September 04, 2019

# Runtime-configured state-machine control of program logging

Christopher J. Phoenix

# Runtime-configured state-machine control of program logging

ABSTRACT

When debugging complex software control flows and interactions, it can be difficult to log exactly the right level of detail to capture the bug. Consequently, a debug cycle can go through dozens-to-hundreds of recompiles and restarts as a programmer attempts to set up logging with sufficient granularity to capture the bug, which adds to the cost and complexity of software development.

Per the techniques described herein, debug components such as a state machine, a ring buffer, etc., are incorporated into a program such that log statements can be enabled or disabled at runtime depending on various states across modules of the program. Via a debugging environment, a programmer can reconfigure the state machine at runtime, enabling efficient handling of debug-logging based on program events. With little or no loss to program speed, the techniques enable a programmer to increase the number of log statements without increasing clutter in the debug-log, modify log generation at runtime, examine program states prior to certain program events or triggers, reduce the number of program recompilations/restarts, and reduce or eliminate the use of global variables for debugging.

KEYWORDS

- Integrated development environment (IDE)
- Debugging
- Logging
- Logic analyzer
- Event-driven programming
- Ring buffer

## BACKGROUND

When debugging complex software control flows and interactions, it can be difficult to log exactly the right level of detail to capture the bug. Operations inside a hot loop may be highly relevant, but only after certain conditions occur in another part of the program. Those conditions themselves may be poorly understood. Once the conditions for recognizing a bug are identified, the program state must often be communicated between near-arbitrary parts of the code in order to control when and what to log. Relevant log statements may be scattered throughout multiple modules. The signal-to-noise of individual log statements may vary during the run of the program. Adequate control of whether to record a log statement may be almost arbitrarily complex, depending on status from many parts of the code. The appropriate control conditions typically change repeatedly during the debugging process.

A debug cycle for a complicated bug often require dozens-to-hundreds of recompiles and restarts as the programmer attempts to set up the logging that is sufficient to capture the bug. This friction results in inadequate logs full of clutter, which must be laboriously scanned and interpreted. Adding or modifying log statements, waiting for compile and download, restarting the program, waiting for the bug to occur, etc. are distracting to the programmer and reduce the amount of attention available for actually understanding the bug, while simultaneously increasing the time and effort required for research.

The present practice of adding unconditional log statements to the code clutters not only the log but also the code, since a log statement might be relevant only when the program is in a certain phase of execution. Adding logic to the program to select when the log statement is executed further clutters the code, especially when the relevance of the statement depends on conditions elsewhere in the program. Either way, reconfiguring what is logged usually requires recompiling, downloading, and restarting the program. Incorrect design of the logging code causes another build-download-restart cycle, further adding to cost and programmer distraction.

The above debugging issues become particularly acute when a rare but consequential bug, e.g., one that occurs once per gigabyte of network traffic, is being tracked down.

## DESCRIPTION

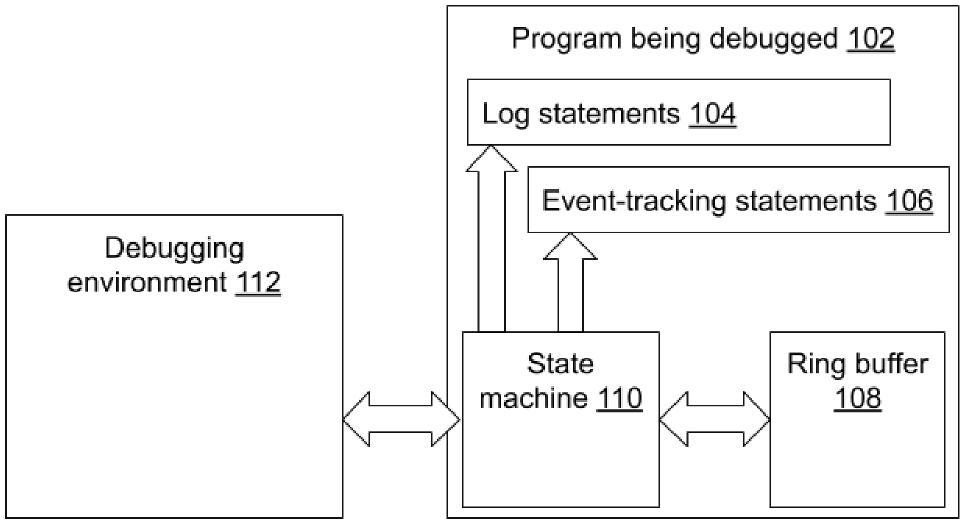
Program being debugged 102
Log statements 104
Event-tracking statements 106
Debugging environment 112
State machine 110
Ring buffer 108

**Fig. 1: Runtime-configured, state-machine control of logging**

Fig. 1 illustrates runtime-configured, state-machine control of debug-logging. Per the techniques of this disclosure, the following are added to the program (102) being debugged.

- **Log statements** (104), e.g., that log the program state or variables.
- **Event-tracking statements** (106), which may include log statements and their parameters.
- A **ring buffer** (108) for log statement output, to retain log output for later decisions as to whether it should be read.
- A **state machine** (110), which monitors events, selects the log statements to enable or disable, and determines how to handle each line of the log output.

- Communications between the program and an **interactive debugging environment** (112), on the same or different computer, to modify the state machine and display logs to the programmer.

Similar in function to a hardware logic analyzer, the state machine:

- Executes state transitions based on program events.
- Optionally, selects and stores in the ring buffer only a small and relevant fraction of the flood of debugging data.
- Preserves and exports the contents of the ring buffer and/or real-time logging statements when the current state indicates the data is desired by the programmer.

The debugging environment may perform the following functions:

- Receive and display log output in near real-time from a running program.
- Enable the user to reconfigure the state machine.
- Handle log statements, e.g., display their status, skip through them, hide or show individual log-statement outputs, store or difference sections of logs from multiple runs, etc.
- Communicate state machine updates to the running program, including displaying and editing the state machine, rules for state transitions, and rules for storing log-statement output.
- Provide various log-related utilities, e.g., differencing versions of code or output, hiding certain statements, collapsing or expanding ranges of statements, displaying corresponding program events, navigating from log or event to the source code that produced it, etc.

For easy navigation between logs, states, and code, the debugging environment can be presented as part of an integrated development environment (IDE).

The ring buffer, which stores recent history and deletes the oldest as newer data arrives, enables relevant log output to be preserved and presented to the programmer in response to an event that happens after the output was logged. The ring buffer is used with the state machine, such that the content of the ring buffer can be preserved from being overwritten in response to a certain specified state in program execution. The ability to preserve an event that happened prior to a trigger is valuable to a programmer, and is today typically only achieved at very high cost by preserving all similar data throughout the execution of the program.

State machines are small to store and fast to execute. Therefore, the tasks of the state machine are done during the logging and event function calls; a separate thread is not necessary. In other words, on each state-setting call, the state machine reacts by advancing the state if appropriate; on each logging call, it determines (based on the current state) whether the call should execute or not. Some state transitions, e.g., transitions that depended on elapsed wall-clock time, may happen during the logging call as well.

Some advantages of the disclosed techniques are as follows:

- With more nuanced control over whether or not a log statement executes, programmers can add many more log statements at a time, reducing the number of recompiles.
- With runtime control via a downloadable state machine, programmers can sensitively modify the log generation from the program without changing code or restarting.
- With a state machine and with a ring buffer, programmers can see back in time prior to a bug or other trigger, with greater detail than with continuous logging.

- The behind-the-scenes connection between events and log behavior (provided by the state machine) allows control of logging based on conditions elsewhere in the program, without any cross-module code, e.g., global variables.

With the provided benefits, the techniques reduce or eliminate friction within software development and enable shrinking the debugging cycle. Programmers can keep their attention on the bug, rather than being distracted by ancillary coding and delays.

In addition to log output, the debugging interface can present a subset of events and/or state transitions directly to the user (as long as they are not too frequent; very frequent events and/or state transitions can be presented in a summary). This reinforces a user's mental model of activity in the program, making it easier to select which log statements they want to enable at various points.

In this manner, the techniques of this disclosure speed up the debugging cycle, potentially allowing hypotheses to be tested in seconds instead of minutes, enabling programmers to maintain a train of thought through several cycles.

Alternate mechanisms to control log statement execution include, at one end of granularity, individual control of each log statement, and at the other end of granularity, turning on or off all log statements simultaneously. A mid-level of granularity can use log levels (WARN, DEBUG, etc.), bit-fields, and arbitrary string tags to select subsets of log statements. The state machine and/or the ring buffer can be used in conjunction with any statement-selection infrastructure anywhere on the granularity spectrum. As alternatives to dynamic external configuration of the state machine, the state machine can be configured via a config file that is read at program start-up, by a snippet of inline code within the program, or by hard-coding a subset of the functionality of the state machine.

## CONCLUSION

Per the techniques described herein, debug components such as a state machine, a ring buffer, etc., are incorporated into a program such that log statements can be enabled or disabled at runtime depending on various states across modules of the program. Via a debugging environment, a programmer can reconfigure the state machine at runtime, enabling efficient handling of debug-logging based on program events. With little or no loss to program speed, the techniques enable a programmer to increase the number of log statements without increasing clutter in the debug-log, modify log generation at runtime, examine program states prior to certain program events or triggers, reduce the number of program recompilations/restarts, and reduce or eliminate the use of global variables for debugging.