cannot have jumped to a remote machine. If both processes are running on the same machine with the same user id, they have the same file-access permissions, and the debugger can preprocess the sourced file just as it did the main file.

When the agent is executing in a *p-safe* state, the source procedure is redefined to the `_agdb_source_wrapper` procedure. In the wrapper procedure, the agent sends the debugger the name of the file it wishes to source. The debugger reads the file, creates a new file in the `/tmp` in which it places the preprocessed version of the file, and sends the agent the name of the new file. The agent then sources the preprocessed version of the file.

When an agent is executing in a *p-unsafe* state, file sourcing occurs as it usually would in Tcl. Since the debugger does not preprocess the file, code in that file is effectively *debug-atomic* even though it may span many lines of code. Since `agent_submit` and `agent_fork` have implicit `agent_jump` calls in the child agent, non-root agents never execute in the *p-safe* state.

It is possible that the agent could preprocess all sourced code without the help of the debugger.

The agent could even send the debugger a string containing the code in the sourced file, so the code could be displayed for the user in the agent's debugger window. However, it seems that the decision to limit sourcing capabilities of the debugger to the *p-safe* state will not significantly diminish the debugger's usefulness. Once the agent has jumped, its user id becomes `agent-tcl`, and it may not to be able to access the same file system. Files the agent sources from a remote machine are likely to contain library procedures, images, sound sequences, etc., offered by the new host. The user is probably unfamiliar to the internal code of remote libraries, and we assume the user wants it to remain *debug-atomic*.

11.2 Source Options

AGDB offers the user three options concerning file sourcing: **ignore**, **track**, and **break**. The user is able to switch file-sourcing options before running an agent or during a breakpoint of a root agent. The default option for root agents, **track**, produces the behavior described in section 11.1. The **ignore** option causes an agent to leave all source code *debug-atomic*. Since non-root agents never run in a *p-safe* state, they have no `source` options.

The **break** option is an extension of the **track** option in that it causes a root agent to break just before it sources the preprocessed version of the file. The **break** option is useful because it allows the user to add line-specific breakpoints in code that is about to be sourced. Line-specific breakpoints remain during multiple sourcings of the same file in a given run of the agent. They also remain during multiple runs of an agent. Line-specific breakpoints are removed when the user explicitly removes the breakpoint in the agent window's breakpoint menu, or when the file is preprocessed again.

12 Forking and Submitting Agents

When an agent spawns a child, the debugger can either track the child or it can ignore the child. The user can control the debugger's initial treatment of a child via the parent agent's **default** menu. By default, the child agent inherits all option values, watch conditions and variables, and line-specific breakpoints from its parent. The child agent uses the parent's password to establish a meeting with the debugger. During this meeting, the child notifies the debugger of its new identity `(machine-name, machine-ip, numeric-id, symbolic-name)`. In exchange, the debugger gives the child a new password, so the child no longer uses its parent's password to communicate with the debugger.

12.1 Fork and Submit Options

AGDB offers the user three options concerning fork and submit: **ignore**, **track**, and **break**. The **ignore** option causes the child agent and the debugger to have no knowledge of or contact with each other. The **track** option causes the debugger to track the child agent as it would a root agent. The **track** option is the default. The **break** option is an extension of the **track** option in that it causes the agent to initiate a breakpoint just after spawning a child. If the fork or submit call was successful, the child will also break just after it receives its new password from the debugger. The **break** option is useful because it allows the user to change options in a newly spawned agent before it executes any code. The user is able to switch fork and submit options before running an agent or during any breakpoint.

12.2 Implementation Details of Fork and Submit

The semantics of forking are such that all variables and procedures available to a parent agent are copied to the child. On the other hand, the semantics of submitting are such that only variables and procedures specified in the arguments to the submit procedure are available to the child. Since the child needs the debugging environment (supplied by the *g-file* global variables and library procedures) to communicate with the debugger, we must add all variables and procedures therein to the argument list. We must also add code to the beginning of the `-script` argument to initiate the child's first meeting with the debugger.

If the user wants to ignore children of a particular parent, the parent must nullify preprocessing so the child does not try to contact the debugger. During preprocessing, the debugger only adds two different library procedure calls to the agent's code: `_agdb_end_comm` and `_agdb_dynamic_break`. The fork and submit wrappers temporarily redefine the two library procedures to be empty. In the submit wrapper, the two empty procedures are then added to the `-procs` section of the parent's submit argument list. After the actual submit or fork call is completed, the parent restores the `_agdb_end_comm` and `_agdb_dynamic_break` procedures to their original definitions.

12.3 Migration Issues

Migration-related procedures such as jump, fork, and submit, threaten contact with the debugger because they may result in an agent's relocation to a remote machine. During execution of migration procedures, the Agent Tcl/Tk interpreter redefines built-in Tcl/Tk procedures. Redefinition of procedures destroys the library-defined wrappers, so the agent must protect its wrappers. Also, migration procedures may invoke these commands (`agent_jump` does indeed call `source`), and expect that the functionality has not been altered by the agent. The agent avoids trouble with wrappers by temporarily removing its wrapper procedures `source`, `exit`, and `main` before it makes a migration call. Upon completion of the migration, the agent reinstates its `source`, `exit`, and `main` wrappers, except that children and agents that have jumped never reinstate their `source` wrappers, and ignored child agents never reinstate their `exit` and `main` wrappers.

Built-in Agent Tcl procedures (beginning with the prefix `agent_`) are neither redefined nor called during execution of migration procedures[2], so we do not need to protect the wrappers for built-in Agent Tcl procedures. However, the Agent Tcl/Tk interpreter redefines the value of the global variable `mask` during a successful migration. After each successful migration, the agent must add the mechanism that traps messages from the debugger to the `_agdb_debug_trap` procedure. Section 6 offers details about modifying `mask`.

13 Summary

AGDB combines traditional and distributed debugging facilities to aid programmers in debugging Agent Tcl agents. The debugger keeps the user informed of the agent's migration and communication by transforming the agent to include a library to communicate with the debugger. The library contains wrappers for the built-in Tcl/Tk and Agent Tcl procedures as well as procedures to carry out interrupts. Library procedures use Agent Tcl's built-in `agent_meet` procedure to communicate with the debugger. These communications can help the programmer get a better sense of the agent's behavior, and thus produce a more reliable agent.

[2] All agent_ procedures have _agent_ clones that are called internally.

14 Future Work

We can improve AGDB in several ways. For example, we can improve AGDB's error handling by catching each command of the user's code and interrupting the agent (rather than killing it) if an error is caught. To implement these ideas, we need support from the interpreter.

If at all possible, we would like to avoid preprocessing files. Rather than inserting procedure calls into the user's code, we would like to use a general-purpose command hook. By *general-purpose command hook*, we mean a procedure that the interpreter automatically calls before each command. Agent Tcl already supports command hooks. Command hooks could call `_agdb_dynamic_break`, but we still need Tcl/Tk support to supply the line and filename arguments to `_agdb_dynamic_break`. Future releases of Tcl/Tk will contain built-in procedures to keep track of line numbers with respect to code files.

Another way in which we can improve AGDB is by catching each command of the user's code and interrupting the agent (rather than killing it) if an error is caught in a block of code that is not already encapsulated by a catch (by the programmer's design). For this, we need support from the Agent Tcl interpreter.

14.1 Debugging Many Agents

Currently, AGDB can support an unlimited number of agents. However, the agents are monitored at too fine a grain for the debugger to be useful on a large scale. For a look at AGDB's graphical user interface, see Figure 2 in Appendix C. A new window pops up for each agent, and there is a practical limit to the number of windows a user can monitor, so we need to incorporate a graphical, global view of the interaction between agents on a large scale. A future release of AGDB will address this issue.

14.2 Handling Blocking Receives

The `agent_receive` call is truly atomic in that the debugger is unable to interrupt the agent while executing the receive call. We could solve this problem by modifying the `agent_receive` wrapper to impose a time limit on all blocking receives. We could also solve this problem by modifying the `agent_receive` wrapper to change the mask before each blocking `agent_receive` call to keep the agent from trapping the debugger's messages to the `_agdb_debug_trap` procedure. Another solution to this problem is to modify the interpreter to execute trapped messages immediately, rather than queuing them.

14.3 Quitting AGDB and Killing Running Agents

When AGDB quits, it attempts to kill all running agents. Transitional agents are left hanging, however, while waiting to meet with the debugger. Future releases of Agent Tcl will solve this problem by including a built-in `agent_status` procedure that the agent can use to check whether or not another agent, namely the debugger, is still running before it makes a blocking `agent_meet` call. Instead of waiting indefinitely, transitional agents will call `exit` if the debugger no longer exists. We could also resolve this issue by adding a `-time` option to the `agent_meet` command so that the transitional agent can exit if a meeting is not established within a given amount of time.

14.4 Quitting AGDB Gracefully

There are many ways for an agent to be killed. AGDB is no different from any another agent; it can be forced or killed by anyone with the proper authority. When the user quits AGDB via the manager window's file menu, AGDB removes all the files it created in the `/tmp` directory, saves transcripts if user asked for them, and kills running agents (when possible). If AGDB is killed in any other way, we would like it to catch the kill signal and exit just as gracefully as if the user had quit via the manager window's file menu. In C programs, it is easy to catch a kill signal. In Tcl, however, this task is not so simple.

Acknowledgments

Many thanks to Robert Gray for offering advice and interpreter support at various stages of AGDB development. Also thanks to Fred Henle, Vishesh Khemani, and others for help with the graphical user interface and advice on standard debugging features.

References

[Gra95a] Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[Gra95b] Robert S. Gray. *Agent Tcl: Alpha Release 1.1*, December 1995. Dartmouth College. Available by WWW at http://www.cs.dartmouth.edu/~rgray/documentation/doc.1.1.ps.gz.

[Gra95c] Robert S. Gray. Transportable agents. Technical Report PCS-TR95-261, Dept. of Computer Science, Dartmouth College, 1995. Thesis proposal.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the 1996 Tcl/Tk Workshop*, pages 9–23, July 1996.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, 1994.

[Wel95] Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall PTR, 1995.

A Sample P-File

Background infomation regarding the *p-file* is supplied in Section 2. The *p-file* is is a copy of the original file, except that it has calls to the AGDB library procedure `_agdb_dynamic_break` interspersed among the lines of code.

The `_agdb_dynamic_break` procedure takes two parameters: the first is the number of the following line in its original file, and the second is the index location of the file in the buffer menu of the agent's window (0 unless the file was sourced). The `_agdb_dynamic_break` procedure decides whether the agent should break. If so, the agent requests a meeting with the debugger, thereby initiating a breakpoint.

```
#!/usr/contrib/bin/agent

# this is an agent which jumps to three machines and
#   collects data at each site.

# define the get_data procedure
proc get_data {x y} {
  _agdb_dynamic_break 7 0
  set list "" ; set z [expr $x + $y]
  _agdb_dynamic_break 8 0
  set machines "muir.cs.dartmouth.edu \
                tuolomne.cs.dartmouth.edu \
                tioga.cs.dartmouth.edu"
  # jump to each machine
  _agdb_dynamic_break 12 0
  foreach m $machines {
    _agdb_dynamic_break 13 0
    if {[catch "agent_jump $m"]} {
      _agdb_dynamic_break 14 0
      append list "\n$m:  \t unable to JUMP to this machine"
      _agdb_dynamic_break 15 0
    } else {
      _agdb_dynamic_break 16 0
      append list "\n$m:  \t[exec last]"
      _agdb_dynamic_break 17 0
    }
    _agdb_dynamic_break 18 0
  }
  _agdb_dynamic_break 19 0
  return $list
```

```
  _agdb_dynamic_break 20 0
}

_agdb_dynamic_break 22 0
agent_begin

_agdb_dynamic_break 24 0
set answer [get_data 44 55]

_agdb_dynamic_break 26 0
agent_end

_agdb_dynamic_break 28 0
return $answer
```

B Structure of the G-File

Background information regarding the *g-file* is supplied in Section 3. This *g-file* corresponds to the *p-file* in Appendix A.

```
#!/usr/contrib/bin/agent-tk

# source the library functions
source /usr/bin/agdb_lib.tcl

# password information initialization
set _agdb_agent_num 1
set _agdb_password  336699

# debugger identity initialization
set _agdb_debugger_machine muir.cs.dartmouth.edu
set _agdb_debugger_id      22

# initialize the state of the agent
set _agdb_never_registered 1
set _agdb_never_jumped     1
set _agdb_stepwise_break   0
set _agdb_nextwise_break   0
set _agdb_nextwise_level   0
set _agdb_line_number(0)  -1
set _agdb_buffer_index(0)  0
set _agdb_level            0

# breakpoint information initialization
set _agdb_num_watches           4
set _agdb_watch_cond_list       {{$x > 5} {$y $< 3}}
set _agdb_watch_var_list        {i j}
set _agdb_general_break_list    {37 54}
set _agdb_temp_break_list       {54}
set _agdb_break_condition(37.0) {$q == 6}

# initialize default option settings
set _agdb_track_jump    1
set _agdb_track_submit  1
set _agdb_track_source  1
set _agdb_track_fork    1
```

```
set _agdb_track_send  128
set _agdb_track_recv  128

# catch the sourcing of the p-file
set _agdb_catch_result [catch {source agdb_lib.tcl} _agdb_src_result]

# Tcl agents make a final communication with the debugger
# Tk also do iff an error is caught
_agdb_end_comm $_agdb_catch_result $_agdb_src_result

# Tcl agents end by returning the source result
if {$agent(language) != "STATE-TK"} {
  return -code $_agdb_catch_result $_agdb_src_result
}
```

C Graphical User Interface

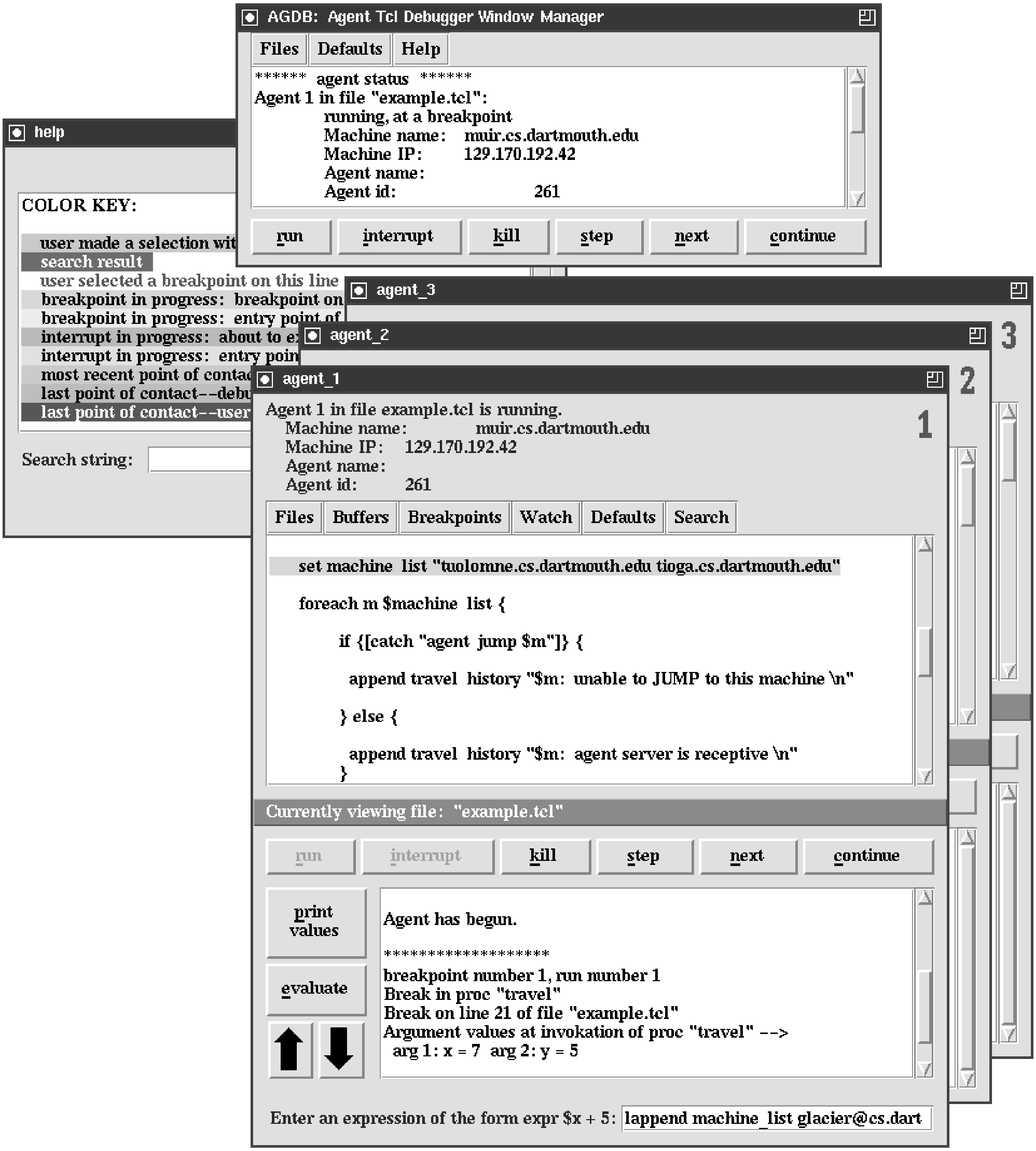

Figure 2: Graphical User Interface for AGDB.