Collecting and Analyzing Data from Distributed Control Programs

David Kortenkamp and Tod Milam

Metrica Inc./TRACLabs
1012 Hercules
Houston TX USA 77058

Reid Simmons and Joaquin Lopez Fernandez

School of Computer Science
Carnegie Mellon University
Pittsburgh PA USA 15213

Abstract

This paper describes a set of tools that allows a developer to instrument a C/C++ program to log data at run-time and then analyze that data to verify correct behavior. The logging tools provide the developer with a means to log a variety of different data to a variety of different outputs. They also allow for synchronized logging of data from distributed programs. One logging output option is an SQL database. We have developed a set of analysis tools that retrieve data from the database to answer common developer questions. The analysis tools use an interval temporal logic to frame database queries. The data logging tools are fully implemented and performance results are given in this paper. The data analysis tools are currently being tested on data from real NASA applications.

1 Introduction

Debugging and verifying distributed control programs is notoriously difficult, yet such control programs are becoming more and more common in complicated applications. Examples include the Remote Agent control architecture [9], the 3T control architecture [3] and the TCA control architecture [13]. In each of these instances, concurrent programs run (often on separate machines) to generate control commands for single or multiple devices.

[1] Email: korten@traclabs.com
[2] Email: reids+@cs.cmu.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

When an error occurs, it can often be very difficult to isolate the problem to one specific control module, due to timing constraints, interprocess communications, and synchronization. The traditional dynamic method for debugging sequential software has no timing constraints. For these systems, *cyclic debugging* (running the program until an error shows up, examining the program state, inserting assertions and re-executing the program to obtain additional information) is commonly used [15]. However, there are several reasons why this approach cannot be applied to distributed control programs:

- Often the distributed processes cannot be paused for examination since they are controlling physical hardware.
- There is no central, global state or even global clock to reference state values, which makes it difficult to reason about the "state" of the system at a given time.
- Due to latencies and timing issues, distributed control programs are inherently non-deterministic and non-repeatable.

In this paper we present a set of tools that allow programmers to instrument their control code and to time-stamp and collect real-time data into a common database. Then, a companion set of tools can be used to analyze the data to find the occurrence, or absence, of temporal patterns in the output. First, we describe the data collection tools, then we describe the related data analysis tools. An integrated example is used throughout. The example looks at verifying the distributed control program for a NASA life support system.

1.1 Related work

Reid Simmons and his group at Carnegie Mellon University have developed several data display tools for distributed control programs, including *comview* for viewing message traffic, *tview* for viewing the hierarchical decomposition of tasks, and *planview* for displaying and analyzing plan execution information. These tools have already been integrated with both the TCA architecture [12] and the NASA Ames Remote Agent architecture. An overview of this work can be found in [14].

Work on visualizing real-time programs has resulted in a product from Real Time Innovations Inc. called *Stethoscope* [11]. Stethoscope allows for data collection, display and modification. However, it is limited to real-time programs running under VxWorks and does not offer support for the kind of high-level, cross-system debugging that autonomous systems require.

There are some recently developed tools for debugging and verifying parallel systems that are related to our research. For example, ParaGraph [4] provides a variety of visualizations of different aspects of a parallel system. Another such tool is PIE [7], which operates with the Mach operating system. A number of tools have been developed for debugging and verifying multi-threaded programs, including tnfview [6]. However, none of these tools can offer the cross-system and high-level debugging and verification support needed by autonomous systems. For more information on parallel and distributed programming visualization tools see [1,16].

2 Data collection

The data collection demands of distributed control programs range from low-level sensory data to the program's internal state. The data collection routines have the following requirements:

- Data collection in real time
- Data logging to a database
- Flexible sampling rates
- Grouping of data into logical sets
- Triggering options (e.g., allowing only data in certain ranges to be collected or only when it has changed)

The data collection tools we have developed are geared towards the C/C++ programming language, although through foreign function calls other programming languages (such as Lisp and Java) can access them. Our goal was to replicate the ease-of-use of the `printf` command in C, while allowing for more control and for distributed operation. In essence, what we have implemented is a *remote printf* capability called *rlog*.

Rlog is implemented as a set of libraries that allow you to instrument your program and send the output to a variety of different places, e.g., the screen, a file, a remote computer or a database. The types of data that can be logged are: character, unsigned character, short integer, unsigned short integer, integer, long integer, unsigned long integer, floating point number, double floating point number and character string.

We support logging on the following platforms: Linux, Solaris, IRIX, and NetBSD. We are currently working on a VxWorks port. As much as possible, the code avoids operating system dependent calls to allow for easy porting to new platforms.

2.1 Rlog functions

The data collection capabilities of our system are contained in a library that the client program compiles into its code. This library contains a wide variety of different logging functions. This section describes each logging function and what it does.

2.1.1 Initialization

The client must call `rlogInitType` for each output type they want to be active. For example the call `rlogInitType(argc, argv, "debug1")` will initialize the output associated with debug1. Optional plugin-specific options can follow the output type, but these are discouraged in favor of command-line arguments or specifying the arguments in the configuration file.

2.1.2 Cleanup

Prior to a client program exiting, or whenever it is finished using an output type, it should call `rlogCleanupType(type)` for each output type, or just `rlogCleanup()` to cleanup all initialized types. This ensures any open files or ports get closed and any allocated memory is freed.

2.1.3 General logging

The following functions provide general logging capabilities:

- `rlogEvent`: This call logs a list of variables once, similar to printf. It takes an arbitrary number of parameters so any number of variables can be logged at once. The first parameter specifies an event name to associate the variables with. This can be any character string. The second parameter is a format string indicating the data type and name of each of the parameters following it. Each variable to be logged will have an entry in the form of "type[:name]" where the type is the data type and the optional name is the name of the data (not necessarily the same as the variable name). Each entry is separated in the string by a space.
- `rlog`: This is a simplified version of `rlogEvent`, which sets the event name to a default value.
- `rlogEnableEvent`: This function enables logging for the specified event. When `rlogEvent` is called the event will be logged.
- `rlogDisableEvent`: This function disables logging of an event. Any call to `rlogEvent` will be ignored for the specified event.
- `rlogOutputFormat`: This function controls how the data is displayed by the various text output plugins (see Section 2.2).
- `rlogEventPrintf`: In this function the output format is determined by the format string, much like the `printf` function, instead of the current output format set by `rlogOutputFormat`. It has an equivalent `rlogPrintf` which uses the default event name as well.

2.1.4 Change-only logging

There are many instances when the developer only wants to log a value when it changes – for example the internal state variables. We have implemented the following functions to log change-only data:

- `rlogRegisterVariable`: This function adds the variable to a list that is checked whenever the "flush" function (see next item) is called.
- `rlogFlushChanges`: Logs all registered variables whose values have changed since the last call. All flushed variables will have the same time stamp.

- `rlogUnregisterVariable`: Removes a variable from the list so that it is no longer logged when the flush function is called.
- `rlogUnregisterAllVariables`: Unregisters all registered variables.

2.1.5 Conditional logging

There are instances when the developer will only want to log a value under certain conditions. While they can do this by embedding an rlog call inside of an if-then, we have provided functions that perform this for them. These include:

- `rlogRegisterEventCondition`: It is in this call that you define the logging condition. For example, that a value be less than a certain number. This call returns an ID that is used in the rest of the conditional logging functions.
- `rlogSetVariableCondition`: This function lets you apply the condition specified in the previous function to a specific variable. Now when any logging call is made on that variable the condition is first checked before logging happens.
- `rlogUnSetVariableCondition`: The condition is no longer checked before logging the variable.
- `rlogUnRegisterEventCondition`: When a condition is no longer needed this function removes it from being checked.

There are some other more specialized functions that are also available. See the rlog WWW site (http://www.traclabs.com/rlog/) for details.

2.1.6 Function entry and exit

An important part of debugging distributed programs is knowing whether and when functions have been called and when they have finished executing. In addition to the functions listed below, we have developed scripts that will read a C/C++ file and automatically add function entry and exit logging commands to each function in that file.

- `rlogLogFunctionEntry`: Logs the entry into a function. Place this (or have the script automatically place it) at the beginning of a function.
- `rlogLogFunctionExit`: Logs the exit from a function.
- `rlogEnableFunctionEntryLogging`: Start logging function entries.
- `rlogDisableFunctionEntryLogging`: Stop logging function entries.
- `rlogEnableFunctionExitLogging`: Start logging function exits.
- `rlogDisableFunctionExitLogging`: Stop logging function exits.

2.2 Output plug-in modules

RLog uses GNU Libtool to portably use dynamically loadable modules for the output plugins. Dynamically loadable modules are similar to shared li-

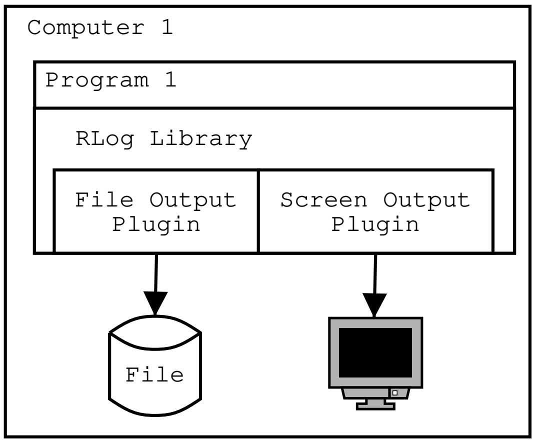

Fig. 1. The developer's program (Program1) is compiled with the RLog libraries. Plug-ins can be loaded at run-time and direct the output to various places. Multiple output destinations can be used. A Null destination turns off logging.

braries. They allow users to change library functions without recompiling their program. Figure 1 shows a typical configuration.

There are four types of plugins available for use with the RLog library, and details for implementing custom plugins are available on the RLog web site. The four types of plug-ins currently available are:

- Text plugins include the screen and file plugins. They output the logged data in text format using the format specified by the `rlogOutputFormat` call, or the default format if `rlogOutputFormat` hasn't been called.
- Database plugins include two MySQL plugins. Each of them use a different schema for storing the data in the database. The database may reside either on the local machine or a remote machine by specifying a different host name. More details of the database are given in Section 2.4.
- Socket plugins are used to send the logged data to the RLog server on a remote machine. The TCP plugin uses raw sockets for sending the data. The IPC plugin uses the IPC library and server available from CMU to send the data. The RLog server handles time stamping the remote data as described in Section 2.5.
- The NULL plugin is used to disable logging. This allows the logging code to remain in the client in case future debugging is needed.

2.3 Configuration file

Rlog uses a configuration file to determine what output modules are to be used. This allows the developer to change output locations without recompiling any code. The configuration file is called rlogDestinations.txt and is located in the current working directory, or the full path may be specified by the environment variable RLOG_DESTINATION_FILE. Each record in this file consists of two fields and an optional third field. Comments in the file begin with a semicolon and continue until the end of the line. The first field is a character string containing any characters except white space (not including a semicolon). This field is the label which will be used in the client code to identify what output plugin module to send logged data to. A generic name, such as "debug1" or "app1dest1" is preferred over a name describing the actual output plugin module like "fileOutput". This will allow changing what output the label is attached to without recompiling code (or having "fileOutput" actually send data to the TCP output).

The second field specifies the full path to the output plugin module including the module name but without the extension. For example "/tmp/rlogFile" might specify the file module.

The third field is optional. It specifies the command line arguments for the module. For example the file module can take the "-f" flag to specify the output file so `-f /tmp/rlogOutput.txt` would specify logging the data to the file "/tmp/rlogOutput.txt".

2.4 Database

One option for storing collected data is an SQL relational database. This gives the user access to powerful search and retrieval capabilities. For this project we have chosen the MySQL database (http://www.mysql.com). We chose this database because it is freely available and runs on the platforms we use.

Once the database was chosen we needed to design a schema for storing our data. A schema is a set of database tables with related data. We have implemented two different schemas; they both have their pros and cons. Performance testing shows that they perform similarly (see Section 2.6).

2.4.1 Schema 1

The first schema consists of two tables. The first is called CommonData which assigns a unique Id for each entry and stores the event name and timestamp for each logging call. The second table is the ItemData table which stores the information for each variable logged. All of the information in the second table is stored as strings (i.e., all types of variables are converted to strings). These records will be tied to the CommonData table entry using the Id generated in that table.

CommonData Table Layout

- Id - an integer field set up as the primary key and auto incremented.
- MachineName - internally generated by rlog, the name of the computer where the program is running that called the log function.
- TimeStamp - a date-time field holding the date/time that the log function was called. Internally generated by rlog.

- Microseconds - The microsecond part of the timestamp, needed since the date-time data type does not go to this precision.
- EventName - The name of the event being logged. In the case of `rlog` a default EventName is used.

ItemData Table Layout

- Id - an integer field which corresponds to the Id field in CommonData. The plugin must match this up since MySQL does not support foreign keys.
- ItemType - a character field of length 50, this is a text description of the data type of this variable.
- ItemName - a character field of length 100, this is the name of the variable as specified in the format statement. If no name was specified then a default name will be generated for internal use.
- ItemValue - a character field of length 100, this is the value of the variable converted into a character string.
- ItemOrder - an unsigned integer field specifying the order in the argument list that this variable appears.

2.4.2 Schema2

The second schema consists of 12 tables: one for the logged event data and one each for the different data types that RLog supports. The EventData table is identical to the CommonData table of the first schema. It assigns a unique Id for each entry and stores the event name and timestamp for each logging call. The individual data type tables will contain the variable information. These records will be tied to the EventData table entry using the Id generated in that table.

EventData Table Layout

- Id - an integer field set up as the primary key and auto incremented.
- MachineName - internally generated by rlog, the name of the computer where the program is running that called the log function.
- TimeStamp - a datetime field holding the date/time that the log function was called. Internally generated by rlog.
- Microseconds - The microsecond part of the timestamp, needed since the datetime data type does not go to this precision.
- EventName - The name of the event being logged. In the case of `rlog` a default EventName is used.

The table layout for each of the eleven logged data types looks like the following:

- Id - an integer field which corresponds to the Id field in EventData. The plugin must match this up since MySQL does not support foreign keys.

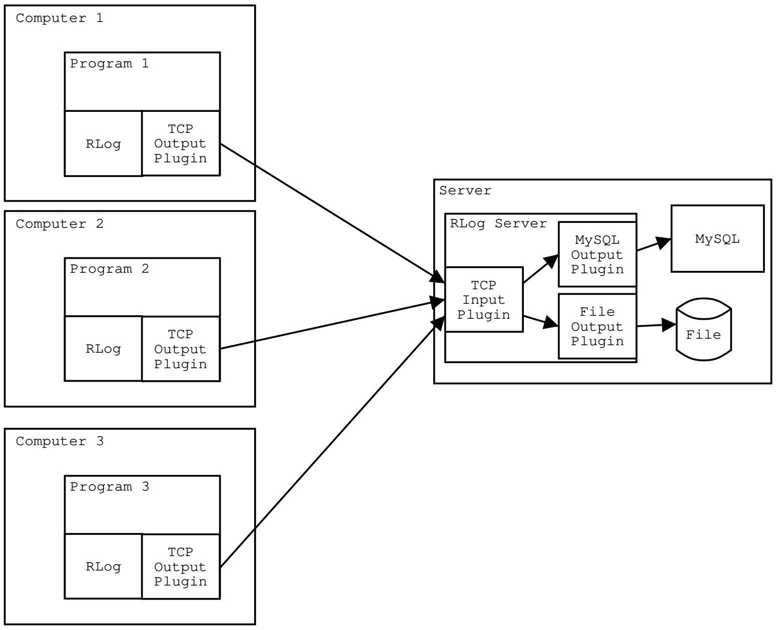

Fig. 2. Programs running on different computers can log data via TCP/IP to a central (server) computer. The data logged on the server can be directed to several destinations, including a MySQL database. The server also determines timing offsets amongst the computers and adjusts time-stamps appropriately.

- ItemName - a variable length character field of max length 100, this is the name of the variable as specified in the format statement. If no name was specified then a default name will be generated for internal use.
- ItemValue - a field of whatever data type this table stores, this is the value of the variable.
- ItemOrder - an unsigned integer field specifying the order in the argument list that this variable appears.

In addition to defining the schemas, we have written C/C++ code to insert data into the database and to extract data from the database. In this way, the user of our logging tools does not need to know SQL or anything about the internals of the relational database. The tools insert and extract all of the data for the user.

2.5 Distributed logging

By using the socket output modules (see Section 2.2), data from different programs running on different machines can be logged to a central location. The computer at the central location must be running an rlogServer program to collect the data. Figure 2 shows an instance of distributed logging.

When collecting data generated by different processes on distributed machines there needs to be some way to time stamp the data using a common clock. The rlogServer is the process that collects all of the data on a host computer and sends it to the database (or other location). When the rlogServer receives a data message from a different (remote) computer it starts a new thread that sends a request to that remote computer for the time offset. It then applies that offset to the received message and all subsequent messages from that remote computer. It polls the remote computer every 2 minutes to update the time offset. If no response is given by the remote computer then the time stamp is unchanged. The remote machine must be running a *rlogTimeServer* that was written by us to determine the offset. This server takes minimal CPU time since it is called very infrequently.

Time offsets are calculated based on the formula published in RFC 2030 [8]. The following is the relevant part of RFC 2030 for our purposes:

"To calculate the roundtrip delay d and local clock offset t relative to the server, the client sets the transmit timestamp in the request to the time of day according to the client clock in NTP timestamp format. The server copies this field to the originate timestamp in the reply and sets the receive timestamp and transmit timestamp to the time of day according to the server clock in NTP timestamp format.

"When the server reply is received, the client determines a Destination Timestamp variable as the time of arrival according to its clock in NTP timestamp format. The following table summarizes the four timestamps:

Timestamp Name	ID	When Generated
Originate Timestamp	T1	time request sent by client
Receive Timestamp	T2	time request received by server
Transmit Timestamp	T3	time reply sent by server
Destination Timestamp	T4	time reply received by client

Then the roundtrip delay d and local clock offset t are defined by:

$$d = (T4 - T1) - (T2 - T3)$$

and

$$t = \frac{((T2 - T1) + (T3 - T4))}{2}$$

RFC 2030 claims accuracy to "within a few tens of milliseconds." Of course this is true only at the point in time that this message is sent and received. Therefore the RLog server will periodically poll the client for a new time offset (currently once every 2 minutes).

2.6 Rlog performance

We have run some performance measures of the rlog libraries for the different outputs. The platform used for these tests was the following:

- CPU: Intel Pentium III @ 800Mhz

Null	0.009
File	0.053
Screen	0.534
TCP	0.711
IPC	0.750
MySQL1	0.347
MySQL2	0.382

Table 1
The number of seconds it takes to make 100 calls of rlog for the different output possibilities. This includes initialization and cleanup.

- Memory: 256 Meg.
- OS: RedHat Linux 6.2
- Model: Dell Dimension XPS B800r desktop computer

Table 1 shows the number of seconds it takes to call the rlog function 100 times for the different outputs possibilities. These numbers are an average of 10 sets of 100 calls for all datatypes and the initialization and cleanup functions required of rlog using the platform specified above.

3 Data analysis

Distributed control programs tend to produce massive amounts of output data that, due to non-deterministic latencies and execution times, are often not reproducible exactly from run to run. Rather than looking for exact matches with known (successful) runs, programmers often are interested in detecting the occurrence (or absence) of particular temporal patterns in the data. These patterns are often repetitive (e.g., "whenever the water level is above a given threshold, the pump should be turned on within 30 seconds") and may themselves be composed of other patterns (e.g., turning on the pump may be composed of a sequence of more primitive events).

To handle this type of data analysis, we have designed a temporal logic geared towards such patterns, and have implemented an initial version of a tool that checks collections of patterns against a database of logged output. The logic, called ITCL (Interval Temporal Checking Logic) combines features of RTIL[10] and RTL [5]. We started with RTIL and RTL because their goal was also to determine if the execution of a real-time (not distributed) program is consistent with a formal description of the program behavior. All the other logics we looked at were designed for "model checking" and they restrict their language to be able to apply verification methods. The main reason why existing logics were not sufficient for our needs was based on our need to relate interval sets in different ways instead of relating two intervals. Also, we need to work with both: interval sets and event sets.

The basic primitives in ITCL are *events* (ϕ), which correspond to entries in the log database, and *intervals*(γ), which are defined in terms of a pair of time points. A time point, in turn, can be the time at which an event occurs, or some point in time relative to an event (e.g., 3 seconds after event `e1` occurs, written as `e1` $\rightarrow$3). The logic enables sets of events (Φ) and intervals (Γ) to be defined, which can then be unioned, differenced, and iterated over (using a "forall" statement in the logic).

One can also define intervals based on conditions (e.g., the intervals of time during which a condition **P** holds, written as `[P]`), and by using "search" operators. A *search operator* ($\Rightarrow$ or $\Leftarrow$) extends an interval from a starting event until the next occurrence of an ending event (or searches from an ending event back to a starting event). For instance, the interval `e1` $\Rightarrow$ `e2` is the period between when the event `e1` occurs and the next occurrence of event `e2`. Similarly, `e1` $\Leftarrow$ `e2` is the period that *ends* when `e2` occurs and *begins* with the previous occurrence of event `e1`. Search operators are most useful in creating interval sets. For instance, `[e1` $\Rightarrow$ `e2]` is the set of all intervals in the log database where event `e1` is followed by an event `e2`. Note that the intervals formed in this way may overlap. For instance, if the database has the events `e1, e1, e2`, then there would be two intervals in the set, each starting with a different instance of event `e1`, but ending at the same event `e2`. In contrast, `[ e1` $\Leftarrow$ `e2]` would have only one interval in the set (extending from the second `e1` event up to the `e2` event).

In addition to the standard Boolean operators, our logic also includes functions for accessing the beginning and ending events of an interval, the time that an event occurs, several Boolean functions for determining the temporal relationship between events and between intervals, and ways of restricting an expression to be evaluated during some subset of an interval (such as at the start, end, or throughout the interval). Table 2 sumarizes some of these ITCL operators, with examples shown in Figure 3. Also, Table 3 shows some macros to express temporal relations between intervals and events.

Operations between interval sets like union, substraction, conjunction and disjunction have been defined in order to make it easy express situations like: *"Action a3 must start after interval a1 and a2 and..."* (union) or *"The system must be working while it is connected except when there is no water in the wicktank or the condensate tank is full"* (substraction), etc.

Conditions are evaluated in intervals to see if they become true eventually ($\diamond$), always ($\square$), before the interval ($\triangleleft$), at the beginning of the interval ($\triangle$), at the end of the interval ($\bigtriangledown$) or after the interval ($\triangleright$).

[3] The result of this expression is an interval set with only one interval (γ4.5). ϕ9 means any event of the Φ9 event set.

Symbol	Meaning	Examples Figure 3
$\Rightarrow$	Search forward for next event	$\Gamma 1 \equiv \Phi 1 \Rightarrow \Phi 2$
$\Leftarrow$	Search backward for next event	$\Gamma 2 \equiv \Phi 1 \Leftarrow \Phi 2$
$\rightarrow$	New event (set) as time after event (set)	$\Phi 6 \equiv \Phi 4 \rightarrow 6$
$\leftarrow$	New event (set) as time before event (set)	$\Phi 5 \equiv 8 \leftarrow \Phi 4$
$\cup$	Union of two interval sets	$\Gamma 4 \equiv \Gamma 1 \cup \Gamma 2$
$\cap$	Substraction of two interval sets	$\Gamma 5 \equiv \Gamma 1 \cap \Gamma 2$
$\uparrow$	Event(s) starting an interval (set)	$\Phi 1 \equiv \uparrow \Gamma 1$
$\downarrow$	Event(s) ending an interval (set)	$\Phi 2 \equiv \downarrow \Gamma 1$
$\triangle$	First subinterval (set) of an interval (set)	$\gamma 9 \equiv \gamma 4.5 \triangle$
$\bigtriangledown$	Last subinterval (set) of an interval (set)	$\gamma 10 \equiv \gamma 4.5 \bigtriangledown$
$\triangleright$	Interval (set) after	$\gamma 11 \equiv \gamma 4.5 \triangleright$
$\triangleleft$	Interval (set) before	$\gamma 8 \equiv \gamma 4.5 \triangleleft$
$\perp$	Null interval set	$\perp \equiv \Phi 4 \Rightarrow \Phi 2$
/	Restricts interval set with conditions	$\{x{:}\Gamma 4/x \text{ include } \phi 9\}$ [3]
$time(\phi)$	Time when event ϕ occurs	$time(\phi 1.2)$ is 24
$\mid \Phi/\Gamma \mid$	Items in event set Φ (interval set Γ)	$\mid \Phi 2 \mid$ is 3

Table 2
Event and interval operators.

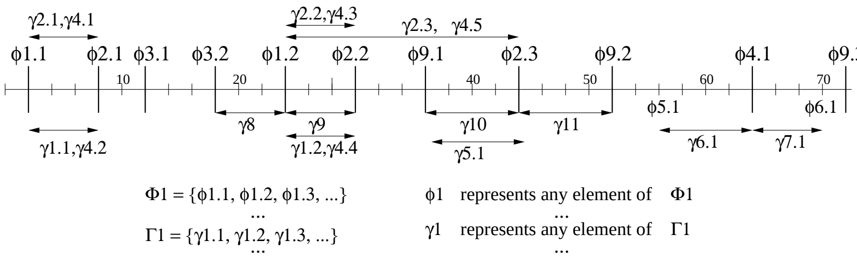

Fig. 3. Events ϕ are grouped into event sets Φ and intervals γ into interval sets Γ .

We have developed an initial implementation of a tool that interprets ITCL expressions and checks their validity with respect to the logging output collected in a file or database. The tool works by constructing interval and event sets and performing operations on them. Whenever a formula evaluates to "false", the tool constructs a counterexample showing how some particular combination of events leads to the error. The idea is that this information will help the user to pinpoint bugs in the distributed system.

MACRO	EQUIVALENCE
$\gamma 1$ intersects $\gamma 2$	time($\uparrow \gamma 1$) $<$ time($\downarrow \gamma 2$) $\wedge$ time($\downarrow \gamma 1$) $>$ time($\uparrow \gamma 2$)
$\gamma 1$ include ϕ	time($\uparrow \gamma 1$) $\leq$ time(ϕ) $\wedge$ time($\downarrow \gamma 1$) $>$ time(ϕ)
$\gamma 1$ include $\gamma 2$	time($\uparrow \gamma 1$) $\leq$ time($\uparrow \gamma 2$) $\wedge$ time($\downarrow \gamma 1$) $\geq$ time($\downarrow \gamma 2$)
$\phi 1$ isbefore[t1,t2] $\phi 2$	time($\phi 1$) + t1 $\leq$ time($\phi 2$) $\wedge$ time($\phi 1$) + t2 $\geq$time($\phi 2$)
$\phi 1$ isbefore(t1,t2] $\phi 2$	time($\phi 1$) + t1 $<$ time($\phi 2$) $\wedge$ time($\phi 1$) + t2 $\geq$time($\phi 2$)
$\phi 1$ isbefore[t1,t2) $\phi 2$	time($\phi 1$) + t1 $\leq$ time($\phi 2$) $\wedge$ time($\phi 1$) + t2 $>$ time($\phi 2$)
$\phi 1$ isbefore(t1,t2) $\phi 2$	time($\phi 1$) + t1 $<$ time($\phi 2$) $\wedge$ time($\phi 1$) + t2 $>$ time($\phi 2$)

Table 3
Some operators for temporal relations between intervals and events.

At this stage, the tool is working over the full ITCL language (this paper has described only a subset of the language). While it is somewhat inefficient, especially when dealing with nested “forall” statements, it is sufficient for us to evaluate its utility. We are currently working to develop more patterns for the Water Recovery System (see next section), and will be testing the data analysis tool against real data in the near future. We are also evaluating the usability of the language. To this end, we are adding higher-level constructs (“syntactic sugar”) to make it easier to specify common types of rules (such as “always do X, T seconds after Y”) and are developing a graphical user interface that will enable users to specify ITCL formula and view any counterexamples found.

4 Data visualization

In addition to data analysis we provide some simple tools to visualize the data in the database. These tools are implemented in Java and extract information from the database and display it. There are three types of displays: 1) raw data; 2) plotting of values against time; and 3) plotting of two values against each other. When you start the data viewer it communicates with the database and provides you with a list of items that can be viewed (see Figure 4). These items can be displayed across time (Figure 5) or two items can be displayed with respect to each other (Figure 6). This is still a very preliminary data viewer and the emphasis in our project has not been on visualization. We hope others will contribute to this aspect of the project.

5 An example

To illustrate the integration of our data collection and analysis tools we use an example for which we have real data. The example is the control program

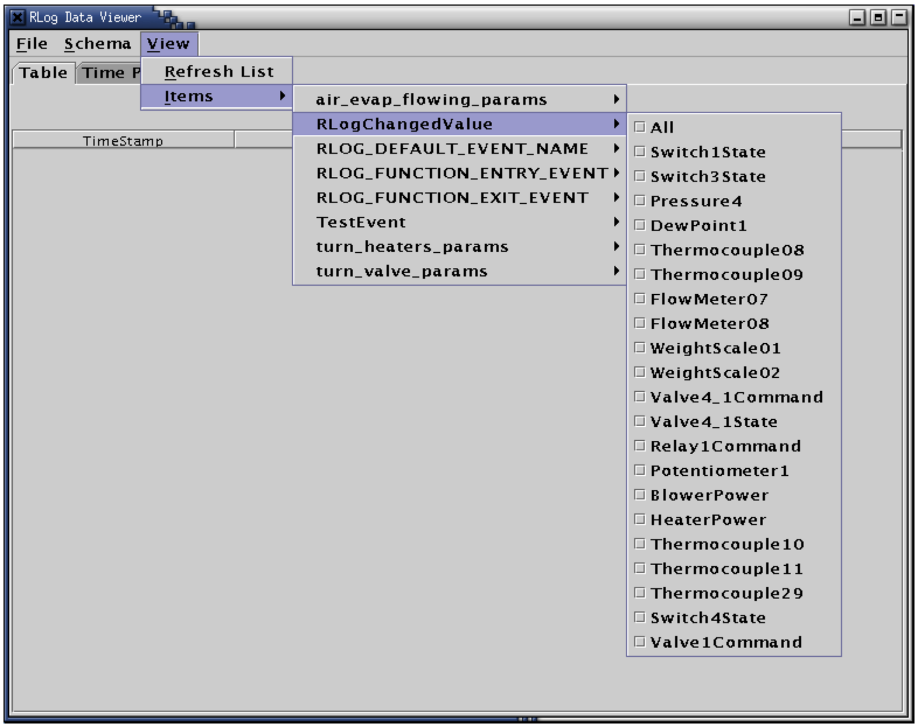

Fig. 4. The data viewer allows the developer to choose the items they wish to display.

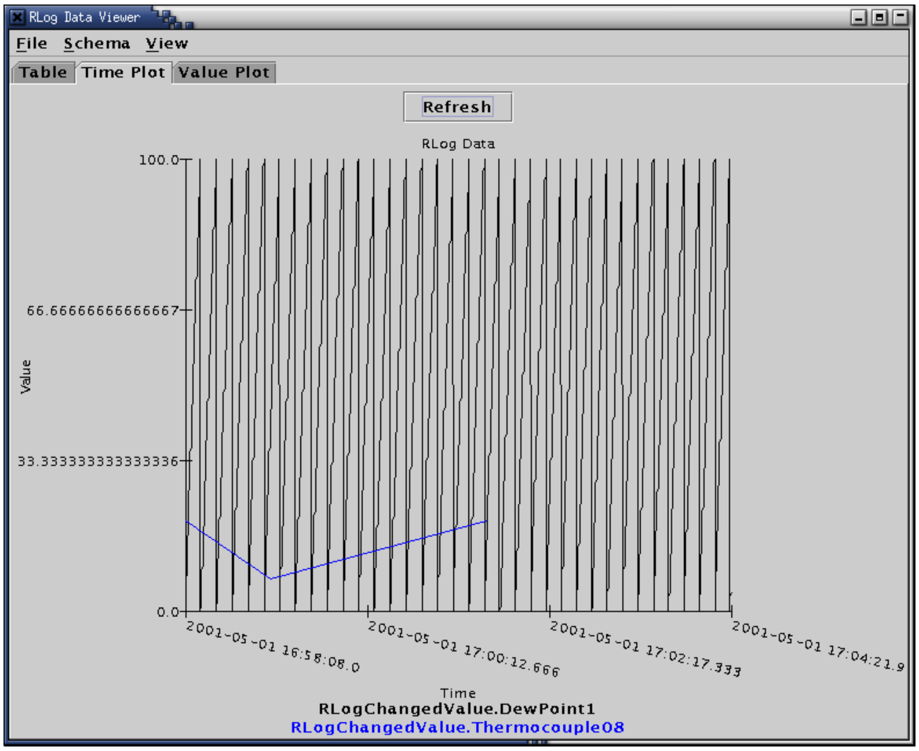

Fig. 5. Multiple items can be displayed across time.

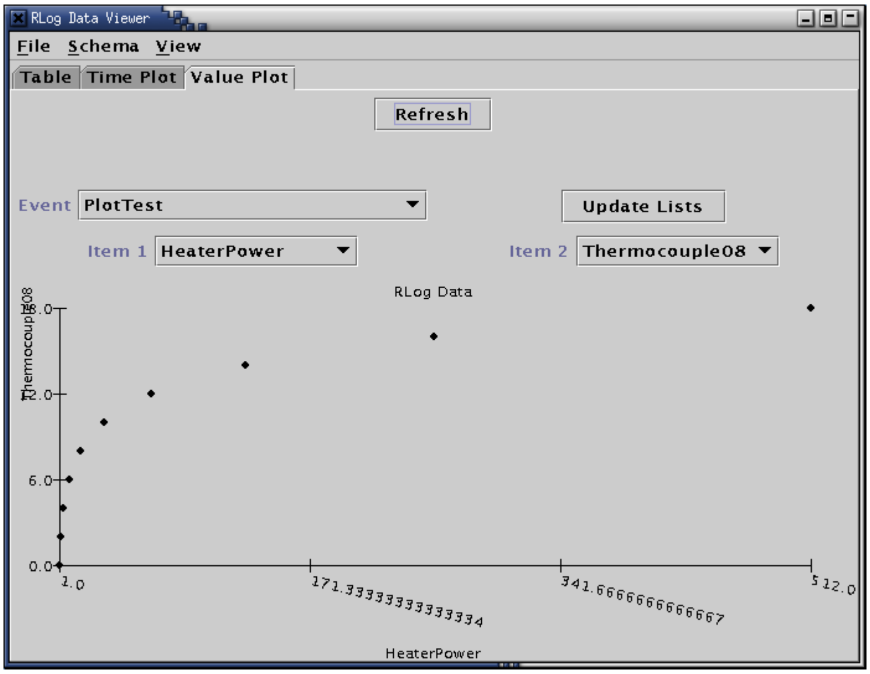

Fig. 6. Multiple items can be displayed across time.

for an advanced life support system at NASA Johnson Space Center [2]. This life support system, the Water Recovery System (WRS), has over 70 sensors and as many actuators. There are four subsystems that are controlled by four independent controllers implemented in C. These controllers themselves receive commands from a LISP-based reactive planner. We instrumented one of the C-based controllers for a single subsystem and logged its data. We are in the process of logging the other three C-based controllers. Also, all logging described in this paper was done using a detailed simulation of the actual hardware. We are currently installing the logging on the actual running system.

We instrumented a single subsystem so that all sensor readings were logged on a "change-only" basis (meaning that the sensor was logged only when its value changed) and all actuator commands were logged also on a change-only basis. An example of a logging command is:

```
rlogRegisterVariable("HeaterPower",RLOG_INT,&heater_power)
```

which means that the variable `heater_power`, which is an integer, is to be logged when it changes and the identifier for this variable is "HeaterPower". This variable is connected to a sensor that measures the power consumption of the heater.

In addition, we logged the entries and exits to control functions called by the LISP-based reactive planner. At the same time we logged the parameters passed to these functions. These functions are used by the LISP-based reactive planner to change control set-points, to query for information and to shutdown or start various subsystem components. An example of a function entry logging command is:

```
rlogLogFunctionEntry("turn_valve_enable");
```

Followed by a logging of the parameters to the function:

```
rlogEvent("turn_valve_params", "string:valve string:value",
          params->valve, params->value);
```

Where the valve is a string and the value is a string (e.g., "on" or "off"). These last two rlog calls will create an entry in the database (with a timestamp) when the `turn_valve` function is called and also a separate entry for the parameters to that function.

For data analysis purposes, one would like to define rules for the Water Recovery System that must be adhered to in any correct implementation of the control program. For instance, one would like to say that the watt meter must report some power draw when the wick tank level is over L or when the overflow tank is over level M and the manual pump is on. This can be written in ITCL as:

```
wick_over_L = [wick_tank.level > L];
overflow_&_pump = [(overflow_tank.level > M) ∧
            (manual_pump.stat ≡ 'on')];
power_draw = wick_over_L ∩ overflow_&_pump;
∀ it2_1:  power_draw {
     it2_1  □ (power_report.pw03 > 0)
};
```

The first formula defines an interval set during which the wick tank level is over `L`. The second one defines an interval set where the overflow tank is over level `M` and the manual pump state is `'on'`. The third statement defines an interval set as the union of the first two. Finally, the last formula says that in each such interval the watt meter must always be reporting positive power. A slightly more complex set of formulae enables us to say that whenever the wick tank reaches a level over `L` or the overflow tank reaches a level `M` and the manual pump is `'on'`, the heaters must be turned on for at least 20 time units.

```
twenty_after = ↑power_draw ⇒ (↑power_draw→ 20);
∀ it2_2:  twenty_after {
     it2_2  □ (heaters.stat ≡ 'on')
};
```

We can use the `power_draw` interval set since it has been defined before. The first formula defines an interval set as the periods of twenty minutes from the beginning of the intervals included in the `power_draw` interval set. The last formula says that in each such interval the state of the heaters must always be `'on'`.

6 Conclusions

Taken together, the data collection and data analysis tools offer distributed control program developers the ability to see what their programs are doing and verify correct behavior. We believe that they will make debugging and verifying distributed programs easier. Of critical importance are the useability of the tools – if the tools are not easy to use then developers will not adopt them. We have tried to make our logging library as easy as *printf* to encourage wide use. The analysis tools require more of a learning curve, but we plan to provide graphical and textual interfaces to those. We encourage anyone interested to download our logging tools at: http://www.traclabs.com/rlog/ and to give us feedback on how they can be improved.

7 Acknowledgments

This work is supported by NASA grant NAS2-99020 administered by NASA Ames Research Center. Pete Bonasso of Metrica Inc./TRACLabs is the chief software engineer of the WRS control system and worked with us to instrument and analyze the control code. Mark Shirley of NASA Ames Research Center was a key participant in this project and contributed many ideas to the preliminary design.

References

[1] Appelbe, W., J. Stasko and E. Kraemer, *Applying program visualization techniques to aid parallel and distributed program development*, Technical Report TR GIT-GVU-91-08, Georgia Institute of Technology (1991).

[2] Bonasso, R. P., *Intelligent control of a NASA advanced water recovery system*, in: *Proceedings of the Sixth International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2001)*, 2001.

[3] Bonasso, R. P., R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller and M. Slack, *Experiences with an architecture for intelligent, reactive agents*, Journal of Experimental and Theoretical Artificial Intelligence **9** (1997).

[4] Heath, M. and J. Etheridge, *Visualizing the performance of parallel programs*, IEEE Software **8** (1991).

[5] Jahanian, F. and A. K. Mok, *Safety analysis of timing properties in real-time systems*, IEEE Transactions on Software Engineering **12** (1986).

[6] Kleiman, S., D. Shah and B. Smaalders, "Programming with threads," SunSoft Press, Mountain View CA, 1996.

[7] Lehr, T., D. Black, Z. Segall and D. Vrsalovic, *MKM: Mach kernal monitor description, examples and measurements*, Technical Report TR CMU-CS-89-131, Carnegie Mellon University (1989).

[8] Mills, D., *Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi (http://www.faqs.org/rfcs/rfc2030.html)* (1996).

[9] Muscettola, N., P. P. Nayak, B. Pell and B. C. Williams, *Remote Agent: to boldly go where no AI system has gone before*, Artificial Intelligence **103** (1998).

[10] Razouk, R. R. and M. M. Gorlick, *A real-time interval logic for reasoning about executions of real-time programs*, SIGSOFT SE Notes **114** (1989).

[11] Schneider, S., *Real-time data monitoring and visualization*, Technical Report White Paper, available at www.rti.com, Real Time Innovations Inc. (1987).

[12] Simmons, R., *An architecture for coordinating planning, sensing and action*, in: *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, 1990.

[13] Simmons, R., *Structured control for autonomous robots*, IEEE Transactions on Robotics and Automation **10** (1994).

[14] Simmons, R. and G. Whelan, *Visualization tools for validating software of autonomous spacecraft*, in: *i-Sairas*, 1997.

[15] Tsai, J., Y. Bi, S. Yang and R. Smith, "Distributed Real-Time Systems: Monitoring, Visualization and Analysis," Wiley & Sons, New York, 1996.

[16] Tsai, J. and S. Yang, "Monitoring and Debugging of Distributed Real-Time Systems," IEEE Computer Society Press, Los Alamitos, CA, 1995.