RERS 2016: Parallel and Sequential Benchmarks with Focus on LTL Verification

Maren Geske[1], Marc Jasper[1,2], Bernhard Steffen[1], Falk Howar[3], Markus Schordan[2], and Jaco van de Pol[4]

[1] TU Dortmund University, Programming Systems, Dortmund D-44227, Germany
{maren.geske, marc.jasper, steffen}@cs.tu-dortmund.de
[2] Lawrence Livermore National Laboratory, CA 94551, USA
{jasper3, schordan1}@llnl.gov
[3] Clausthal University of Technology, Clausthal-Zellerfeld, Germany
falk.howar@tu-clausthal.de
[4] University of Twente, Formal Methods and Tools, Enschede, The Netherlands
J.C.vandePol@utwente.nl

Abstract. The 5th challenge of Rigorous Examination of Reactive Systems (RERS 2016) once again provided generated and tailored benchmarks suited for comparing the effectiveness of automatic software verifiers. RERS is the only software verification challenge that features problems with linear temporal logic (LTL) properties in larger sizes that are available in different programming languages. This paper describes the revised rules and the refined profile of the challenge, which lowers the entry hurdle for new participants. The challenge format with its three tracks, their benchmarks, and the related LTL and reachability properties are explained. Special emphasis is put on changes that were implemented in RERS — compared to former RERS challenges. The competition comprised 18 sequential and 20 parallel benchmarks. The 20 benchmarks from the new parallel track feature LTL properties and a compact representation as labeled transition systems and Promela code.

1 Introduction

The RERS challenge is an annual verification challenge that focuses on LTL and reachability properties of reactive systems. The benchmarks are generated automatically from automata which allows to generate new problems each year that are previously unknown to the participants. The challenge was designed to explore, evaluate and compare the capabilities of state-of-the-art software verification tools and techniques. Areas of interest include but are not limited to static analysis [14], model checking [5,7,2], symbolic execution [11], and testing [4].

The focus of RERS is on principal capabilities and limitations of tools and approaches. The RERS challenge is therefore "free-style", i.e., without time and resource limitations and encouraging the combination of methods and tools. Strict time or resource limitations in combination with previously known solutions encourage tools to be tweaked for certain training sets, which could give a false impression of their capabilities. It also leads to abandoning time consuming problems in the interest of time. The main goals of RERS[5] are:

1. encouraging the combination of methods from different (and usually disconnected) research fields for better software verification results,
2. providing a framework for an automated comparison based on differently tailored benchmarks that reveal the strengths and weaknesses of specific approaches,
3. initiating a discussion about better benchmark generation, reaching out across the usual community barriers to provide benchmarks useful for testing and comparing a wide variety of tools, and
4. collecting (additional) interesting syntactical features that should be supported by benchmark generators.

To the best of our knowledge there is no other software verification challenge with a profile that is similar to that of RERS. The SV-COMP[6] [3] challenge is also concerned with reachability properties and features a few benchmarks concerning termination and memory safety. In direct comparison, SV-COMP does not allow the combination of tools and directly addresses tool developers. It has time and resource limitations, does not feature achievements, but has developed a detailed ranking system for the comparison of tools and tries to prevent guessing by imposing high penalties on mistakes. An important difference to SV-COMP is that RERS features benchmarks that are generated automatically for each challenge iteration, ensuring that all results to the verification tasks are unknown to participants.

Another challenge concerned with the verification of parallel benchmarks in combination with LTL properties is the Model Checking Contest [12] (MCC). The participants have to analyze Petri nets as abstract models and check LTL and CTL formulas, the size of the state space, reachability, and various upper bounds. The benchmark set consists of a large set of known models and a small set of unknown models that were collected among the participants. Participants submit tools, rather than problem answers. Tools that participate in MCC have to adhere to resource restrictions, which is not the case when participating in RERS. MCC uses randomly generated LTL formulas, but uses no mechanism to generate models that match them. In direct comparison to RERS, MCC features hand-written or industrial problems instead of automatically generated benchmarks.

Finally, VerifyThis [10] features program verification challenges. Participants get a fixed amount of time to work on a number of challenges, to prove the functional correctness of a number of non-trivial algorithms. That competition focuses on the use of (semi-)interactive tools, and submissions are judged by a jury. In direct comparison, RERS participants submit results that can be checked and ranked automatically; only the “best approach award” involves a jury judgment.

[5]As stated online at `http://www.rers-challenge.org/`

[6]`https://sv-comp.sosy-lab.org/`

This paper describes the challenge procedure of RERS 2016 and presents the three different tracks: Sequential LTL, Sequential Reachability, and Parallel LTL. Parallel benchmarks are a new addition to the RERS challenge. Their structure and the format of the Parallel LTL track are introduced within the following sections. Simplifications for the sequential benchmarks that were made to lower the entry hurdle for new participants are explained and a new solution verification tool for training problems is introduced. Throughout this paper, special focus is set on the changes compared to former RERS challenges.

Section 2 describes the overall layout and timeline of the RERS challenge 2016. The three tracks and their benchmarks are explained in Section 3. Section 4 presents the structure of individual benchmark programs, whereas Section 5 showcases examples of the provided properties that participants have to analyze. The scoring scheme and the submission format are defined in Section 6. Section 7 briefly discusses the benchmark generation process before Section 8 presents a conclusion and an outlook to future developments.

2 Challenge Procedure

The RERS challenge 2016 features sequential and parallel benchmarks. The sequential problems are divided into two tracks according to their verification tasks, LTL properties and reachability of errors. The parallel track only focuses on LTL properties. All challenge tasks are newly generated for each competition so they are unknown to the participants prior to the competition. No training of verifiers on the benchmarks is possible as the solutions are only released after the challenge is completed. Instead, sets of training problems help participants to test their tools before submitting actual results.

2.1 Sequential Benchmarks

In every year RERS provides a couple of training problems that are available with solutions. These problems allow contestants to get a feeling for the sequential programs, the related verification tasks, and the syntactical features that are newly introduced in a particular year. In order to ease the initial adaptation effort for participants, a tool is provided that takes a training solution file and a proposed solution of the participant and calculates the correct and wrong answers and total points scored for the problem[7].

The challenge procedure starts off with the release of the training problems and their solutions. This is consistent with all former editions of the challenge [9,6]. For this year's challenge, 8 training problems are provided for both sequential tracks, 16 problems in total with 100 properties each. A detailed description of the problem format can be found in Section 3. The training problems are small in size, but have the same complexity and syntax as the challenge problems for each category (i.e., plain, arithmetic, and data structures).

[7]www.rers-challenge.org/2016/index.php?page=trainingphase

The challenge phase starts with the release of the actual benchmarks. The time between the release of the problems and the submission deadline was set to four and half months. There are 18 benchmarks, 9 for each of the two tracks, LTL and Reachability verification. Each benchmark comes with 100 properties of the track category that need to be verified or falsified. The verifiers do not have to be submitted. Only the given answers are evaluated for the final ranking and achievements. In addition, an award for the best approach is given out by the competition committee based on submitted descriptions.

2.2 Parallel Benchmarks

The parallel benchmarks of RERS are a new addition in 2016 and were released a few months later than the sequential problems. These 20 different problems from the parallel category feature from 1 to 20 parallel automata. They each come with 20 LTL properties that participants can analyze.

Participants had about four weeks for analyzing the parallel benchmarks. Because of this shorter time frame, separate training problems were omitted in favor of a structured sequence of problems with an increasing number of parallel automata. The first 5 problems can be analyzed completely by existing tools such as SPIN [8]. Their results can be used as a reference by participants which lowers the hurdle to enter the challenge.

3 Challenge Format and Categories

This section describes the verification tasks of the individual benchmarks and the different categories that the RERS 2016 challenge consists of.

3.1 Verification Tasks

Each sequential or parallel problem comes with 100 or 20 properties respectively. The participants have to check whether or not the individual properties are satisfied. The possible answers are defined as follows:

True. The property is satisfied, there exists no path that violates the property.
False. The property is violated, there exists at least one path that violates the property.
No answer given. The participant was not able to find an answer to this question.

The submission of counterexample traces for violated properties is not a requirement. Only the answers described above are used for the ranking and achievement evaluation (see Section 6).

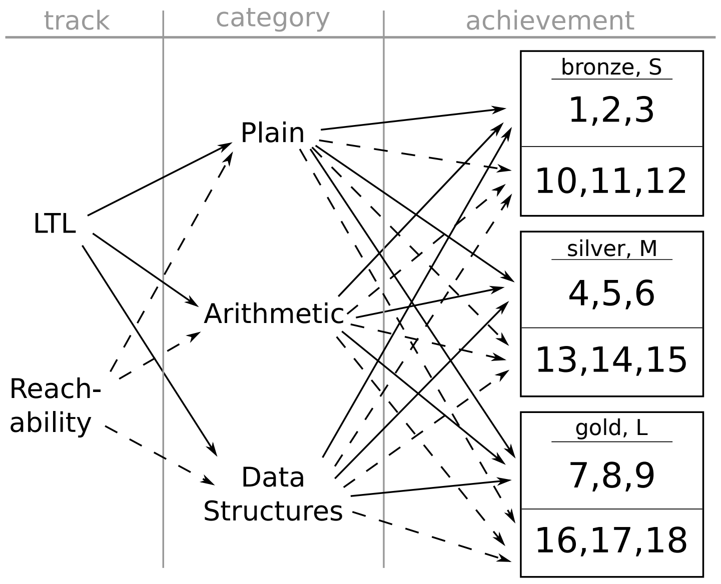


Fig. 1: Sequential Benchmarks for RERS 2016

3.2 Sequential Benchmarks

The sequential benchmarks are grouped into two tracks, i.e., LTL and Reachability, that correspond to the property type that has to be analyzed (see Section 5). The LTL properties are specified in additional files and distributed with the benchmark programs. Figure 1 gives an overview of the generated benchmarks and their respective category and track (dashed lines for Reachability, solid lines for LTL) and the achievements that can be gained (see Section 6 for details and thresholds). For each track there are three categories that represent the syntactical features included in the benchmarks belonging to the respective category.

Plain. The program only contains assignments, with the exception of some scattered summation, subtraction, and remainder operations in the reachability problems.

Arithmetic. The programs frequently contain summation, subtraction, multiplication, division, and remainder operations.

Data structures. Arrays and operations on arrays are added. Other data structures are planned for the future.

Some of this year's problems from the plain category in the Reachability track also contain a few arithmetic operations. The LTL track is not affected. The reason for the existence of these operations is an improved method of inserting (un-)reachable errors into the program. Arithmetic operations in the plain category are planned to be removed for next year's challenge.

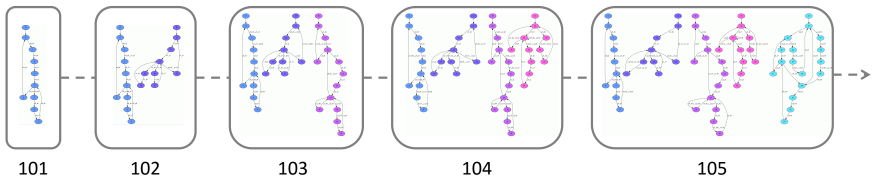


Fig. 2: Sequence of Parallel Benchmarks

In the rightmost column of Figure 1, the problem numbers are grouped according to the type of achievement that can be gained. The achievement levels also correspond to the size of the benchmarks: bronze are small, silver are medium-sized, and gold are large programs. In each line the first problem is of the *Plain* category, the middle of the *Arithmetic* and the last of the *Data Structures* category.

3.3 Parallel Benchmarks

The benchmarks in the parallel track of RERS 2016 form a sequence of problems with increasing difficulty. Instead of scaling complexity through code obfuscation, these benchmarks become harder to analyze by featuring an increasing number of parallel components within a given parallel system as is shown in Figure 2. One component is added for each new problem. In addition, the entire communication within the parallel system changes due to a different transition relabeling (see also Section 4.2). This can lead to an entirely new space of reachable states even though the structure of automata from smaller problems is reused.

Individual components of a parallel system are defined as labeled transition systems. Multiple components run in parallel, communicating with each other, and reacting to input from the environment. The reachable state space of the parallel system becomes larger as the number of parallel components increases, potentially posing a challenge to the verification of the provided LTL properties. The parallel benchmarks focus on the communication between components. Actual parallel computation is therefore not modeled but could occur at every state of the individual transition systems. The concrete semantics of the benchmarks are explained in Section 4.2.

4 Program Structure and Available Formats

The different types of programs that are part of the RERS 2016 benchmarks are explained in the following paragraphs, along with their structural properties and changes compared to previous challenge iterations.

4.1 Sequential Benchmarks

The sequential programs are available as C and Java code. The overall structure of the source code is the same for all of these benchmarks. They represent in-

```
int x1 = 1;
int x2 = 2;
...
void calc_output(int);

int calc_output(int in) {
 if(in == 3 && x7 != 0) {
  x9 = 4;
  return 24;
 }
 ...
}
```

(a) Variables and Control Logic

```
int main() {
 while (1) { // main i/o-loop
  int input;
  scanf("%d", &input);
  if((input != 2) && (input != 5)
 && (input != 1) &&
  (input != 3) && (input != 4))
      return -2;
  calc_output(input);
 }
}
```

(b) Main Function with Infinite Loop

Fig. 3: Example of a Sequential Benchmark Program in C

stances of event-condition-action systems that are used for example in logic controllers [1] and database management systems [13]. An illustrating code snippet is depicted in Figure 3. Each program consists of a main function with an infinite `while`-loop that reads an input and passes it to the `calc_output`-function[8] that contains the program logic and computes an output. The logic is organized in nested `if-else`-blocks and contains syntactical operations according to the benchmark's category. Compilation instructions are provided on the website[9].

Improved Benchmark Code. In contrast to former challenges, all inputs that are not eligible are rejected before the internal logic is evaluated. This way the problems are now self-contained and it is not necessary anymore to pass the input alphabet to the verifier. Moreover, this change makes the main function equal to the versions of RERS 2012 benchmarks that are used in SV-COMP[3], allowing all participants to use their tools without modifications on the new benchmarks for 2016. In order to assure that the code is valid C++ code, all functions are previously defined.

Predefined Functions. In order to ease the entry level for new participants, the former "error syntax" (i.e., an assertion with the error number) has been removed from the benchmarks. It was replaced by an external function for C99 and C++ programs, e.g., __VERIFIER_error(5) for the error number 5, and by a static function of a fictional `Errors` class in Java, e.g., Errors.__VERIFIER_error(5) for the error number 5. An implementation that simulates the semantics

[8]The name was shortened for space reasons, in the challenge the function is named `calculate_output`

[9]Java version: `rers-challenge.org/2016/index.php?page=java-code`
C version: `rers-challenge.org/2016/index.php?page=c-code`

used in RERS prior to 2016 can be viewed in Figure 4. The reference implementation can be replaced by a suitable implementation for the verifier or can just be interpreted semantically.

```
void __VERIFIER_error(int);

void __VERIFIER_error(int i) {
 fprintf(stderr,"error_%d_",i);
 assert(0);
}
```

(a) Reference Implementation in C99/C++

```
public class Errors {
 public static void
 __VERIFIER_error(int i) {
    throw new
    IllegalStateException(
       "error_" + i );
 }
}
```

(b) Reference Implementation in Java

Fig. 4: Reference Implementations for Simulating pre-2016 Error Behavior

4.2 Parallel Benchmarks

The new parallel programs are available in two different formats. First, the benchmarks are represented as a cluster of labeled transition systems. Second, a Promela [8] version is provided that implements the transition systems as parallel processes.

Cluster of Labeled Transition Systems. Each benchmark is available as a DOT[10] graph (`.dot` file) containing the components of the parallel systems as clusters in a directed graph (Figure 5). These clusters are understood as non-deterministic labeled transition systems. A run of such a transition system starts at the single state without incoming transitions. There are three types of labels within the transition systems:

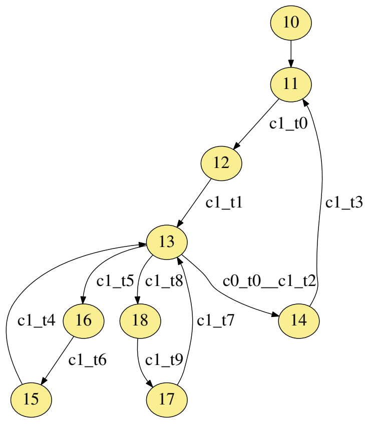
Environment Input. Labels that only occur in a single transition system can be triggered independently, when they are enabled (e.g. “c1_t0” in Figure 5b).

Empty Label. Similarly, transitions without a label are understood as internal transitions that can be triggered at any time. For RERS 2016, these only exist as initial transitions because they do not add to the communication with other components or the environment. As initial transitions however, empty labels help to ensure that both the DOT graph and the Promela code feature the same semantics regarding LTL properties.

[10] `http://www.graphviz.org/content/dot-language`

```
digraph G {
subgraph component1 {
  10 [label="10" ... ];
  11 [label="11" ... ];
  12 [label="12" ... ];
  13 [label="13" ... ];
  14 [label="14" ... ];
  ...
  11->12 [label="c1_t0"...
  12->13 [label="c1_t1"...
  14->11 [label="c1_t3"...
  15->13 [label="c1_t4"...
  13->16 [label="c1_t5"...
  ...
} }
```

(a) Graph as DOT File



(b) Graph Visualization

Fig. 5: Component of a Parallel System as DOT Graph

Communication. Non-empty labels that occur in multiple transition systems are synchronized (rendezvous communication). They only exist in pairs and can only be triggered simultaneously, when they are both enabled (e.g., "c0_t0__c1_t2" in Figure 5b).

A transition is enabled if and only if the corresponding components are currently in the states preceding that transition. The only known fact about the environment is that it does not introduce new deadlocks: One of the enabled transitions (if any) will eventually be triggered.

Promela Code. Each parallel benchmark is available as Promela code. An example is shown in Figure 6: Within this version, every component of the parallel system is implemented as a parallel process (`proctype`). The environment is represented by an additional parallel process and sends random messages to the parallel components which triggers their transitions. All message channels are unbuffered to realize rendezvous communication. For simplicity, transition-system-internal empty labels in the graph representation are also triggered by the environment process in the Promela version (`nop` message). The rendezvous communication between different parallel components is also realized via message passing.

The Promela program contains a single global variable `lastAction` and an additional parallel process `Listener`. This listener gets notified about every message that is sent in between the parallel components or between the parallel system and the environment. The listener always stores the most recent message in `lastAction`. The sequence of values stored in variable `lastAction` describes an abstract trace of the Promela program that matches the trace of transitions in the respective cluster of labeled transition systems. The LTL properties are therefore defined based on the content of variable `lastAction` (see also Section 5.1). Note that the addition of this variable increases the state space, which would be unnecessary for solutions that are based on action-based properties.

```
/* Actions (messages) */
mtype = { nop, c1_t7, ...

/* Inter-process channels */
chan p1_0 = [0] of {mtype};
...

/* Env<->process channels */
chan p0 = [0] of {mtype};
...

/* Environment */
active proctype Env()
{
do
:: p1 ! c1_t7
:: p0 ! nop
...
od
}

/* Action channel */
chan act = [0] of {mtype};

/* Most recent message */
mtype lastAction = nop;

/* Action listener */
active proctype Listener() {
atomic {
do
:: act ? lastAction ->
step: skip
od
} }
```

(a) channels, environment, and listener

```
/* Process 1 */
active proctype Proc1() {
int state = 10;
do
:: state == 10 ->
  atomic {
  if
  :: p1 ? nop ->
    state = 11;
  fi
  }
:: state == 11 ->
  atomic {
  if
  :: p1 ? c1_t0 ->
    act ! c1_t0;
    state = 12;
  fi
  }
...
:: state == 13 ->
  atomic {
  if
  :: p1 ? c1_t5 ->
    act ! c1_t5;
    state = 16;
  :: p1 ? c1_t8 ->
    act ! c1_t8;
    state = 18;
  :: p1_0 ! c0_t0__c1_t2 ->
    state = 14;
  fi
  }
...
}
```

(b) parallel component from Figure 5

Fig. 6: Promela Code Example

5 Properties and Their Representation

The sequential benchmarks contain the two tracks: LTL and Reachability. The new parallel problems only provide LTL properties. As mentioned in Section 4, the sequential benchmarks are now designed to only contain properties of their respective track type instead of both (which was the case in former challenges). The separation of properties leads to a more understandable semantics of the LTL properties: In past challenges, a definition of "error-free behavior" was required to specify on which execution paths the LTL properties needed to be checked.

5.1 LTL Properties

This section explains the structure of the LTL properties provided as part of the RERS challenge benchmarks.

Sequential Benchmarks. For each sequential problem in the LTL track there are 100 LTL formulas that have to be checked. The properties are provided in a property file (extension `.txt`) and contain the input and output symbols that are used within these formulas. Figure 7 shows an exemplary property file for a sequential benchmark. Each formula has an identifying number marked by `#` and ranging from 0 to 99. This identifier is followed by a textual description of the property. The following line contains the actual LTL property formulated in the syntax and semantics already explained in [16].

```
#inputs [[A, B, C, D, E]]
#outputs [[X, Y, Z, U, V, W, S, T]]
#0: output W, output V responds to input C after output U
(false R (! oU | (false R (! iC | (oW & X (true U oV))))))

#1: input B precedes output Y before output X
(! (true U oX) | (! oY U (iB | oX)))
```

Fig. 7: Extract from an LTL Property File (Sequential Benchmark)

Parallel Benchmarks. The parallel benchmarks include property files similar to the sequential ones described above. Instead of specifying input-output behavior, they contain LTL formulas over transition labels in the respective parallel automata. For the LTL verification, a trace of a parallel benchmark is always understood as its sequence of transition labels. The declaration of specific alphabets is therefore omitted. To simplify the representation, these LTL formulas contain some additional operators such as `=>` for the regular implication[11]. Currently, no textual description of the LTL properties is provided for parallel benchmarks.

In addition to being included in the property files (`.txt`), the LTL properties are directly part of the Promela code (`.pml`). Within the Promela file, the formulas are represented in an equivalent SPIN syntax. Figure 8 shows an example of one LTL property in both RERS and SPIN representations[11] (`W` and `V` meaning "weak until" and "release", respectively). The `#define` statements are taken from the Promela code and ensure that only those messages received by the listener count as transitions for the LTL properties (see Section 4.2).

```
#4:
( G(c2_t9 => (!c2_t6 W c4_t3)) )
```

(a) RERS Property File

```
#define p_c2_t9 (lastAction == c2_t9)
#define p_c4_t3 (lastAction == c4_t3)
#define p_c2_t6 (lastAction == c2_t6)

ltl p4 { [](!p_c2_t9 || (p_c4_t3 V (!p_c2_t6 || p_c4_t3))) }
```

(b) Promela Version

Fig. 8: An LTL Property from a Parallel Benchmark

5.2 Reachability Properties

Each sequential Reachability problem comes with 100 properties that have to be analyzed. Other than the LTL properties, the Reachability properties are implicitly provided in the program's source code in form of error function calls that are described in Section 4.1. The number that is passed to the predefined function call corresponds to the error number and falls in the range of 0 and 99. A property to be verified or falsified is that the error function `__VERIFIER_error(x)` is never executed for some particular x in the range of 0 to 99.

6 Scoring Scheme

RERS has a 3-dimensional reward structure that consists of a competition-based ranking on the total number of points, achievements for solving benchmarks, and an evaluation-based award for the most original idea or a good combination of methods. Apart from the evaluation-based ranking, both the achievements and

[11] For detailed definitions, please refer to `https://spot.lrde.epita.fr/trans.html`

the competition rules have been changed for this year's competition to make the challenge more appealing to participants.

The new parallel benchmarks only feature a ranking based on the achieved score, achievements in this track are left for future iterations of RERS. As participating with someone else's tool is possible in all tracks, each tool used by the participant is listed with his or her submission. This ensures that no tool can be discredited by improper usage.

6.1 Achievements

To honor the accomplishments of verification tools and methods without the pressure of losing in a competition despite good results, RERS introduced achievements for different nuances of difficulty. For every sequential category there are 3 achievements: bronze, silver and gold. Achievements are awarded for reaching a threshold of points that is equal to the number of counterexamples that can be witnessed for the corresponding group of benchmarks, as long as no wrong answer is given. Counterexamples are paths reaching an error function for the Reachability track and paths violating LTL properties for the LTL track. Only the highest achievement for every category is awarded and the thresholds for every category are calculated as follows:

- $bronze = $ #falsifiable properties of small problem
- $silver = bronze +$ #falsifiable properties of medium problem
- $gold = silver +$ #falsifiable properties of large problem

The participant's achievement score within a category is computed from all submitted results (verified or falsified):

$$\text{achievement\_score}(category) = \begin{cases} (\#\text{submitted}(category, small) & \text{if } 100\% \\ +\#\text{submitted}(category, medium) & \text{correct} \\ +\#\text{submitted}(category, large)) & \text{results} \\ 0 & \text{otherwise} \end{cases}$$

For example let achievement_score($plain$) >= $bronze(plain)$ and achievement_score($plain$) $<$ $silver(plain)$, then the participant is awarded the Bronze Achievement in the plain category. In total it is possible to gain 6 achievements, 3 for each sequential track, matching the number of different categories within a track.

6.2 Competition-based Ranking

The most significant change compared to previous iterations is that RERS 2016 has no overall ranking for the whole challenge anymore, but separate rankings for each track. This will highlight specialized tools that are excellent in a single track, but do not contribute to other tracks. Moreover, only one submission is allowed for both the achievements and the ranking for sequential problems. Reducing the submission to a single set of answers should prevent guessing of unknown properties. Previous RERS issues allowed a restricted form of guessing by having a mild penalty for wrong answers. The motivation was to differentiate incomplete approaches that cover large parts of the state space from approaches that only covered a small part. The current issue discourages guessing, since we want to focus on solutions for complete verification or falsification. Therefore, we considerably raised the penalty for wrong solutions. Participants can opt out of the ranking and will thus only appear on the website if they successfully gained an achievement. The scoring scheme for the competition ranking works as follows.

Correct answer. The participant receives 1 point.

No answer. The score remains unchanged if no answer was submitted.

Incorrect answer. The penalty is calculated over all mistakes in a track and corresponds to an exponential penalty for all mistakes of 2 to the power of n, where n is the number of mistakes made.

The revised penalty for mistakes hardly punishes unexpected mistakes, which can always happen by programming mistakes for example. However, the penalty for systematic guessing, which usually produces several mistakes, is severe.

6.3 Solution Format

The solution format is straightforward and has to be submitted in comma-separated value (CSV) format. An example can be seen in Figure 9 where

- **no** is the problem number (unique for the challenge)
- **spec** is the property specification number (error code or number of the LTL property in the LTL property file)
- **answer** expresses whether or not the property is satisfied, i.e., the LTL formula is satisfied or the error function call is unreachable. It can be specified as true/false, yes/no or 1/0.

```
<no1>,<spec1>,<answer1>
<no2>,<spec2>,<answer2>
...
```

Fig. 9: Format for Submitted Solutions

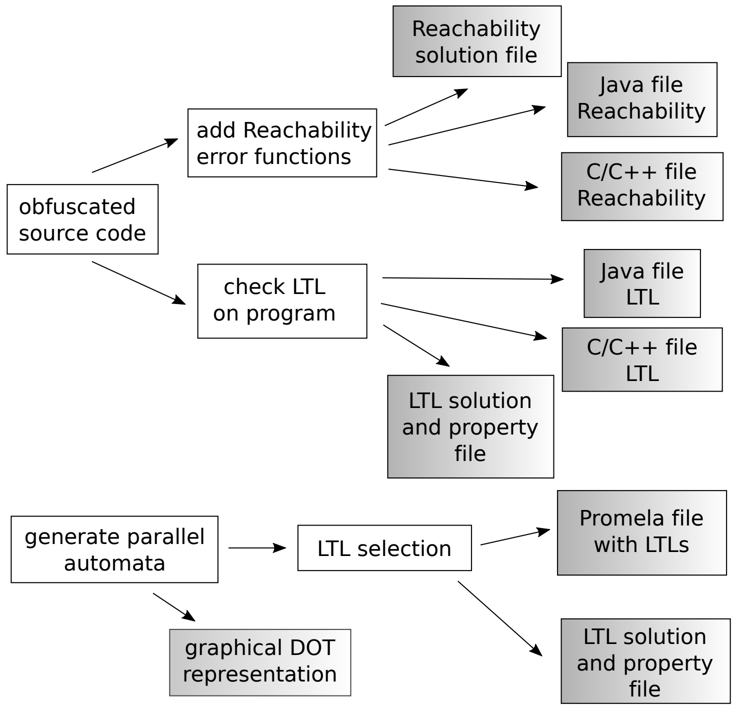

Reachability solution file
Java file Reachability
add Reachability error functions
C/C++ file Reachability
obfuscated source code
Java file LTL
check LTL on program
C/C++ file LTL
LTL solution and property file
generate parallel automata
LTL selection
Promela file with LTLs
graphical DOT representation
LTL solution and property file

Fig. 10: Benchmark Generation for RERS 2016

7 Generation Process

This section explains how the sequential benchmark generation in 2016 differs when compared to previous iterations of the RERS challenge and briefly sketches the generation of the parallel benchmarks . An overview of the abstract generation process and all generated files is included in Figure 10.

The parallel benchmark generation uses a new concept and has been implemented in the tool CodeThorn that is based on the ROSE compiler infrastructure [15]. As a first step, the set of parallel automata is generated and a graphical DOT representation is exported to give an overview of the system. LTL formulas are then automatically generated, tested, and 20 difficult properties are selected. Afterwards, a Promela version of the parallel system is generated that includes the LTL formulas. These properties are also exported as separate solution and property files. Details of the generation process will be explained in an upcoming paper.

The process to generate the sequential benchmarks remained the same as in [16]. Figure 10 shows how it diverges after the basic skeleton of the obfuscated source code has been created. In former challenges all properties were used for all types of benchmarks. In comparison, RERS 2016 separated LTL and Reachability properties to make the challenge more transparent. The only change to the generator when compared to former challenges is that instead of inserting error function into all problems, they are only added to the Reachability benchmarks for which no property file is exported. Only for the LTL benchmarks the LTL properties are checked and exported as a solution and property file. The language export of the sequential benchmarks is not specific to the benchmark track or category and generates a Java and C99 version for each benchmark.

8 Conclusion and Perspectives

The RERS challenge 2016 was the fifth iteration of the challenge and was used to establish a clearer profile and strengthen the position as an LTL challenge. The reachability properties were separated from LTL properties to ease the entry hurdle and to remove misunderstandings concerning the semantic of an “error-free” behavior from former challenges. The rules were slightly adapted to further discourage guessing of results. This paper describes all changes compared to the challenge of 2015 in detail and gives a clear overview of the structure and rules that are valid for the 2016 RERS challenge.

Furthermore, we added a new aspect to automatic benchmark generation with parallel benchmarks that are provided both as a graph representation and as Promela code. Parallel benchmarks are used to strengthen the LTL aspect of the challenge, by adding 20 LTL properties for each of the 20 problems in the parallel track. We plan to build on these initial parallel benchmarks during future iterations of RERS, for example by adding additional versions in other programming languages.

Looking back at the evolution of RERS, the challenge in 2012 contained only simple programs from the plain category that were later enhanced with arithmetic calculations for the online challenge. The challenge in 2013 featured white-box and black-box problems with more complex control structures. 2014 added data structures to the set of available syntactical features and extended the variety of available small modifications like larger input alphabets. The challenge in 2015 finally added benchmarks for monitoring as it was co-located with the conference on Runtime Verification[12].

The long-term goal of the RERS challenge is to establish open source benchmark generators that can be used to generate tailored benchmarks for an easy comparison of different tools and techniques.

References

1. E Emanuel Almeida, Jonathan E Luntz, and Dawn M Tilbury. Event-condition-action systems for reconfigurable logic control. *IEEE Transactions on Automation Science and Engineering*, 4(2):167–181, 2007.
2. D. Beyer. Status report on software verification. In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.

[12]`http://rv2015.conf.tuwien.ac.at/`

3. Dirk Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*, pages 887–904, New York, NY, USA, 2016. Springer-Verlag New York, Inc.
4. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, LNCS 3472. Springer, 2005.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.
6. Maren Geske, Malte Isberner, and Bernhard Steffen. Rigorous examination of reactive systems: The RERS challenge 2015. In *Runtime Verification: 6th International Conference*, pages 423–429. Springer, 2015.
7. G. J. Holzmann and M. H. Smith. Software model checking: Extracting verification models from source code. *Softw. Testing, Verification and Reliability*, 11(2), 2001.
8. Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.
9. F. Howar, M. Isberner, M. Merten, B. Steffen, D. Beyer, and C. Păsăreanu. Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. *Software Tools for Technology Transfer*, 16(5):457–464, 2014.
10. Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. VerifyThis 2012 - A program verification competition. *STTT*, 17(6):647–657, 2015.
11. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
12. F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trịnh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest. http://mcc.lip6.fr/2016/results.php, June 2016.
13. Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *ACM Sigmod Record*, volume 18, No. 2, pages 215–224. ACM, 1989.
14. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
15. Markus Schordan and Dan Quinlan. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference*, pages 214–223. Springer, 2003.
16. Bernhard Steffen, Malte Isberner, Stefan Naujokat, Tiziana Margaria, and Maren Geske. Property-driven benchmark generation: Synthesizing programs of realistic structure. *Software Tools for Technology Transfer*, 16(5):465–479, 8 2014.