RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs

Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister
University of Waterloo
Waterloo, ON, Canada
{snavabpo, y2joshi, cwwwu, sberkovi, rmedhat, bbonakda, sfischme}@uwaterloo.ca

ABSTRACT

We introduce the tool RiTHM (Runtime Time-triggered Heterogeneous Monitoring). RiTHM takes a C program under inspection and a set of LTL properties as input and generates an instrumented C program that is verified at run time by a *time-triggered* monitor. RiTHM provides two techniques based on static analysis and control theory to minimize instrumentation of the input C program and monitoring intervention. The monitor's verification decision procedure is sound and complete and exploits the GPU many-core technology to speedup and encapsulate monitoring tasks.

1. INTRODUCTION

Runtime verification (RV) is a complementary approach to exhaustive verification and testing, where a *monitor* inspects the program's execution at run time to evaluate a set of correctness properties. Most approaches in the literature are *event-triggered* RV (ETRV), where the monitor is invoked with each occurrence of events that can change the valuation of a property. Such techniques exhibit *nonuniform* and *unpredictable* monitoring overhead, which can cause undesirable program behavior and, hence, catastrophic consequences in real-time systems. To overcome this defect, in [4], we introduced the notion of *time-triggered* runtime verification (TTRV), where the monitor stops the program execution within time periods, polls the state of the program, evaluates properties, and resumes the program's normal execution. A time-triggered monitor (TTM) ensures predictable and evenly distributed monitoring overhead and invocations throughout the program run. Such monitoring can control resource usage and predictability of the monitor invocations which are among the indicators of the quality of a monitoring solution, especially in the context of real-time systems.

The main challenge in implementing TTRV is that if valuation of a property changes more than once between two monitoring points, a TTM may overlook a property violation. To deal with this issue, we have introduced several techniques, such as (1) employing history variables for the case where the TTM is an *external* thread [3, 9], (2) inlining TTM's monitoring instructions in the program code [5] (called *self-monitoring*), (3) path prediction using symbolic execution [8], and control-theoretic monitoring [7].

Despite the long history of runtime monitoring, we know of no tools that enable runtime monitoring of real-time systems. In this paper, we introduce the tool RiTHM (Runtime Time-triggered Heterogeneous Monitoring) that realizes a subset of the aforementioned techniques for TTRV. RiTHM takes a C program under inspection and a set of LTL properties as input and generates an instrumented C program that is verified at run time by a TTM. The current implementation of RiTHM supports two TTM and instrumentation techniques: (1) TTM with optimized fixed polling period using static analysis, and (2) TTM with least variation in dynamic polling period using PID and fuzzy controllers. TTM's verification decision procedure for 4-valued semantics of LTL [1] is sound and complete [4], and takes advantage of the GPU many-core technology to speedup monitoring and isolating the monitoring tasks [2]. The tool has been used in several real-world case studies such as the Apache web server, a UAV autopilot software, and a laserbeam stabilizer for eye surgery.

2. TOOL OVERVIEW

Figure 1 shows modules and detailed data flow of RiTHM. The tool takes a C program and a set of LTL properties as input and generates an *instrumented* C program as output. The specification language for expressing properties is the 3-valued LTL designed particularly for runtime verification [1]. Each given LTL property is specified in terms of variables of the input C program. For instance, `G(x >= 10 and foo_y = z)` is one such property, where `x` and `z` are two global variables and `y` is local to function `foo`. Evaluation of LTL properties at run time is handled by a GPU-based verification engine. If a machine is not equipped with the GPU, the verification is automatically shipped to multi-core CPU. The verification engine is invoked by the TTM thread that RiTHM generates using Unix high resolution timers (module 7). The monitor stops the C program's execution with a fixed/dynamic polling period, reads the program state, sends the extracted data to the verification engine, and resumes the program thread. The monitor evaluates properties in parallel with the program execution.

Module 1 (in Figure 1) is *Globalizer* (implemented over LLVM Clang [6]) that takes the C program and LTL properties as input and generates a C program, where all the variables participating in the LTL properties are changed into global variables. Globalizer generates the list of the globalized vari-

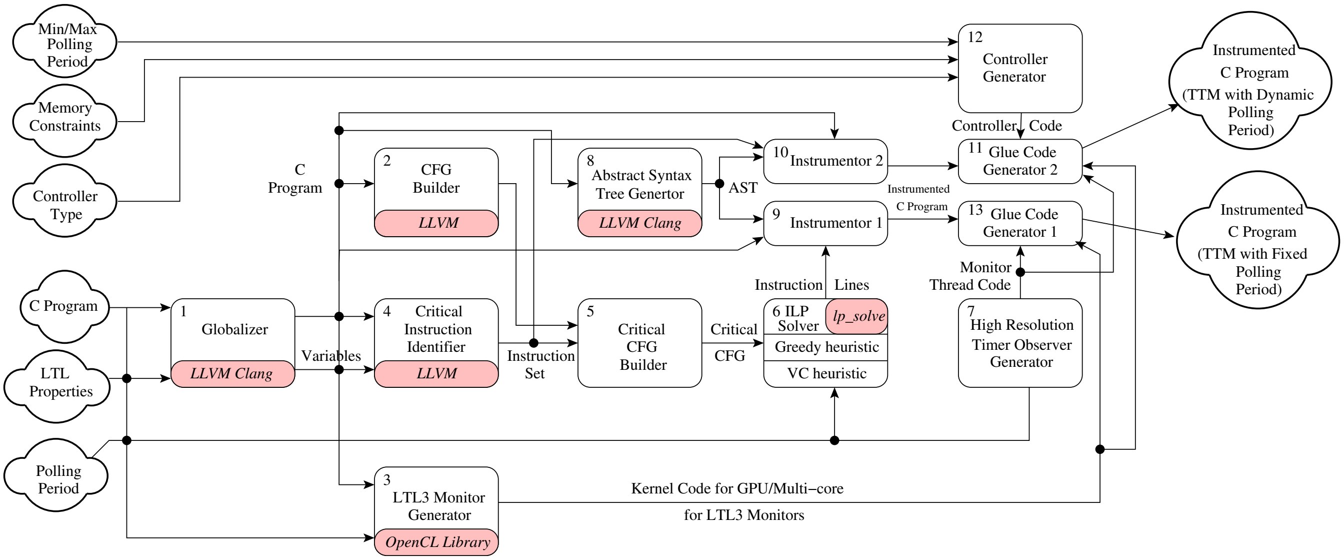


Figure 1: Building blocks and data flow in RiTHM.

ables and passes it to $\text{L}\textsc{tl}_3$ *Monitor Generator* (module 3) and *Critical Instruction Identifier* (module 4) that identifies and annotates the set of the C program's instructions which may change the evaluation of the properties at run time. From this point RiTHM gives two options for generating a TTM, as described in Subsections 2.1 and 2.2.

2.1 Instrumentation for TTM with Fixed Polling Period

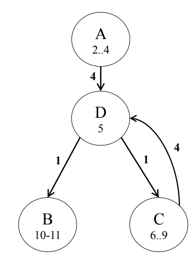
Figure 3(a) shows the screen shot of configuring RiTHM to generate this type of TTM. Globalizer sends the modified C program to *CFG Builder* (module 2) that generates the control-flow graph (CFG) of the program. This module is implemented over LLVM. Figure 2(b) shows the CFG of the program in Figure 2(a) (Fibonacci function). The weight of each arc is the best-case execution time of the instructions in the originating vertex. For simplicity, in this example, we assume that each instruction takes one time unit.

Next, RiTHM constructs a critical CFG, where each critical instruction resides in one and only one vertex (module 5). Figure 2(c) shows the critical CFG of Figure 2(a), where variables Fnew, Fold, and ans are used by the LTL properties. Notice that in order to ensure soundness, the polling period should be not greater than the shortest time between the execution time of two critical instructions (called the longest polling period, LPP). If the input polling period is greater than LPP, then some critical instructions must be instrumented, so that their results are temporarily stored in a *history* buffer until the monitor's next poll. In terms of a CFG, instrumenting a critical instruction involves deleting its corresponding vertex in the critical CFG and merging outgoing and incoming arcs of the vertex by summing up their pairwise weights. We require that the number of instrumentations is minimum. We have shown that this optimization problem is NP-complete [4].

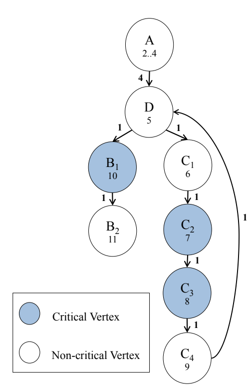
RiTHM uses the following approaches to solve the optimization problem (module 6): (1) integer linear programming (ILP) [4], (2) a greedy heuristic [9], and (3) a heuristic based on finding the minimum vertex cover [9]. The output of either technique is a set of instructions in the C program that need to be instrumented. For the program in Figure 2(a), to apply a polling period of 2 time units, the ILP solution for the critical CFG in Figure 2(c) instruments vertices C_2 (Line 7) and B_1 (Line 10). The lines of code corresponding to these instructions are located using the abstract syntax tree generator (module 8) of LLVM Clang, and instrumented by *Instrumentor 1* (module 9). Finally, *Glue Code Generator 1* (module 13) augments the instrumented code with function calls to the verification engine for verifying properties at run time.

```
1. fib(int n) {
2.    int  i, Fnew, Fold, temp,ans;
3.    Fnew = 1;  Fold = 0;
4.    i = 2;
5.    while( i <= n ) {
6.        temp = Fnew;
7.*       Fnew = Fnew + Fold;
8.*       Fold = temp;
9.        i++; }
10.*   ans = Fnew;
11.    return ans;}
```

(a) Fibonacci function in C

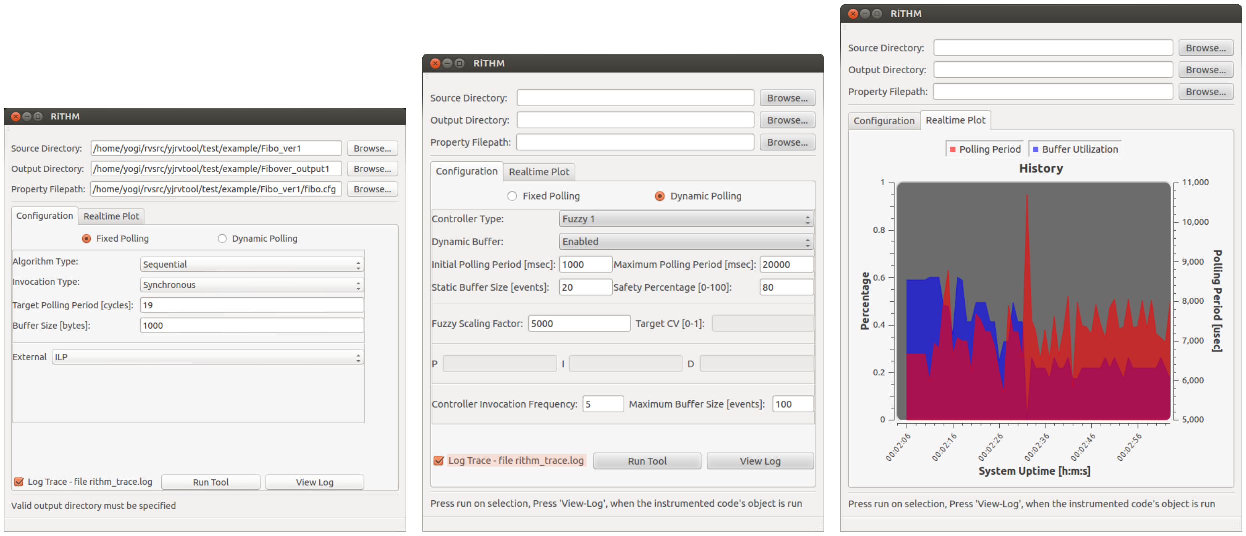


(b) CFG



(c) Critical CFG

Figure 2: Example of a program and CFG.



(a) Configuration of TTM with fixed polling period (b) Controller-based TTM configuration for dynamic polling period (c) Real-time resource utilization plot

Figure 3: Selected RiTHM screen shots.

2.2 Instrumentation for TTM with Dynamic Polling Period

Since in reactive systems, environment events play an important role in determining the polling period, techniques based on static analysis are not expected to be effective. To deal with reactive systems, RiTHM can also generate TTMs augmented with PID and fuzzy controllers (module 12) that can dynamically change the polling period based on the environment behavior. Specifically, given the range of allowed polling periods and constraints on the static/dynamic history buffer size, the controllers target minimizing variations in adjustments to the dynamic polling period as well as maximizing history buffer utilization. Figure 3(b) shows the screen shot for configuring controller-based TTMs. Instrumentation and controller-based TTMs generation are achieved through modules 10 and 11, respectively.

3. SELECTED EXPERIMENTS

Figure 4(a) [4] shows that the absolute overhead incurred by a TTM (with and without history) is bounded and uniform and, hence, predictable, as opposed to an ETM. Figure 4(b) shows that the execution time of the program blowfish (from the MiBench benchmark suite) monitored by TTRV without using history, is larger than the execution time of the program monitored by ETRV. This excessive overhead is due to the fact that a TTM gets invoked more often than an ETM. However, by extending the polling period (e.g., by a factor of 100), TTRV performs better than ETRV. Figure 4(c) shows the execution time and memory usage of the program blowfish when instrumented by ILP and the other RiTHM instrumentation heuristics for the polling period of $40 \times LPP$.

Figure 4(d) shows the coefficient of variation (CV) of polling period and memory utilization (in terms of the number of empty locations in the history buffer, where positive is excessive dynamic allocation and negative denotes an under utilized buffer) for the Apache web server using a controller-based TTM. As can be seen, the monitor that is controlled by two fuzzy controllers for stabilizing the polling period and buffer size (i.e., BSC+PPC:F2) shows a significantly low CV and well-utilize history buffer. The data set of this experiment is from the 1998 FIFA World Cup web server.

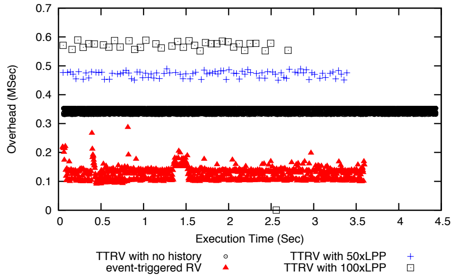
Figure 5(a) shows that our GPU-based algorithms for verifying LTL$_3$ properties are clearly scalable with respect to the number of cores. The error bars represent a 95% confidence interval. This graph also shows that the mean throughput increases with the number of cores engaged in monitoring. At some point, the parallel verification algorithms reach the optimum, where all the core are utilized. Figure 5(b) shows that the CPU utilization of the autopilot process of an unmanned aerial vehicle (UAV) application monitored using our GPU-based algorithms is almost identical to the CPU utilization of the unmonitored process. Notice that the same program monitored by a CPU-based monitor is almost 100% utilized. This result holds when the CPU frequency is reduced to half of the normal frequency. An interesting side-effect of this result is that GPU-based monitoring is considerably power efficient.

4. AVAILABILITY

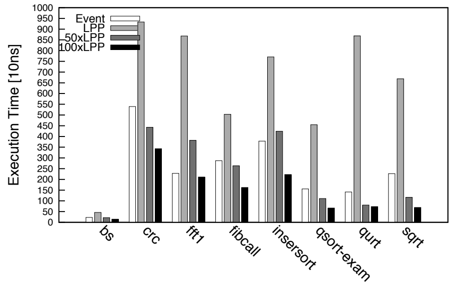
RiTHM is an open source tool. To access the tool, related publications, a screencast, more detailed experimental results, user guide, and other resources, please visit http://uwaterloo.ca/embedded-software-group/projects/rithm.

5. SUMMARY

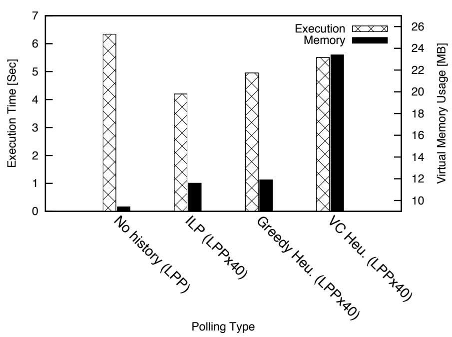
In this paper, we introduced the tool RiTHM that augments C programs with monitors that ensure time predictability and optimal memory utilization for sound and complete verification of LTL properties and run time. This type of monitoring is especially useful in the context of real-time systems. RiTHM applies two methods: (1) fixed monitor polling using static code analysis, and (2) dynamic polling using controllers that response to environment actions. RiTHM has been tested using large software applications such as the Apache web server, a UAV autopilot, and a laser-beam stabilizer for eye surgery.



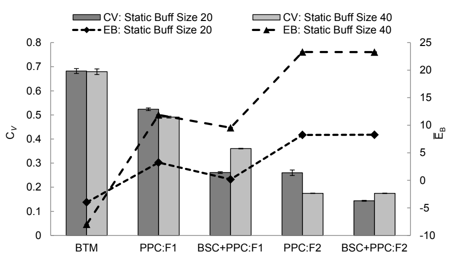
(a) TTRV of blowfish.



(b) TTRV of SNU programs.



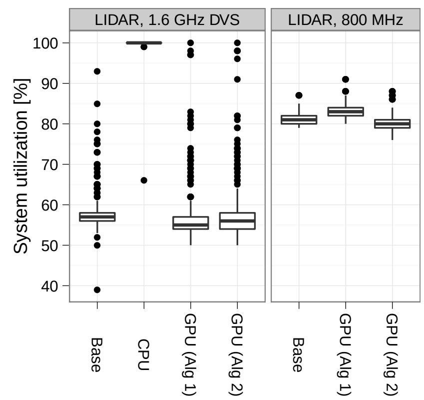
(c) Memory vs. execution time.



(d) Controller-based TTM.

Figure 4: Selected experiments.

(a) Scalability.



(b) CPU utilization.

Figure 5: Selected GPU-based verification experiments.

In future releases, RiTHM will include our optimization techniques using symbolic execution, inlined monitors, and combined static and dynamic analysis methods.

6. REFERENCES

[1] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[2] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. GPU-based runtime verification. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2013. To appear.

[3] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.

[4] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design (FMSD)*, 2013. To appear.

[5] B. Bonakdarpour, J. J. Thomas, and S. Fischmeister. Time-triggered program self-monitoring. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 260–269, 2012.

[6] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, page 75, 2004.

[7] R. Medhat, D. Kumar, B. Bonakdarpour, and S. Fischmeister. Runtime verification with controllable time predictability and memory utilization. Technical Report CS-2013-02, University of Waterloo, 2013.

[8] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Path-aware time-triggered runtime verification. In *Runtime Verification (RV)*, 2012. To appear.

[9] S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Runtime Verification (RV)*, pages 208–222, 2011.

APPENDIX

A. THE DEMO

The demo illustrates RiTHM by designing, enabling monitoring, and executing a simple Fibonacci example for fixed polling period, and dynamic polling period. The demo consists of four parts: (1) application design, (2) development of LTL properties, (3) applying RiTHM to synthesize monitors and instrument programs, and (4) demonstrating RiTHM's features on collecting statistics about satisfaction/violation of properties, trace logs, and timing behavior of monitors. If time permits, we will also show a highlight of applying RiTHM on a more sophisticate example such as Apache.

A.1 Application Design

We will develop the Fibonacci function as illustrated in Figure 6 as a C program with a given input value n. The demo will analyze the corresponding CFG (see Figure 2(b)).

A.2 Specification Development

We will demonstrate monitoring three properties (see Figure 7):

- The first property property0 states that the loop counter eventually reaches a value greater than 10.
- The second property property1 states that it is always the case that the loop counter i is less than or equal to n.
- The third property property2 intends to monitor that an element in the output Fibonacci series is strictly greater than the previous element.

A.3 Instrumentation Phase

The RiTHM configuration screenshot is shown in Figure 3(a). Running RiTHM results in the log shown Figure 8. This window also shows possible errors and warnings. For instance, an error could be using a variable in a property that is never declared. A warning may be issued in case of using unresolved pointers or aliases. The RiTHM user guide describes all the limitations (mostly due to LLVM static analysis) in detail.

After running the tool, the instrumented program is the one shown in Figure 9. As can be seen, the instrumentation adds the value of the variables of interest based on the given polling period such that no properly valuation is overlooked. It can also be seen that Globalizer has prefixed variables by the name of the function that declares the variables of interest (i.e., function main).

A.4 Running the Instrumented Application and Observing Online Statistics

RiTHM provides detailed report on the valuation of LTL properties and maintains a trace log that enables the user to trace back the value of variables that participate in LTL properties. This is achieved by the help of the name of the source C file and the line number of the instruction that changes the value of the variable. Figures A.4 and A.4 show the property and trace log of the case study. RiTHM reports that property0 has been satisfied, but property1 and property2 are violated. The trace log illustrates the reason. Property1 is violated because when the loop terminates the value of i is 31 while the value of n is 30. Property2 is violated because the values of Fnew and Fold are zero in the initial state of the program.

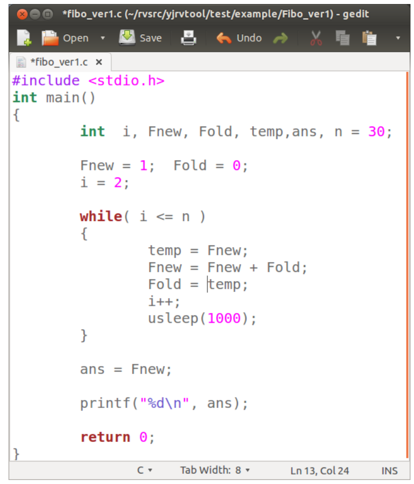


Figure 6: Input C program (Fibonacci function)

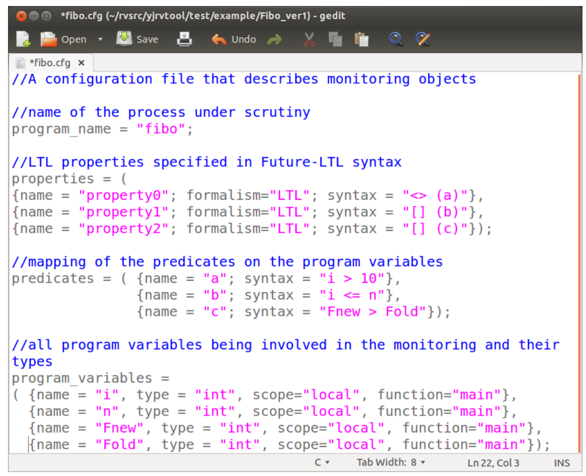


Figure 7: LTL Properties

When dynamic polling is used, RiTHM provides a real-time plot of the current polling period and the utilization of the available buffer as seen in Figure 3(c).

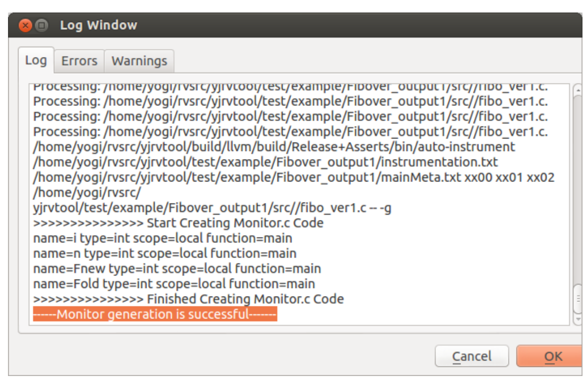


Figure 8: Instrumentation log

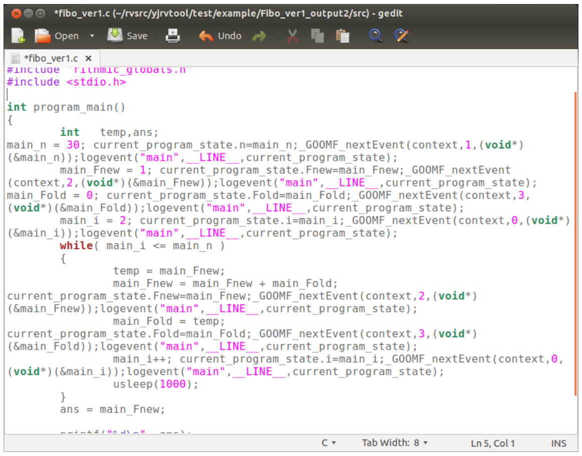


Figure 9: Instrumented code

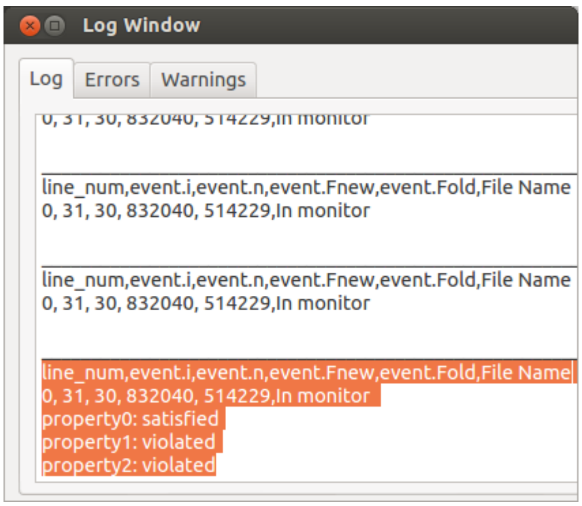


Figure 10: Property valuations report

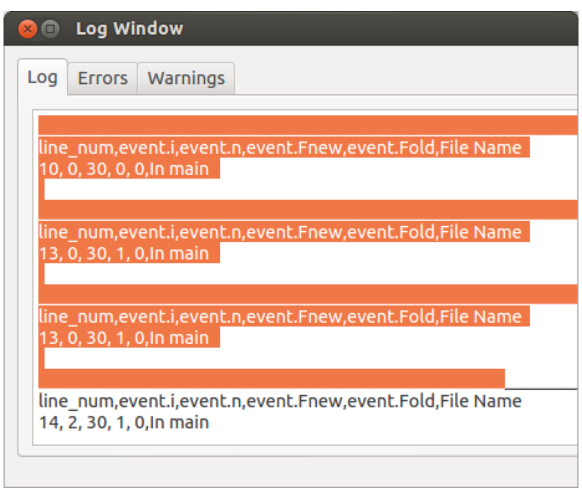


Figure 11: Trace log