Real-Time Embedded Software Group

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

User's Guide

Electrical and Computer Engineering Department,
University of Waterloo, January 2013

Table of Contents

List of Tables iii

List of Figures v

1 RiTHM Overview 3

1.1 Time-triggered Monitor 4

1.2 Verification Engine 4

2 Getting Started 5

2.1 Pre-requisites 5

2.2 Installation 7

2.3 Configuration File for RiTHM 9

3 Repository Structure 11

3.1 doc Directory 11

3.2 src Directory 11

3.3 test Directory 13

4 Running RiTHM 15

4.1 Running RiTHM via GUI 15

4.1.1 Fixed Polling 15

4.1.2 Dynamic Polling 17
4.1.3 Realtime Plot 18
4.2 Running RiTHM via Command line 18
4.3 Viewing Results 19

5 RiTHM with an Example 23

6 Limitations of RiTHM 33

List of Tables

List of Figures

2.1 An example of a configuration file. 10

4.1 RiTHM's GUI window. 16
4.2 RiTHM's dynamic polling GUI window. 21
4.3 RiTHM's realtime plot GUI view. 22
4.4 RiTHM's log GUI view. 22

5.1 Globalizer output. 24
5.2 GooMF output. 24
5.3 Critical Instruction Identifier output. 26
5.4 CFG Builder and Critical CFG Builder output. 27
5.5 ILP Solver output. 29
5.6 Instrumentor output. 30
5.7 Monitor Generator and verification output. 31

About This Guide

This guide is intended for software developers that wish to use RiTHM to verify their systems at runtime. RiTHM is a tool chain that automates the process of runtime verification of projects written in C. To this end, RiTHM leverages time-triggered monitoring to observe the system's runtime behavior, and the GPU many-core technology to verify the runtime behavior of the system. The rest of the guide provides (1) an overview of RiTHM, (2) RiTHM's installation procedure, (3) RiTHM's directory structure, (4) how to run RiTHM, (5) an example of running RiTHM, and (6) RiTHM's limitations.

CHAPTER 1

RiTHM Overview

In a computing system, correctness refers to the assertion that the system satisfies its specification at all times. Achieving system correctness is a major problem for today's large software systems. Verification and testing are arguably the two most common approaches to ensure program correctness. However, verification may suffer from the state explosion problem, and testing may not be able to cover all possible execution scenarios of the system. These limitations argue for *runtime verification* where it inspects a system's runtime behavior to verify a set of properties at run time. Runtime verification frameworks mainly consist of two major components, the *monitor* which extracts information from the system at run time, and the *verification engine* which verifies a set of properties at run time with respect to the information provided by the monitor.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of an event of interest triggers the monitor to extract information and call the verification engine for property evaluation. This technique leads to defects such as unpredictable monitoring overhead and potentially bursts of monitoring invocation at run time. Such defects can cause unpredictable behavior at run time; especially in real-time embedded safety/mission-critical systems, where it can result in catastrophic consequences. To tackle these drawbacks, we propose **RiTHM**: Runtime Time-triggered Heterogeneous Monitoring.

RiTHM is a runtime verification framework which uses *time-triggered* monitoring to observe a program's runtime behavior [3], [2]. The time-triggered monitor runs in parallel with the program and extracts (i.e., samples) the program state at fixed time intervals (i.e., the sampling period). In addition, at each sample the time-triggered monitor calls the verification engine to evaluate a set of LTL_3 properties with respect to the sampled program state. Our studies show that the time-triggered monitor

of RiTHM results in observing bounded monitoring overhead and predictable monitoring invocation at run time [3], [2], a feature required for runtime verification of real-time embedded safety/mission-critical systems.

1.1 Time-triggered Monitor

The time-triggered monitor is a separate thread which runs in parallel with the program under inspection and samples the program with respect to a pre-defined fixed sampling period. An issue surrounding time-triggered monitoring is *sound state reconstruction* (i.e., sampling of all states that are vital to achieve sound verification of the set of properties). Hence, RiTHM leverages static analysis of the program to determine the *longest sampling period* which ensures sound state reconstruction [3], [2], [6]. To further decrease the runtime overhead of monitoring, RiTHM devises a technique to add minimal instrumentation into the program to further increase the sampling period of the time-triggered monitor while ensuring sound state reconstruction. RiTHM carries out techniques for adding instrumentation such that minimal additional memory is imposed on to the program under inspection [3], [2], [7].

To further decrease the runtime overhead imposed by RiTHM, we devised a method to eliminate the inherit overhead of concurrency. Hence, we developed a self-monitoring technique where the time-triggered monitor is weaved within the program under inspection [4].

1.2 Verification Engine

When the time-triggered monitor reads the state of a program, it passes them to the verification engine to evaluate a set of LTL_3 properties. The verification engine evaluates the set of properties in a parallel fashion and makes use of the GPU many-core technology. The verification engine uses two parallel monitoring algorithms that effectively exploit the many-core platform available in the GPU. This verification engine further reduces monitoring overhead, monitoring interference, and power consumption due to leveraging the GPU technology [1].

CHAPTER 2

Getting Started

This chapter will guide you through the procedure to download RiTHM's source files and build the tool from source. At the moment, RiTHM is targeted for both 32- and 64-bit Ubuntu 12.04 LTS, and 32-bit Ubuntu 11.10. Most of the build and execution infrastructure should be portable to any *nix system, but will require additional efforts to build RiTHM's dependencies. The support for other *nix distributions is under construction and will be available in the future.

2.1 Pre-requisites

The following is a list of things that you will require to build and develop RiTHM:

Operating System

- Any computer running Ubuntu 12.04 LTS or 32-bit Ubuntu 11.10.
- `sudo` access with your user account in Ubuntu.

get-apt Packages

- `ia32-libs`: If you are running 64-bit Ubuntu, `i132-libs` package must be installed. This package allows the user to skip compilation processes specific to 64-bit machines and run 32-bit executables on 64-bit platforms.
- `realpath`: This package is used by some of the invoked shell scripts to resolve path parsing issues and path dependencies.

- `QMake`: If you wish to build the GUI from the committed source file, you require `qmake`.

OpenCL Packages

RiTHM allows you to choose whether the verification engine performs its verification on CPU or on GPU. Thus, the following OpenCL supporting packages must be perviously installed.

- Systems with AMD/ATI GPU: AMD GPU OpenCL SDK can be downloaded from here:

  http://developer.amd.com/tools/hc/AMDAPPSDK/downloads/Pages/default.aspx

- Systems with NVIDIA GPU: NVIDI OpenCL SDK can be downloaded from here:

  http://www.nvidia.com/Download/index.aspx?lang=en-us

Remark: If your system uses AMD GPU/APU card, but you do not have AMD GPU OpenCL SDK installed, it will be installed automatically when running ./build_deps.sh.

General Applications

- Text editor of your choice.
- An integrated development environment (IDE), if you are not as comfortable with the command line.

Optional Dependencies

If you intend to acquire RiTHM from Bitbucket (see Section 2.2), you will need the following:

- `git`: If `git` has not been installed on your machine, you can install it on Ubuntu by typing the following command in the terminal:

  ```
  sudo apt-get install git
  ```

- SSH keys, if you wish to connect to Bitbucket via SSH.
- Bitbucket account (this is optional).

Other Dependencies

The following packages are packaged along with RiTHM's source code and will be installed automatically when running ./build_deps.sh:

- lp_solve
- libconfig
- open CSV (java lib)
- apache commons (java lib)
- LLVM
- Clang

2.2 Installation

Step 1. Make sure you have installed all the pre-requisites for OpenCL Packages and General Applications.

Step 2. Acquire RiTHM. You can do so by using one of the following methods:

- Acquire it directly from the Real-time Embedded System's group webpage. Download the tar ball of the tool provided within the following webpage:

  https://uwaterloo.ca/embedded-software-group/projects/rithm

- Acquire it from Bitbucket. To work with the git repository on Bitbucket, you can either retrieve the tool using HTTPS or SSH. If you plan on using HTTPS to communicate with the Bitbucket repository, change to the directory that you would like to make the clone in and then enter the following command in the terminal:

  ```
  git clone
  https://<username>@bitbucket.org/embedded_software_group/rvtool.git
  [name of local directory]
  ```

  where `<username>` is your Bitbucket username, and an optional argument `[name of local directory]`, which designates the name of the folder the cloned repository will locally reside in. The `git clone` command is slightly different for SSH:

  ```
  git clone git@bitbucket.org:embedded_software_group/rvtool.git
  [name of local directory]
  ```

Step 3. After cloning the repository to your local machine, it is time to start building the tool from source. First, change the directory in your terminal to the root directory of RiTHM. In this directory, there are two files you will need to invoke to build the RiTHM:

- `build-deps.sh`
- `Makefile`

`build-deps.sh` contains the necessary commands required to pull in all of the tool's external dependencies and build them as necessary. This will also establish the subdirectory named 'build', which will contain all the compiled objects and executables. From the terminal, run the following command:

```
sudo ./build-deps.sh
```

`sudo` access is required to install missing packages via `apt-get`. The following packages may be installed via `apt-get`:

- `realpath`: converts any relative directory/file paths into absolute paths.
- `subversion`: version control required to pull LLVM and Clang sources.
- `ia32-libs`: needed for machines running 64-bit Ubuntu for compatibility with other libraries.
- `AMD APP SDK`: (Note: this is installed in `/opt/AMDAPP/` as opposed to the 'build' directory).
- `Qt and QMake`
- `lp_solve`
- `libconfig`
- `open CSV` (java lib)
- `apache commons` (java lib)
- `LLVM`
- `Clang`

This script will likely take one or more hours to finish, because the LLVM framework and Clang takes a long time to compile and build for the first time.

Step 4. When `build-deps.sh` successfully finishes, run `make` from the root directory in the terminal (no need for `sudo`) to build the tool. When `make` finishes running, you can change to the 'build' directory and run RiTHM by either using command-line (calling `run.sh`) or the GUI (initiating `rvtool`). For more details about running RiTHM, please refer to Section 4.

2.3 Configuration File for RiTHM

RiTHM requires a configuration file that contains the set of properties that the verification engine will verify at run time. Hence, you must provide this configuration file before running RiTHM. This configuration file must contain the list of the properties along with the mapping of the predicates of the properties to the local and global variables of the program under inspection. Figure 2.1 shows an example of a configuration file.

A configuration file consists of the following parts:

- program_name: It contains the name of the program under inspection (ex. fibo).
- functions: You can define your own customized functions that you require to define a property. You can define the functions using the standard C language.
- properties: It contains the set of properties that the verification engine much verify at run time. The properties can be either in LTL or AT&T FSM format. Each property consists of three parts: (1) name, which is the unique name of the property, (2) formalism, which notes the format of the property, and (3) syntax, which is the definition of the property.
- predicates: It maps the predicates used in the properties to its definition with respect to the program variables. Each predicate consists of two parts: (1) name, which is the name of the predicate, and (2) syntax, which is the definition of the predicate, using the program variables (ex. for predicate a in Figure 2.1, variables x and z are variables of the program QLogAudit).
- program_variables: It contains the set of program variables used by the predicates of the properties. Each program_variable consists of two parts: (1) name, which is the name of the variable, and (2) type, which is the type of the variable.

```
//A configuration file that describes monitoring objects

//name of the process under scrutiny
program_name = "QLogAudit";

//developer can specify his own verification functions
functions = (
"bool IMUSanityCheckPhi(float phi1, float teta1, float phi2, float p1, float q1, float r1, float delta)
{
float temp = ((phi2 - phi1) / 0.01) - (p1 + q1*sin(phi1)*tan(teta1) +    r1*cos(phi1)*tan(teta1));
      return (temp < delta && temp > -delta);
}",
"bool xisNegative(x)
{
      return (x < 0);
}")

//LTL properties specified in Future-LTL syntax or in AT&T FSM format
properties = (
{name = "prop1"; formalism="LTL"; syntax = "[] (a && b)"},
{name = "prop2"; formalism="LTL"; syntax = "[] (a U b)"},
{name = "prop3"; formalism="LTL"; syntax = "[] (c -> X (d U ! c))"},
{name = "satellites"; formalism="FSM"; syntax =
"digraph G {
\"(0, 0)\" -> \"(1, 1)\" [label = \"(e&&f)\", action = \"bind(timestamp)\"];
\"(0, 0)\" -> \"(1, 1)\" [label = \"(e)\", action = \"bind(timestamp)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(f)\"];
\"(0, 0)\" -> \"(0, 0)\" [label = \"(<empty>)\"];
\"(1, 1)\" -> \"(1, 1)\" [label = \"(e&&f)\"];
\"(1, 1)\" -> \"(-1, 2)\" [label = \"(e)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(f)\"];
\"(1, 1)\" -> \"(0, 0)\" [label = \"(<empty>)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(e&&f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(e)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(f)\"];
\"(-1, 2)\" -> \"(-1, 2)\" [label = \"(<empty>)\"];
\"(-1, 2)\" [label=\"(-1, 2)\", style=filled, color=red]
\"(1, 1)\" [label=\"(1, 1)\", style=filled, color=pink]
\"(0, 0)\" [label=\"(0, 0)\", style=filled, color=darkseagreen1]
}"
} );


//mapping of the predicates on the program variables
predicates = ( {name = "a"; syntax = "x > y"},
           {name = "b"; syntax = "z < 0"},
           {name = "c"; syntax = "xisNegative(float x)"},
           {name = "d"; syntax = "z == 0"},
           {name = "e"; syntax = "num_of_sats < 3"},
           {name = "f"; syntax = "timestamp < param1 + 10"});

//all program variables being involved in the monitoring and their types
program_variables = ( {name = "x"; type = "float"},
                {name = "y", type = "float"},
                {name = "z", type = "float"},
                {name = "num_of_sats", type = "unsigned int"},
                {name = "timestamp", type = "unsigned long"});
```

Figure 2.1. An example of a configuration file.

CHAPTER 3

Repository Structure

When you download RiTHM, the root directory is called `rvtool`. This directory contains three main subdirectories `doc`, `src`, and `test`.

3.1 doc Directory

This directory contains two main folders `dev-guide` and `user-guide`. The `dev-guide` contains a pdf file `dev-guide-main` which is a manual for developers who tend to contribute to RiTHM's development. The `user-guide` contains a pdf file `user-guide` which is this user guide manual.

3.2 src Directory

This directory contains the source files of RiTHM. This directory contains the following main subdirectories:

- `frontend`: This directory contains the source files for RiTHM's GUI and the shell scripts to run RiTHM via command line. The `gui` folder contains the source files for the GUI, and the `run` folder contains the shell scripts to run RiTHM from command line.

- `globalizer`: This folder contains source files for the portion of the *Globalizer* module that parses and extracts data from the configuration file. It contains

two folders. The `globalizer_formatter` contains the source files for the formatting functionality of the Globalizer. The `test` folder contains a sample used to test the formatter.

- `GooMF`: Contains the source files for the GooMF verification engine. These files implement the *GPU-based* LTL$_3$ *Monitor Generator* module. For a detailed overview of GooMF, please visit the following website:

  https://bitbucket.org/sberkovich/goomf/wiki/Home

- `java`: This directory contains the source files of the modules written in java. It contains two main folders `lib` and `src`. The `lib` folder contains jar files for open CVS and apache commons. The `src/ca/uwaterloo/esg/rvtool/directory` directory contains the following folders:
  - The `globalizer` folder contains source files for the *Globalizer* module that changes the program's source code appropriately. This module uses Clang to carry out its functionality.
  - The `jni` folder contains the libconfig java files required by some of the modules to carry out their functionality.
  - The `monitor` folder contains the source files for the *High Resolution Timer Observer Generator* module. This module creates the RiTHMicMonitor.c file found in the <path to program>/src directory, after you run RiTHM on the program. In other words, this module creates the time-triggered monitor and also hooks it to GooMF and the program (Glue Code Generator module).
  - The `SMIRF` folder contains source files for the self-monitoring mode of RiTHM. This mode is still under improvement.
  - The `TTRV` folder contains the source files for the *CFG Builder*, the *Critical CFG Set Builder*, and *ILP solver, Greedy heuristic, VC heuristic* modules. In general, `TTRV` contains the source files which carry out the static analysis of RiTHM. The folders `Cleaner`, `ExtractAlias`, `FindInstLines`, `FunctionCleaner`, `NullFinder`, and `VarExtractor` either edit the representation of the input to the *CFG Builder* and *Critical CFG Set Builder* modules, or edit the representation of the output of these modules to fit their interfaces. The `CfgParser` folder contains the source files for the the *CFG Builder*, the *Critical CFG Set Builder*, and *ILP solver* module. The `Heuristics` folder contains the source files for the *Greedy heuristic, VC heuristic* module, and the `Instrumentor` folder contains the source files that create the input to the *Instrumentor* module (i.e., `instrumentation.txt` file).
  - The `llvm` folder contains the patches and source files to run the `Globalizer`, `Critical Instruction Identifier`, and *CFG Builder* modules over LLVM and Clang. The `include` folder contains the edited LLVM source files required to create a weighted control-flow graph required for the

CFG Builder module. The `lib` folder contains the source file to the dynamic library carrying out the `Critical Instruction Identifier` module. The `tools` folder contains the Clang patches required to run the `Globalizer` module.

- The `monitor` folder contains the template for the time-triggered monitor.
- The `SMIRF` folder contains the source files for carrying out self monitoring. This functionality is at the moment under improvement.
- The `swig` folder contains libconfig files.
- The `TTRV` folder contains outputs of examples when running the *CFG Builder*, the *Critical CFG Set Builder*, and *ILP solver*.

3.3 test Directory

In the root directory, you will find a subdirectory named `test`. In the `test` directory, there are two directories `example` and `expected-output`. The `example` directory contains multiple program examples that you can use to check out RiTHM. The `expected-output` contains the output expected when running the programs in the `example` folder in various modes of RiTHM (i.e., ILP, Heu1, and Heu2). You can use these outputs to check if RiTHM properly runs on your machine.

CHAPTER 4

Running RiTHM

RiTHM can be initiated both from a GUI environment (recommended) and from command line. The GUI environment allows the specification of all the required parameters for running RiTHM. Then the GUI wraps the parameters and invokes the appropriate shell script that, in turn, this script invokes the shell scripts with their corresponding command line parameters.

4.1 Running RiTHM via GUI

The following sections present using the RiTHM GUI with a fixed polling configuration, a dynamic polling period configuration, and using it to view realtime measurements of the polling process.

4.1.1 Fixed Polling

Figure 4.1 shows the main GUI window when fixed polling is selected. The description of the GUI fields are as follows:

- Source Directory: Specifies the directory containing the source files of the program under inspection.
- Output Directory: The output directory where the generated files are placed.
- Property Filepath: Specifies the path to the configuration file (see Section 2.3).

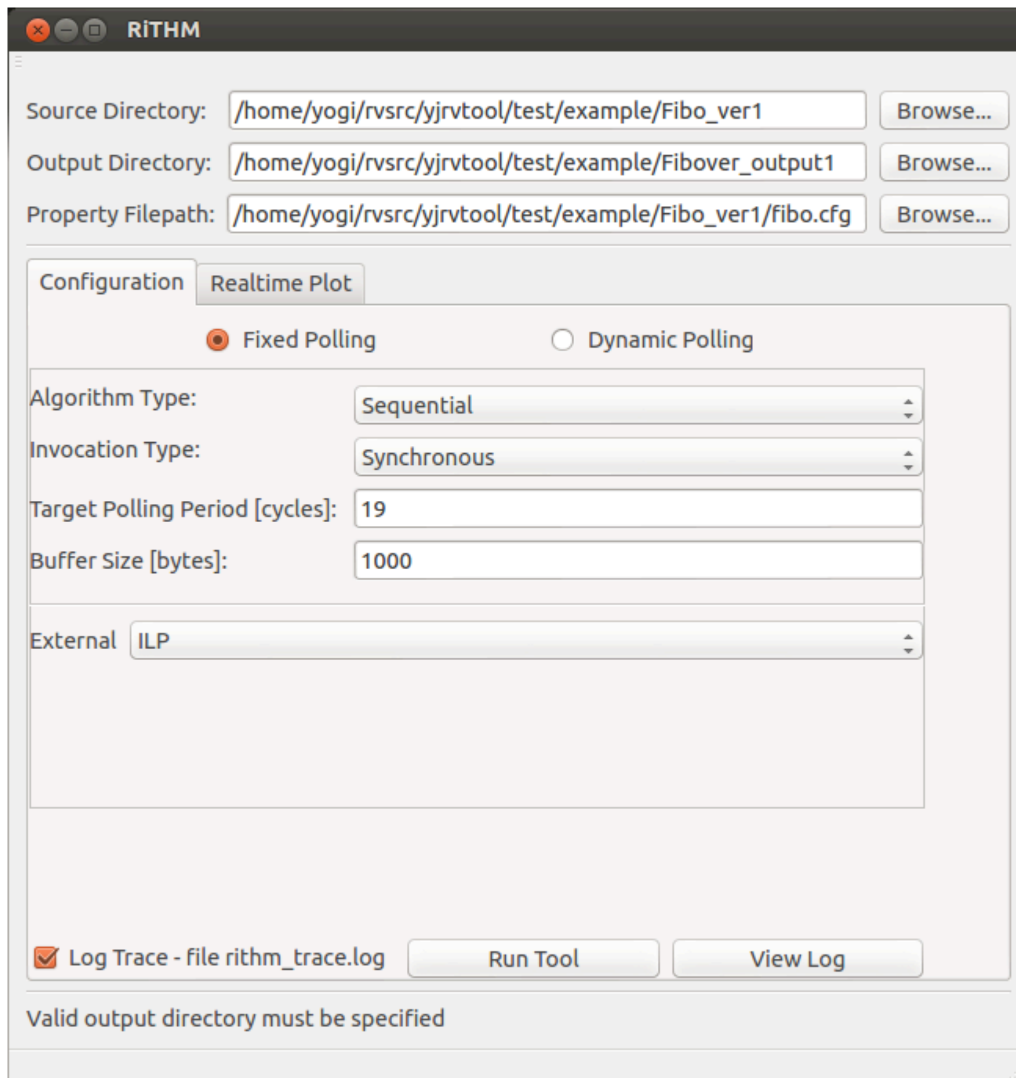

Figure 4.1. RiTHM's GUI window.

- Algorithm Type: Specifies the verification algorithm used by the verification engine (i.e., GooMF). Currently we have developed four different algorithms for property verification. They can be listed in the declining order of the CPU engagement: Sequential, Partial Offload, Finite-History, and Infinite-History algorithm.

- Invocation Type: Specifies the blocking type of the verification engine (i.e., GooMF). The verification engine might be either blocking (main thread (i.e., time-triggered monitor thread) invokes the flush function and waits until its done) or non-blocking (main thread designates a worker thread from the pool to perform the verification).

- Target Sampling Period: Specifies the intended sampling period for the time-triggered monitor. The value is in CPU cycles.

- Buffer Size: Specifies the size of the buffer in which program states are stored

via the instrumentations. By default, the time-triggered monitor flushes the buffer containing the program states (i.e., provides stored states to verification engine, then empties buffer) at each sample. This option allows you to specify a different behaviour in which you define the buffer size and the buffer is flushed once it is full.

- External monitoring optimization algorithm. In this GUI field, you can define the optimization algorithm used to insert instrumentation into the program: ILP, Heu1, or Heu2.
- Log Trace. Controls whether a trace log file should be outputted or not.

After RiTHM finishes its run, enter the following command to create the executable of the instrumented program augmented with the time-triggered monitoring thread:

```
gcc -L"./.." -o"main" <program name>.c RiTHMicMonitor.c
                  rithmic_globals.c -lGooMF -lrt
```

where <program name>.c is the C program under inspection. If your program accommodates multiple C files, change this command appropriately.

4.1.2 Dynamic Polling

Figure 4.2 shows the main GUI window when dynamic polling is selected. The description of the GUI fields are as follows:

- Source directory, output directory, and property file configuration is the same as for fixed polling (see Section 4.1.1).
- Controller Type: The type of controller to use. THere are five possible controllers: PID, Fuzzy 1, Fuzzy 2, Fuzzy 3, and Fuzzy 3 - remapped.
- Dynamic Buffer: Specifies whether dynamic buffer allocations are enabled or not.
- Initial Sampling Period: Specifies the initial sampling period for the time-triggered monitor. The value is in microseconds.
- Maximum Sampling Period: Specifies the maximum sampling period for the time-triggered monitor. The value is in microseconds.
- Static Buffer Size: Specifies the size of the fixed buffer.
- Safety Percentage: Specifies the percentage of buffer occupation that the controller should target.

- Fuzzy Scaling Factor: Specifies the scale by which to multiply the fuzzy controller output. Applicable to all fuzzy controllers only.
- Target CV: The target coefficient of variation for Fuzzy 3 and Fuzzy 3 - Remapped.
- P, I and D: The proportional, integral, and differential gains of the PID controller.
- Controller invocation frequency: The number of monitor invocations that must occur before the controller is invoked. Only applicable when dynamic mode is enabled, otherwise it defaults to 1.
- Maximum Buffer Size: The maximum size of a dynamically allocated buffer.

For more information on these fields, please refer to [5].

After RiTHM finishes its run, enter the following command to create the executable of the instrumented program augmented with the time-triggered monitoring thread:

```
gcc -L"./.." -o"main" <program name>.c rithmic_globals.c -lGooMF
                            -lrt -lm
```

where <program name>.c is the C program under inspection. If your program accommodates multiple C files, change this command appropriately.

4.1.3 Realtime Plot

The Realtime Plot tab in the RiTHM GUI enables the user to view the current polling period, and the current buffer utilization percentage (see Figure 4.3). Using the realtime plot is only applicable when dynamic polling is used. The plot shows in realtime how the controller adapts to the rate of events by dynamically adjusting the polling period.

4.2 Running RiTHM via Command line

You can invoke RiTHM via the command line from the root directory (fixed polling only). An example of invoking RiTHM via command-line is as follows:

```
./run.sh ../test/Fibo ../test/out ../test/Fibo/fibo.cfg ilp 1
    _GOOMF_enum_alg_seq _GOOMF_enum_sync_invocation 100 1
```

- `run.sh` is the main shell script which can be found in <root directory>/src/frontend/run.
- `../test/Fibo` is the source directory containing the program under inspection.
- `../test/out` is the output directory for files of (1) the program under inspection after RiTHM finishes its processing, and (2) the time-triggered monitor. Recall that RiTHM instruments the program to meet the provided target sampling period.
- `../test/Fibo/fibo.cfg` is the path to the configuration file.
- `ilp` is the optimization technique used by RiTHM to determine instrumentation points (can be replaced by heu1 or heu2 when using external monitoring, and sat or greedy when using self monitoring).
- `1` is the target sampling period.
- `_GOOMF_enum_alg_seq` means that the verification algorithm used by the verification engine GooMF is sequential and runs on the CPU (it can be replaced by other algorithms, see [1] for other options).
- `_GOOMF_enum_sync_invokation` means that the verification process is synchronous (it can be replaced by other modes, see [1] for other options).
- `100` is the size of the verification buffer where the instrumentation stores program state.
- `1` enables otputting a trace log of the application. Put `0` otherwise.

After RiTHM finishes its run, enter the following command to create the executable of the instrumented program augmented with the time-triggered monitoring thread:

```
gcc -L"./.." -o"main" <program name>.c RiTHMicMonitor.c
                rithmic_globals.c -lGooMF -lrt
```

where <program name>.c is the C program under inspection. If your program accommodates multiple C files, change this command appropriately.

4.3 Viewing Results

There are two methods to view the convergence results of the properties being verified:

- Using the file `/tmp/rithm_trace.log`, a user can view the results of periodic verification of the given properties.

- Using the GUI, by pressing on the *View Log* button.

An exmaple of this output is shown in Figure 4.4.

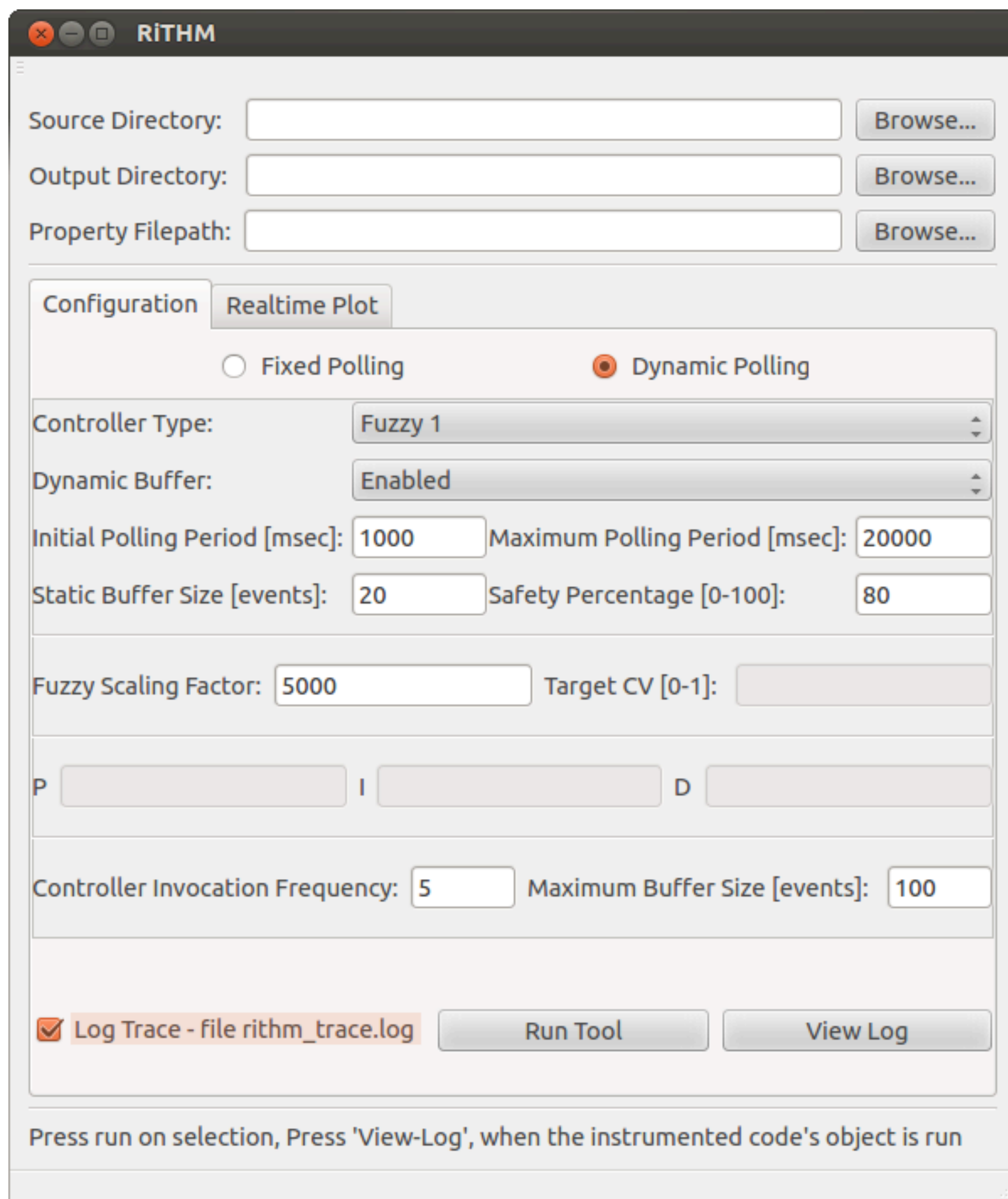

Figure 4.2. RiTHM's dynamic polling GUI window.

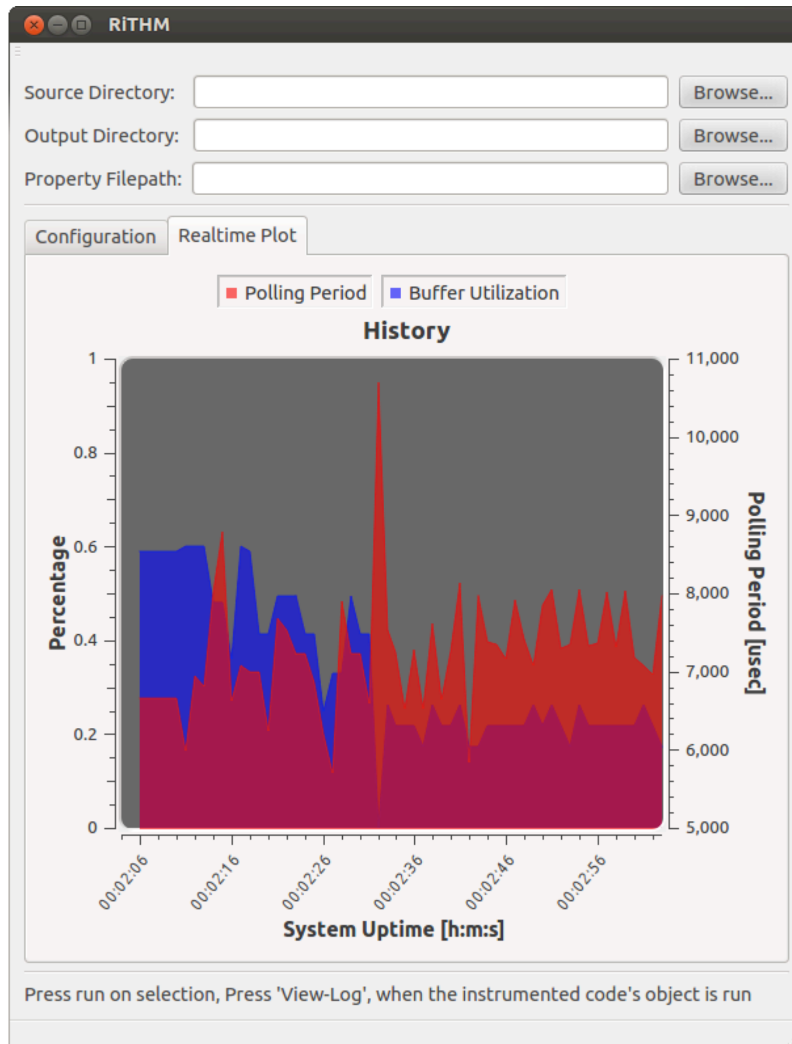

Figure 4.3. RiTHM's realtime plot GUI view.

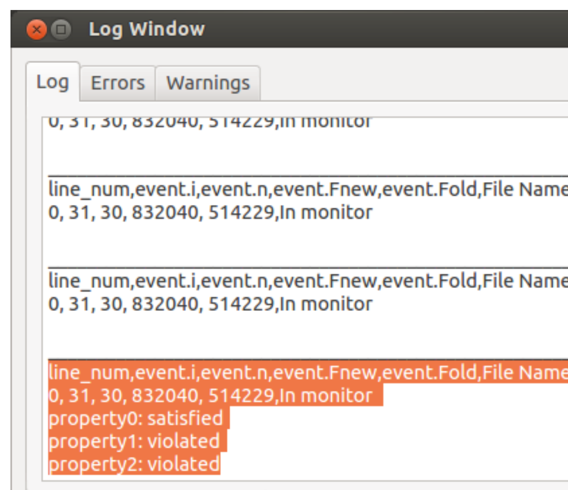

Figure 4.4. RiTHM's log GUI view.

CHAPTER 5

RiTHM with an Example

In the root directory you will find a subdirectory named `test`. In the `test` directory, there are two subdirectories, `example` and `expected-output`. The `example` directory contains multiple program examples that you can use to check out RiTHM. The `expected-output` contains the output expected when running the programs in the `example` directory in various modes of RiTHM (i.e., ILP, Heu1, Heu2). You can use these outputs to check if RiTHM properly runs on your machine.

We will use the `fibo_ver1` program from the `test/example` directory to elaborate more on RiTHM. We will run RiTHM on `fibo_ver1` in the external mode, using the ILP algorithm with a target sampling period of 15. To do so, enter the following command in the terminal:

```
./run.sh ../test/Fibo ../test/out ../test/Fibo/fibo.cfg ilp 15
    _GOOMF_enum_alg_seq _GOOMF_enum_sync_invocation 100
```

First Step: RiTHM does a syntax check on `fibo_ver1` and calls the Globalizer to edit `fibo_ver1`. You should see an output similar to Figure 5.1. After this step, in directory `out`, files `mainMeta.txt` and `goomfMeta.txt`, and in folder `out/src`, files `fibo_ver1.c` and `rithmic_globals.*` have been created.

Second Step: RiTHM generates GooMF related files and recompiles GooMF libraries. You should see an output similar to Figure 5.2.After this step, in directory `out`, files `libGooMF.so`, and in folder `out/src`, files `GooMF*` and `ProgramState.h` have been created.

Third Step: RiTHM calls the Critical Instruction Identifier. You should see an output similar to Figure 5.3(a). After this step, in directory `out/Tool`, folders

```
Source files found:
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c
========== Checking Syntax of Source File(s) =========
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:17:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
        usleep(1000);
        ^
1 warning generated.
========== Parsing libconfig File for Critical Variables =========
<i,int,local,main>
<n,int,local,main>
========== Insert Global Header Include Statement ==========
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:17:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
        usleep(1000);
        ^
1 warning generated.
========== Globalizing Source File(s) =========
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:18:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
        usleep(1000);
...
1 warning generated.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
==================================================
```

Figure 5.1. Globalizer output.

```
properties file path = /home/shay/git-repos/rvtool/test/Fibo_ver1/fibo.cfg
application path = /home/shay/git-repos/rvtool/test/out_fibo_ilp/src
GooMF library path = /home/shay/git-repos/rvtool/build/GooMF
GooMF output lib path = /home/shay/git-repos/rvtool/test/out_fibo_ilp
=====================================================
generating program state and kernels...
=====================================================
copying generated files...
/home/shay/git-repos/rvtool/build/GooMF/GooMFLibrary/GooMF_CPU_monitor.h
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/ProgramState.h
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/GooMF_GPU_monitor_alg_finite.cl
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/GooMF_GPU_monitor_alg_infinite.cl
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/GooMF_GPU_monitor_alg_partial.cl
=====================================================
recompiling GooMF shared library...
currently in:
/home/shay/git-repos/rvtool/build/GooMF/GooMFLibrary
compiling...
make: `lib/libGooMF.so' is up to date.
finished!
```

Figure 5.2. GooMF output.

IR, Nulls, Alias, LLVM, and Pass have been created. These folders contain the following files:

- IR: contains the internal representation of fibo_ver1.c (i.e., fibo_ver1.c.bc and renamed_fibo_ver1.c.bc), and rithmic_globals.c (i.e., rithmic_globals .c.bc and renamed_rithmic_globals.c.bc). Note that in all renamed_* files the registers used in the corresponding internal representation is renamed.
- Nulls: contains file Nulls.txt which incorporates the list of unnamed registers that need monitoring. This file should be empty for fibo_ver1.c.
- Alias contains files alias.txt and func.txt. File alias.txt contains the alias sets, if your using the default LLVM alias analysis, this file must contain the following line:
  AliasSet[0x353f690, 2] may alias, Mod/Ref Pointers: (i32* @main_n, 4), (i32* @main_i, 4).
  File func.txt contains the list of functions of fibo_ver1.c: main, usleep, printf.
- LLVM: contains files alias.txt, critInst.txt, and functionCall.txt. File alias.txt contains the reformatted alias sets from Alias/alias.txt. File critInst.txt contains the variables to be monitored: main_i, main_n. File functionCall.txt contains the lines of code that contain calls to program functions. This file should be empty for fibo_ver1.c.
- Pass: contains files critInst.txt, critLines.txt, functionCalls.txt, inst Lines.txt, and Nulls.txt. File critInst.txt contains the variables to be monitored: main_i, main_n. File critLines.txt contains the critical instructions, see Figure 5.3(b). File functionCall.txt contains the lines of code that contain calls to program functions. This file should be empty for fibo_ver1.c. File instLines.txt contains the critical instructions with the duplicates removed, see Figure 5.3(c). File Nulls.txt contains the list of unnamed registers that need monitoring. This file should be empty for fibo_ver1.c.

Fourth Step: RiTHM calls CFG Builder and Critical CFG Builder. You should see an output similar to Figure 5.4(a). After this step, in directory out/Tool, folders CFG and Graph have been created. These folders contain the following files:

- CFG: contains files cfg.main.dot and rawCFG. File cfg.main.dot is the dot file containing the CFG of the main function of fibo_ver1.c. File rawCFG contains the contents of all the dot files in the CFG folder (see Figure 5.4(b)).
- Graph: contains files critical_CFG, graph.txt, and vertices. File critical _CFG contains the weighted critical CFG of fibo_ver1.c (see Figure 5.4(c)). File graph.txt contains the weighted CFG of fibo_ver1.c. File vertices maps the critical vertices of the weighted critical CFG to the variable representing these vertices in the ILP model.

```
>>>>>>>>>>>>>>>>> Start Renaming registers
>>>>>>>>>>>>>>>>> Renaming Complete
>>>>>>>>>>>>>>>> Start Finding Null Instructions
>>>>>>>>>>>>>>>> Finished Finding Null Instructions
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:20:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
        usleep(1000);
        ^
1 warning generated.
>>>>>>>>>>>>>>>>> Start Renaming registers
>>>>>>>>>>>>>>>>> Renaming Complete
>>>>>>>>>>>>>>>> Start Finding Null Instructions
>>>>>>>>>>>>>>>> Finished Finding Null Instructions
>>>>>>>>>>> Start processing Aliases
>>>>>>>>>>> Finished processing Aliases
>>>>>>>>>>> Starting the processing of Nulls
main_i
main_n
>>>>>>>>>>> Finished the processing of Nulls
>>>>>>>>>>>>>>> Start Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Finish Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Start Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Finish Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Start Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Finish Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Start Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Finish Extracting Lines for Instrumentation
>>>>>>>>>>>>>>> Start Cleaning out the Final Result
>>>>>>>>>>>>>>> Finished Cleaning out the Final Result
```

(a) Output on terminal.

```
main,main_n,8
main,main_i,8
main,main_n,12
main,main_i,12
main,main_n,19
main,main_i,19
main,main_n,8
main,main_i,8
main,main_n,12
main,main_i,12
main,main_n,19
main,main_i,19
```

(b) `critLines.txt` for `fibo_ver1.c`.

```
main,main_n,8
main,main_i,8
main,main_n,12
main,main_i,12
main,main_n,19
main,main_i,19
```

(c) `instLines.txt` for `fibo_ver1.c`.

Figure 5.3. Critical Instruction Identifier output.

```
///---------------- Starting Creation of Critical CFG -------------------///
/home/shay/git-repos/rvtool/test/out_fibo_ilp/Tool/IR/fibo_ver1.c.bc
>>>>>>>>>>>>>>> Start Extracting Function Calls for Merging CFGs
>>>>>>>>>>>>>>> Finished Extracting Function Calls for Merging CFGs
Writing 'cfg.main.dot'...
>>>>>>>>>>>>>>> Start Extracting Function Calls for Merging CFGs
>>>>>>>>>>>>>>> Finished Extracting Function Calls for Merging CFGs
/home/shay/git-repos/rvtool/test/out_fibo_ilp/Tool/IR/rithmic_globals.c.bc
>>>>>>>>>>>>>>> Start Extracting Function Calls for Merging CFGs
>>>>>>>>>>>>>>> Finished Extracting Function Calls for Merging CFGs
Writing 'cfg.main.dot'...
///--------------------- Merging the CFGs ---------------------///
///-------------------------------------------------------------///
>>>>>>>>>>>>>> Parsing the function Call locations
>>>>>>>>>>>>>> Finished Parsing the function Call locations
>>>>>>>>>>>>>> Finished Merging dot files
>>>>>>>>>>>>>> Reading the raw CFG files
4
>>>>>>>>>>>>>> Adding edges to the graph
>>>>>>>>>>>>>> Reading variables of interest
->>>>>>>>>>>>>> Syncing CFG graph with dummy nodes
->>>>>>>>>>>>>> Merging CFG of functions at function call locations
>>>>>>>>>>>>>> Providing CFG to SMIRF
>>>>>>>>>>>>>> Finished Providing CFG to SMIRF
///--------------------- Creating Critical CFG ---------------------///
///-----------------------------------------------------------------///
>>>>>>>>>>>>>> Removing Non-Critical nodes from the CFG graph
>>>>>>>>>>>>>> Putting the critical instructions in separate nodes in the CFG graph
///--------------------- Calculating LSP ---------------------///
///-------------------------------------------------------------///
>>>>>>>>>>>>>> Writting the Critical CFG
>>>>>>>>>>>>>> Finding LSP
```

(a) Output on terminal.

```
digraph "CFG for 'main' function" {
	label="CFG for 'main' function";
	Node0x1c6eaf0 [shape=record,label="{entry\n 7--12--14\n,
7@5,8@0,11@1,12@0,14@0\n}"];
	Node0x1c6eaf0 -> Node0x1c6eb50;
	Node0x1c6eb50 [shape=record,label="{while.cond\n 14--14\n,14@5\n|{<s0>T|<s1>F}}"];
	Node0x1c6eb50:s0 -> Node0x1c6ebb0;
	Node0x1c6eb50:s1 -> Node0x1c6ec10;
	Node0x1c6ebb0 [shape=record,label="{while.body\n 16--20--21\n,
16@3,17@3,18@1,19@2,20@0,21@0\n}"];
	Node0x1c6ebb0 -> Node0x1c6eb50;
	Node0x1c6ec10 [shape=record,label="{while.end\n 23--25--27\n,
23@3,25@1,27@0\n}"];
}
```

(b) rawCFG for fibo_ver1.c.

```
v,2,main,16,21
v,6,main,7,8
v,7,main,9,14
v,-1,main,0,0
v,-2,main,0,0
e,2,2,16
e,2,-2,14
e,6,7,1
e,7,2,14
e,7,-2,12
e,-1,6,6
```

(c) critical_CFG for fibo_ver1.c.

Figure 5.4. CFG Builder and Critical CFG Builder output.

Fifth Step: RiTHM calls the ILP solver to create and solve the ILP model. You should see an output similar to Figure 5.5(a). After this step, in directory `out/Tool`, folder `ILP` has been created. This folder contain the following files:

- `out_histModel.lp`: contains the ILP model.
- `out_ilp.txt`: contains the ILP solution (see Figure 5.5(b)).
- `insLines.txt`: contains the lines that the ILP solution says that need to be instrumented.

Sixth Step: RiTHM calls the Instrumentor. You should see an output similar to Figure 5.6(a). After this step, in directory `out`, file `instrumentation.txt` has been created (see Figure 5.6(b)). In addition, `fibo_ver1.c` has been instrumented with GooMF APIs (see Figure 5.6(c)).

Seventh and Last Step: RiTHM calls the Monitor Generator. You should see an output similar to Figure 5.7(a). After this step,in directory `out/src`, file `iRiTHMicMonitor.c` has been created. At this point, `fibo_ver1.c` is ready to be compiled (see last paragraph of Section 4). The resulting executable will be verified at runtime. If a property is violated, you will be notified via a message in the terminal (see Figure 5.7(b)).

```
///--------------------- Creating ILP History Model ---------------------///
///---------------------------------------------------------------------///
>>>>>>>>>>>>>> Finding the Loop Structures
>>>>>>>>>>>>>> Finding indegree of each node
>>>>>>>>>>>>>> Creating ILP Model
>>>>>>>>>>>>>> ILP Model Complete
>>>>>>>>>>>>>>> Start Extracting the Instrumentation Points
>>>>>>>>>>>>>> Reading variables of interest
>>>>>>>>>>>>>> Creating Instrumentation for History
>>>>>>>>>>>>>> Instrumentation for Instrumentor Complete
>>>>>>>>>>>>>>> Finished Extracting the Instrumentation Points
```

(a) Output on terminal.

```
Value of objective function: 3.00000000

Actual values of the variables:
x0                 0
x1                 1
x2                 0
x3                 1
x4                 1
e0_0_0             6
y0                 6
y1                 0
e3_0_0             7
e2_1_0             14
y2                 14
y3                 0
y4                 7
y5                 0
e4_0_0             21
e4_1_0             19
y6                 0
y7                 21
y8                 19
y9                 0
e2_0_B_0_0         16
y10                0
y11                16
```

(b) `out_ilp.txt` for `fibo_ver1.c`.

Figure 5.5. ILP Solver output.

```
========== Auto-instrumenting Source File(s) =========
Instrumentation Map:
8 => <main_i,main,main_i,8>
12 => <main_i,main,main_i,12>
---------- Checking Syntax of Instrumented Source File(s) ---------
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/rithmic_globals.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/rithmic_globals.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/rithmic_globals.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/rithmic_globals.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/rithmic_globals.c.
========== Checking Syntax of Source File(s) =========
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:20:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
            usleep(1000);
            ^
1 warning generated.
========== Inserting Braces in Source File(s) =========
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:20:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
            usleep(1000);
            ^
1 warning generated.
---------- Checking Syntax of Modified Source File(s) ---------
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:20:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
            usleep(1000);
            ^
1 warning generated.
========== Auto-instrumenting Source File(s) =========
Instrumentation Map:
8 => <main_i,main,main_i,8>
12 => <main_i,main,main_i,12>
/home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c:20:3: warning: implicit declaration of function 'usleep' is
invalid in C99 [-Wimplicit-function-declaration]
            usleep(1000);
            ^
Instrumenting [** main_i **] after 12:2-12:11
---------- Checking Syntax of Instrumented Source File(s) ---------
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
Processing: /home/shay/git-repos/rvtool/test/out_fibo_ilp/src/fibo_ver1.c.
```

(a) Output on terminal.

```
GOOMF_Header,<#include "GOOMFInstrumentor.h">
main,main_n,8,< _GOOMF_nextEvent(context,1,(void*)(&main_n));>
main,main_i,8,< _GOOMF_nextEvent(context,0,(void*)(&main_i));>
main,main_n,12,< _GOOMF_nextEvent(context,1,(void*)(&main_n));>
main,main_i,12,< _GOOMF_nextEvent(context,0,(void*)(&main_i));>
```

(b) `instrumentation.txt` for `fibo_ver1.c`.

```
#include "rithmic_globals.h"
#inclide "GooMFInstrumentor.h"
#include <stdio.h>

int program_main()
{
        int Fnew, Fold, temp, ans;
    main_n = 30;
        Fnew = 1;  Fold = 0;
        main_i = 2;_GOOMF_nextEvent(context,1,(void*)(&main_n));

    while( main_i <= main_n )
        {
        temp = Fnew;
                Fnew = Fnew + Fold;
        Fold = temp;
        main_i++;_GOOMF_nextEvent(context,0,(void*)(&main_i));
                usleep(1000);
    }
    ans = Fnew;
        printf("%d\n", ans);
        return 0;
}
```

(c) Instrumented `fibo_ver1.c`.

Figure 5.6. Instrumentor output.

```
Running generate-mon.sh
>>>>>>>>>>>>>>>> Start Creating Monitor.c Code
name=i type=int scope=local function=main
name=n type=int scope=local function=main
>>>>>>>>>>>>>>>> Finished Creating Monitor.c Code
```

(a) Monitor Generator output.

```
rtesg@ubuntu: ~/git/rvtool/test/out_ilp_ver1_works/src
rtesg@ubuntu: ~/git/rvtool/build    rtesg@ubuntu: ~/git/rvtool/test/out_ilp_ver1_wor...
-rw-rw-r--  1 rtesg rtesg   116 2012-12-10 13:59 ProgramState.h
-rw-rw-r--  1 rtesg rtesg    24 2012-12-10 13:59 rithmic_globals.c
-rw-rw-r--  1 rtesg rtesg   121 2012-12-10 13:59 rithmic_globals.h
-rw-rw-r--  1 rtesg rtesg  5220 2013-01-08 14:13 RiTHMicMonitor.c
-rw-rw-r--  1 rtesg rtesg  5218 2013-01-08 12:47 RiTHMicMonitor.c~
drwxrwxr-x  2 rtesg rtesg  4096 2012-12-10 13:59 src
drwxrwxr-x 11 rtesg rtesg  4096 2012-12-10 13:59 Tool
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$ ./maIN
bash: ./maIN: No such file or directory
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$ ./main
0
Context initialized..
3524578
property 0: probably satisfied
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$ ./main
0
Context initialized..
3524578
property 0: probably satisfied
rtesg@ubuntu:~/git/rvtool/test/out_ilp_ver1_works/src$
```

(b) Verification output.

Figure 5.7. Monitor Generator and verification output.

CHAPTER 6

Limitations of RiTHM

The current distribution of RiTHM has certain limitations where you as a user must consider to properly use RiTHM. Here we highlight the main current limitations of RiTHM:

Source Code Limitations

At the time being RiTHM can only consider monitoring the following data types. In other words, the predicates of the properties can use variable types of the following:

- `integer`
- `float`
- `character`
- `double`
- `string`
- `arrays with the above type`

In addition, RiTHM has a limited capability of handling aliases. RiTHM uses the alias analysis provided by LLVM. Hence, RiTHM is limited to the capability of LLVM's alias analysis. If the analysis suffers from inaccuracy (ex, does not find an alias), then RiTHM will also suffer from it. Therefore, we highly recommend that you install LLVM patches which provide highly accurate alias analysis. You can find a detailed overview of the LLVM's alias analysis at this link `http://llvm.org/docs/AliasAnalysis.html`.

As for the structure of the source code of the program under inspection, we highly advise that the definition of an array to reside on a separate line. At the time being in some cases the Instrumentor can not handle arrays being defined with other variables on the same line. We are currently working on fixing the issue.

Time-triggered Monitor Limitations

Currently the monitor preempts the main thread using hi-res timers. The minimum period for hi-res timers depends on many factors, including CPU load at the time of preemption, background processes, and other scheduling issues. To decrease the minimum period, we used a linux kernel patched with realtime scheduling (Linux-RT). In our experiments, the period could only be decreased to 5 microseconds and still maintain predictability. Thus for a program to be preempted in a reliable manner, the sampling period should not fall below 5 microseconds.

Bibliography

[1] S. Berkovich. GooMF: GPU-based Online and Offline Monitoring Framework. https://bitbucket.org/sberkovich/goomf/wiki/Home.

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

[3] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Accepted for publication in Springer journal of Formal Methods in System Design (FMSD)*, 2012.

[4] 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.

[5] 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.

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

[7] 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.