Benchmarks and Performance Analysis of the LogicGuard Framework *

Bashar Ahmad
RISC Software GmbH, Hagenberg, Austria
Bashar.Ahmad@risc-software.at

Michael Krieger
RISC Software GmbH, Hagenberg, Austria
Michael.Krieger@risc-software.at

June 29, 2016

Abstract

This paper presents benchmarks, performance measurements and analysis for LogicGuard framework. The specification and data used to perform the benchmarks are all artificial and were designed to show various complexity levels. For this purpose, a set of parameters was defined to generate specifications and sample data, which include quantifier depth, search direction, windows size and delay. A set of tools was designed and implemented to perform the benchmarks and the analysis of the results. In addition to the LogicGuard software, a tool to generate specifications based the performance parameters, a tool to process the measurements and lastly a tool to analyse, extrapolate and plot the results were developed. The results presented show the resource cost in terms of processing time per message, number of instances and memory usage. To be able to capture the performance of LogicGuard specification as accurately as possible, the external function used for monitoring was designed to return as fast as possible such that the time needed for external processing can be neglected. Further analysis using polynomial extrapolation is applied to understand the behaviour of the framework.

1 Introduction

LogicGuard is a framework for dynamically monitoring network traffic streams in real time [3, 4]. The framework uses a specification language for describing stream properties which is based on a rich and well-know formulation languages,

*Supported by the Austrian Research Promotion Agency (FFG) in the frame of the BRIDGE program by the projects 832207 “LogicGuard” and 846003 “LogicGuard II”.

namely classical first order predicate logic and set theory. The specification language allows us to specify streams properties and to construct streams operating on a higher level of abstraction. The logical formulas are quantified over stream positions such that they are able to describe the stream properties by relating the values of messages at different positions with each other. The framework provides a translator that translates the specifications into executable monitors that observe streams in real time for violations of the specified properties. The framework also provides a statical analysis to determine whether the generated monitor can operate with a finite amount of history or not [2].

The LogicGuad specification language supports arbitrarily nested quantifiers which allows us to describe and verify very complex properties of steams; please refer to [4] for more details about the core features of the specification language. In the following example the monitor `M1` verifies that every position `x` with value zero is followed within 100 time units by position `y` that has value one such that between `x` and `y` there are only positions `z` with value two.

```
monitor<IP> M1 =
  monitor<IP> x :
    IsZero(@x) =>
      exists<IP> y with x < _ <= x+100 :
        IsOne(@y) &&
        forall<IP> z with x < _ < z :
          IsTwo(@x)
```

As we see from the previous example the specification language also allows to define a range of stream positions where the predicate needs to applied. In some cases we could have an unbounded range, as in following example. This means that the monitor needs infinite history to operate; in such a case, the static analyzer will complain and we need to transform the specification to one that cops with a finite amount of history [4].

```
monitor<S> M =
  monitor<S> x : P(@x) =>
    position<S> y = max<S> y with _ < x : Q(@y) :
    R(@x,@y)
```

Assuming a specification runs within an finite limit, there is still the question of how efficient that evaluation of that specification is in terms of processing time and memory usage. Basically we would like to find the limits were the monitor can work within an acceptable time frame and memory consumption. From the previous examples, we can see that there are some factors which determine the performance of the specification such as size of search space, direction of search, depth of quantifier nesting and the delay between messages. Changes in these factors could effect the time required for the monitor to evaluate a specification and produce results. There also some external factors which contribute to the overall performance of the monitor such as overhead of the external functions, however in this paper we will ignore overhead caused by external functions (by

using negligible external functions) and focus on the internal performance of monitors.

The paper [1],presented by out colleagues, discuss the space complexity of run-time monitor of a LogicGuard specification, in this paper they study the space complexity based the core language and they propose an algorithm to predicate the space requirements for monitors, in the paper they also provide some experimental results to validate the results of the prediction algorithm. However, in this report, we study the performance of monitor of the full language, we would like to see the effect nested quantifiers from practical point of view, so we can show that LogicGuard can be a practical framework to analyze data stream demonically using predicate logic.

The rest of the paper is divided as follow; Section 2 shows the specifications design, Section 3 describes the environment setup, Section 4 presents the benchmarks results, Section 5 presents the analysis results and Section 6 concludes the paper.

2 Specifications

The aim of these benchmarks is to measure LogicGuard's performance with respect to various specification complexities. In each benchmark we measure the processing time per message (CPU usage), number of instances and total memory usage. The complexity of a LogicGuard specification can be defined in terms of the depth of a quantifier nesting, the size of the search space and the direction of the search (backward, forward or both). These factors directly effect the number of instances generated in order to evaluate the specification. Using these factors we have defined the following parameters in order to generate specifications with different complexity:

L (Quantifier depth): this represents the depth of quantifier used by the main monitor in a given specification. In this benchmark **L** ranges from 0 (no quantifiers) to 2 (two nested quantifiers).

T (Search direction): in LogicGuard quantifiers we need to specify the range where the formula needs to apply. **T** represents the fact whether the quantifier needs to look forward, backward or both directions to evaluate the formula.

S (Window Size) represents how far the quantifier should look backward or forward or both (depending on the type as mentioned in the previous point). In this benchmark the value of **S** starts at 10, is increased by 10 and ends with 100 (S = 10, 20, ... 100 messages). To set this value we have used the range construct of the quantifier; however, since the range is defined by time not position, if a message arrive every 10 ms and we want to set the range backward to 5 then the lower bound value of the range construct is set to 50.

D (Delay): this represents the delay between messages (the inverse of the frequency of message arrivals). In this benchmark **D** starts with 200 ms, is increased by 10 ms and ends with 400 ms (D = 200, 210, ... 400 ms).

Using **L** and **T** we can define 13 different combinations, from these we choose the following 7 for benchmarking:

- No quantifiers.
- One quantifier looking backward.
- One quantifier looking forward.
- One quantifier, looking backward and forward.
- Two quantifiers, the first one looking backward and the second one looking forward.
- Two quantifiers, the first one looking forward and the second one looking backward.
- Two quantifiers, both looking backward and forward.

Using **S** and **D** we can determine the upper and the lower bound for the quantifiers. As explained earlier, the range of **D** in this benchmark is 200 ... 400ms and we use increments of 10 (i.e 200, 210, ... 400ms). As for **S**, we use a range of 10 ... 100 messages with increments of 10 (10, 20, ... 100 messages).

3 Environment Setup

In this section we describe the environment of the benchmarks including the data stream used for the benchmarks, the external functions, the hardware and software used and the LogicGuard settings.

3.1 Data Streams

In this benchmarks we used integer values produced using LogicGuard's built-in random number generator. Please refer to LogicGuard manual for more details on how to use the this feature [3].

3.2 External Functions

Another very important factor of the performance of any given LogicGuard specification is external function(s) used by the specification and their overhead. However, since the purpose of these benchmarks is only to measure the performance of the internal evaluation of a specification, we now used a dummy external function which always returns true such that we can neglect the overhead caused by call an external function. Please refer to Appendix B for more details.

3.3 Environment

To perform these benchmarks a Lenovo X1 Carbon laptop was used. The laptop has an Intel(R) Core(TM) i7-5600U CPU @ 2.6 GHz processor and 8 GB of RAM. The system runs under Windows 10 Pro (x64). LogicGuard.exe version 1.03 [1] was used to run the benchmarks. A small tool written in Java 7 was used for processing the collected measurements and produce CSV as output files. For analysis and plotting another tool was developed using Python 2.7 and the packages scipy, numpy and matplotlib. To generate specifications according to the parameters defined in section 2 we have designed a tool using Java 7 generates specifications from the parameters mentioned above.

3.4 LogicGuard Settings

In this section we will describe briefly the settings of the LogicGuard program. For more details on the options and settings supported by LogicGuard, please refer to the manual [3]. The settings are given by the following sequence of command line arguments.

```
LogicGuard -engine random -randnumber RNDNUM
  -randdelay DELAY -csvout profile.csv
  -reportnumber REPNUM -nowait -nowarn spec.lgs External.dll
```

As mentioned earlier we have used a stream of random integers as our main external data stream by specifying the engine type `-engine random`. In order to control how many message to generate and the delay between each message we have used the options `-randnumber` *RNDNUM* and `-randdelay` *DELAY*. The values *RNDNUM* and *DELAY* depends on the benchmark (the parameter **D** is specified through *DELAY*). LogicGuard is able to capture performance measurements while running by the option `-reportnumber` *REPNUM* which instructs the program to capture measurements every *REPNUM* messages. By default LogicGuard prints the captured measurements to the console, however, we use the option `-csvout profile.csv` to export the measurements to a CSV file. To help automatizing the benchmarks we use the option `-nowait` which instructs the program not to wait for the user to acknowledge that the program is done and the option `-nowarn` to instruct the program not to show the profiler warnings on the screen (warnings such as buffer overflow, or the processing time exceed the delay).

[1] http://www.risc.jku.at/projects/LogicGuard2/

4 Results

In this section we present the results of the benchmarks. They are presented in terms of number of instances, processing time and memory usage. The plots shown in this section are the result of running the specifications descried in Section 2 with 400 ms delay (`D=400`). We repeated the benchmarks with different window sizes starting from 10 messages to 100 messages with increments of 10. For each plot shown in this section, the y-axis represents a specific measurement (number of instances, processing time or memory usage), while the x-axis represents the window size (`S`). For each measurement we present three plots. The first shows the measurement for all specifications, where a log scale is used to plot the results. The second plot shows the measurements for the specification with no or one quantifier, where liner scale is used for this plot. The third plot shows the results for specifications with double nested quantifiers, where also liner scale was used.

4.1 Number of Instances

When the LogicGuard monitor starts, it instantiates an instance for each incoming message to evaluate the specification; the number of instances instantiated and how long shall live depends on the specification. For example, when evaluating a quantifier which is forward-looking and the upper bound is 5, each instance created needs to wait for 4 more messages (instances) before it can be evaluated and terminated. The number of instances is a quite important measurement for LogicGuard, a higher number of instances indicate a busy monitor which requires more resources in terms of CPU time and memory. The more complex and nested the specification is the higher the number of instances becomes.

Figure 1 shows the number of instances measured for all the specifications described earlier. As we can see from the figure, the specification with no quantifier and the specification with one quantifier looking backward have a constant low number of instances. The specifications with one quantifier which is backward and forward looking and respectively only forward looking have a higher number of instances; this yield a line (shown as a curve in the logarithmic scale) in relation to the window size. The specifications with two quantifiers have a much higher number of instances which also yields a line.

Figure 2 shows the number of instances measure for the specification which has no quantifiers and the specifications which have one quantifier.

The specification with no quantifiers evaluates messages as soon as they arrive so any instance we create gets evaluated and terminates immediately. The specification with one quantifier but only looking backward also evaluates messages as soon as they arrive, since it only looking backward; the monitor keeps the history of the evaluated instances, as soon as a new instance get created the monitor can evaluate it right way. It is worth to mention that we need to have a finite history, otherwise the memory could grow indefinitely.

From the figures we notice that for specifications which has forward-looking, the number of instances take a line, with one quantifier the curve is on the

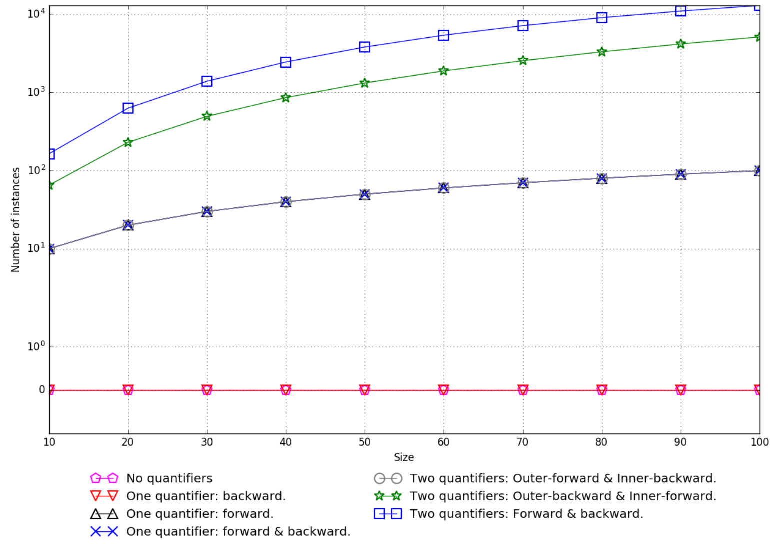

Figure 1: Number of instances: overall view

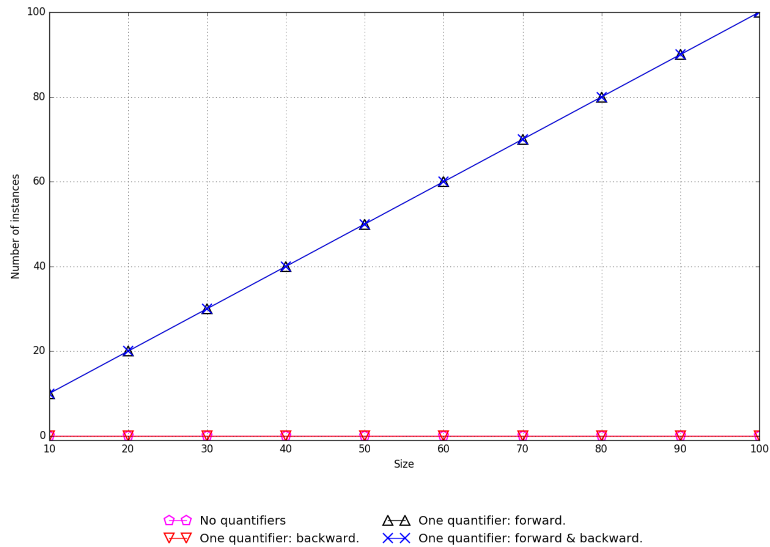

Figure 2: Number of instances: 0 and 1 quantifier.

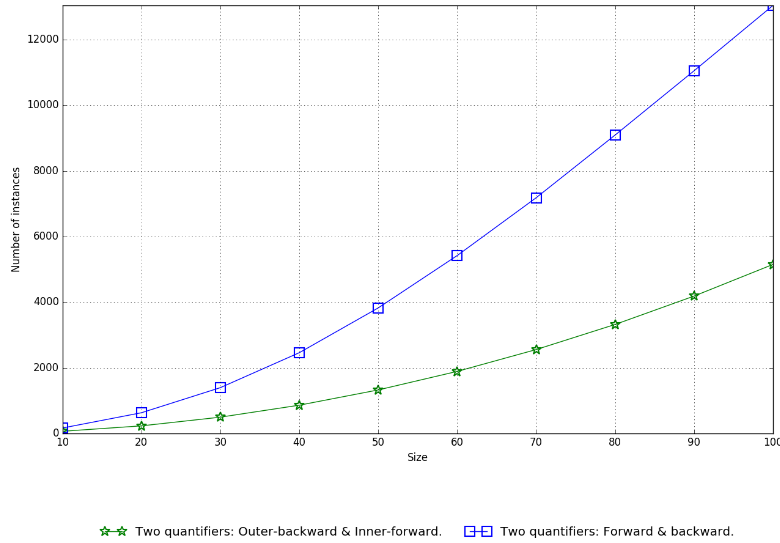

Figure 3: Number of instances: 2 quantifiers.

lower part of the plot (around 100 instances). However, for two quantifiers, the curve is on the upper part of the plot (greater than 14000 instances). We also notice that for specifications with one quantifier which forward looking, adding backward looking does not effect the performance (as shown in figure 2), as explained earlier, the instances for backward looking is zero because it get evaluated right away, however, because of the forward looking, any new instance created need to wait for a certain number of instances before it can be evaluated. The number of instances depends on the window size . As we can see from the figures, as the windows size increases the number of instances increases as well.

4.2 Processing Time

In this section we show the measurements in terms of processing time. We measure how much CPU time is needed to process a message.

Figure 4 shows the processing time for all specifications. The specification with no quantifier requires a very small processing time, close to 0.1 ms; the curve is fairly constant, because the window size does not matter when running no quantifier specification, the only factor is the overhead caused by the external function. However, for the rest we notice that the curves start to incline as the window size increases, as the number of instances increases so the processing time per message increases as well. Not like the number of instance, backward looking has an effect on processing time, however, forward looking still has a greater effect. There is quite a gap when we increase the number of nested quan-

tifiers, because the number of instances needed to evaluated upon the arrival of each message increases significantly with every additional quantifier.

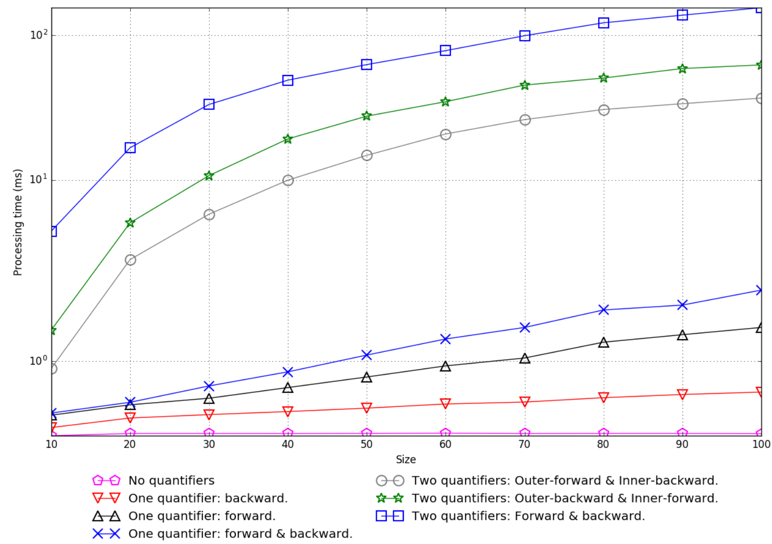

Figure 4: Processing time per message: 0 and 1 quantifier.

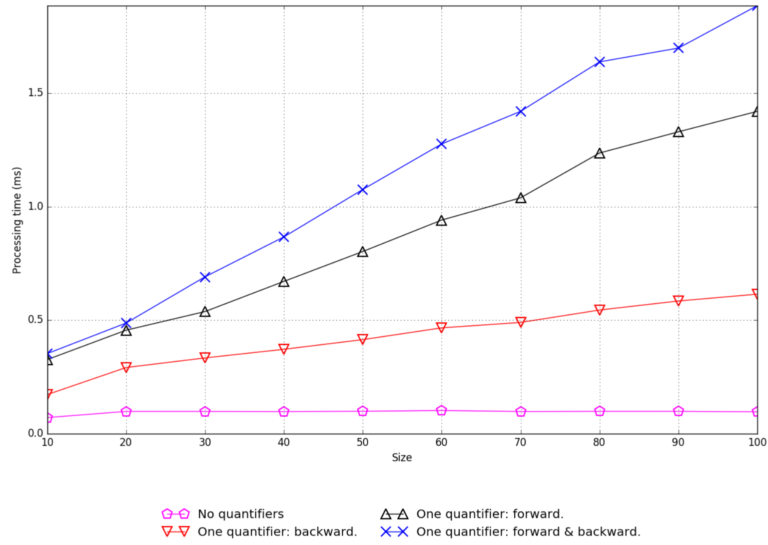

Figure 5: Processing time per message: 0 and 1 quantifier.

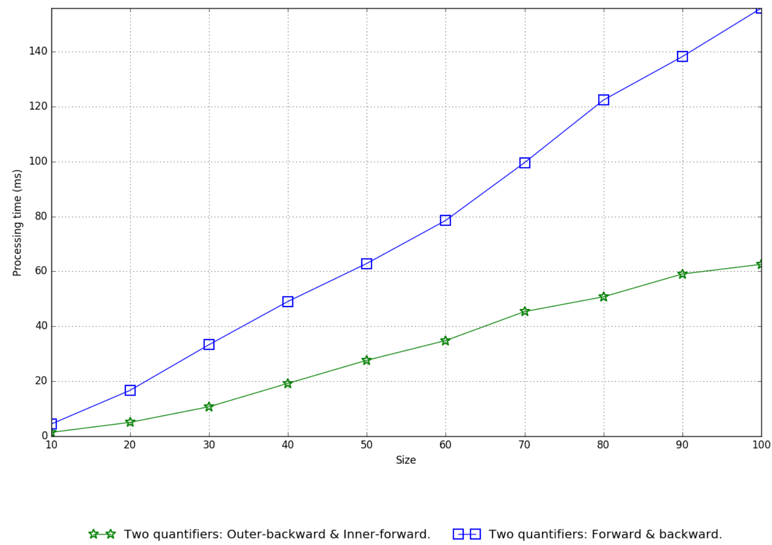

Figure 6: Processing time per message: 2 quantifiers.

4.3 Memory Usage

In this section we show the total memory usage of the specifications. As figure 7 shows all the specifications except the ones with two quantifiers (with the outer one looking backward and the inner one looking forward respectively both looking backward and forward) have more or less the same memory usage regardless of the window size. For memory usage the effective factor is the history we need to keep in order to evaluate the specification (windows size and search direction). For most cases, the memory reserved by the process is enough to satisfy the history needs. However, in the case of two quantifiers where the outer quantifier is backward looking, the needed memory increases significantly, as the outer needs to wait for the inner to be finished to continue evaluation; it needs to keep all the evaluated instances up to the lower range bound such that with the arrival of every new message the history increases.

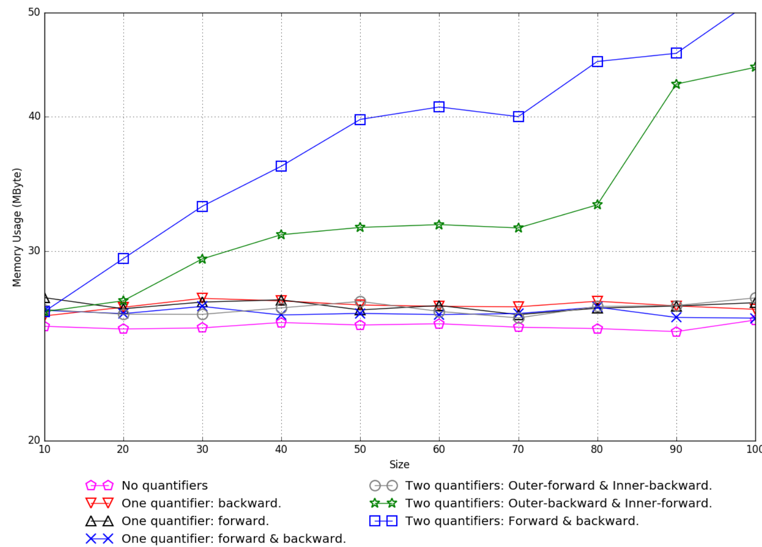

Figure 7: Memory usage: 0 and 1 quantifier.

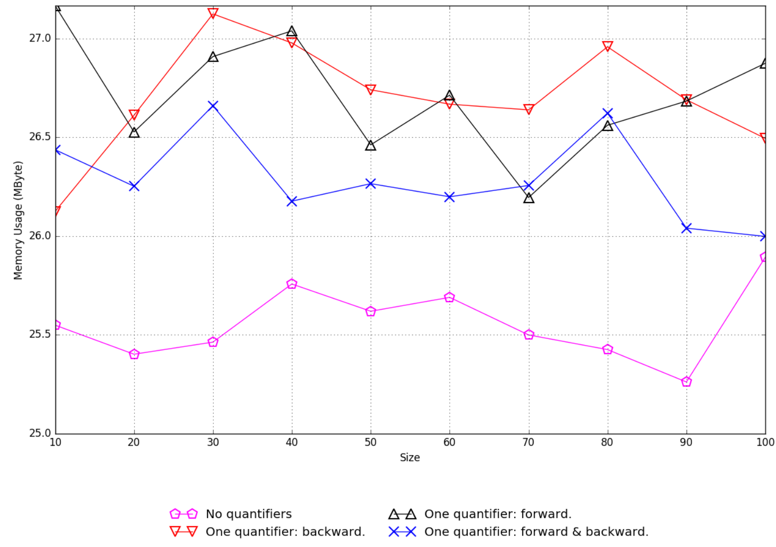

Figure 8: Memory usage: 0 and 1 quantifier.

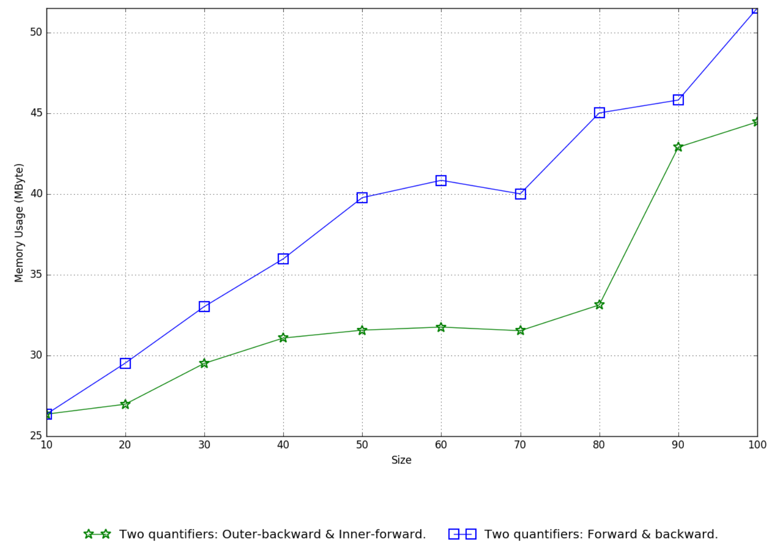

Figure 9: Memory usage: 2 quantifiers.

5 Analysis

In this section we describe the analysis of the results. We used a polynomial interpolation function to plot a smooth curves and polynomial fitting function to forecast the measurements values. The polynomial degree is chosen based on the level of nested quantifier **L**. For the specifications with no quantifiers we chose polynomial degree 0, for the specification with one quantifier we chose polynomial degree 1 and for the specification with two quantifiers we chose polynomial degree 2. In each plot presented in this section, the blue line represents the polynomial fitting curve, the red dots represent actual measurements, and the blue shadow on the side of the curve represents the standard deviation ($\pm 1\sigma$). Using the benchmarks, we capture measurements to window size **S** 100 and using the fitting function we forecast the values up to window size **S** 200.

5.1 Number of Instances

In this section we show the analysis for number of instances. Figures 10 - 16 show the fitting curve for all the specifications used in the benchmarks. All the curves show the expected result.

5.2 Processing Time

In this section we show the analysis for Processing time. Figures 17 - 23 show the fitting curve for all the specifications used in the benchmarks. Figure 17 shows the fitting curve of the specification with no quantifier, however, as the plot shows, we could not find the suitable coefficient for polynomial fitting function, for this plot we use degree zero because the specification does not use any quantifier, the resulted function is a line, however, since the reading of the CPU has a lot of fluctuations, the fitting function was not able produce a line which go through the all measurements values, the value of standard deviation is quiet high which indicate that we cannot trust the predicted values. Figures 18, 19 and 20 show smooth curves with slight increase in the standard deviation. Figures 21, 22 and 23 show also smooth curves, however, the standard deviation values increase significantly as we go far from the last actual value which make it hard to trust the forecast for higher values.

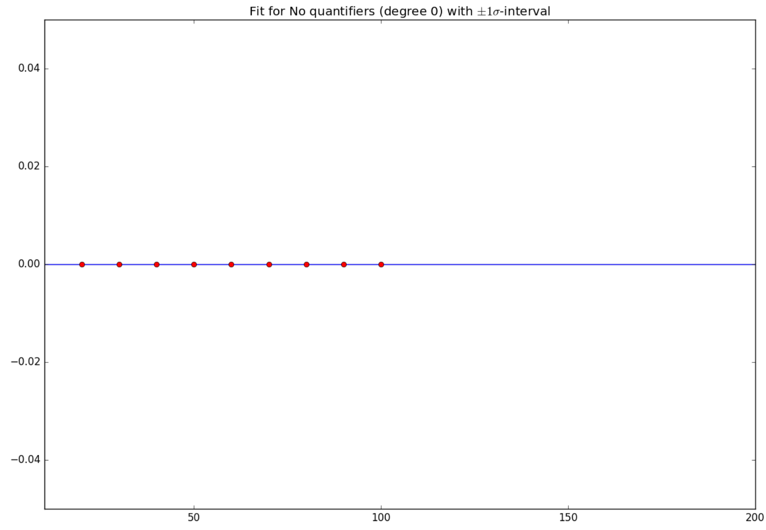

Figure 10: Number of instances fitting: No quantifiers.

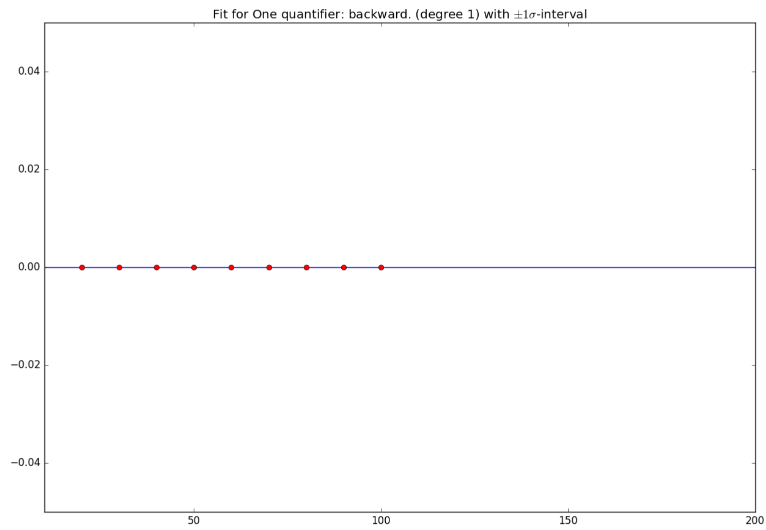

Figure 11: Number of instances fitting: One quantifier with backward looking.

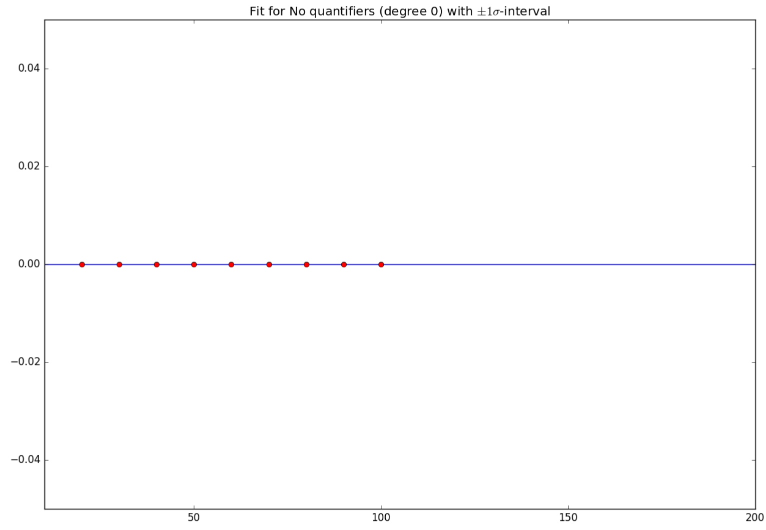

Figure 12: Number of instances fitting: One quantifier with forward looking.

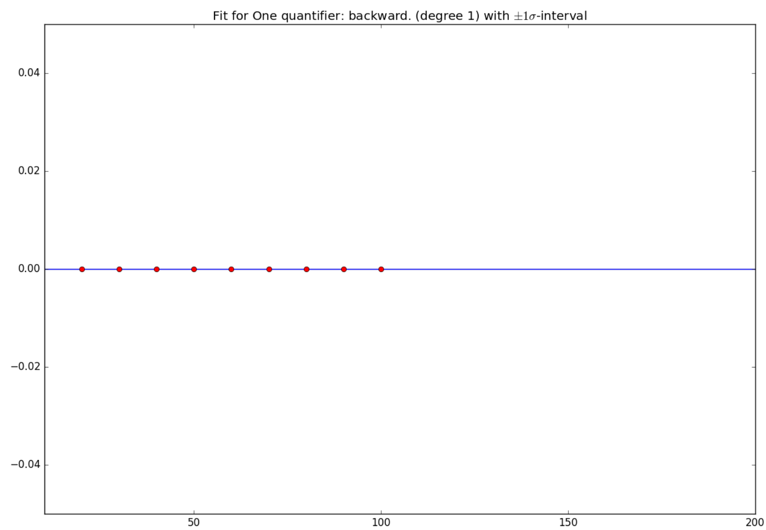

Figure 13: Number of instances fitting: One quantifier with backward and forward looking.

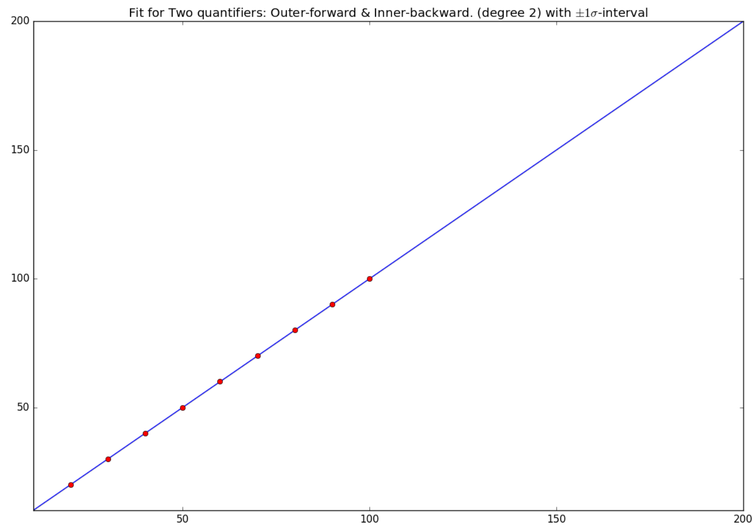

Figure 14: Number of instances fitting: Two quantifiers which the outer one with forward looking and the inner one with backward looking.

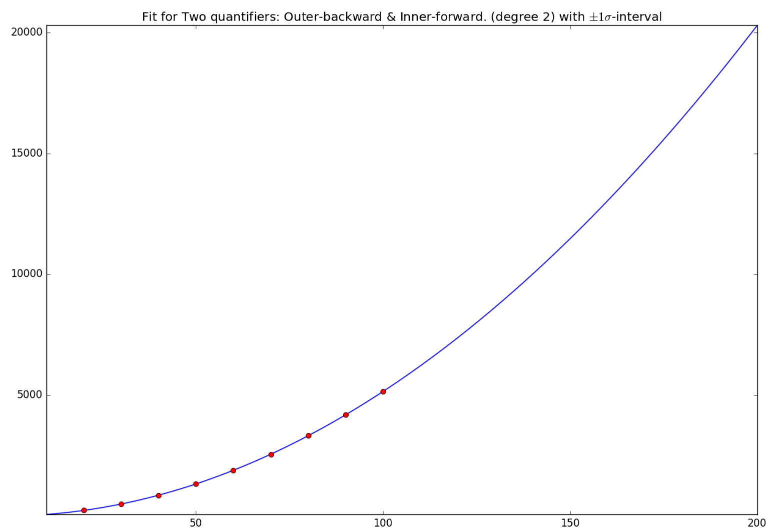

Figure 15: Number of instances fitting: Two quantifiers which the outer one with backward looking and the inner with one forward looking.

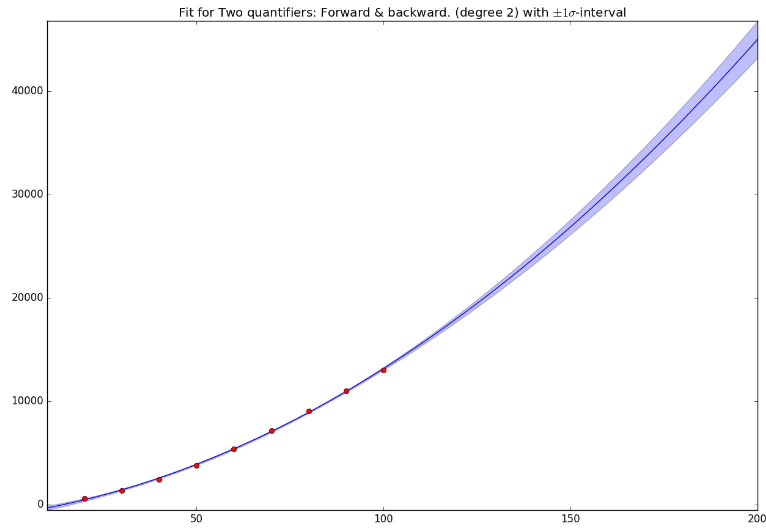

Figure 16: Number of instances fitting: Two quantifiers which both with forward and backward looking.

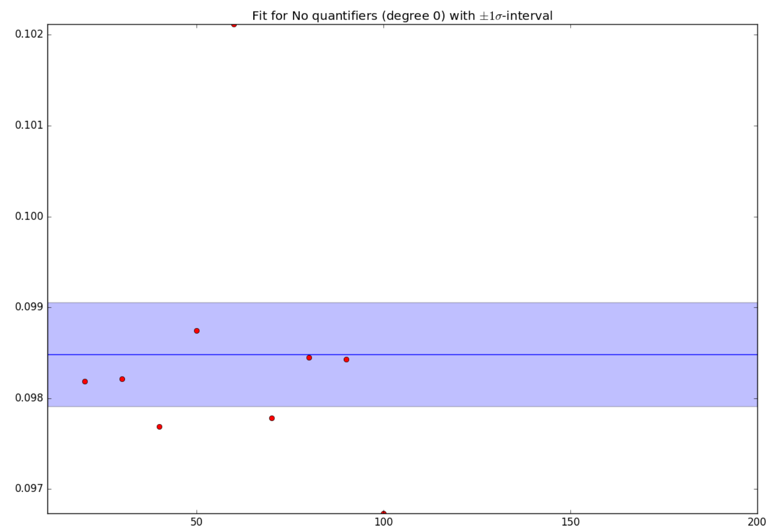

Figure 17: Processing time fitting: No quantifiers.

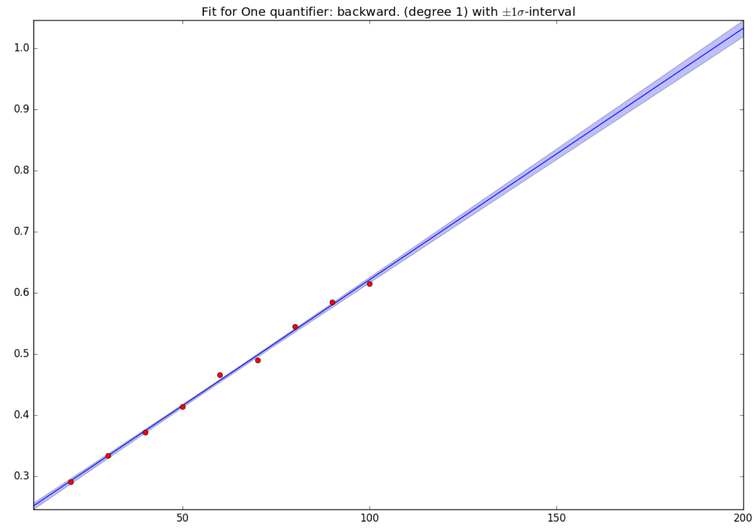

Figure 18: Processing time fitting: One quantifier with backward looking.

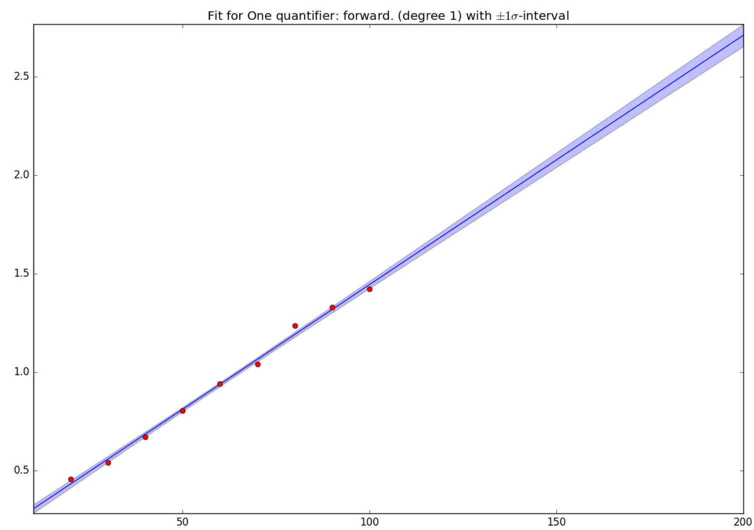

Figure 19: Processing time fitting: One quantifier with forward looking.

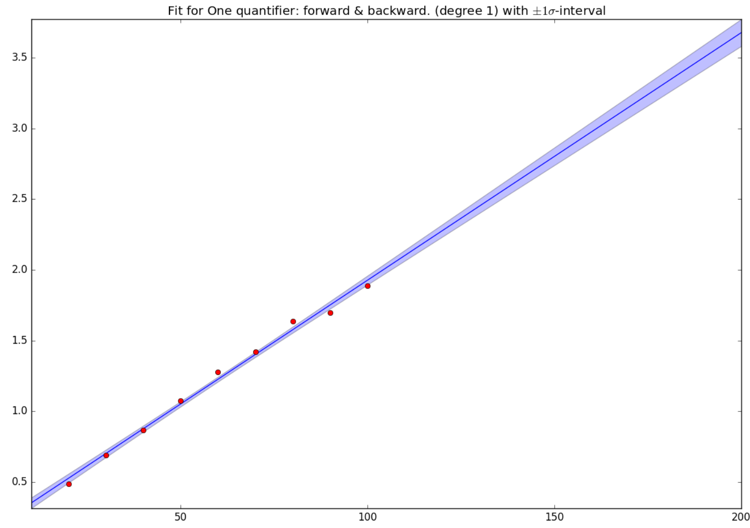

Figure 20: Processing time fitting: One quantifier with backward and forward looking.

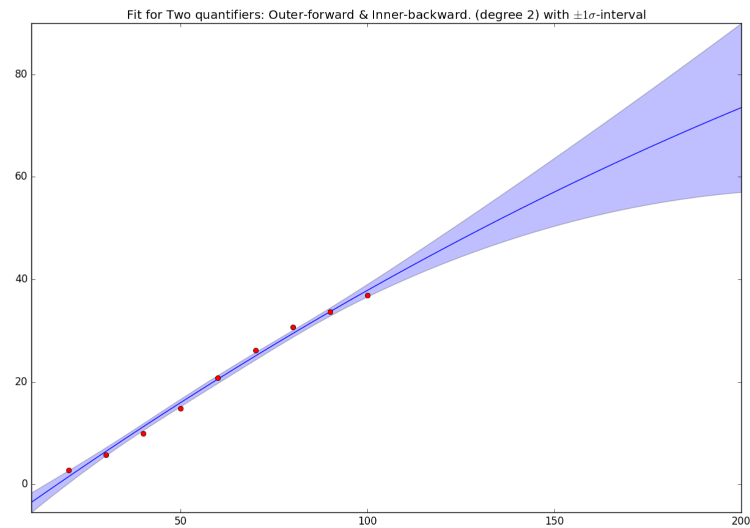

Figure 21: Processing time fitting: Two quantifiers which the outer one with forward looking and the inner one with backward looking.

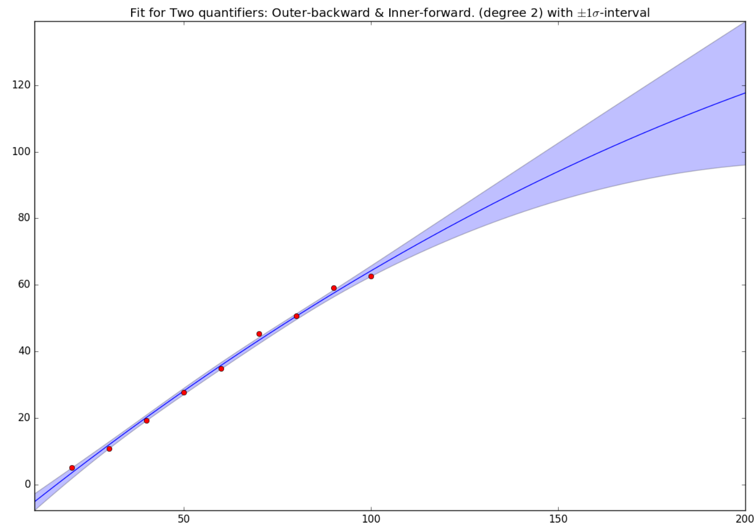

Figure 22: Processing time fitting: Two quantifiers which the outer one with backward looking and the inner with one forward looking.

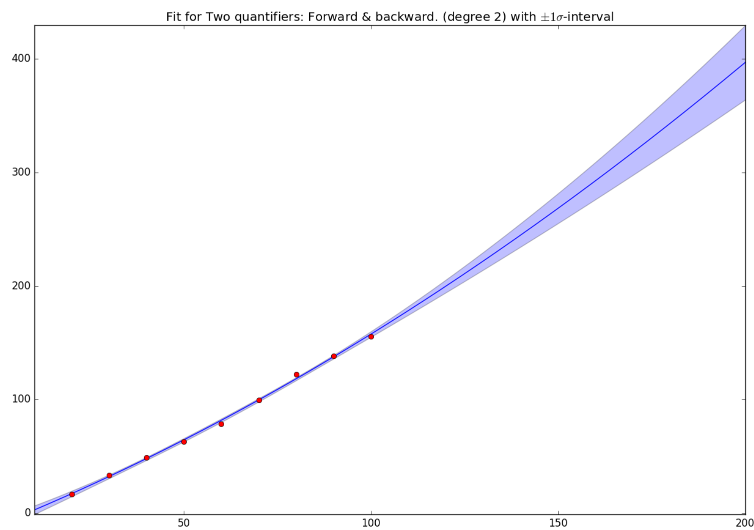

Figure 23: Processing time fitting: Two quantifiers which both with forward and backward looking.

6 Conclusions

In this report we presented the benchmarks and performance analysis for the LogicGuard framework. The objective of these benchmarks is to determine the performance of monitor evaluation in the LogicGuard framework. We used an artificial data set produced using the the built-in random number generator. We used a set of specification designed to stress test the framework in order to capture the performance limits of the framework. We defined a set of parameters to control the generated specification, which include quantifier depth, search direction, window size and delay. We presented the results in terms of number of instances, CPU usage and memory usage, as we saw in the result section, the measurements are within expectation. Furthermore, we preformed data analytical methods using polynomial function to plot smooth curves of the results and to forecast the measurements for higher window size.

References

[1] D. M. Cerna, W. Schreiner, and T. Kutsia. Predicting Space Requirements for a Stream Monitor Specification Language. In Y. Falcone and C. Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, pages 135–151. Springer International Publishing, September 2016.

[2] T. Kutsia and W. Schreiner. Verifying the Soundness of Resource Analysis for LogicGuard Monitors (Revised Version) . RISC Report Series 14-08, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, 2014.

[3] W. Schreiner, T. Kutsia, D. Cerna, M. Krieger, B. Ahmad, H. Otto, M. Rummerstorfer, and T. Gössl. The LogicGuard Stream Monitor Specification Language Tutorial and Reference Manual. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, October 2015.

[4] W. Schreiner, T. Kutsia, M. Krieger, A. Bashar, H. Otto, and M. Rummerstorfer. Monitoring Network Traffic by Predicate Logic. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, September 2014.

Appendix A Specifications

In this section we list all the specifications used for this benchmarks. Please notice that the values of the ranges illustrated here are not the ones we actually have used (it is specified as 1 here only for the correctness of the specification syntax). The actual values depends on the Size (S) and Delay (D) for each benchmark. In LogicGuard the range value determine the position of the message in the stream, so if the size is 10 and the delay is 100, the value of the range is 1000.

A.1 No Quantifiers

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x : AlwaysTrue(@x);
```

A.2 One Quantifier: Backward Looking

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x-1 <=# _ <= x: AlwaysTrue(@x);
```

A.3 One Quantifier: Forward Looking

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x <= _ <=# x+1 : AlwaysTrue(@x);
```

A.4 One Quantifier: Backward and Forward Looking

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x-1 <=# _ <=# x+1: AlwaysTrue(@x);
```

A.5 Two Quantifiers: Outer one is forward looking and inner backward looking

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x <= _ <=# x+1 :
     forall<IP> z with y-1 <=# _ <= y: AlwaysTrue(@x);
```

A.6 Two Quantifiers: Outer one is backward looking and the inner one is forward looking

```
type int;

stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x-1 <=# _ <= x:
      forall<IP> z with y <= _ <=# y+1: AlwaysTrue(@x);
```

A.7 Two Quantifiers: backward and forward looking

```
type int;
```

```
stream<int> IP;

logical AlwaysTrue(value<int> x);

monitor<IP> M =
  monitor<IP> x :
    forall<IP> y with x-1 <=# _ <=# x+1 :
     forall<IP> z with y-1 <=# _ <=# y+1 : AlwaysTrue(@x);
```

Appendix B External functions

```
public static bool AlwaysTrue(object m)
{
    return m != null;
}
```