Benchmarking in Computer Science and Competition on Software Verification

Dirk Beyer

2023-03-21, at Masaryk University, Brno

Part 1: Reliable Benchmarking

Dirk Beyer, *Stefan Löwe*, and *Philipp Wendler*.
Reliable Benchmarking: Requirements and Solutions. [2]
STTT 2019



Evaluation of Research Result

- Result “Theorem”
  Evaluation “Proof”
- Result “Algorithm”
  Evaluation “Algorithm Analysis, properties, Big-O”
- Result “Heuristics for Complex Problems”
  Evaluation “Performance Experiments”

Comparative Evaluation

- Old: Done by competitors
- New: Done by independent competitions

Background: Requirements

Repeatability

- everything documented
  (machine, version of tool and OS, parameters)
- deterministic tool
- **reliable benchmarking**

Reproducibility

- everything above
- availability of tool, benchmark set,
  configuration, environment
  (published and archived, appropriate license)

Replicability

(not discussed here)

Benchmarking is Important

- Evaluation of new approaches
- Evaluation of tools
- Competitions
- Tool development (testing, optimizations)

Reliable, reproducible, and accurate results needed!

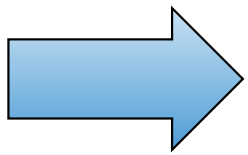
Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Not relevant for
most verification tools



Easy?

Benchmarking is Hard

- Influence of I/O
- Networking
- Distributed tools
- User input

Not relevant for most verification tools

- Different hardware architectures
- Heterogeneity of tools
- Parallel benchmarks

Relevant!

Goals

- Reproducibility
  - Avoid non-deterministic effects and interferences
  - Provide defined set of resources
- Accurate results
- For verification tools (and similar)
- On Linux

Checklist

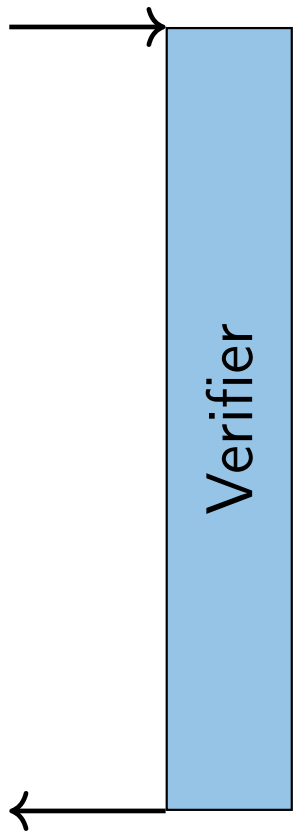
1. Measure and Limit Resources Accurately
   - Time
   - Memory
2. Terminate Processes Reliably
3. Assign Cores Deliberately
4. Respect Non-Uniform Memory Access
5. Avoid Swapping
6. Isolate Individual Runs
   - Communication
   - File system

Measure and Limit Resources Accurately

- Wall time and CPU time
- Define memory consumption
  - Size of address space? Too large
  - Size of heap? Too low
  - Size of resident set (RSS)?
- Measure peak consumption
- Always define memory limit for reproducibility
- Include sub-processes

Measuring CPU time with "`time`"

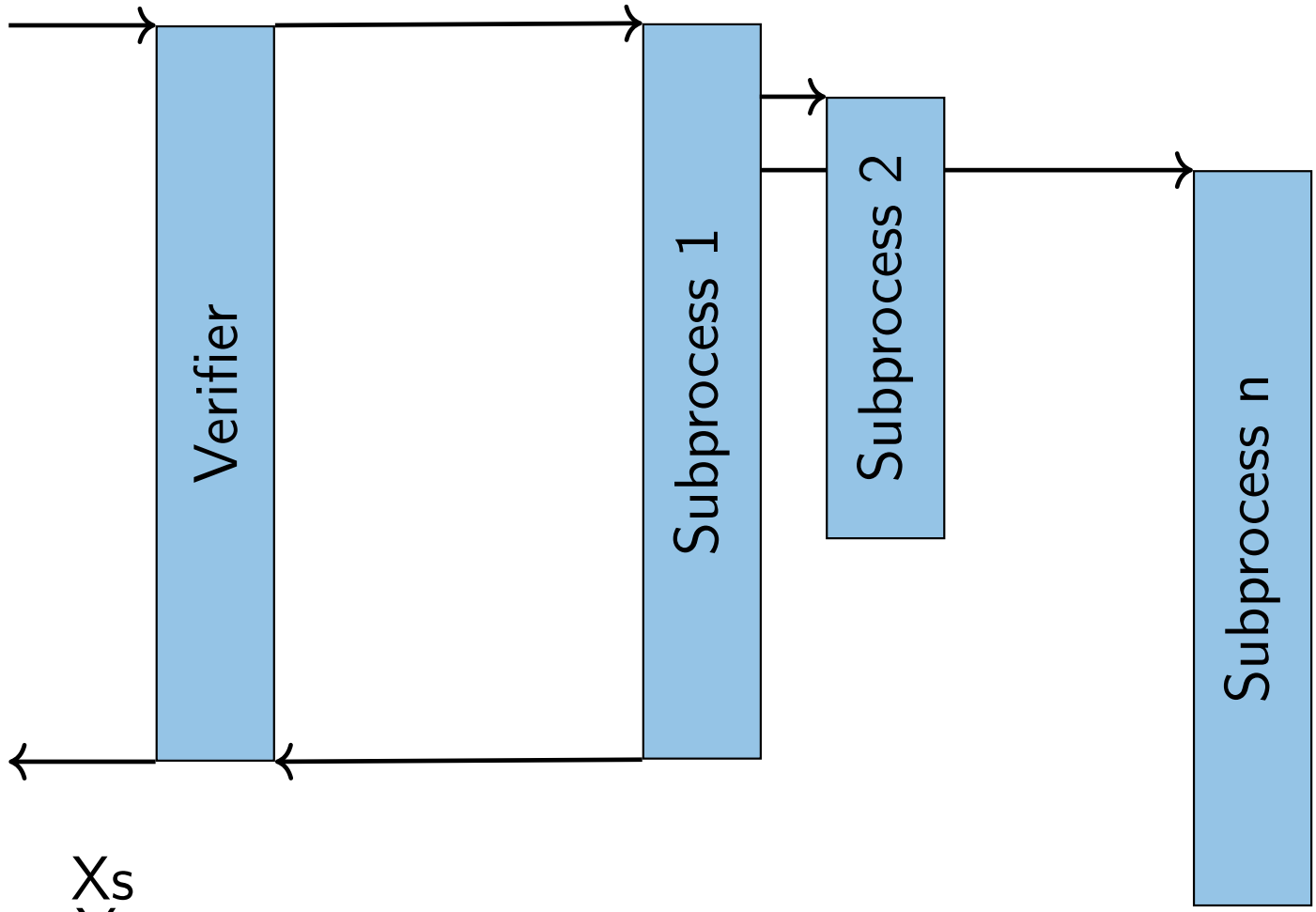
```
~$ time verifier
```



real	Xs
user	Ys
sys	Zs

Measuring CPU time with "`time`"

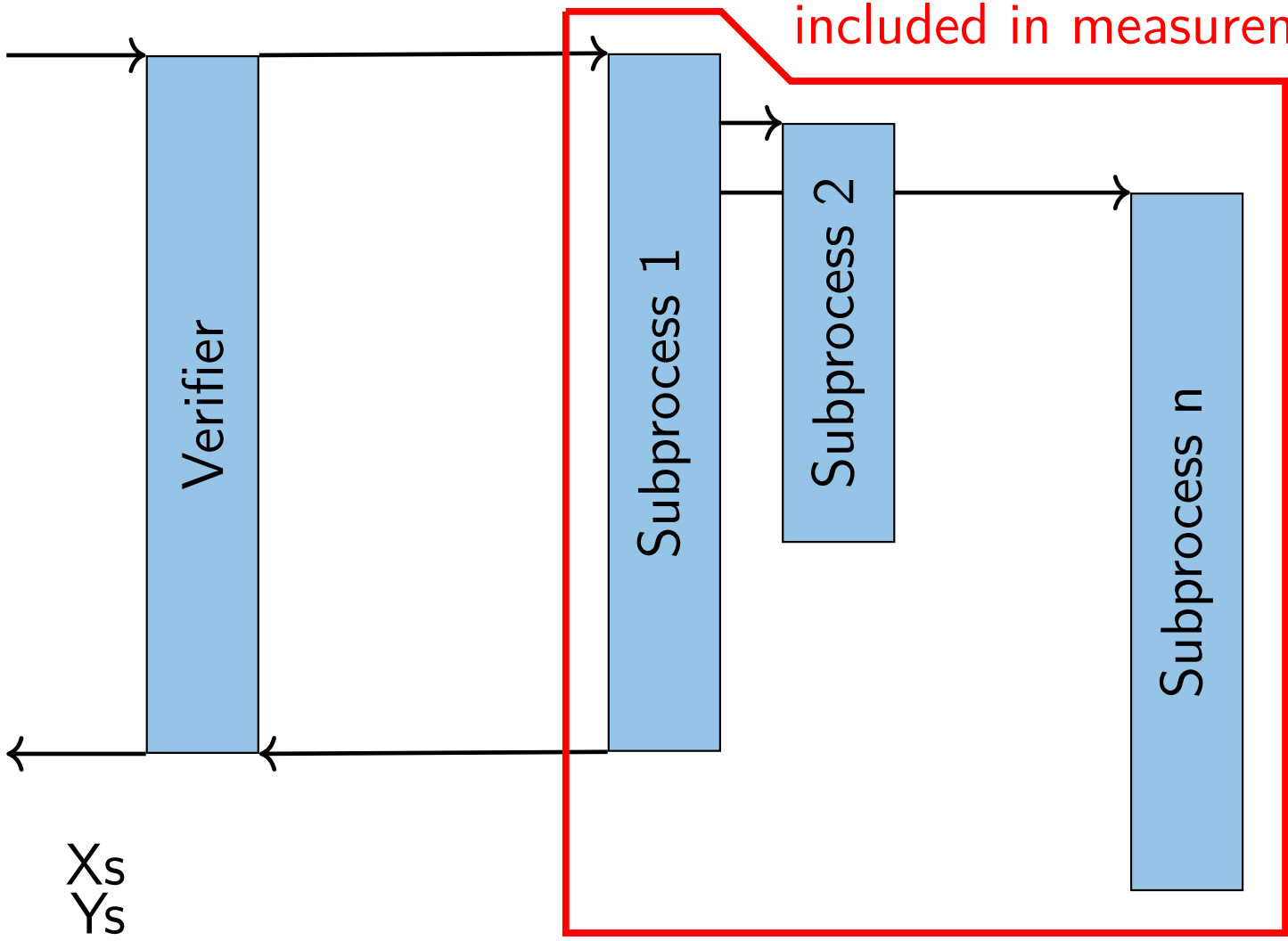
```
~$ time verifier
```



real Xs
user Ys
sys Zs

Measuring CPU time with "time"

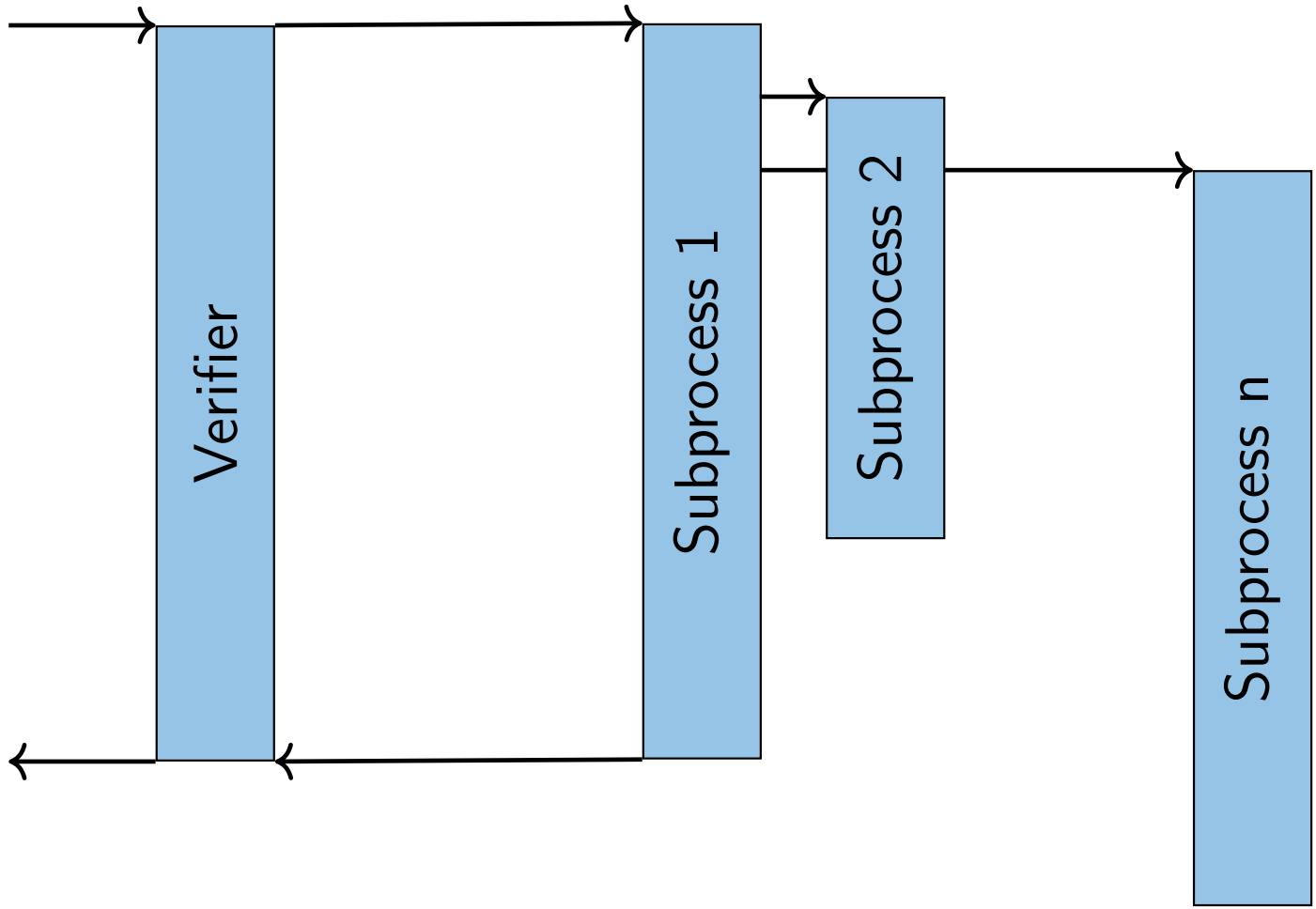
```
~$ time verifier
```



real Xs
user Ys
sys Zs

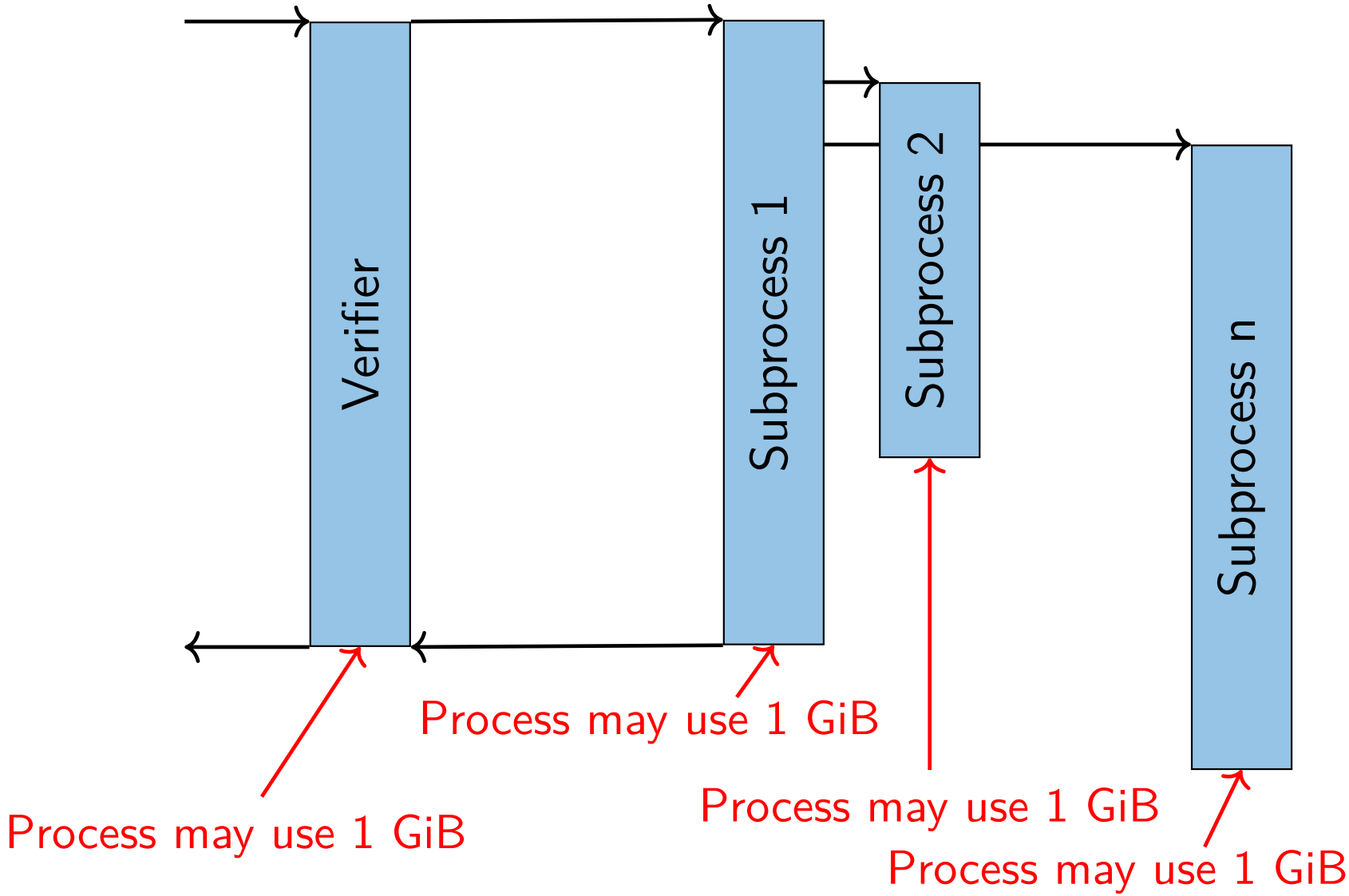
Limiting memory with “ulimit”

```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



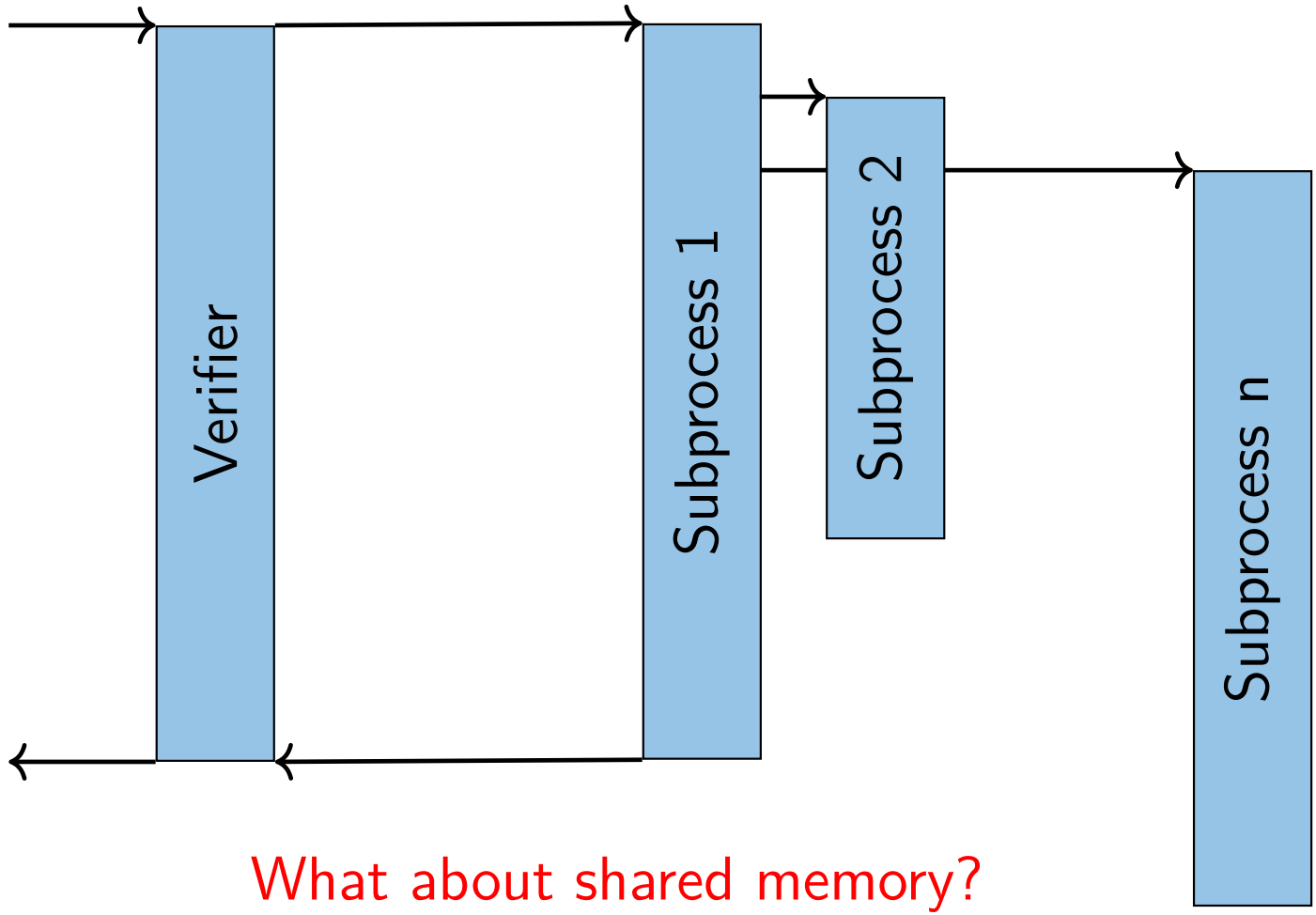
Limiting memory with "ulimit"

```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



Limiting memory with "ulimit"

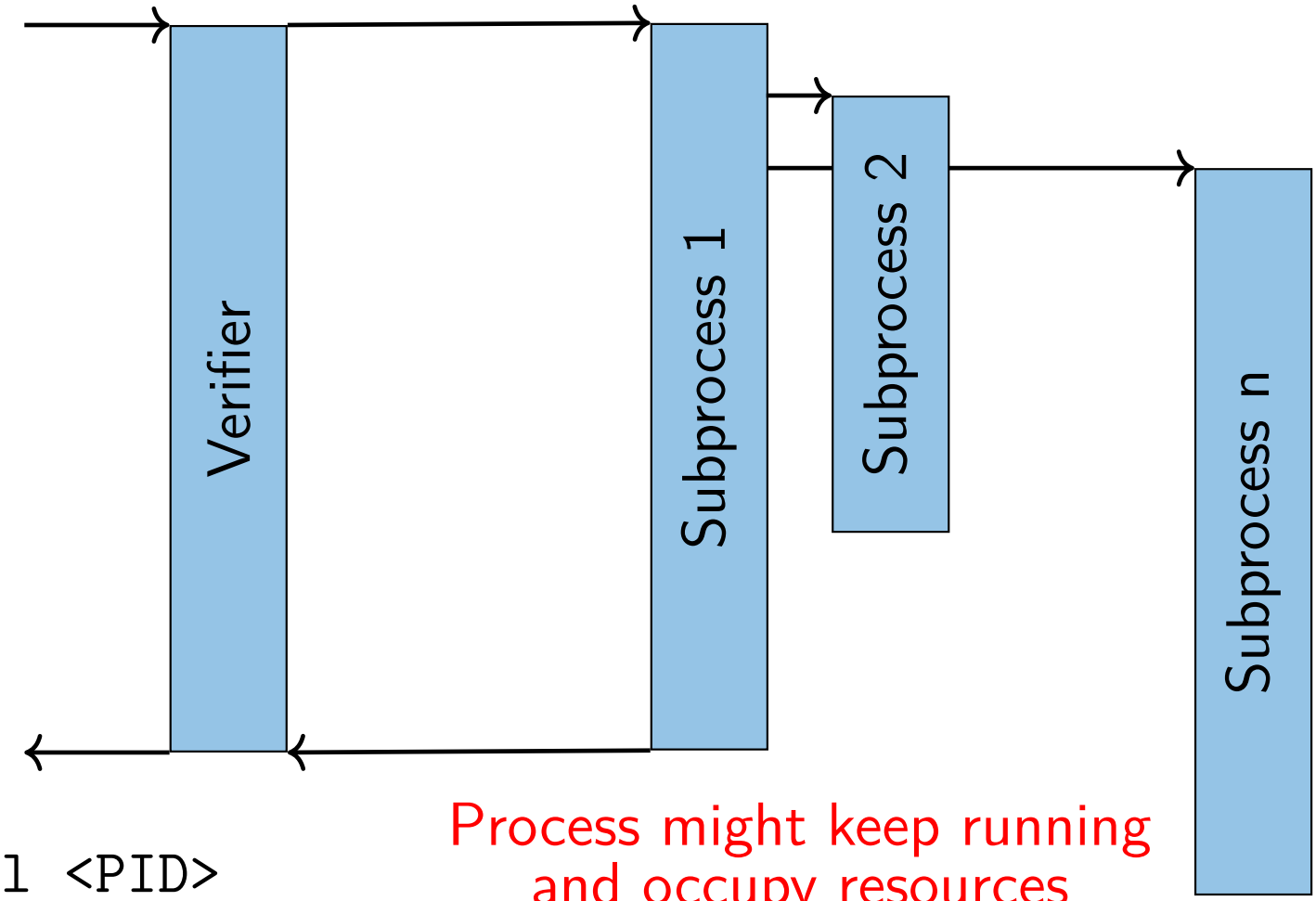
```
~$ ulimit -v 1048576 # 1 GiB
~$ verifier
```



What about shared memory?

Terminate Processes Reliably

```
~$ verifier
```




```
~$ kill <PID>
```

Process might keep running
and occupy resources

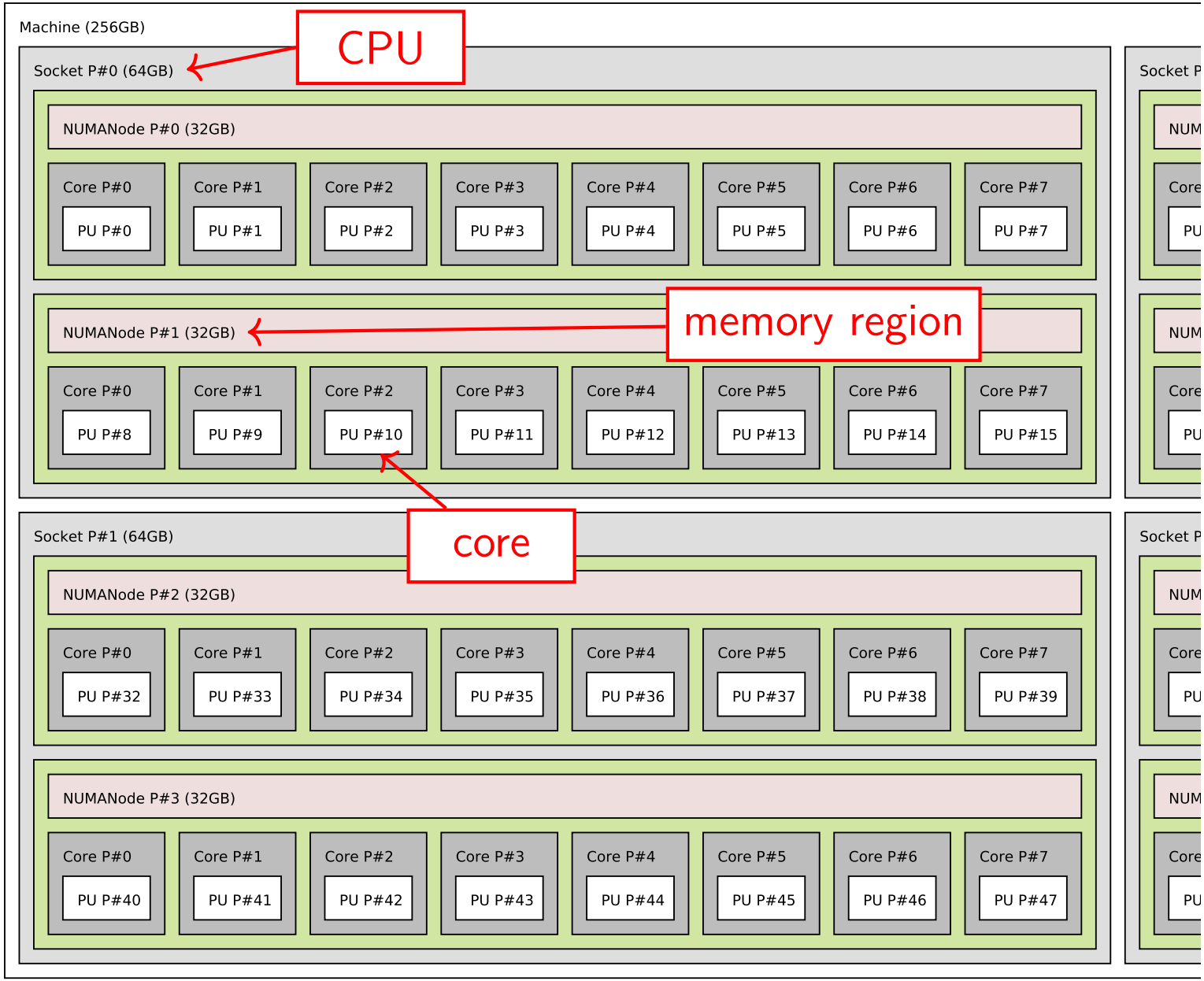
Assign Cores Deliberately

- Hyper Threading:
  Multiple threads sharing execution units
- Shared caches

Respect Non-Uniform Memory Access (NUMA)

- Memory regions have different performance depending on current CPU core
- Hierarchical NUMA makes things worse

Type `lstopo` on your machine (Ubuntu: package `hwloc`)



Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # ... (tool started here)

  ```
  killall z3 2> /dev/null
  killall minisat 2> /dev/null
  killall yices 2> /dev/null
  ```

- Thanks for thinking of cleanup

Isolate Individual Runs

- Excerpt of start script taken from some verifier in SV-COMP:

  # ... (tool started here)

  ```
  killall z3 2> /dev/null
  killall minisat 2> /dev/null
  killall yices 2> /dev/null
  ```

- Thanks for thinking of cleanup
- But what if there are parallel runs?

Isolate Individual Runs

- Temp files with constant names like `/tmp/mytool.tmp` collide
- State stored in places like `~/.mytool` hinders reproducibility
  - Sometimes even auto-generated
- Restrict changes to file system as far as possible

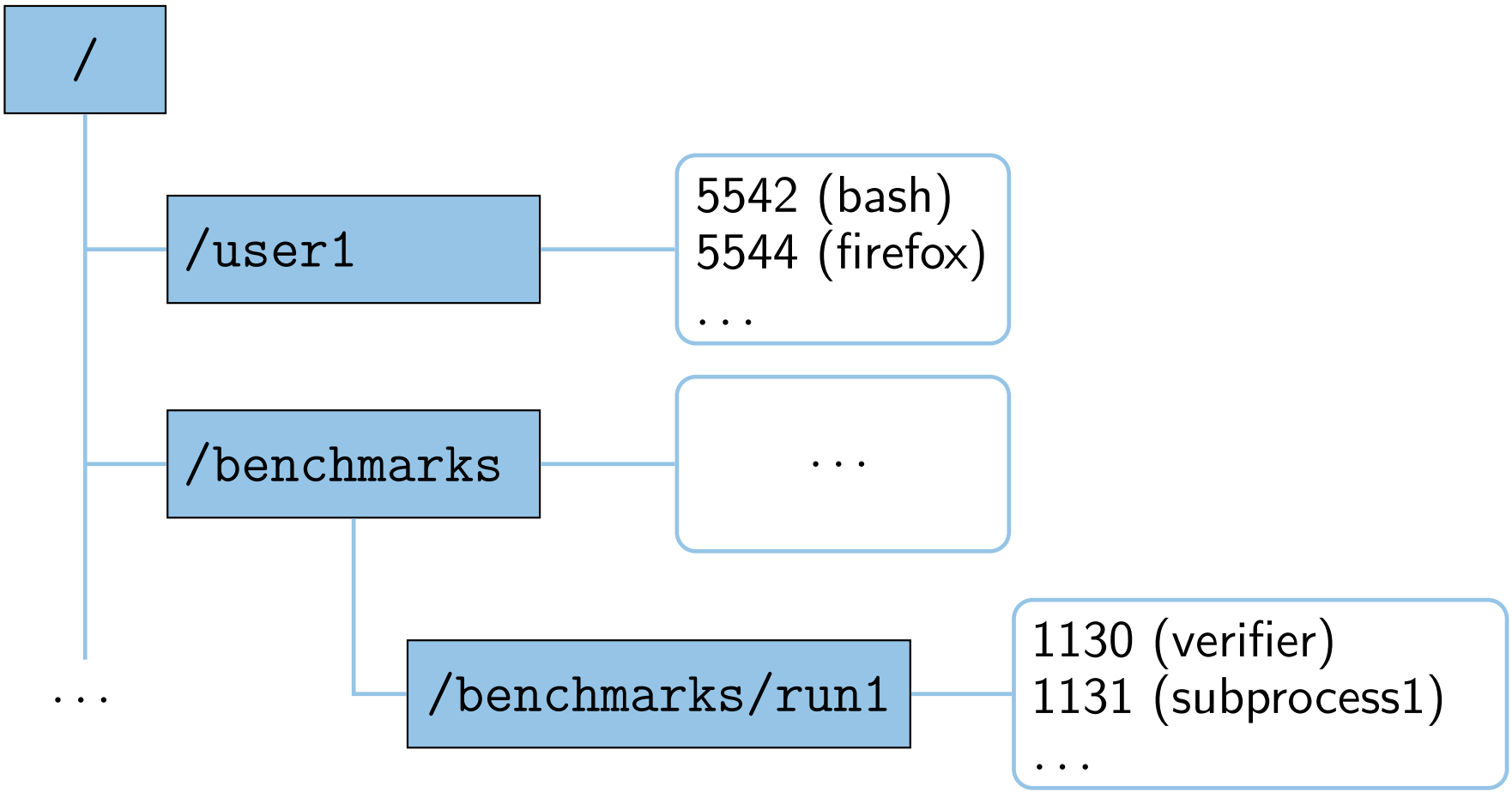
Cgroups

- Linux kernel “control groups”
- Reliable tracking of spawned processes
- Resource limits and measurements per cgroup
  - CPU time
  - Memory
  - I/O etc.

Only solution on Linux
for race-free handling of multiple processes!

Cgroups

- Hierarchical tree of sets of processes



Namespaces

- Light-weight virtualization
- Only one kernel running, no additional layers
- Change how processes see the system
- Identifiers like PIDs, paths, etc. can have different meanings in each namespace
  - PID 42 can be a different process in each namespace
  - Directory / can be a different directory in each namespace
  - ...
- Can be used to build application containers without possibility to escape
- Usable without root access

Benchmarking Containers

- Encapsulate groups of processes
- Limited resources (memory, cores)
- Total resource consumption measurable
- All other processes hidden
  and no communication with them
- Disabled network access
- Adjusted file-system layout
  - Private `/tmp`
  - Writes redirected to
    temporary RAM disk

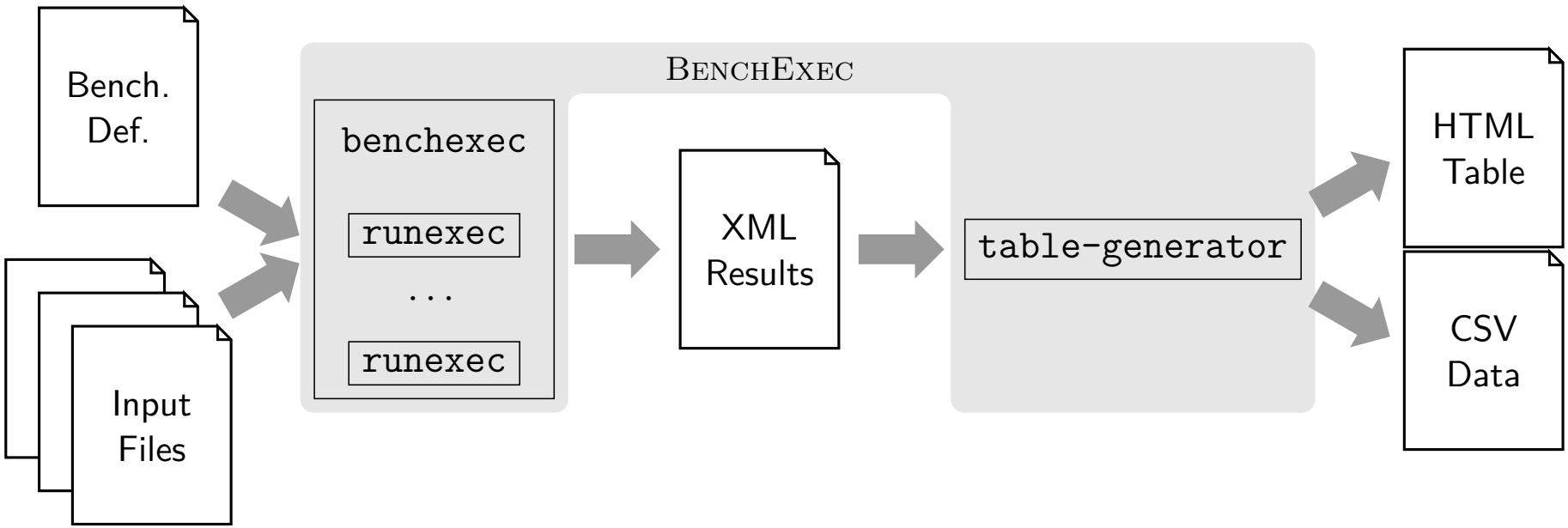
BenchExec

- A Framework for Reliable Benchmarking and Resource Measurement
- Provides benchmarking containers based on cgroups and namespaces
- Allocates hardware resources appropriately
- Low system requirements (modern Linux kernel and cgroups access)

BenchExec

- Open source: Apache 2.0 License
- Written in Python 3
- https://github.com/sosy-lab/benchexec
- Used in International Competition on Software Verification (SV-COMP) and by StarExec
- Originally developed for software-verification, but applicable to arbitrary tools

BenchExec Architecture



`runexec`

Benchmarks a single run of a tool (in container)

`benchexec`

Benchmarks multiple runs

`table-generator`

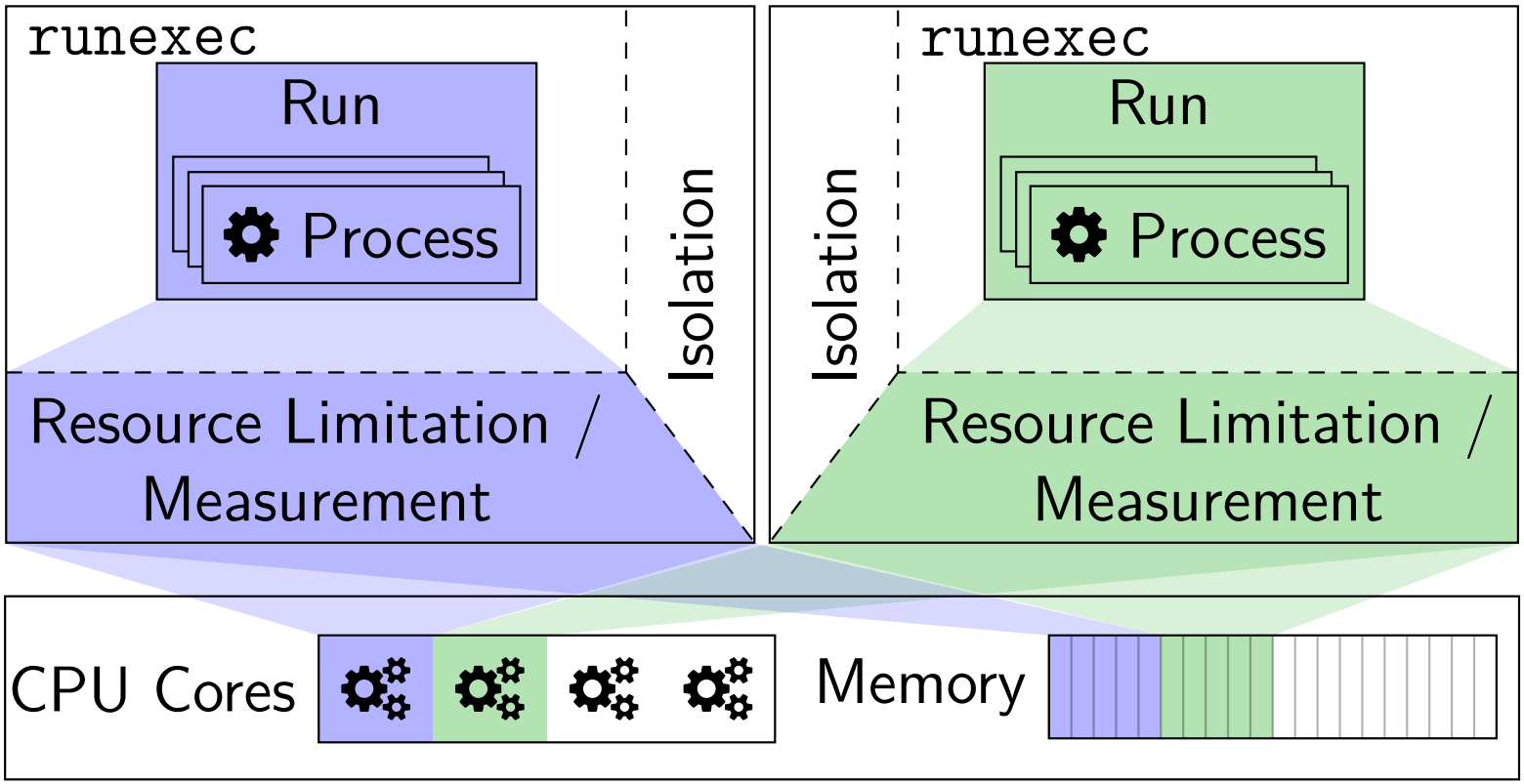
Generates CSV and interactive HTML tables

BenchExec: `runexec`

- Benchmarks a single run of a tool
- Measures and limits resources using cgroups
- Runnable as stand-alone tool and as Python module
- Easy integration into other benchmarking frameworks and infrastructure
- Example:

```
runexec --timelimit 100 --memlimit 16000000000
        --cores 0-7,16-23 --memoryNodes 0
        --<TOOL_CMD>
```

BenchExec: `runexec`



BenchExec: `benchexec`

- Benchmarks multiple runs
  (e.g., a set of configurations against a set of files)
- Allocates hardware resources
- Can check whether tool result is as expected
  for given input file and property

BenchExec: `table-generator`

- Aggregates results
- Extracts statistic values from tool output
- Generates CSV and interactive HTML tables (with plots)
- Computes result differences and regression counts

BenchExec Configuration

- Tool command line
- Expected result
- Resource limits
  - CPU time, wall time
  - Memory
- Container setup
  - Network access
  - File-system layout
- Where to put result files

Please Read More

Dirk Beyer, *Stefan Löwe*, and *Philipp Wendler*.
Reliable Benchmarking: Requirements and Solutions. [2]
STTT 2019

- More details
- Study of hardware influence on benchmarking results
- Suggestions how to present results
  (result aggregation, rounding, plots, etc.)

Conclusion — Part 1

Be careful when benchmarking!

Don't use time, ulimit etc.
Always use cgroups and namespaces!

BenchExec
https://github.com/sosy-lab/benchexec

Part 2: SV-COMP

11th Competition on Software Verification


Proc. TACAS 2022,
https://doi.org/10.1007/978-3-030-99527-0_20

Motivation - Goals

1. Community suffers from unreproducible results
   $\rightarrow$ Establish set of benchmarks
2. Publicity for tools that are available
   $\rightarrow$ Provide state-of-the-art overview
3. Support the development of verification tools
   $\rightarrow$ Give credits and visibility to developers
4. Establish standards
   $\rightarrow$ Specification language, Witnesses,
   Benchmark definitions, Validators
5. Train PhD students on benchmarking and reproducibility
6. Provide computing resources to groups that do not have large clusters

Schedule of Sessions

Session 1:

- Competition Report, by organizer
- System Presentations, 7 min by each team
- Short discussion

Session 2:

- Open Jury Meeting, Community Discussion, moderated by organizer

Procedure – Time Line

Three Steps – Three Deadlines:

- Benchmark submission deadline
- System submission
- Notification of results (approved by teams)

Verification Problem

Input:

- C program → GNU/ANSI C standard
- Property
  → Reachability of error label, of overflows
  → Memory safety (inv-deref, inv-free, memleak)
  → Termination

Output:

- TRUE + Witness (property holds)
- FALSE + Witness (property does not hold)
- UNKNOWN (failed to compute result)

Environment

Machines (1000 $ consumer machines):

- CPU: 3.4 GHz 64-bit Quad-Core CPU
- RAM: 33 GB
- OS: GNU/Linux (Ubuntu 20.04)

Resource limits:

- 15 GB memory
- 15 min CPU time (consumed 470 days)

Volume: 309 081 verification runs, 1.43 million validation runs
Incl. preruns: 2.85 million verification runs using 19 years, and 16.3 million validation runs using 11 years

Scoring Schema

Common principles: Ranking measure should be

- easy to understand
- reproducible
- computable in isolation for one tool

SV-COMP:

- Ranking measure is the quality of verification work
- Expressed by a community-agreed score
- Tie-breaker is CPU time

Scoring Schema (2022, unchanged)

Reported result	Points	Description
UNKNOWN	0	Failure, out of ressources
FALSE correct	$+1$	Error found and confirmed
FALSE incorrect	-16	False alarm (imprecise analysis)
TRUE correct	$+2$	Proof found and confirmed
TRUE incorrect	-32	Missed bug (unsound analysis)

Fair and Transparent

Jury:

- Team: one member of each participating candidate
- Term: one year (until next participants are determined)

Systems:

- All systems are available in open GitLab repo
- Configurations and Setup in GitLab repository
  $\rightarrow$ Integrity and reproducibility guaranteed

47 Competition Candidates

Qualification:

- 33 qualified, additional 14 hors concours
- 10 result validators, 1 witness linter
- One person can participate with different tools
- One tool can participate with several configurations (frameworks, no tool-name inflation)

Benchmark quality:

- Community effort, documented on GitLab

Role of organizer:

- Just service: Advice, Technical Help, Executing Runs

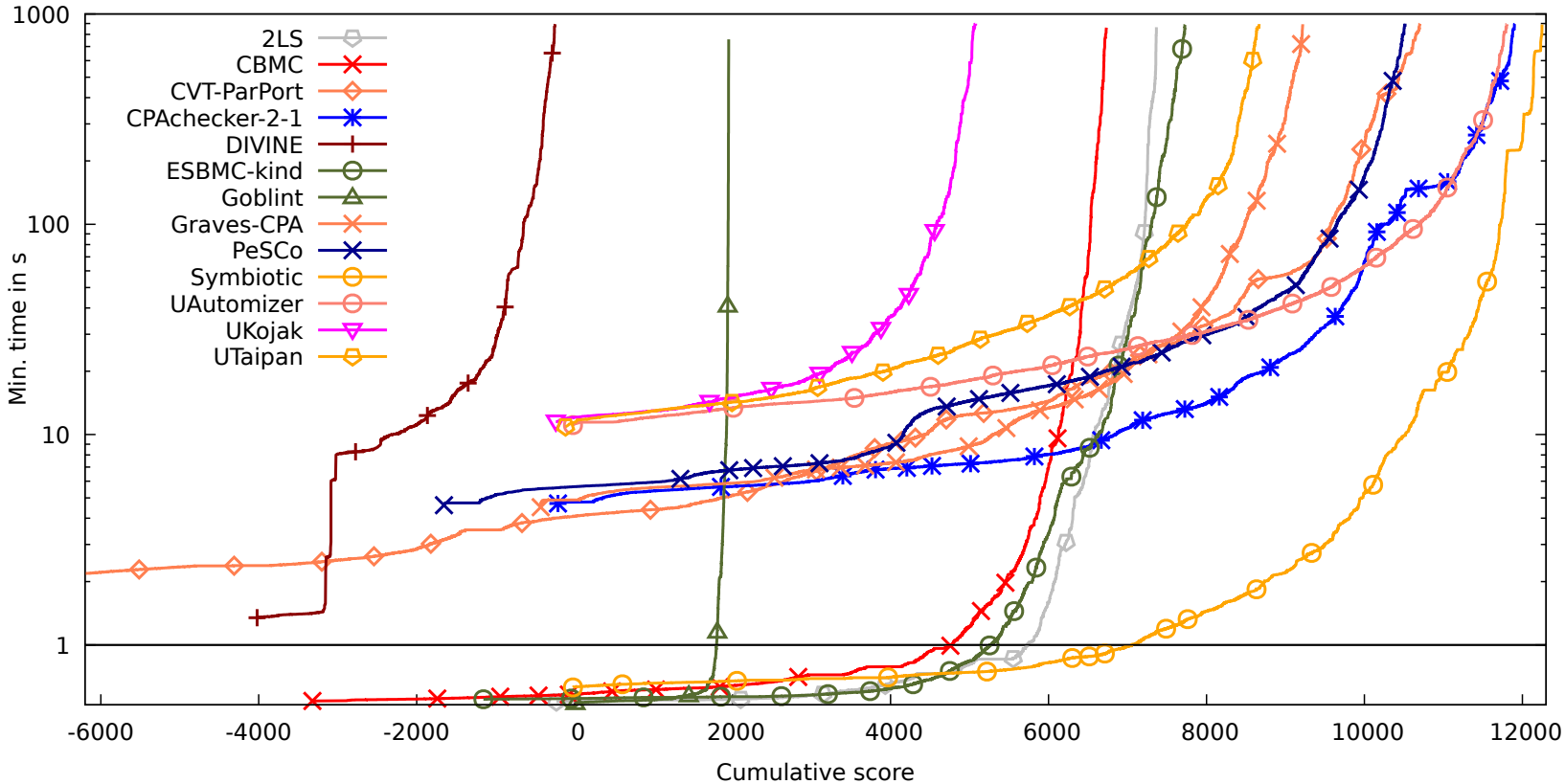
Benchmark Sets

- Everybody can submit benchmarks (conditions apply)
- Eight categories when closed (scores normalized):
  - Reachability: 5400 tasks
  - Memory Safety: 3321 tasks
  - Concurrency: 763 tasks
  - NoOverflows: 454 tasks
  - Termination: 2293 tasks
  - Software Systems: 3417 tasks
  - Overall: 15648 tasks
  - Java: 586 tasks

Replicability

- SV-Benchmarks:
  `https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks`
- SV-COMP Setup:
  `https://gitlab.com/sosy-lab/sv-comp/bench-defs`
- Resource Measurement and Process Control:
  `https://github.com/sosy-lab/benchexec`
- Archives:
  `https://gitlab.com/sosy-lab/sv-comp/archives-2022`
- Witnesses:
  `https://doi.org/10.5281/zenodo.5838498`

Results – Example: Overall



Impact / Achievements

- Large benchmark set of verification tasks
  $\rightarrow$ established and used in many papers
  for experimental evaluation
- Good overview over state-of-the art
  $\rightarrow$ covers model checking and program analysis
- Participants have an archived track record
  of their achievements
- Infrastructure and technology for
  controlling the benchmark runs (cf. StarExec)

[Competition Report and System Descriptions
are archived in Proceedings TACAS 2022]
https://doi.org/10.1007/978-3-030-99527-0_20

Alternative Rankings — Definitions

- Correct Verifiers — Low Failure Rate:

$$\frac{\text{number of incorrect results}}{\text{total score}}$$

with unit E/sp.

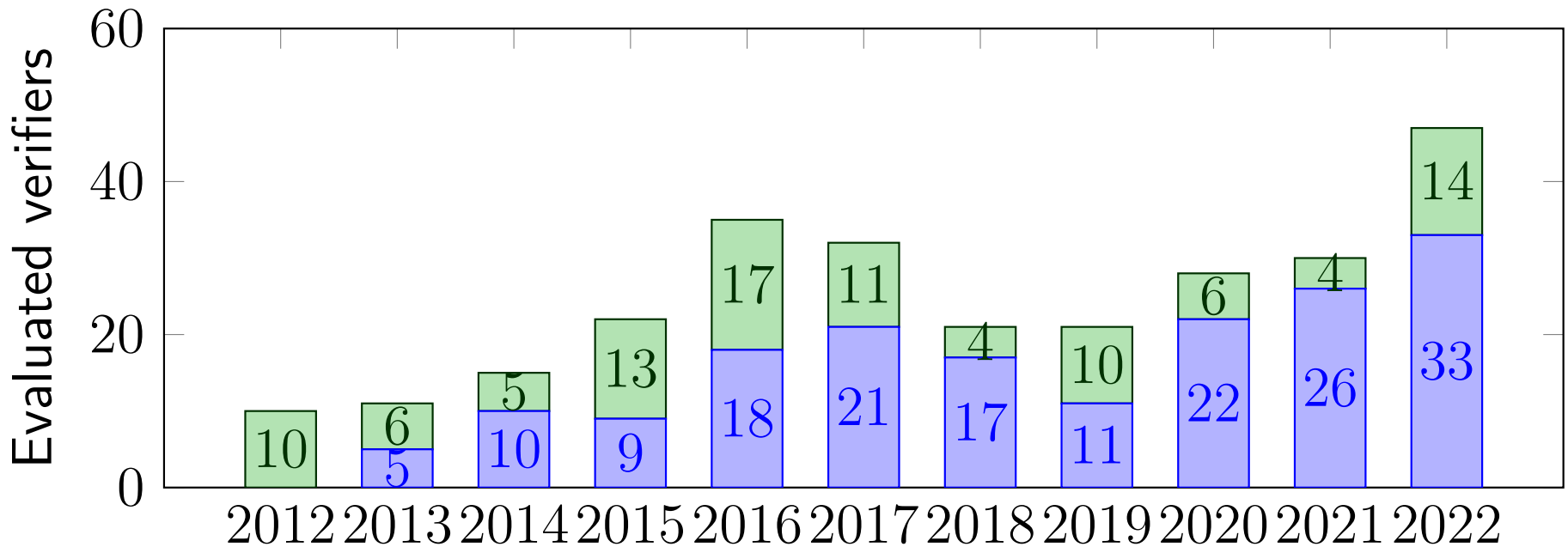
- Green Verifiers — Low Energy Consumption:

$$\frac{\text{total CPU energy}}{\text{total score}}$$

with the unit J/sp.

Number of Participants

Number of evaluated verifiers for each year
(first-time participants on top)



Different Techniques

Participant	CEGAR	Predicate Abstraction	Symbolic Execution	Bounded Model Checking	k-Induction	Property-Directed Reach.	Explicit-Value Analysis	Numeric. Interval Analysis	Shape Analysis	Separation Logic	Bit-Precise Analysis	ARG-Based Analysis	Lazy Abstraction	Interpolation	Automata-Based Analysis	Concurrency Support	Ranking Functions	Evolutionary Algorithms
2LS				✓	✓			✓			✓						✓	
AProVE			✓		✓		✓	✓		✓	✓						✓	
CBMC				✓							✓					✓		
CBMC-Path				✓							✓					✓		
CPA-BAM-BnB	✓	✓					✓				✓	✓	✓	✓				
CPA-Lockator	✓	✓					✓				✓	✓	✓	✓		✓		
CPA-Seq	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
DepthK				✓	✓		✓	✓	✓		✓		✓	✓		✓	✓	
DIVINE-explicit							✓				✓					✓		
DIVINE-SMT							✓				✓					✓		
ESBMC-kind				✓	✓						✓					✓		
JayHorn	✓	✓				✓		✓					✓	✓				
JBMC				✓							✓					✓		
JPF				✓			✓	✓			✓					✓		
Lazy-CSeq				✓							✓					✓		
Map2Check				✓							✓							
PeSCo	✓	✓		✓	✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	
Pinaka			✓	✓					✓		✓		✓			✓		
PredatorHP									✓									
Skink	✓						✓							✓	✓			
Smack	✓			✓		✓					✓		✓			✓		
SPF			✓						✓							✓		
Symbiotic			✓					✓			✓							
UAutomizer	✓	✓									✓		✓	✓	✓		✓	
UKojak	✓	✓									✓		✓	✓				
UTaipan	✓	✓									✓		✓	✓	✓			
VeriAbs	✓			✓	✓		✓	✓										
VeriFuzz				✓				✓										✓
VIAP																		
Yogar-CBMC	✓			✓							✓		✓			✓		
Yogar-CBMC-Par.	✓			✓							✓		✓			✓		

Competition Report [1]

https://doi.org/10.1007/978-3-030-17502-3_9

References I

[1] Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9

[2] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y