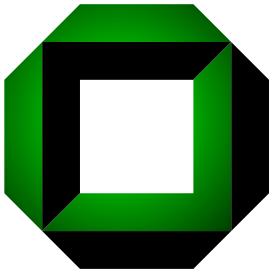

Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

Fakultät für Informatik
Institut für Programmstrukturen
und Datenorganisation
Lehrstuhl Prof. Goos

# On Improvements of the Varró Benchmark for Graph Transformation Tools

Rubino Geiß  Moritz Kroll

December 5, 2007

Technical Report 2007-7
ISSN 1432-7864

# ABSTRACT

In 2004 G. Varró, A. Schürr, and D. Varró proposed the first graph transformation benchmark ever. Varró et al. also published results for tools like AGG, PROGRES, and Fujaba. While repeating some of his measurements, we were not able to reproduce his figures and findings. This paper reports our suggestions for improving the so-called Varró benchmark.

# CONTENTS

# CHAPTER 1

# INTRODUCTION AND PROBLEM

In 2004[1] the first graph transformation benchmark was proposed by G. Varró, A. Schürr, and D. Varró [VSV05a]. Within a year G. and D. Varró et al. also published performance figures for tools like AGG, PROGRES, and FUJABA in two articles [VFV05, VSV05b] as well as on a web page [Var05]. This web page includes a detailed test chart. Additionally, the source codes for the measurements of the contemplated tools are available there. We reused some of the original figures of Varró in our own publications [GBG+06, Kro07]. We reprint these results in Figure 1.1; Table 1.1 contains the details for Figure 1.1 as well as further results. Figure 1.1 contains data points for an improved version of GRGEN(SP) that was not available for the ICGT 2006 proceedings [GBG+06], but was presented at the conference. We measured AGG on our own (see section 1.1.4). Besides adding measurements for our tool GRGEN [BG07, Gei07] we rescaled Varró's figures by a small correction factor[2] to match the original figures of Varró due to different hardware.

We recently tried to repeat some of the measurements of Varró et. al. [VFV05, VSV05b], especially those for FUJABA, on our own. Surprisingly, we failed in reproducing his figures, at least by orders of magnitude. This obviously cannot be explained by different hardware, since we used roughly comparable computers. Therefore we examined the experimental design and setup of Varró closely in order to explain the found discrepancies (see this chapter). Chapter 2 presents our results of the improved Varró benchmark.

[1] Depending on the publication dates you could also say 2005.

[2] To reuse Varró's results we multiplied his figures by 0.68 which is the speed difference of both processors according to the SPEC organization [Sta05]. Whether this is the correct factor or not, is beyond our scope. It simply does not matter if its correct value is, e.g. 0.33 or 3.0, because we talk about several orders of magnitude and even different complexity classes.

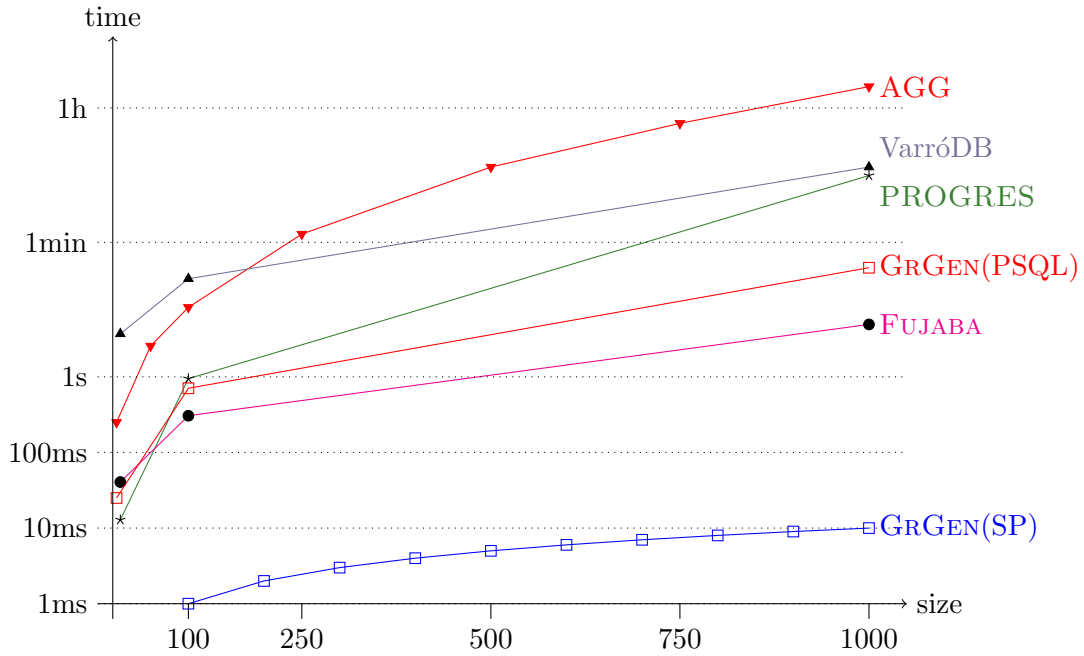


Figure 1.1: Running times of STS mutex benchmark (multiplicity optimizations off, parameter passing off, simultaneous execution off; for parameter details see [VSV05a])

Table 1.1: Running times for several of the Varró benchmarks (in milliseconds)

| Benchmark → | STS | | | ALAP | | | ALAP simult. | | | LTS |
|---|---|---|---|---|---|---|---|---|---|---|
| Tool ↓ | 10 | 100 | 1000 | 10 | 100 | 1000 | 10 | 100 | 1000 | 1000,1 |
| PROGRES | 12 | 946 | 459,000 | 21 | 1,267 | 610,600 | 8 | 471 | 2,361 | 942,100 |
| AGG | 330 | 8,300 | 6,881,000 | 270 | 8,027 | 13,654,000 | – | – | – | $> 10^7$ |
| Fujaba | 40 | 305 | 4,927 | 32 | 203 | 2,821 | 20 | 69 | 344 | 3,875 |
| VarróDB | 4,697 | 19,825 | 593,500 | 893 | 14,088 | 596,800 | 153 | 537 | 3,130 | 593,200 |
| GrGen(PSQL) | 30 | 760 | 27,715 | 24 | 1,180 | 406,000 | – | – | – | 96,486 |
| GrGen(SP) | $< 1$ | 1 | 10 | $< 1$ | 1 | 11 | $< 1$ | $< 1$ | 9 | 21 |

## 1.1 The Published Test Charts

The webpage [Var05] of G. Varró contains several test charts (log files) consisting of the raw data acquired during measurements. The structure of these log files is similar for all tools, but unfortunately not identical. In this section, we'll take a look at the structure of these log files to give clues on how to interpret the data.

### 1.1.1 What has been Measured?

There are three different mutex benchmarks defined by Varró: STS (short transformation sequence), ALAP (as long as possible), and LTS (long transformation sequence). These benchmarks can be modified to specifically test the impact of some optimizations: PP (parameter passing between rule applications), par (parallel matching and execution of graph updates), and one/many (indicating that multiplicity optimizations are on or off, respectively). Varró suggests using only the following combinations: STSone, STSonePP, STSmany, STSmanyPP, ALAP, ALAPpar, LTS (cf. [VSV05b]). Moreover his benchmarks have one (or two in case of LTS) size parameters.

In the following list we give hints on abnormalities[3] of the log files with respect to the above classification (the next subsections elaborate on how to read these log files):

- AGG is neither capable of parallel rewriting nor has multiplicity optimizations, so these benchmark variants are left out.
- The PROGRES log files for ALAP, ALAPpar, and LTS differ because they contain the timestamps for the applications of the initial `newRule`, whereas the log files for AGG and FUJABA only contain the "payload" rule applications.
- ALAPpar differs for FUJABA and PROGRES due to issues regarding different modes of parallel rewriting of both tools.
- We do not consider the measurements for the Varró DB approach, because it is not a publicly available tool.

### 1.1.2 The STS Benchmark

The STS benchmark is one of the mutex benchmarks used by Varró (cf. section 1.1.1). We use this benchmark to illustrate our explanations. The STS benchmark of size $n$ executes the following sequence of rule applications:

- apply `newRule` $n-2$-times
- apply `mountRule` once
- apply `requestRule` $n$-times
- repeat the following sequence of rule applications $n$-times:
  - apply `takeRule`
  - apply `releaseRule`
  - apply `giveRule`

This can be concisely specified by the following graph rewrite sequence [BG07]:

```
newRule[n-2] & mountRule & requestRule[n] & (takeRule & releaseRule & giveRule)[n]
```

[3] These are not errors per se; this rather tries to prevent some potential misunderstandings.

### 1.1.3 The Log Files

The log files for the three tools (AGG, Fujaba, and PROGRES) consist of three lines per single rule application. The logs for the Java based tools (AGG and Fujaba) are prepared with log4j [Fou06]. For the PROGRES tool — with its generated matcher in C code — Varró used custom code for logging (see section 1.2.2). Listings 1.1, 1.2, and 1.3 are excerpts of a run of the STSmany benchmark of size $n = 10$ for the discussed tools. Table 1.2 describes the content of each column of the AGG (Listing 1.1) and Fujaba (Listing 1.2) log files, while Table 1.3 does the analogous for PROGRES(Listing 1.3) log files. Different column separators like "-", ":", and spaces are ignored for conciseness.

Please note, that even if you consider all log files for PROGRES, you will not find any zero directly after the decimal point (like in 1110237924.096410) and sometimes the numbers after the decimal point are less than 6 digits long. This is erroneous (described in section 1.2.2).

### 1.1.4 Incomplete Log Files

Some log files on Varró's web page [Var05] are truncated, presumably due to prohibitive long running time of the according benchmarks. Only AGG is affected. In the publications of the benchmark results one can find corresponding interleaves [VFV05, VSV05b]. The authors state that they aborted measurements with long running times. In the following we give a list of all discontinued log or missing files:

- STSmany1000 for AGG (see Listing 1.4)
- STSmanyPP1000 for AGG
- ALAP1000 for AGG
- STSone10, STSone100, STSone1000, ALAPpar10, ALAPpar100, and ALAPpar1000 for AGG are left out because AGG does not support the necessary features.

Due to these discontinued log files for AGG, we did our own measurements to get all data points for AGG.

### 1.1.5 The Publications of Benchmark Results

There are two publications made by the original authors containing results of the Varró benchmark [VFV05, VSV05b]. In this section we paraphrase our thoughts on how to relate the figures presented there with the raw test charts and source code published on the web site [Var05].

Table 1.2: Description of a log file row for the Java-based tools

| Column | Example from Listing 1.1 | Description |
|---|---|---|
| 1 | `618` | Time in milliseconds (ms, $10^{-3}$sec) since the first log entry; this value is computed internally by log4j. |
| 2 | `[main]` | *Irrelevant.* The name of the thread calling log4j. This is `main` for all files. |
| 3 | `DEBUG` | *Irrelevant.* The debug-level. This is `DEBUG` for all files. |
| 4 | `hu. ... .STSmany` | *Irrelevant.* The fully qualified name of the class containing the method calling log4j. |
| 5 | `giveRule` | The name of the current rule. |
| 6 | `update` | The part of the rule application that has just been completed: init (initialization code), pattern matching (or pm for FUJABA), and update (update the graph i.e. do the rewrite; clean-up code is included). |
| 7 | `1110566155307449000` | The result of a call to the Java runtime library `System.nanoTime()`. The result is not interpretable in a straightforward manner (please see section 1.2.1). Albeit other possibilities, in this case this function counts the nanoseconds (ns, $10^{-9}$sec) since the UNIX epoch, roughly speaking. The values of column one and this column are incommensurable by definition (cf. section 1.2.1). |

Table 1.3: Description of a log file row for PROGRES

| Column | Example from Listing 1.3 | Description |
|---|---|---|
| 1 | `giveRule` | The name of the current rule. |
| 2 | `update` | The part of the rule application that has just been completed: init (initialization code), pm (pattern matching), and update (update the graph i.e. do the rewrite; clean-up code is included). |
| 3 | `1110237924.983845` | The seconds since the UNIX epoch. The numbers after the decimal point should be the fractions of the second with microsecond ($\mu$s, $10^{-6}$sec) resolution (note the difference between resolution and precision). Due to erroneous implementation this is not true (see section 1.2.2). |

Listing 1.1: Excerpt (beginning and end) of the STSmany10 log file of AGG

```
0 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - newRule: init: 1110566154687070000
58 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - newRule: pattern matching: 1110566154747309000
105 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - newRule: update: 1110566154794235000
105 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - newRule: init: 1110566154794426000
...
606 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - takeRule: init: 1110566155295635000
607 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - takeRule: pattern matching: 1110566155296755000
610 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - takeRule: update: 1110566155299266000
610 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - releaseRule: init: 1110566155299377000
613 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - releaseRule: pattern matching: 1110566155302524000
614 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - releaseRule: update: 1110566155303407000
614 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - giveRule: init: 1110566155303486000
617 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - giveRule: pattern matching: 1110566155306527000
618 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - giveRule: update: 1110566155307449000
```

Listing 1.2: Excerpt (beginning and end) of the STSmany10 log file of Fujaba

```
0 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - newRule: init: 1110559421066197000
19 [main] DEBUG many.Container  - newRule: pm: 1110559421088969000
19 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - newRule: update: 1110559421089363000
19 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - newRule: init: 1110559421089512000
...
55 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - takeRule: init: 1110559421125921000
56 [main] DEBUG many.Container  - takeRule: pm: 1110559421126085000
56 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - takeRule: update: 1110559421126222000
56 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - releaseRule: init: 1110559421126327000
56 [main] DEBUG many.Container  - releaseRule: pm: 1110559421126454000
56 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - releaseRule: update: 1110559421126591000
56 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - giveRule: init: 1110559421126693000
56 [main] DEBUG many.Container  - giveRule: pm: 1110559421126825000
56 [main] DEBUG hu.bme.cs.benchmark.mutex.fujaba.STSmany  - giveRule: update: 1110559421126946000
```

Listing 1.3: Excerpt (beginning and end) of the STSmany10 log file of PROGRES

```
newRule : init : 1110237924.964010
newRule : pm : 1110237924.965726
newRule : update : 1110237924.965878
newRule : init : 1110237924.965983
newRule : init : 1110237924.966637
...
takeRule : init : 1110237924.983052
takeRule : pm : 1110237924.983293
takeRule : update : 1110237924.983380
releaseRule : init : 1110237924.983388
releaseRule : pm : 1110237924.983583
releaseRule : update : 1110237924.983653
giveRule : init : 1110237924.983660
giveRule : pm : 1110237924.983773
giveRule : update : 1110237924.983845
```

Listing 1.4: Excerpt (the end) of the truncated STSmany1000 log file of AGG

```
...
556991 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: pattern matching: 1110566742890247000
558757 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: update: 1110566744656603000
558758 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: init: 1110566744657621000
566944 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: pattern matching: 1110566752843949000
569278 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: update: 1110566755177695000
569279 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: init: 1110566755178707000
573915 [main] DEBUG hu.bme.cs.benchmark.mutex.agg.STSmany  - requestRule: pattern matching: 1110566759814578000
```

## 1.2 The Implementation of the Chronometry

### 1.2.1 Time Measurement for FUJABA and AGG

The measurements for FUJABA and AGG use log4j. The log files contain three timestamps for each rule application. Each timestamp is made after the according sub-step of the rule application (init, pattern matching, and update) has been completed. The timestamps are dumped to the console. It is unclear from the available log files and source code if the dump was redirected to a file or has just been sent to the console (and has been put to a file from the buffer of the console afterwards) or even both. This setup is arguable in general, especially its basic design (see section 1.3).

For each row there are two times given. The first in column 1 and the second in column 7 (see section 1.1.3): Firstly, the time in milliseconds since the first log entry; this value is computed internally by log4j. Secondly, the result of a call to the Java runtime library `System.nanoTime()`. This value has a nanosecond resolution, but as stated by the documentation of the method no precision whatsoever is guaranteed (see the quotation below). As stated by the documentation `System.nanoTime()` "[...] is not related to any other notion of system or wall-clock time.", so both timing methods yield incommensurable outputs. It is unclear from the available log files and source code which time source has been used for the publications [VFV05, VSV05b] (see also section 1.1.5).

> *Comment from G. Varró [Var07]:* In the original measurements, the timestamp dump was redirected to a file and the nanosecond values were used for the publications.

The following is quoted for the method `nanoTime` of the class `System` from the documentation of the Java runtime library [Mic07]:

> **public static long** nanoTime()
>
> Returns the current value of the most precise available system timer, in nanoseconds.
>
> This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years ($2^{63}$ nanoseconds) will not accurately compute elapsed time due to numerical overflow.
>
> For example, to measure how long some code takes to execute:
>
> ```
> long startTime = System.nanoTime();
> // ... the code being measured ...
> long estimatedTime = System.nanoTime() - startTime;
> ```
>
> **Returns:** The current value of the system timer, in nanoseconds.
>
> **Since:** 1.5

### 1.2.2 Time Measurement for PROGRES

Listing 1.5 shows the unaltered routine from G. Varró's webpage [Var05] which does the time measurement for PROGRES. This routine and its usage is questionable for different reasons:

Listing 1.5: Time measurement for PROGRES

```
1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <time.h>
4  #include <unistd.h>
5
6  void print_time ()
7  {
8   struct timeval tv;
9   struct tm* ptm;
10  char time_string[40];
11  long milliseconds;
12
13  /* Obtain the time of day, and convert it to a tm struct. */
14  gettimeofday (&tv, NULL);
15  ptm = localtime (&tv.tv_sec);
16  /* Format the date and time, down to a single second. */
17  strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);
18  /* Compute milliseconds from microseconds. */
19  milliseconds = tv.tv_usec / 1000;
20  /* Print the formatted time, in seconds, followed by a decimal point
21     and the milliseconds. */
22  // printf ("%s.%03ld\n", time_string, milliseconds);
23  printf ("%ld.%ld\n", tv.tv_sec, tv.tv_usec);
24  // printf ("%ld%ld\n", tv.tv_sec, tv.tv_usec);
25  // printf ("%ld.%ld\n", ts.tv_sec, ts.tv_nsec);
26 }
```

1. The computation in line 15, 17, and 19 is superfluous. The rather costly call to the "string format date and time" routine will most certainly not be removed by any compiler optimizations. This forges the measurements more than necessary.

2. Line 23 is erroneous: The struct elements `tv_sec` and `tv_usec` both carry long[4] values. After the call to `gettimeofday` in line 14 the field `tv.tv_sec` contains the seconds since the epoch and `tv_usec` contains the microseconds ($\mu$s) since the beginning of the last second, i.e. $0 \leq$ `tv_usec` $< 1,000,000$ always holds. So for example, `tv.tv_sec = 12` and `tv.tv_usec = 9876` should be interpreted as 12.009876sec after the epoch and not as 12.9876sec. Therefore the output of `tv_usec` has to be padded with six "0"; line 23 specifies no padding at all. The correct code may look like:

   ```
   printf ("%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
   ```

   > *Comment from G. Varró [Var07]:* The published results handled the microsecond part in the same way as you mentioned, even if the printf code is not correct. (There was a post processing phase which performed the correct 0 padding.)

3. The use of fine grained timestamp logging for performance evaluation is arguable in principle (see section 1.3 for a detailed discussion).

[4] This is operating system dependent. For Suse Linux with kernel 2.6.11.4-21.17-smp on a 32-bit platform this holds.

## 1.3 Design of the Chronometry

In principle, there are four approaches to chronometry spanned by two independent features: *granularity* and *mode of accumulation*.

The *granularity* of measurement can reach from "a single hardware instruction" over "an inner loop" to whole programs. But obviously the finer the granularity is, the more invasive the measurements are. If we want to study the "natural" timing behavior of a real-time-system, it is obviously a bad idea to encapsulate every processor instruction with a costly call to some debugging library. Not only will the program execute orders of magnitude slower than without instrumentation, moreover, it will loose its specific performance characteristic: E.g. the cache is full of code and data belonging to the measurement, not to the original program. Also many compiler optimizations may not be applicable any more (inlining, re-use of expressions, ...). Thus, fine granularity is always achieved at the cost of losing the "natural" behavior; ultimately rendering the results useless. So, if we are interested in "natural" behavior, the granularity has to be as coarse as possible.

The other dimension of chronometry is the *mode of accumulation*: We can either use timestamps (before/after certain steps) or compute the elapsed time of a step directly (the difference between two timestamps). There is obviously no difference between the *timestamp* and the *elapsed time* approach for sufficient coarse granularity. The timestamp approach seems to be good at very fine granularity though — with elapsed times near or even below clock resolution. The elapsed time approach fails in such a situation. However, even if we are at the brink of the clock resolution (between two consecutive timestamps) we can always use bigger intervals leaving out some timestamps to calculate meaningful statistics. This seems very convincing, logical, and theoretically sound, but it is plainly *not* true in practice! Why?

- Timestamps have to be stored somewhere. Usually they are printed to the console or a file. Afterwards they can be post-processed to get the desired statistics.
- Storing the timestamps is costly. Moreover printing something to a console on any operating system or graphical user interface is magnitudes slower than storing it to the main memory.
- Consoles buffer the output with various strategies. At some point, every console (or operating system) inescapably freezes the printing process if this process "dumps all-too fast".
- The same is true for file systems but the effect is less dramatic.
- Computing the timestamp (which usually involves calling the kernel) is a very expensive operation. The running time of the payload can be hidden by this overhead.

Altogether, printing timestamps should not be used for granularities that dump several times per second or more. The most "reasonable" and less intrusive way to get timing results is to use high granularity and elapsed time mode. To this point the statements above — despite being common wisdom — are unproven; the next section will close this gap.

### 1.3.1 Timestamp Chronometry with `printf` Considered Harmful

We have implemented the four approaches and additionally quiet variants (storing the data in main memory instead of dumping it) of the two fine granularity approaches in small example programs:

1. fine granularity, timestamp (FGTS, Listing A.1)
2. fine granularity, elapsed time (FGET, Listing A.2)

Table 1.4: Running time (in microseconds) of the discussed chronometry approaches in combination with different output medium and payloads (values of the `step` variable)

| `steps` → | 10 | 100 | 1,000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|---|---|
| strategy ↓ medium → | disk (on xterm) | | | | | |
| CGET | 319 | 2284 | 21,943 | 218,135 | 2,197,019 | 23,118,140 |
| CGTS | 382 | 2313 | 22,027 | 218,203 | 2,196,090 | 23,113,267 |
| FGET | 51,225 | 53,601 | 73,292 | 269,497 | 2,260,022 | 23,289,843 |
| FGTS | 64,767 | 66,974 | 94,388 | 367,841 | 3,016,865 | 25,383,131 |
| QFGET | 51,377 | 53,813 | 72,821 | 269,473 | 2,250,387 | 23,014,723 |
| QFGTS | 51,372 | 55,310 | 82,023 | 361,693 | 3,017,942 | 25,233,317 |
| strategy ↓ medium → | xterm | | | | | |
| CGET | 330 | 2,285 | 23,238 | 225,727 | 2,198,134 | 23,117,591 |
| CGTS | 377 | 2,308 | 23,339 | 219,583 | 2,196,862 | 23,093,594 |
| FGET | 51,214 | 55,051 | 73,988 | 321,182 | 2,273,723 | 23,167,904 |
| FGTS | 1,315,086 | 1,312,531 | 1,314,823 | 2,532,553 | 3,953,345 | 26,130,123 |
| QFGET | 51,836 | 53,940 | 73,771 | 270,665 | 2,252,275 | 23,010,462 |
| QFGTS | 52,815 | 56,602 | 83,364 | 356,344 | 3,012,283 | 25,215,074 |
| strategy ↓ medium → | KDE Konsole | | | | | |
| CGET | 319 | 2,296 | 23,283 | 224,675 | 2,202,347 | 23,303,268 |
| CGTS | 374 | 2,298 | 23,426 | 219,755 | 2,201,963 | 23,286,486 |
| FGET | 51,436 | 61,188 | 73,686 | 317,245 | 2,444,776 | 23,590,659 |
| FGTS | 776,981 | 779,903 | 806,244 | 1,179,087 | 4,028,748 | 26,452,178 |
| QFGET | 51,564 | 53,932 | 73,471 | 271,288 | 2,252,662 | 23,003,765 |
| QFGTS | 52,804 | 56,679 | 82,648 | 356,500 | 3,012,686 | 25,354,412 |
| strategy ↓ medium → | GNOME terminal | | | | | |
| CGET | 330 | 2,284 | 23,061 | 218,635 | 2,189,709 | 22,607,555 |
| CGTS | 364 | 2,306 | 23,237 | 219,333 | 2,188,023 | 22,596,380 |
| FGET | 51,459 | 53,408 | 72,853 | 2,631,577 | 5,056,676 | 25,718,893 |
| FGTS | 1,882,485 | 1,882,690 | 1,911,790 | 3,858,690 | 6,632,817 | 28,665,439 |
| QFGET | 51,754 | 53,808 | 73,730 | 270,245 | 2,248,306 | 22,928,958 |
| QFGTS | 52,545 | 56,688 | 82,552 | 355,797 | 3,005,662 | 25,057,223 |
| ideal running time | 231 | 2,311 | 23,118 | 231,181 | 2,311,814 | 23,118,140 |

3. coarse granularity, timestamp (CGTS, Listing A.3)
4. coarse granularity, elapsed time (CGET, Listing A.4)
5. quiet, fine granularity, timestamp (QFGTS, Listing A.5)
6. quiet, fine granularity, elapsed time (QFGET, Listing A.6)

The "payload" is a single loop accumulating a volatile variable:

```
for(j=0; j<steps; j++) cnt += j;
```

We can simulate different computation lengths (running times of the payload) by changing the value of the `steps` variable. QFGTS (Listing A.5) gets the timestamps and stores them in memory. After computing all payload the timestamps are printed. QFGET (Listing A.6) is a special case of *fine granularity, elapsed time* which does no output at single (fine grained) steps. It rather prints the sum of the accumulated differences at the end of the program.

We also want to test the influence of the output medium on the measuring error. Therefore we use four different methods:

1. output to disk started on xterm, KDE Konsole, and GNOME terminal
2. output to xterm (and disk)
3. output to KDE Konsole (and disk)
4. output to GNOME terminal (and disk)

The complete source code of our measurements can be found in appendix A, and the resulting raw test charts can be found on our web site `www.grgen.net/chronometry`. All measurements were performed on our benchmark computer (see appendix B) and repeated 30 times.

Table 1.4 shows the median of the (internally measured) running time of the six example programs. For the examples using timestamp chronometry (TS), we subtracted the first and the last timestamp to get the running time. Just like the method used by Varró, this produces an initial error, because the code executed before the fist timestamp is not measured. In case of elapsed time chronometry (ET), the elapsed times for the fine granularity examples (FG) are summarized to get the running time. We repeated each payload loop 15,000 times.

To get a clearer picture, we want to compute the running time of a single step of the payload loop, i.e. something like `cnt += j` in combination with the overhead of the loop such as computing `j < steps`, the conditional jump, and incrementing the loop variable `j++`. Please note that the core of the payload `cnt += j` cannot be optimized away by any optimizing compiler, because of the volatile definition of the global variable `cnt`.

Our most exact measurements (the CGTS and CGET with `steps` = 1,000,000) suggest that the execution of the payload's core statements needs 1.54ns. The Figures 1.2–1.11 show the relative error of the running time of such a single step of the payload loop measured by different approaches and setups. In every figure we marked the zone of no measurement deviation (relative to 1.54ns) by a dotted red line. With 1.54ns per single payload loop step we can calculate the theoretical running time of the different payloads. To relate the `steps`-variable with the running time caused by the payload, we inserted the dashed blue vertical lines representing 10ns, 1$\mu$s, and 100$\mu$s running time.

### 1.3.2 Interpretation

If there is no measurement deviation at all, we would see all data points on the red dotted line in the Figures 1.2–1.11. Obviously this is not true. Particularly considering the logarithmic scaled time (y)-axis, we can see measurement deviation of factors up to several hundreds of thousands.

The measurement deviation can vary just by changing the kind of terminal (xterm, GNOME terminal, or KDE Konsole; see appendix B) by at least a factor of two (see Figure 1.2, 1.3, and 1.4). Moreover, the characteristics of the increase of the measurement deviation is different. However, using the hard disk and not the terminal for output, changes the situation dramatically (FGTS/disk, see Figure 1.5): The measurement deviation reduces to about 5,000 but this measurement is obviously still useless—at least for payloads below 100$\mu$s. Please note that Fujaba and GrGen can perform a single rewrite step much faster than that; nevertheless Varró used this measurement setup. Some confusing test charts of Varró even suggest that maybe some output to the console was made (repeating some of Varró's test on our machine showed a speedup by more then an order of magnitude).

Not performing any output (neither to disk nor to terminal) lowers the measurement deviation even more (QFGTS/disk, see Figure 1.6). The FGET/xterm is almost as exact as the disk variants of the FGTS examples. This suggests that ET is superior to TS because it does not time the asynchronous output routines of the GUI/operating system. The same supremacy of ET can be seen if we look at coarse grained chronometry (CGTS/disk vs. CGET/disk). The CG chronometry, especially the CGET variants, have (in contrast to all other variants) tolerable measurement deviations even if the payload is extremely small. Using the CGET/disk-method, we can even measure payloads as fast as 10ns, if we repeat it 15,000 times, with a relative error below 50%. If we increase the number of repetitions, we can even get below this relative error.

The influence of the kind of terminal even for `steps` = 1,000,000 is still measurable, but not significant anymore. The data points below the red dotted 100% line in Figure 1.11 (not possible in an ideal world) are due to reasons beyond the scope of this paper.

### 1.3.3 Varró's Analysis

A breakdown of the running time to single rule applications as shown by Varró et al. [VSV05b] cannot be done naive. Fast graph rewrite tools such as FUJABA, GRGEN, and GRGEN.NET need very little time (about $1\mu$s) for a typical match and rewrite step of the STS benchmark. A call to `gettimeofday` requires about the same time. So the running time of the graph rewrite step is covered up by the overhead of the measurement. Without special counteractive measures the generated figures are meaningless.

Note that on Linux, Java uses the `gettimeofday` system call for both `System.nanoTime()` and the `System.currentTimeMillis()` method. Thus the results received for `gettimeofday` and C apply to Java, too.

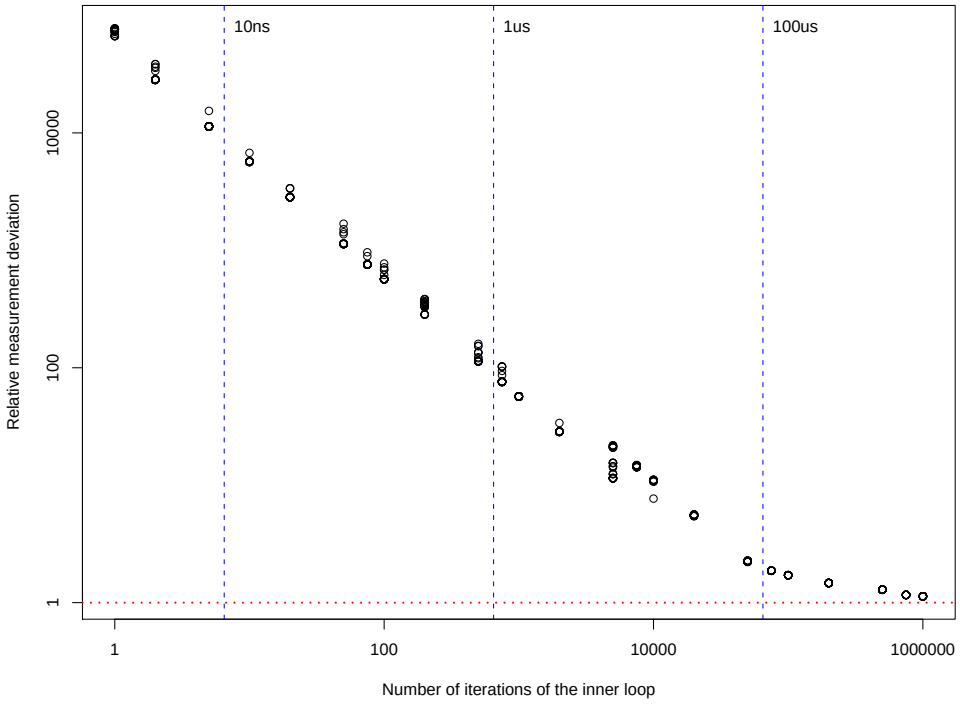


Figure 1.2: FGTS/xterm: Running time of a single step of the payload loop measured with fine granularity, timestamp chronometry, and with output on xterm and disk

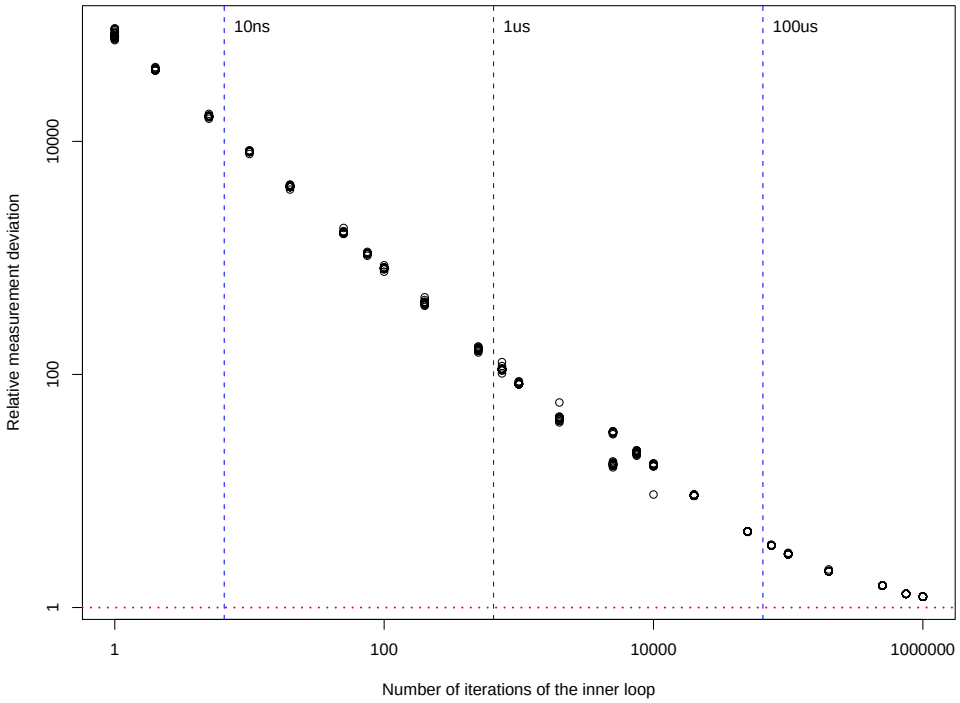


Figure 1.3: FGTS/GNOME: Running time of a single step of the payload loop measured with fine granularity, timestamp chronometry, and with output on GNOME terminal and disk

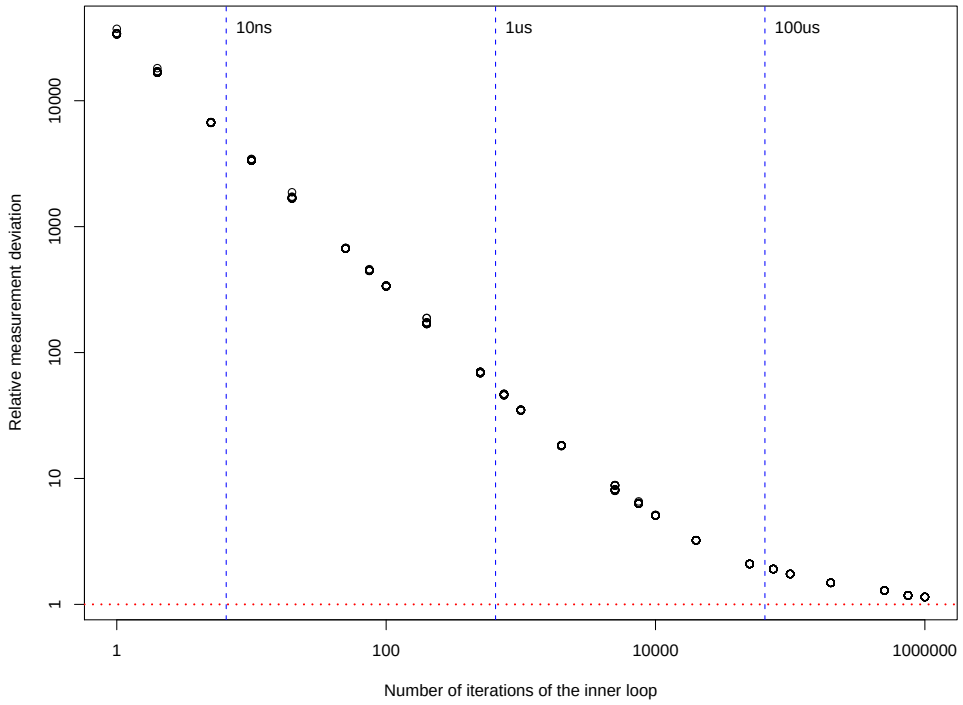


Figure 1.4: FGTS/KDE: Running time of a single step of the payload loop measured with fine granularity, timestamp chronometry, and with output on KDE Konsole and disk

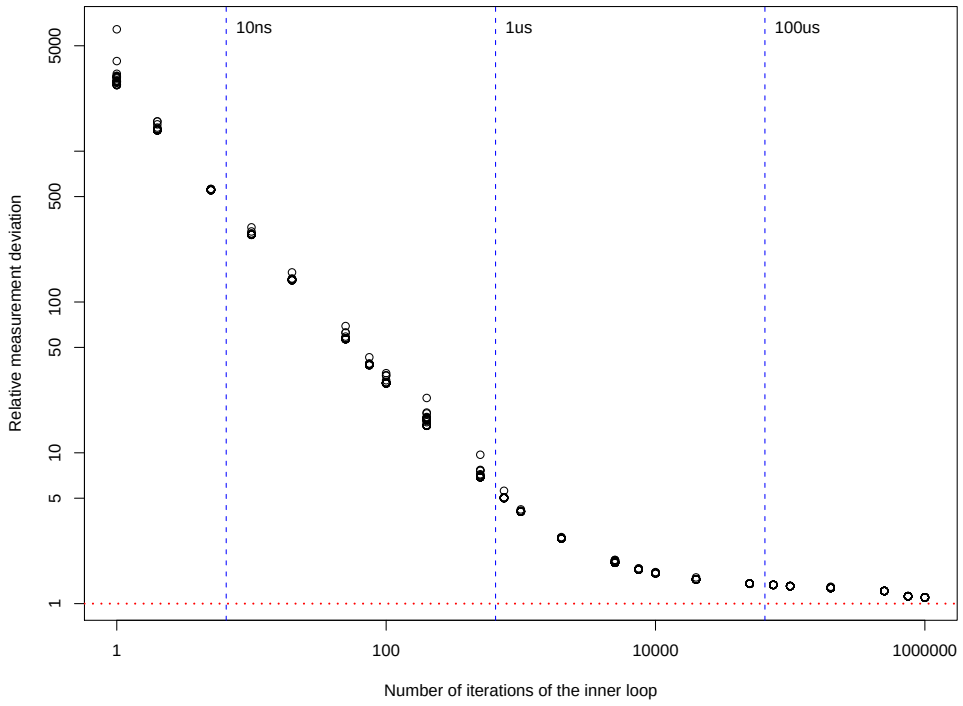


Figure 1.5: FGTS/disk (on xterm): Running time of a single step of the payload loop measured with fine granularity, timestamp chronometry, and with output on disk

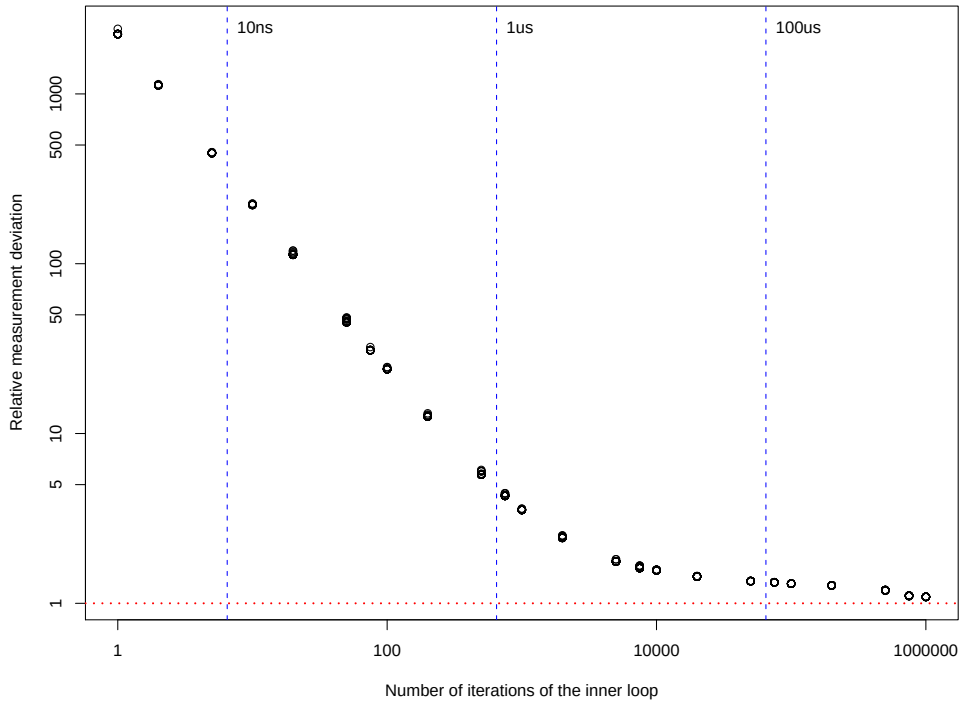


Figure 1.6: QFGTS/disk (on xterm): Running time of a single step of the payload loop measured with quiet, fine granularity, timestamp chronometry, and with output on disk

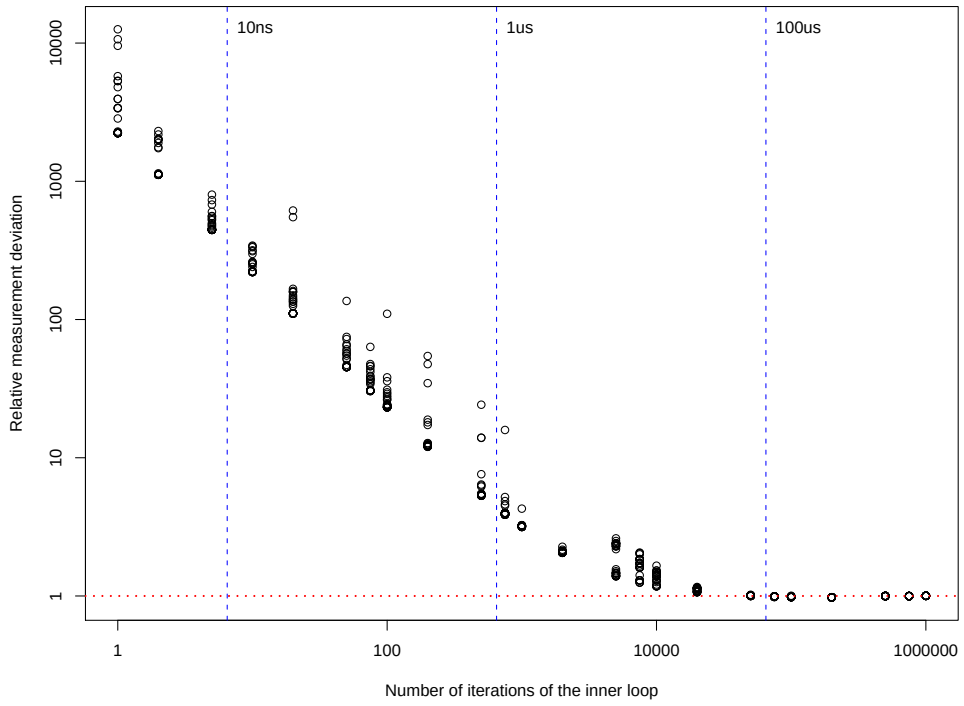


Figure 1.7: FGET/xterm: Running time of a single step of the payload loop measured with fine granularity, elapsed time chronometry, and with output on xterm and disk

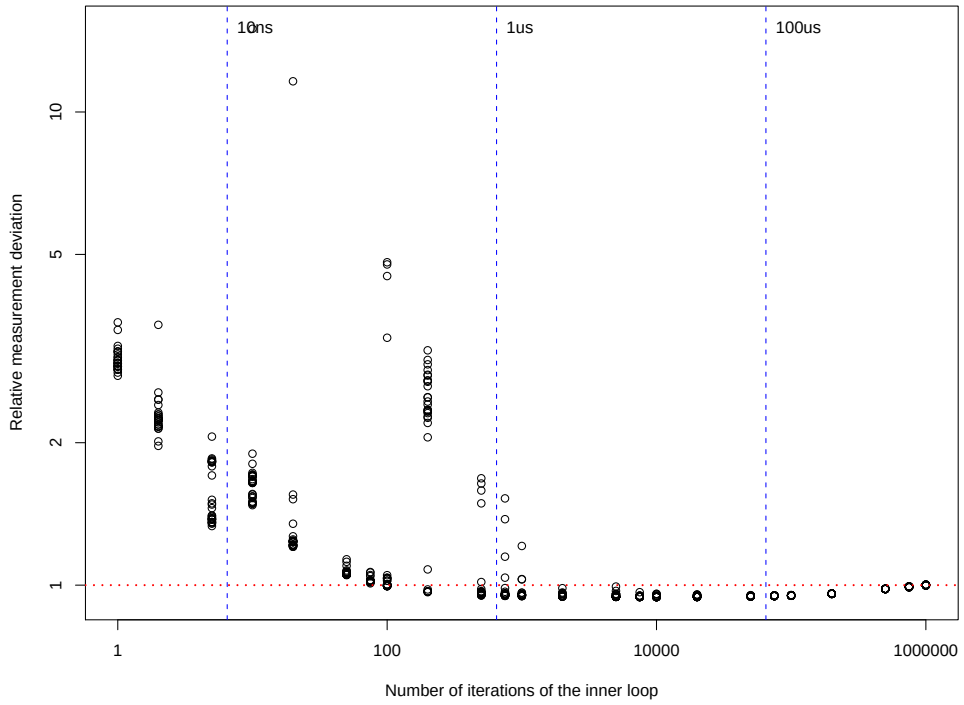


Figure 1.8: CGTS/disk (on xterm): Running time of a single step of the payload loop measured with coarse granularity, elapsed time chronometry, and with output on disk

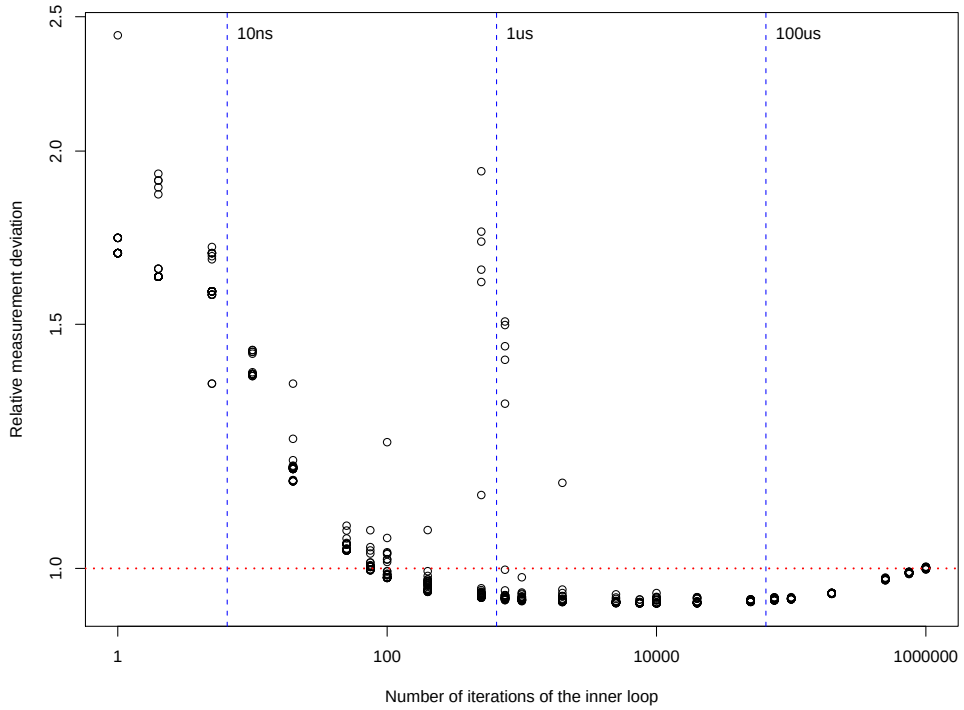


Figure 1.9: CGET/disk (on xterm): Running time of a single step of the payload loop measured with coarse granularity, elapsed time chronometry, and with output on disk

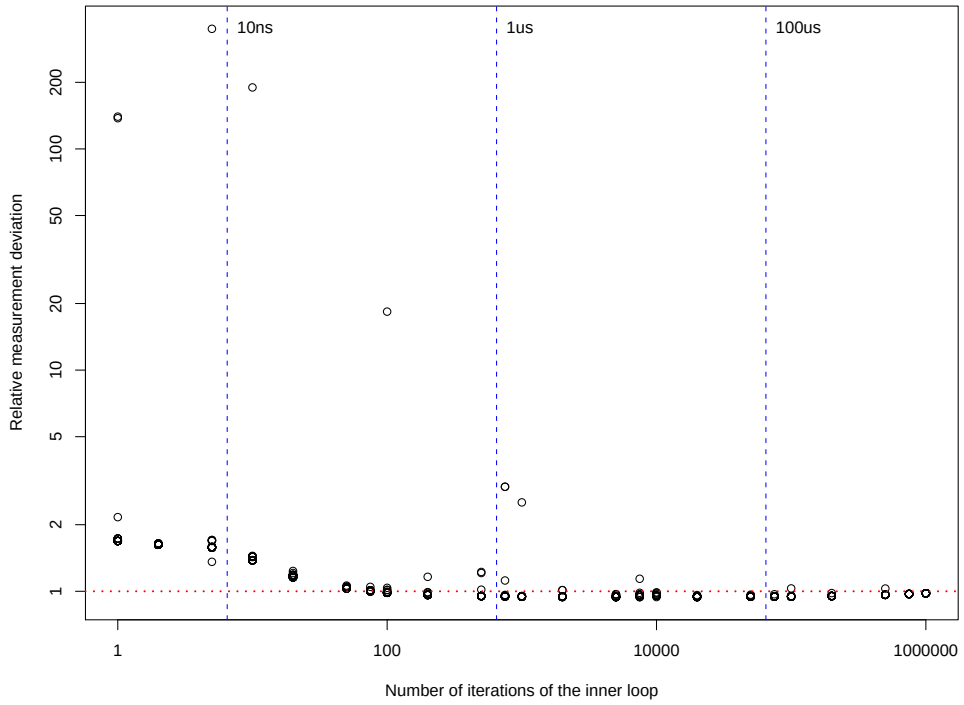


Figure 1.10: CGET/disk (on GNOME terminal): Running time of a single step of the payload loop measured with coarse granularity, elapsed time chronometry, and with output on disk

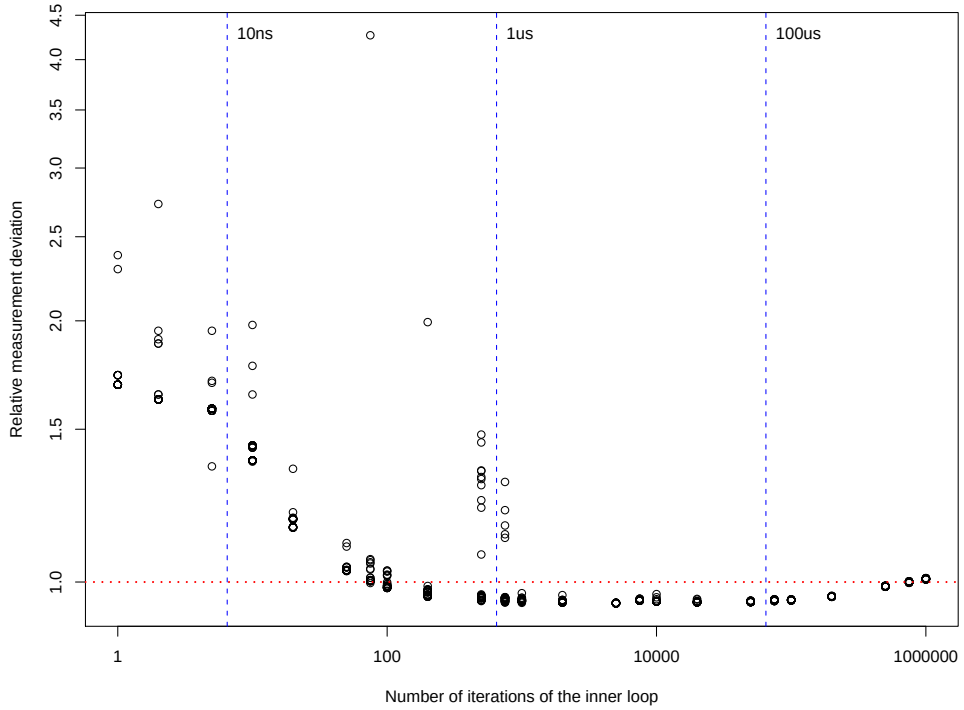


Figure 1.11: CGET/disk (on KDE Konsole): Running time of a single step of the payload loop measured with coarse granularity, elapsed time chronometry, and with output on disk

## 1.4 Design of Rules for Fujaba

While conducting benchmarks for Fujaba we noticed inconsistent running times. Fujaba performed much better as we expected using the results of the Varró benchmark. Analyzing the design of the rules and the meta-model of the STS benchmark we found two issues that need improvement. To confirm our ideas we repeated the original benchmark and executed our improved versions. The measurements in this section were executed 40-times on our benchmark computer under Windows XP with Sun Java 1.6.0_01 (for more details on the setup see appendix B).

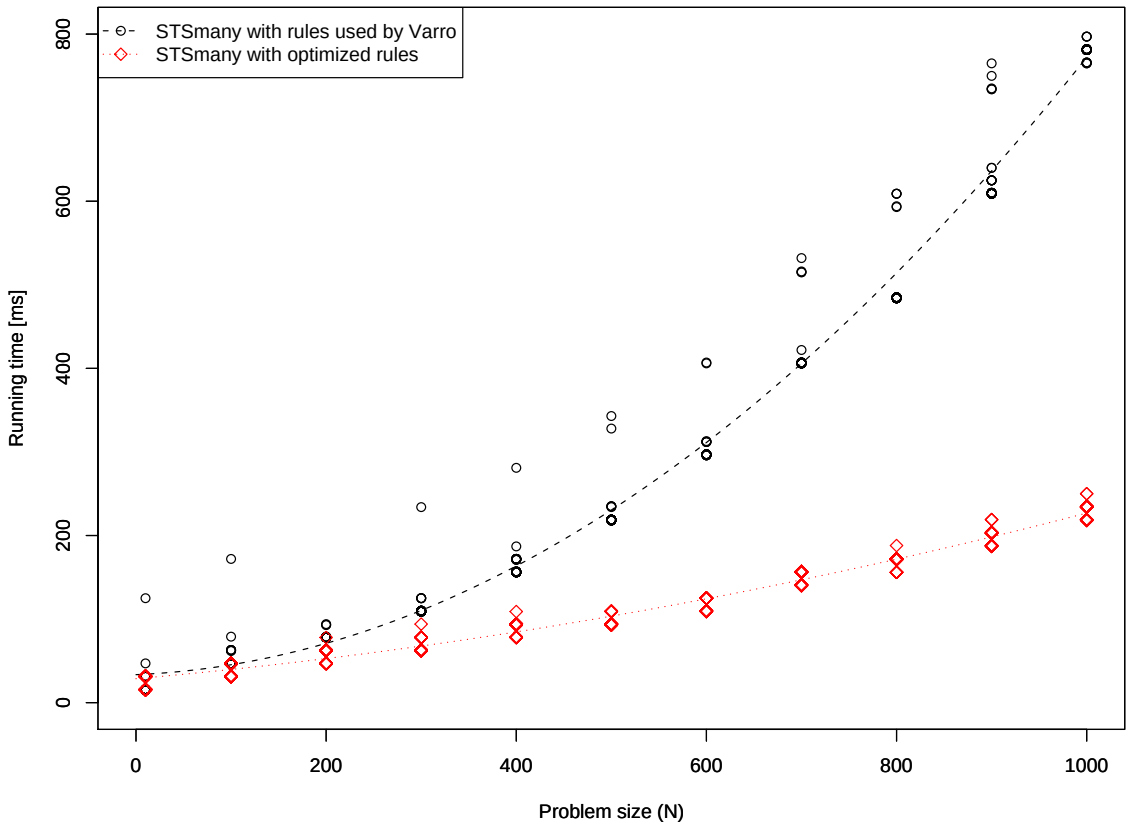


Figure 1.12: Running times of original and optimized rules of STSmany for Fujaba

### 1.4.1 Linear-time vs. Constant-time Matching

During the design of rules for Fujaba the user has to define starting points for the rule patterns (left-hand side of the graph rewrite rules). Choosing an inappropriate starting point can substantially raise the complexity of the matching process. In case of the Fujaba implementation of the STS benchmark performed by Varró the rules `takeRule`, `releaseRule`, and `giveRule` start with a process element instead of the single resource (see appendix A.2). As these rules search for a specific process connected to the resource via an unique edge, up to $n$ different processes are examined per rule leading to a matching runtime complexity of $O(n)$. Using the resource as a starting point the matching runtime complexity is only $O(1)$, as the resource has only one outgoing edge of the according type and the rest of the pattern can be matched in $O(1)$, too.

*Comment from G. Varró [Var07]:* Modeling questions are always a highly sensi-

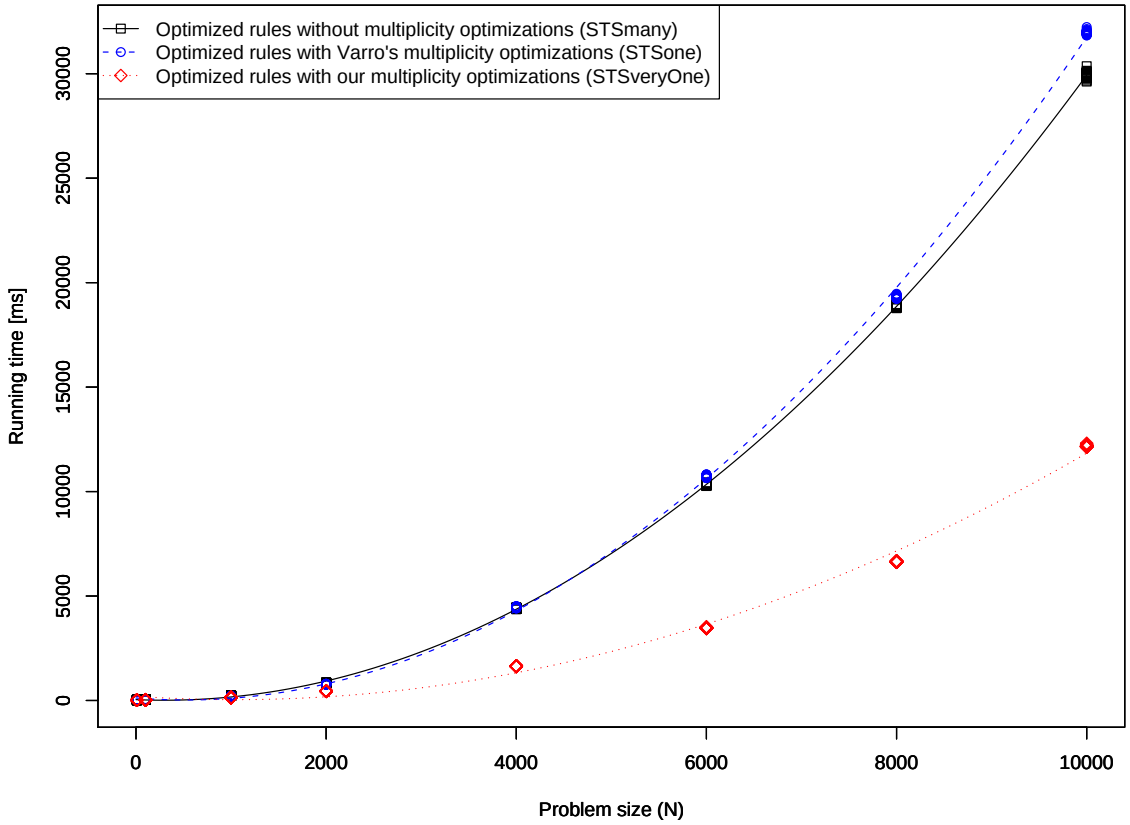


Figure 1.13: Running times using models with different given multiplicities for Fujaba

tive topic, since a good (but hand-made and user defined) modeling can significantly speed-up a given tool, which can make the comparison unfair. By defining a starting point, the problem itself becomes simpler as Fujaba gets a hint where to start. In case of other tools (AGG, PROGRES) the starting point should be determined by the tool itself. (I guess your tool also belongs to this second category.)

Figure 1.12 shows a comparison of the running times of the STSmany benchmark for Fujaba between the original rules used by Varró and the new optimized rules. With this improvement the `requestRule` remains the only rule with linear matching runtime complexity, causing a quadratic runtime for the whole benchmark.

*Remark:* GrGen avoids this problem by automatically generating the whole search plan. It considers the heuristically best starting points for the according situation taking the actual host graph into account. Moreover it uses knowledge about the last successful match enabling the search algorithm to heuristically reduce the search-space. By guessing where no match will be found, it can cut whole sub trees of the search-space. This is done in a conservative way, which never endangers correctness but may result in considerable speedups. Using this heuristics GrGen can even find a match for the `requestRule` in constant time (see Figure 2.3).

### 1.4.2 Underspecified Meta Model for STSone

The meta model should be designed to fit the structure of the problem as good as possible, allowing problem specific optimizations. We adopted this guideline from the "Benchmarking for Graph Transformation" paper by Varró et al [VSV05b]. A quote of the crucial statements from this paper is given below:

> Our general guideline for the comparison of tools was to use the standard services available in the default distribution, fine-tuned according to the suggestions of different tool developers. [...] In case of FUJABA, the models themselves were slightly altered to provide better performance.

In case of the STSone benchmark the goal was to see, how multiplicity optimizations affect the runtime performance. Despite the fact, that this benchmark uses only a single resource, the model used by Varró for STSone and STSonePP for FUJABA supports multiple resources and multiple incoming `token`, `held_by`, and `release` edges per process, which only makes sense with multiple resources. As FUJABA represents edges with a multiplicity greater than one as HashSets and edges with a multiplicity of one as simple fields with the appropriate type, restrictive multiplicities can greatly improve the performance.

Figure 1.13 shows a comparison of STSmany, STSone, and STSveryOne. The STSmany benchmark uses no multiplicity optimizations, and the STSone benchmark contains the multiplicity optimizations used by Varró. The new STSveryOne benchmark is an improved version of STSone only supporting a single resource and therefore only one incoming `token`, `held_by`, and `release` edge per process. All three benchmarks take advantage of the optimized rules from section 1.4.1.

As expected, STSveryOne is much faster than STSmany and STSone. But STSone being slower than STSmany was really unexpected. With the help of profilers we found out that STSone was faster than STSmany for all rules but the `requestRule`. The `requestRule` is the remaining rule with linear matching complexity and therefore, needs most of the running time (96% for STSone with $N = 10000$). As this rule is nearly unaffected by the improved multiplicity information of STSone, only a very small speed up was expected. But in fact it needed 29% more running time as in STSmany. Further investigation showed that Sun's HotSpot compiler inlined other functions in STSone than in STSmany and that these functions were less important to be inlined than those chosen for STSmany.

CHAPTER 2

# IMPROVEMENTS AND RESULTS

In this chapter we present our suggestions for improving the Varró benchmark as well as new resulting figures.

## 2.1 Improvements

- The running time should only be measured with coarse granularity, i.e. only the whole benchmark at once.
- No output during the measurement and before the measurement (due to asynchronous output) should be made.
- Choose appropriate starting points for rules in FUJABA to reduce the number of processed graph elements. This can reduce the running time from linear to constant complexity for several rules.
- In FUJABA the graph meta model should be restricted to allow only necessary connections. In case of the STSone benchmark the running time is divided by two.
- To reduce the influence of outliers the median should be used and therefore many benchmark runs are needed.
- For fast tools with just-in-time compilation (JIT) little benchmark sizes are pointless, unless the influence of the overhead for the JIT is reduced by performing a warm-up. This preperatory step is only executed to force just-in-time compilation of the program and is not included in the actual measurement of the running time.

## 2.2 Improved Results for FUJABA

With the considerations presented so far we can construct new figures for the Varró benchmark (particularly the STSmany variant). We only considered the FUJABA tool closely, since it is the fastest tool besides the two faster GRGEN-implementations. Figure 2.1 shows an improved and extended version of the ICGT 2006 presentation. It is extended by two curves: One showing the recently implemented GRGEN.NET and the other one showing the improved FUJABA version. All new measurements were performed on our benchmark computer (see appendix B) and repeated 40 times.

To get a more realistic view for the fast tools, we need bigger problem sizes. By choosing a double logarithmic scale we should be able to easily distinguish tools with linear runtime (GRGEN and GRGEN.NET) and all other tools. But as you can see in Figure 2.2 the curves for FUJABA and GRGEN.NET look more like a super exponential function[1]; in case of GRGEN.NET the long, almost constant, segment (size: 10–10000) is surprising. These anomalies

[1] Please keep in mind, that on log-log scale all polynomials are linear curves. Their degree is reflected by the gradient. Different factors are reflected by different axis intercepts. Super polynomial functions are plotted super linear.

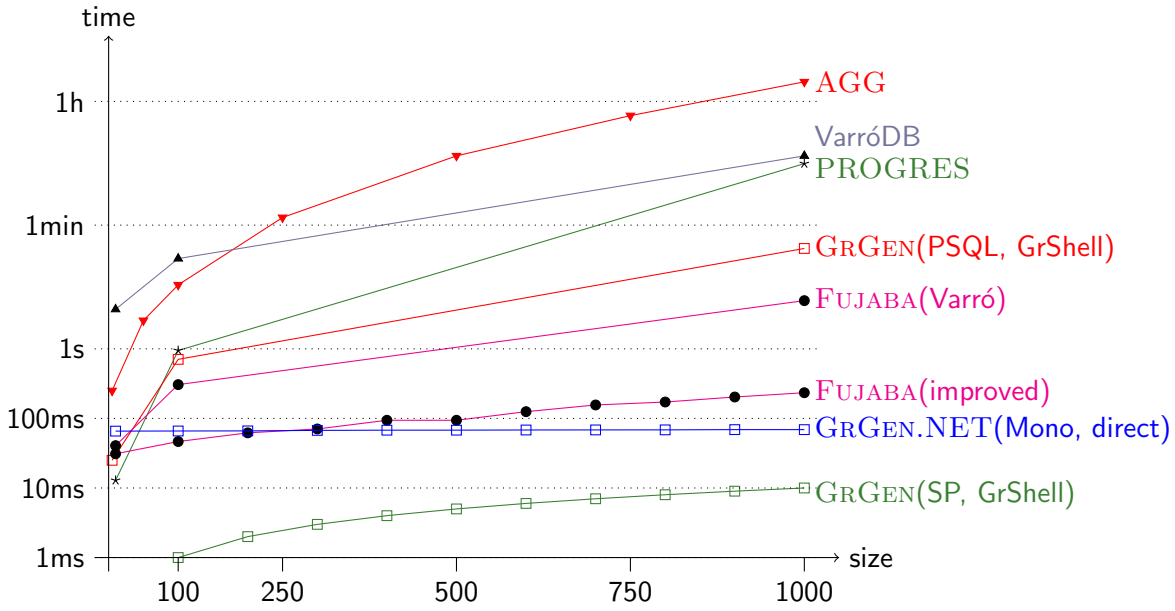


Figure 2.1: Running times of the STSmany benchmark for different graph rewrite systems

are caused by the just-in-time compilation of the Java and .NET runtime environments. For Figure 2.3 we use a warm-up run: We let the tools perform a STSmany benchmark run of size 1000 and afterwards (without exiting the program) perform the actual benchmark. To reduce memory anomalies we force garbage collection in between the runs.

Now we can see that the running time of GrGen.NET is indeed linear and the running time of Fujaba is quadratic[2] (cf. section 1.4.1). The constant segment at the beginning of the GrGen(SP), GrGen.NET and Fujaba curve is due to the timer resolution and the use of the median.

## 2.3 Conclusion

We believe that Varró's benchmark is a crucial step for the graph rewrite community towards empirical foundations. By no means do we want to reduce the outstanding character of this benchmark introduced by the Varró brothers.

This paper has suggested several improvements to the implementation and the measurement setup of the Varró benchmark. The idea of the benchmark in principle is *not* questioned by us. Albeit we regard the numerical data presented in the well-known publications [VFV05, VSV05b] to be flawed in several ways:

- The design of the chronometry is not suitable for fast tools.
- The rules and the meta model for Fujaba are not adequately tuned.
- The implementation of the chronometry for PROGRES is erroneous.
- The log files have inconsistent layout and content.
- The specification file, source code, and log files for Fujaba STSmany/one do not match.
- Tools with runtime environments featuring JIT compilation require special care.

We have shown improved figures for the Fujaba, GrGen, and the newly released GrGen.NET tools. Our suggestions address implementation and design flaws of the Varró benchmark. We believe that this will strengthen the trust in the significance and objectivity of this valuable benchmark.

[2]Please note, that the different gradients (double steepness) prove this statement.

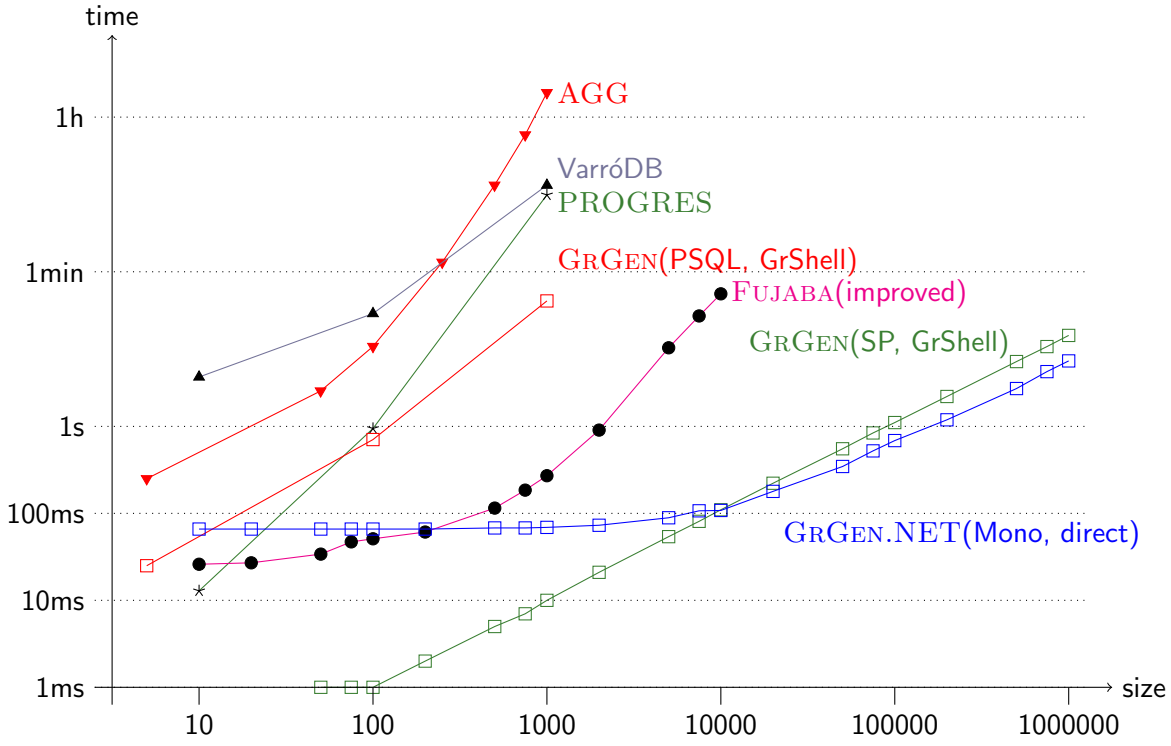


Figure 2.2: Running times of the STSmany benchmark for different graph rewrite systems

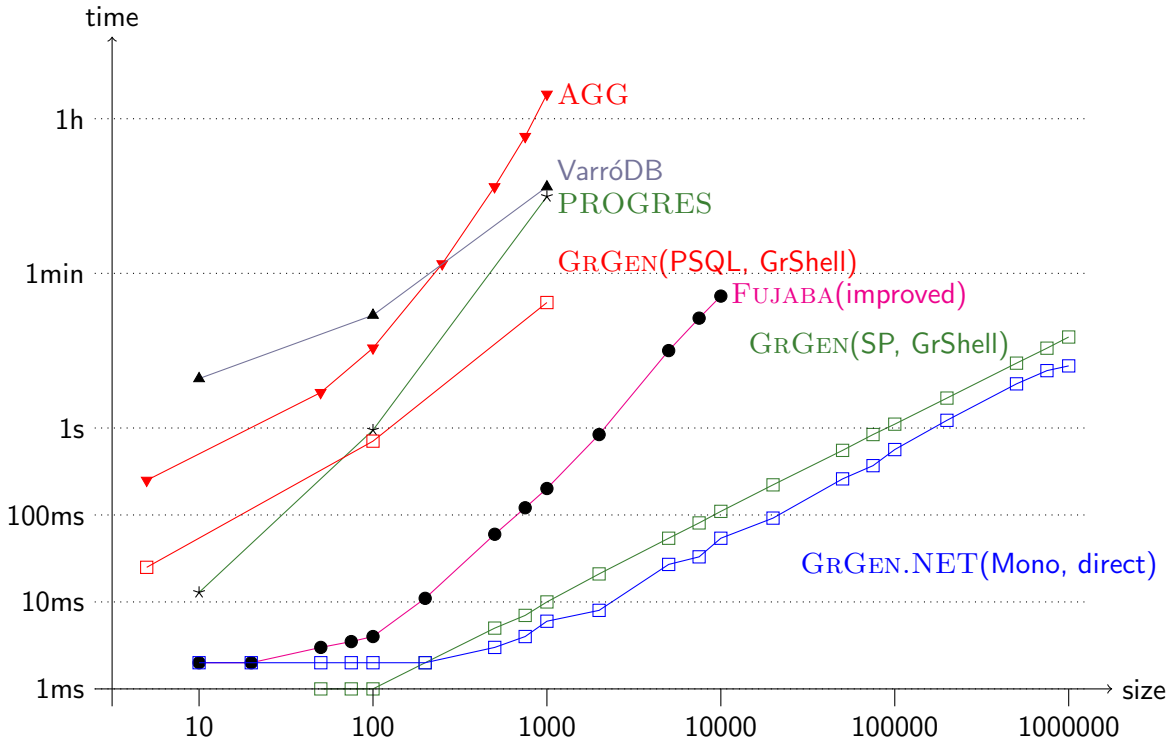


Figure 2.3: Running times of the STSmany benchmark for different graph rewrite systems with a warm-up run (n=1000) for Fujaba and GrGen.NET to reduce JIT overhead

APPENDIX A

# SOURCE CODE

## A.1 Timestamp Chronometry Considered Harmful

Listing A.1: fine granularity, timestamp

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   struct timeval tv;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);

   for(i=0; i<rep; i++) {
      for(j=0; j<steps; j++) cnt += j;
      gettimeofday(&tv, NULL);
      printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%ld.%06ld␣s\n", tv.tv_sec, tv.tv_usec);
   }
   exit(EXIT_SUCCESS);
}
```

Listing A.2: fine granularity, elapsed time

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   long long int runtime=0;
   struct timeval tvb, tve;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);

   for(i=0; i<rep; i++) {
      gettimeofday(&tvb, NULL);
      for(j=0; j<steps; j++) cnt += j;
      gettimeofday(&tve, NULL);
      runtime += tve.tv_sec*1000000ULL+tve.tv_usec - (tvb.tv_sec*1000000ULL+tvb.tv_usec);
      printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%lld␣usec\n", runtime);
   }
   exit(EXIT_SUCCESS);
}
```

Listing A.3: coarse granularity, timestamp

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   struct timeval tv;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);

   gettimeofday(&tv, NULL);
   printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%ld.%06ld␣s\n", tv.tv_sec, tv.tv_usec);
   for(i=0; i<rep; i++) {
      for(j=0; j<steps; j++) cnt += j;
   }
   gettimeofday(&tv, NULL);
   printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%ld.%06ld␣s\n", tv.tv_sec, tv.tv_usec);
   exit(EXIT_SUCCESS);
}
```

Listing A.4: coarse granularity, elapsed time

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   long long int runtime;
   struct timeval tvb, tve;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);

   gettimeofday(&tvb, NULL);
   for(i=0; i<rep; i++)
      for(j=0; j<steps; j++) cnt += j;
   gettimeofday(&tve, NULL);
   runtime = tve.tv_sec*1000000ULL+tve.tv_usec - (tvb.tv_sec*1000000ULL+tvb.tv_usec);
   printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%lld␣usec\n", runtime);
   exit(EXIT_SUCCESS);
}
```

Listing A.5: quite, fine granularity, timestamp

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   struct timeval *tv;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);
   tv = malloc(sizeof(*tv) * rep);

   for(i=0; i<rep; i++) {
      for(j=0; j<steps; j++) cnt += j;
      gettimeofday(&tv[i], NULL);
   }
   for(i=0; i<rep; i++)
      printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%ld.%06ld␣s\n", tv[i].tv_sec, tv[i].tv_usec);
   exit(EXIT_SUCCESS);
}
```

Listing A.6: quiet, fine granularity, elapsed time

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
volatile int cnt;

int main(int argc, char *argv[]){
   int i, j, rep, steps;
   long long int runtime=0;
   struct timeval tvb, tve;
   rep = atoi(argv[1]);
   steps = atoi(argv[2]);

   for(i=0; i<rep; i++) {
      gettimeofday(&tvb, NULL);
      for(j=0; j<steps; j++) cnt += j;
      gettimeofday(&tve, NULL);
      runtime += tve.tv_sec*1000000ULL+tve.tv_usec - (tvb.tv_sec*1000000ULL+tvb.tv_usec);
   }
   printf ("Lorem␣ipsum␣dolor␣sit␣amet:␣%lld␣usec\n", runtime);
   exit(EXIT_SUCCESS);
}
```

Listing A.7: The script that does the measurements of chronometry.

```
APPS="cget␣cgts␣fget␣fgts␣qfget␣qfgts"
REPEATS=15000
STEPS="1␣2␣5␣10␣20␣50␣75␣100␣200␣500␣750␣1000␣2000␣5000␣7500␣10000␣20000␣50000␣75000␣100000␣
    200000␣500000␣750000␣1000000"
MEASUREMENTS=30

rm *.log aux $APPS

for app in $APPS ; do
   gcc -O3 -falign-loops=15 ${app}.c -o ${app}
done

for kind in disk term ; do
   for app in $APPS ; do
      echo "steps;time" > ${kind}_${app}_${REPEATS}.csv
   done
done

for (( iter=0 ; iter <= $MEASUREMENTS; iter = iter + 1)) ; do
   for step in $STEPS ; do
         for app in $APPS ; do
            ./$app ${REPEATS} ${step} | tee term_${app}_${REPEATS}_${step}_${iter}.log
            ./$app ${REPEATS} ${step} >     disk_${app}_${REPEATS}_${step}_${iter}.log
      done
   done

   for kind in disk term ; do
      for step in $STEPS ; do
         for app in $APPS ; do
            echo -n "$step;" >> ${kind}_${app}_${REPEATS}.csv
         done

         for app in cgts fgts qfgts ; do
            echo -n "(" > aux
            tail -1 ${kind}_${app}_${REPEATS}_${step}_${iter}.log | awk '{printf("%s",$6)}'
                >> aux
            echo -n "␣-␣" >> aux
            head -1 ${kind}_${app}_${REPEATS}_${step}_${iter}.log | awk '{printf("%s",$6)}'
                >> aux
            echo ")*1000000" >> aux
            bc < aux >> ${kind}_${app}_${REPEATS}.csv
         done

         for app in cget qfget ; do
            < ${kind}_${app}_${REPEATS}_${step}_${iter}.log awk '{print $6}' >>
                ${kind}_${app}_${REPEATS}.csv
         done

         tail -1 ${kind}_fget_${REPEATS}_${step}_${iter}.log | awk '{print $6}' >>
             ${kind}_fget_${REPEATS}.csv
      done
   done
done
```

## A.2 STSmany Improved

The following figures show storyboard representations of some STS benchmark rules generated with the FUJABA tool. We present the original and the improved versions of these rules.

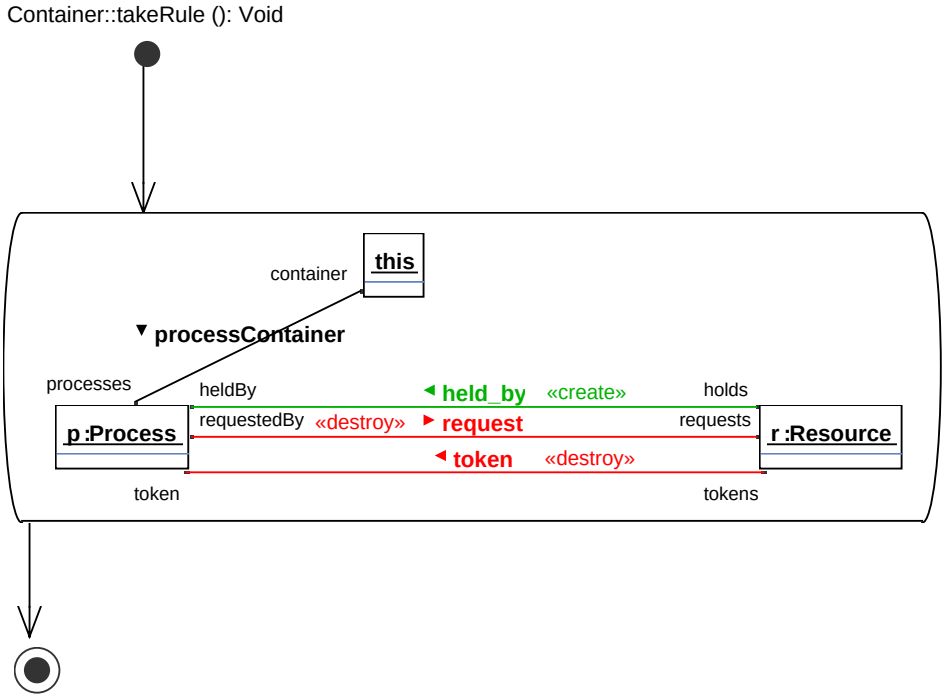


Figure A.1: takeRule used by Varró

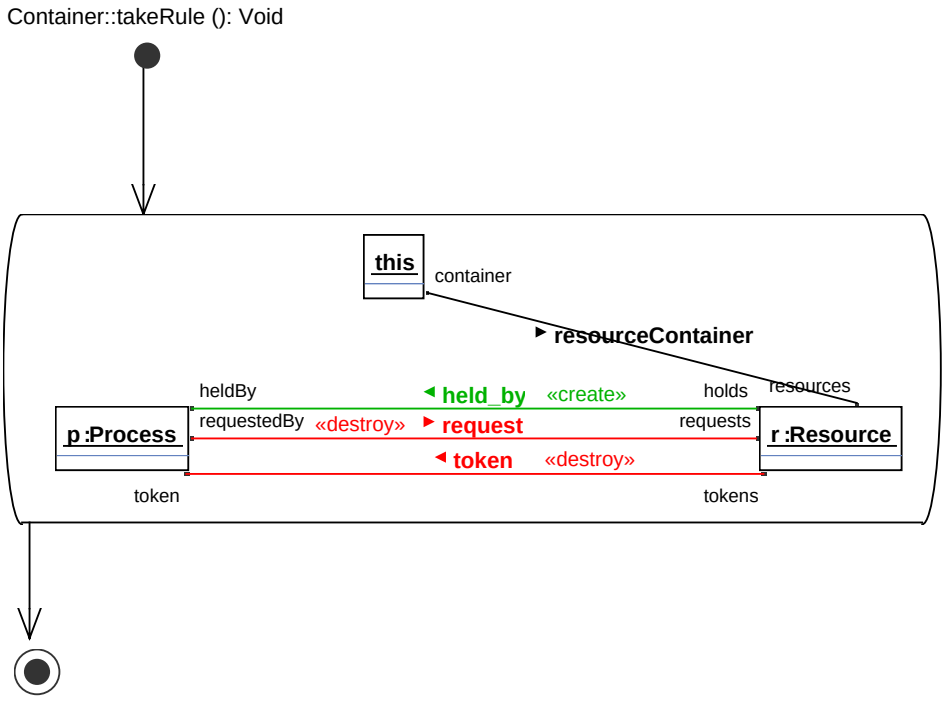


Figure A.2: Improved takeRule

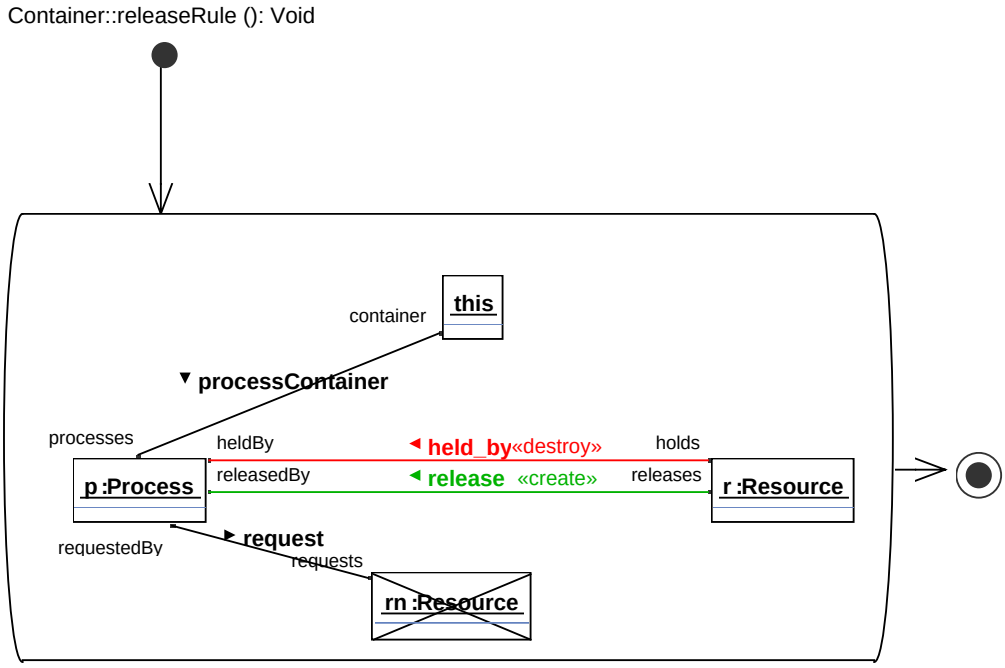


Figure A.3: releaseRule used by Varró

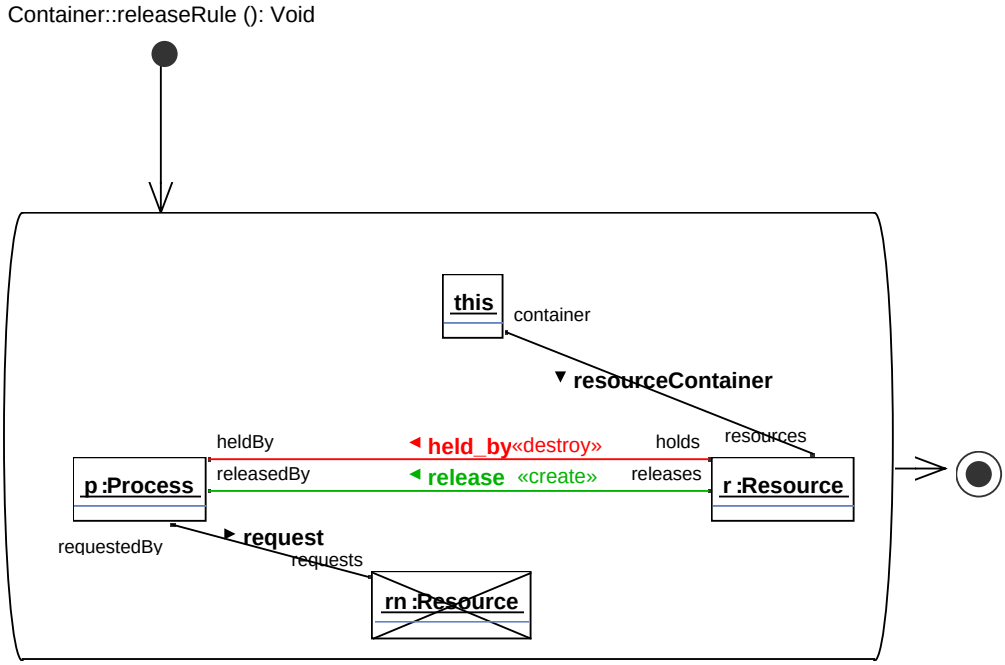


Figure A.4: Improved releaseRule

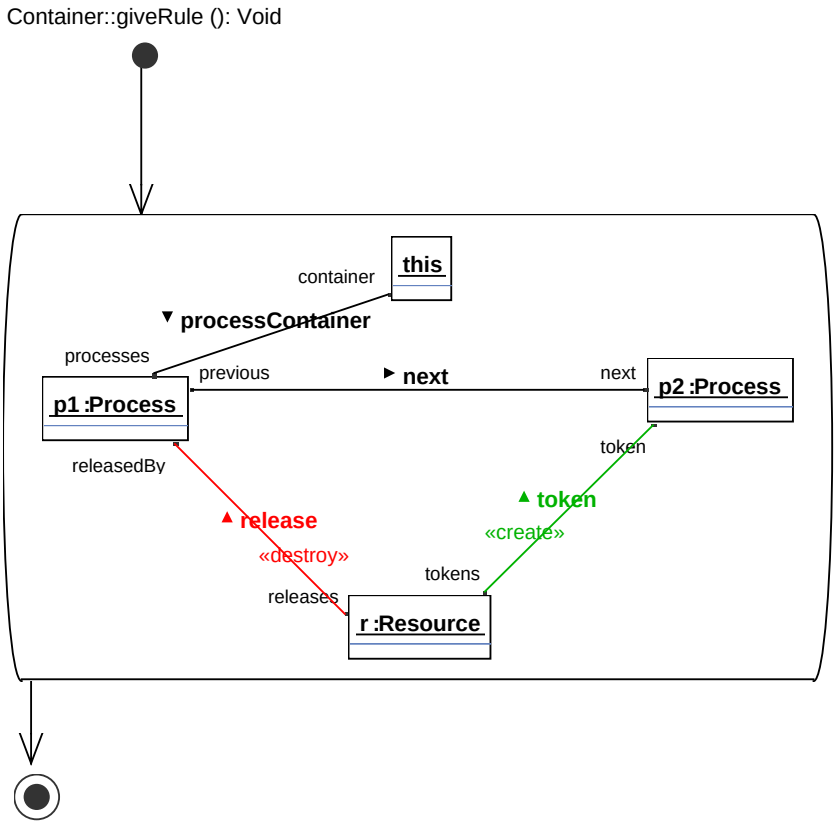


Figure A.5: giveRule used by Varró

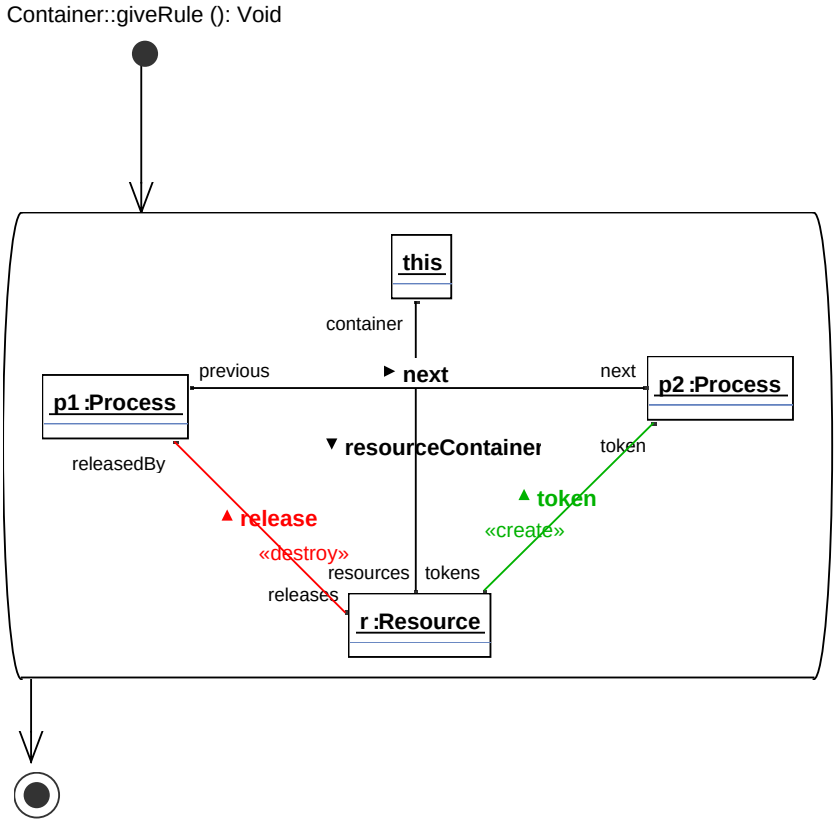


Figure A.6: Improved giveRule

## A.3 STSveryOne

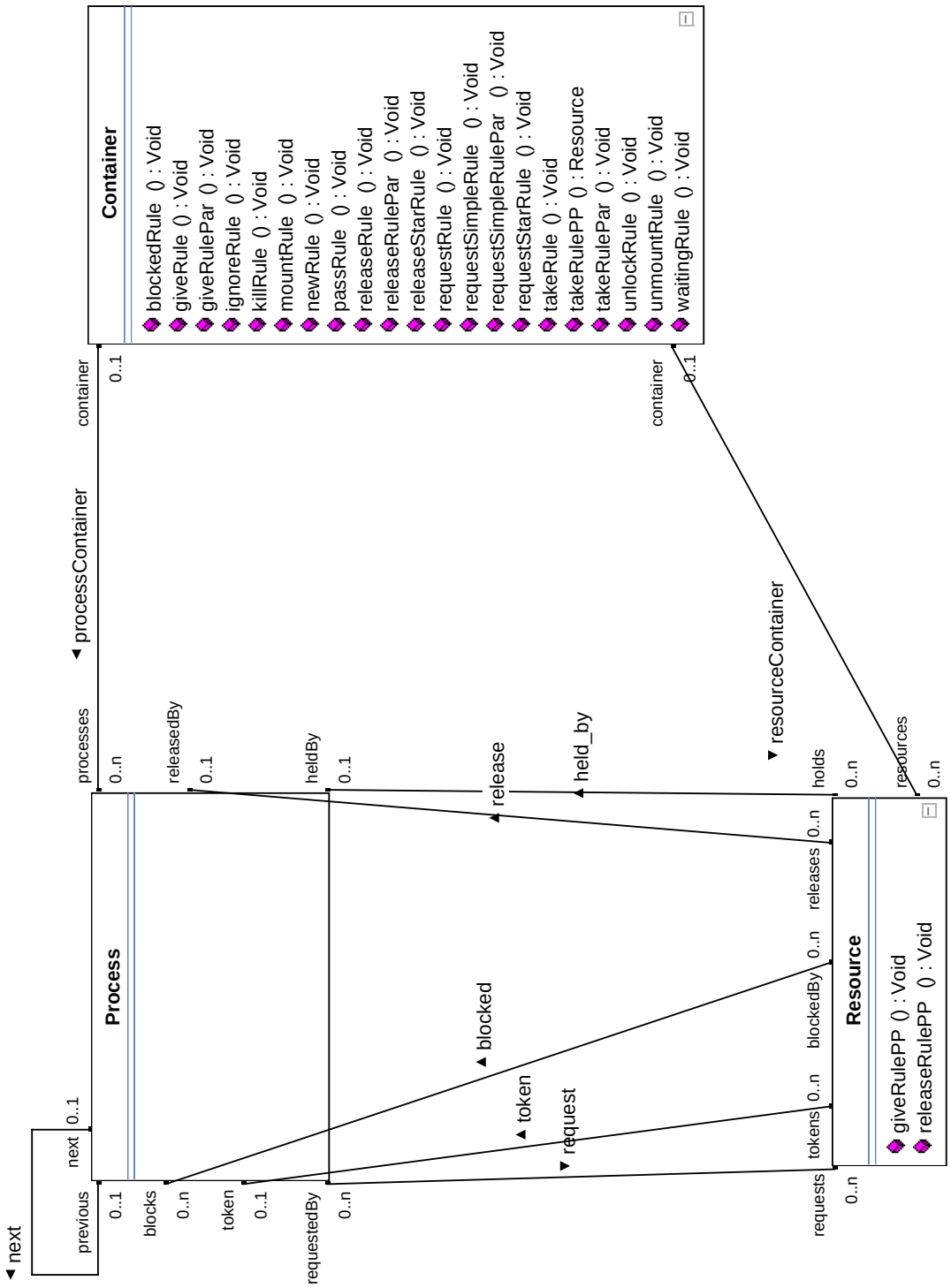


Figure A.7: STSone model used by Varró

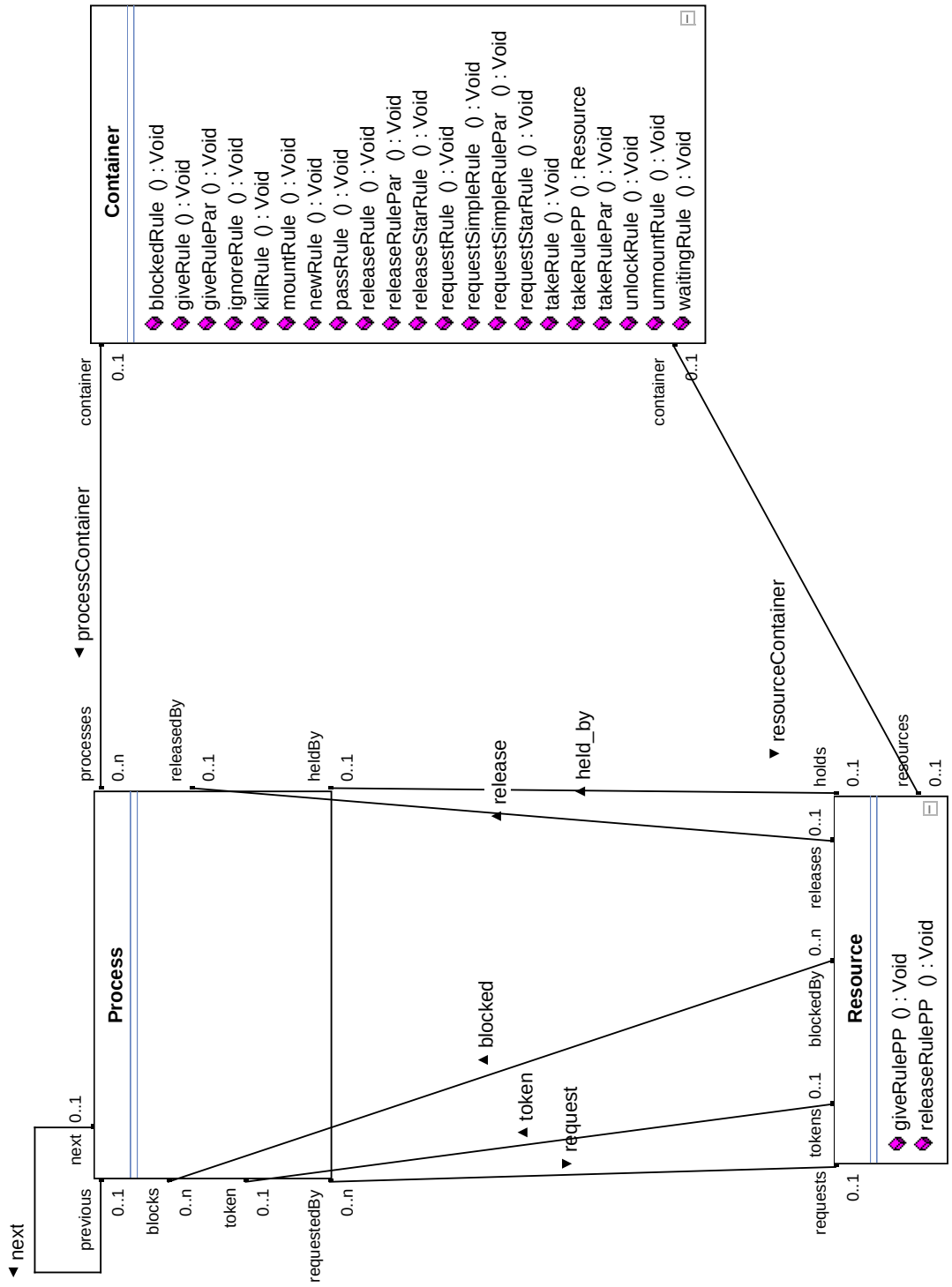


Figure A.8: Improved STSone model. Please note the changed cardinalities of the resourceContainer, request, held_by, and taken associations.

APPENDIX B

# THE BENCHMARK COMPUTER

We use a single core, single threaded desktop personal computer to perform our benchmarks. To compare the benchmark runs with older ones we used some seasoned hardware. This chapter lists all relevant software and hardware components involved in the benchmark.

*CPU*

- AMD Athlon$^{TM}$ XP 3000+
- L1-Cache: 64 + 64 KiB (Data + Instructions)
- L2-Cache: 512 KiB with Core Frequency
- 2100 MHz
- FSB 400

*Memory*

- 2 × Infineon 512 MiB
- DDR 500 CL3

*Motherboard*

- Asus A7N8X-X

*Graphics Card*

- ATI/AMD Radeon 7000 OEM

*Software/Linux*

- Suse Linux 9.3
- Linux Kernel 2.6.11.4-21.17-default
- GNU C Library 2.3.4 (20050218)
- X Window System Version 6.8.2 (9. Feb. 2005)
- Mono 1.2.3.1
- Java 2 Runtime Environment, SE (build 1.5.0_04-b05)
- xterm 200-3
- GNOME 2.10.0
- KDE 3.4.0

*Software/Windows*

- Microsoft Windows® XP, Version 5.1 (Build 2600.xpsp_sp2_gdr.070227-2254: Service Pack 2)
- Microsoft.NET Framework v2.0.50727
- Java$^{TM}$ SE Runtime Environment (build 1.6.0_01-b06)

# BIBLIOGRAPHY


[BG07] Jakob Blomer and Rubino Geiß. The GrGen User Manual. Technical Report 2007-5, Universität Karlsruhe, IPD Goos, June 2007. ISSN 1432-7864.

[Fou06] Apache Software Foundation. Log4j Project. http://logging.apache.org/log4j/, 2006.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *ICGT*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.

[Gei07] Rubino Geiß. GrGen.NET. http://www.grgen.net, November 2007.

[Kro07] Moritz Kroll. GrGen.NET: Portierung und Erweiterung des Graphersetzungssystems GrGen, May 2007. Studienarbeit, Universität Karlsruhe.

[Mic07] SUN Microsystems. Java™Platform, Standard Edition 6 – API Specification. http://java.sun.com/javase/6/docs/api/, 2007.

[Sta05] Standard Performance Evaluation Corporation. All SPEC CPU2000 results published by SPEC page. http://www.spec.org/cpu2000/results/cpu2000.html, September 2005.

[Var05] Gergely Varró. Graph transformation benchmarks page. http://www.cs.bme.hu/~gervarro/benchmark/2.0/, August 2005.

[Var07] Gergely Varró. On Improvements of the Varró Benchmark for Graph Transformation Tools. Private correspondence, June 2007.

[VFV05] Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformation in relational databases. In T. Mens, A. Schürr, and G. Taentzer, editors, *Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)*, volume 127, No 1 of *Electronic Notes in Theoretical Computer Science*, pages 167–180. Elsevier, March 2005.

[VSV05a] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for Graph Transformation. Technical report, Department of Computer Science and Information Theory, Budapest University of Technology and Economics, March 2005.

[VSV05b] Gergely Varró, Andy Schürr, and Dániel Varró. Benchmarking for graph transformation. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 79–88, Washington, DC, USA, 2005. IEEE Computer Society.