International Journal on Software Tools for Technology Transfer (2021) 23:917–930
https://doi.org/10.1007/s10009-021-00617-z

COMPETITIONS AND CHALLENGES

Special Issue: TOOLympics 2019

The RERS challenge: towards controllable and scalable benchmark synthesis

Falk Howar[1] · Marc Jasper[1] · Malte Mues[1] · David Schmidt[1] · Bernhard Steffen[1]

Accepted: 6 May 2021 / Published online: 24 June 2021

Abstract
This paper (1) summarizes the history of the RERS challenge for the analysis and verification of reactive systems, its profile and intentions, its relation to other competitions, and, in particular, its evolution due to the feedback of participants, and (2) presents the most recent development concerning the synthesis of hard benchmark problems. In particular, the second part proposes a way to tailor benchmarks according to the depths to which programs have to be investigated in order to find all errors. This gives benchmark designers a method to challenge contributors that try to perform well by excessive guessing.

Keywords Benchmark generation · Verification competitions · Error witnesses · Temporal logic · LTL synthesis · Büchi automata · Modal contracts · Parallel decomposition · Model checking · Bisimulation checking

1 Introduction

Competitions and challenges have provided a valuable contribution to the development of verification and analysis tools, and numerous events of this kind have evolved over the last decades [4,6,8,29,32,36,43]. The approaches followed by these in many cases recurring events vary from off-site to on-site, with or without concrete resource constraints, from solution orientation to tool orientation, from known benchmark problems to problems with unknown true properties to controlled, generated benchmarks, and from qualitative/human evaluation to automated evaluation processes etc. (cf. Sect. 2 and [5]).

The RERS challenge is characterized by its property-oriented benchmark generation: benchmarks are automatically generated in a "requirements-driven" fashion. More precisely, the starting point of the benchmark generation process is a set of desired structural properties, here formulated in LTL, which are successively transformed via Büchi automata that characterize all satisfying executions to Modal Transitions Systems and, with a few more steps, transformed to code of various implementation languages (cf. Sect. 2.4). This construction principle aims at benchmarks that closely resemble realistic code, but can be flexibly tailored in their degree of difficulty. Originally, we considered size, amount of arithmetic operations, and the data structures used as a measure for intricacy. Over the years, the importance of controlling the length of shortest counterexamples as a means for scaling the difficulty of the verification task (in contrast to the complexity of the benchmark problem) became more and more apparent.

This paper consists of two parts: The first part (Sect. 2) summarizes the history of RERS, its profile and intentions, its relation to other competitions, and, in particular, its evolution due to the feedback of participants. This also comprises a discussion of experienced 'oddities', both at RERS and in relation to other competitions, as well as ways to overcome them. The second part (Sect. 3) presents our most recent development concerning the control of counterexample lengths. The proposed tailoring of benchmarks focusing on the depths to which programs have to be investigated in order to find all errors gives benchmark designer a way to

✉ Marc Jasper
marc.jasper@tu-dortmund.de

Falk Howar
falk.howar@tu-dortmund.de

Malte Mues
malte.mues@tu-dortmund.de

David Schmidt
david3.schmidt@tu-dortmund.de

Bernhard Steffen
bernhard.steffen@tu-dortmund.de

[1] TU Dortmund University, Dortmund, Germany

challenge contributors who are claiming satisfaction without sufficient evidence.

2 The RERS challenge

The *Rigorous Examination of Reactive Systems* challenge (RERS) is a verification challenge that focuses on temporal and reachability properties of reactive systems. RERS was founded in 2010 and has annual instances since 2012. 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 [52], model checking [9,14,25], symbolic execution [38], and (learning-based) testing [62].

The key idea of RERS is to use *generated*, *realistic* problems of *scalable* complexity on which participants have to check sets of properties.

Automatic generation of benchmarks with known properties provides new problems each year that are (a) previously unknown to participants, and (b) for which the correct verdicts for properties are not known to the participants during the challenge—preventing "performance tuning" of participating verification tools towards a high score on the basis of known expected results or known characteristics of benchmarks.

Realism of benchmarks (in contrast to typical randomly generated benchmarks) is achieved in a requirements-driven fashion: programs are generated according to characteristic temporal patterns resembling the structure of real code.

Scalability of difficulty is the basis for detailed performance profiling of participating tools.

In this section, we provide a brief motivation for RERS, an overview of the history of RERS, sketch some of the scientific contributions on the automatic generation of benchmarks that were facilitated through RERS, and briefly describe how different ranking methods in RERS enable detailed profiling of tools.

Remark Parts of this section have been published before in papers or on the RERS website. We provide pointers to more detailed accounts where it is appropriate and possible. The focus of this section is on providing a general overview.

2.1 Rigorous examination of reactive systems

The motivation of RERS is to enable profiling of principal capabilities and limitations of tools and approaches. The RERS challenge is therefore "free-style", i.e., without neither time nor resource limitations, and encourages 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. Our focus on principal capabilities instead of defined and identical resources is reflected by making RERS a challenge instead of a competition. We only provide the tasks and collect the results from participants. Solutions are computed by them in any way they want. The main goals of RERS are:

1. encourage the combination of methods from different (and usually disconnected) research fields for better software verification results,
2. provide a complete framework for an automated challenge organization that covers the process from generating differently tailored tasks that reveal the strengths and weaknesses of specific approaches to an automated result comparison (excluding the computation of results itself),
3. initiate 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. collect (additional) interesting syntactical features that should be supported by benchmark generators.

There exists no other software verification challenge with a profile that is similar to that of RERS: while (1) is a quite generic goal that is pursued by a number of verification competitions, goals (2)–(4) are unique to RERS. Nevertheless, RERS shares some intentions and characteristics with SV-COMP, MCC, and VerifyThis.

The software verification competition [8] (SV-COMP) is also concerned with reachability properties and features a few verification tasks concerning termination and memory safety. In direct comparison, SV-COMP does not allow the manual combination of tools and directly addresses tool developers. In contrast to RERS, it has time and resource limitations, does not feature certificate-like achievements (cf. Sect. 2.5), 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. Over time, the RERS benchmark generator contributed problems to the SV-COMP benchmark repository.

Another competition concerned with the verification of parallel systems in combination with LTL properties is the Model Checking Contest [43] (MCC). 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 consists of a large set of known models and a small set of unknown models that were collected among the participants.

In contrast to RERS, MCC participants submit tools that have to adhere to resource restrictions, rather than problem answers. Moreover, the correct answers to the used verification tasks are not always known, and a majority vote-based approach to correctness is used.[1] This may well penalize outstanding approaches that are e.g. unique in identifying the correct result. This problem is overcome for RERS due to its property-oriented benchmark generation. We were happy to hear that MCC started using some verification tasks of RERS to partially overcome this problem.

Finally, VerifyThis [29] 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 interactive or semi-interactive tools. Similar to RERS, VerifyThis encourages the use of a mixture of tools, however 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.

2.2 Genesis (from ZULU to RERS)

The idea for RERS arose in 2010 after participating in the ZULU automata learning competition [28]. The ZULU competition had some very exciting and some rather frustrating aspects. The competition was based on randomly generated automata, the participating learning tools competed in a black-box scenario, and questionnaires (sets of words for which participants had to decide language membership) were the basis for ranking tools.

One the one hand, ZULU had an incredibly engaging training and competition mode: contestants could generate new training problems in a push-button approach and ranking of tools on all benchmark instances was instantly visible. Improvements to algorithms did translate to almost instant gratification, fueling a month-long race for the win.

The mode of ranking performance by counting correct answers in questionnaires, on the other hand, did not serve well for differentiating tools and in some cases even favored learning algorithms that were already known to perform badly on real problems. Less precise models produced better predictions for certain distributions of words in questionnaires. Moreover, algorithms could (and were) tuned towards the structural properties of a randomly generated benchmark. It turned out that this tuning was often counterproductive for inferring models of real systems.

The RERS initiative aimed at developing an engaging challenge, or a set of challenges (cf. Sect. 2.3), in the area of formal methods that would overcome the perceived weaknesses of ZULU. As a consequence, RERS is based on generated benchmarks (cf. Sect. 2.4), and one of the long-term goals of RERS is making the generation of new benchmarks accessible to participants. At the same time, the approach to benchmark generation in RERS aims at generating benchmarks that have realistic properties—resulting in relevant performance profiles of tools. This aim is also supported by RERS providing multiple modes of ranking and rating, tailored to profile contributions according to their capabilities and limitations (cf. Sect. 2.5).

2.3 Tracks and history

After an initial workshop in 2010, RERS had yearly challenges since 2012 with a constantly evolving set of tracks and verification challenges. Since 2012, a total of 49 people from 16 different research groups participated in RERS.[2] Table 1 provides a comprehensive overview that will be detailed by the remainder of this section.

Sequential Programs RERS started in 2012 with sequential benchmark programs in two tracks (LTL and Reachability) that correspond to the property type that has to be analyzed. Sequential benchmark programs are made available as Java and C programs. Since 2014, there are three categories in each track that represent the syntactical features included in the benchmark programs that belong 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).

In each category, small, medium-sized, and large programs are generated for a challenge benchmark.

Starting in 2020, LTL properties will be controlled for minimal depth of counterexamples (presented in this paper), enabling an additional dimension in which complexity can be scaled.

Parallel Programs Since 2016, RERS features benchmarks that contain parallel systems which are made available as labeled transition systems (LTSs), Promela [24] code, and Petri nets [20,53]. The parallel track started with LTL properties and was tentatively extended to CTL properties in 2018.

[1] This approach is indeed quite common e.g., in the SAT-solving community.

[2] See http://www.rers-challenge.org/<challenge-year>/index.php?page=results for comprehensive lists of participants and results.

Table 1 RERS Challenges 2010-2020

Year	Colocation	Basic tracks				Special feature	Teams	Ref.
		Sequential		Parallel				
		Reach	LTL	LTL	CTL			
2010	ISoLA 2010					Initial workshop on founding RERS	n/a	[28]
2012	ISoLA 2012	✓	✓			–	5	[26,27]
2013	ASE 2013	✓	✓			Black-box/grey-box/white-box	3	[27]
2014	ISoLA 2014	✓	✓			*Extended lang. features (arithmetic, data structures)*	5	–[a]
2015	RV 2015	✓	✓			Monitoring challenge	2	[21]
2016	ISoLA 2016	✓	✓	✓		–	6	[22]
2017	SPIN/ISSTA 2017	✓	✓	✓		–	7	[31]
2018	ISoLA 2018	✓	✓	✓	✓	–	6	[33]
2019	TOOLympics at TACAS 2019	✓	✓	✓	✓	Industrial problems by ASML	4	[32]
2020	ISoLA 2020	✓	✓	✓	✓	*Scalable depth of LTL counterexamples*	Open	–

[a] Emphasized features brought or are expected to bring a permanent change to how benchmarks are generated for basic tracks

As a new addition in 2019, CTL properties were fully supported as a full track for the category of parallel programs (e.g. Petri nets) and were therefore on par with our support for LTL model checking tasks.

Experimental Tracks In several years, RERS had experimental tracks that did not (yet) result in permanent additions to the challenge.

- In 2013, RERS featured *grey-box* and *black-box* problems in addition to the (default) *white-box* problems. The additional problems were intended to encourage participation of black-box approaches and facilitated integration of white-box and black-box techniques.
- In 2015, RERS was co-located with the international conference on runtime verification (RV) [6] and featured monitoring problems for which traces were provided.
- In 2019, for the first time in the history of RERS, the challenge featured benchmark programs that are based on real-world models [32]. The corresponding challenge tracks were based on a cooperation with ASML, a large Dutch semiconductor company who provided the underlying models. Properties that participants could analyze for these systems ranged from reachability queries over LTL formulas to CTL properties (omitted in Table 1).

A detailed history and description of all past tracks and all sets of challenge problems can be found on the RERS website[3] along with properties and expected verdicts.

2.4 Synthesis of benchmarks with known properties

RERS relies on generated benchmark problems of scalable complexity and with known properties. The motivation for this, as stated above, is to enable detailed profiling of tools. The RERS benchmark generation technology combines scalable complexity with known properties, two goals that appear conflicting at first glance: it is impossible to automatically decide properties on problems that are too complex for current tools to analyze. Other competitions (e.g. MCC) solve this by determining verdicts that ought to be accepted as correct by majority vote. This, of course, has the drawback that a high performance of few tools, resulting in uncommon but correct verdicts on some problems, leads to a competition ranking that is inversely correlated to performance. We observed this firsthand in the ZULU competition.

Motivated by this experience, we have developed a generic method and tool-boxes for generating benchmark problems of scalable complexity with known properties. A frequent argument against the use of generated benchmarks is the potential threat to the validity of profiling results that arises from their artificial nature. We address this threat in RERS by using sets of LTL properties for inducing structure or actual industrial system models at the core of our benchmark synthesis.

In this section, we provide a brief overview of the generic method, using the generation of sequential benchmark problems as a concrete example. Detailed accounts of concrete tool-boxes for different classes of benchmark problems can be found in the papers listed in Table 2.

General Approach Our general approach to the generation of benchmarks that we use in RERS is sketched in Fig. 1 and exists in two variants, *property-based benchmark generation* and *model-based benchmark generation*. Both variants fol-

[3] http://rers-challenge.org.

Table 2 Generation of benchmarks in RERS

Year	Title	Ref
2013	Property-driven benchmark generation	[63]
2014	Tailored generation of concurrent benchmarks	[61]
2014	Property-driven benchmark generation: synthesizing programs of realistic structure	[64]
2017	Property-preserving generation of tailored benchmark petri nets	[67]
2018	Synthesizing subtle bugs with known witnesses	[35]

low the same high-level pattern. The process is divided into two phases. In the first phase (upper half of both sub-figures), benchmark properties are established on a small model. In the second phase, models are expanded by semantics-preserving transformations that increase complexity at the model-level and then generated into code. Code generation can add another dimension of complexity by encoding the behavior specified in the model through different language features (e.g., using arithmetic expressions or data structures).

Property-based Benchmark Generation In this variant (left of Fig. 1), we start the generation process by randomly choosing and then instantiating LTL property specification patterns [19] that we partition into a small defining set used in the subsequent synthesis step, and a larger set of additional properties whose validity is later checked on the synthesized model via model checking. Typically, we generate around 100 properties, about ten of which can be defining, in order to still allow for automated synthesis.

Our current implementation uses LTL2Buchi [23] and the Spot library [18] for translating the LTL specification into a Büchi automaton. The resulting intermediate Büchi automaton is then transformed into a concrete reactive system model (a Mealy machine) that represents all words/paths satisfying the defining properties. The construction of this Mealy machine is randomized and can be customized in various dimensions, e.g., the size of the model, the size of the input and output alphabets, the density of the transition graph etc., while guaranteeing that all defining properties remain valid.

Model-based Benchmark Generation In this variant, we start from a reactive system model. Such models were provided by ASML in 2019 [32]. Properties and verdicts can then be computed in two different ways from these models (right of Fig. 1). Generated properties can be model-checked as in the property-driven approach. This was, e.g., done for LTL properties in the industrial track of RERS 2019. Alternatively, properties can be computed from the models directly. This was done for CTL properties in the industrial track of RERS 2019.

Model-Expansion and Code Generation In the second phase of generating sequential benchmark problems, Mealy Machines are enlarged via randomized property-oriented expansion (POE) [60] and by introducing unreachable states. Both transformations are incremental and can be stopped at any moment, e.g., when a certain threshold of states is reached. The transformation from Mealy Machines to programs interprets Mealy machines as simple loops of guarded commands, whose guards precisely check for the correct state identification, and replaces the simple guard structure with a complex, semantically equivalent decision structure.

As a final step, we employ data-flow analysis, transformation and code motion techniques [12,39–42,49,58,68] to randomly elaborate the program model structure along both the logical and the control structure, delocalizing information and obtaining quite general `while`-program-like structures [2].

Generation of Parallel Benchmarks We have also applied our property-preserving generation process to obtain parallel systems in various formats like (Nested-Unit) Petri nets [20, 53], Promela [24] code, or simply as graphs in DOT[4]. This also happens in two conceptual steps: first, we synthesize an interesting core model from an LTL specification in the same way as for the sequential case, and then decompose this core model into parallel components in a property-preserving fashion. Key for this decomposition was a new notion of modal contracts [65] which allows us to generate parallel systems with arbitrarily many components.

Basing property preservation on modal refinement [46] instead of language inclusion guarantees that not only linear-time properties are preserved, but branching-time properties as well. This allows us to use the generated parallel models not only for the reachability and the LTL model checking track, but also for the CTL model checking and bisimulation checking track [66], the later being planned as a future addition.

2.5 Ranking

RERS has a three-dimensional reward structure that consists of a competition-based ranking on the total number of points, achievements for solving problems without submitting wrong answers, and an evaluation-based award for the most original idea or a good combination of methods. Computation of scores and modes of ranking (per track, per category) have evolved slightly over the years. Adaptations

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

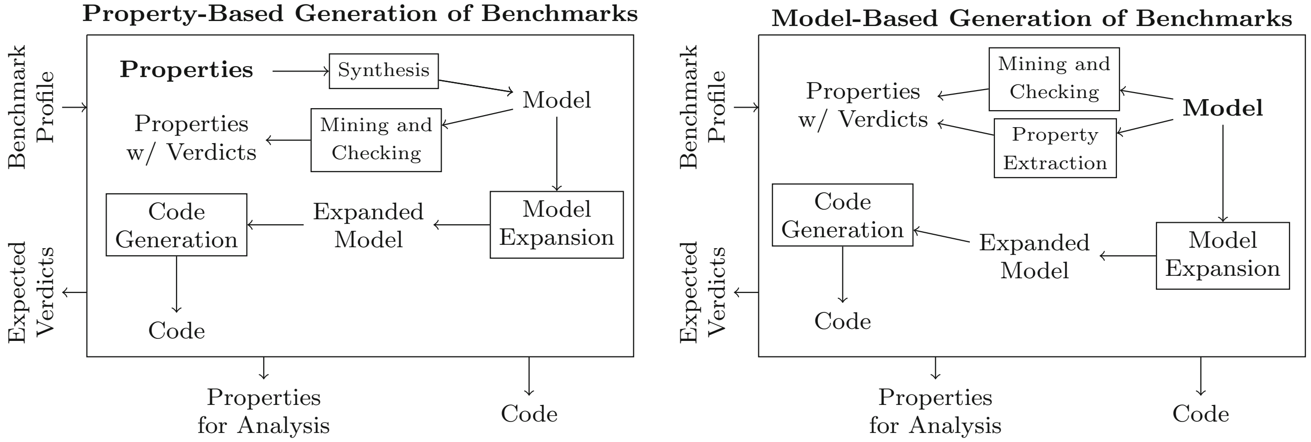

Fig. 1 Benchmark Synthesis in RERS. Property-driven generation (left) starts with a set of properties from which a model is synthesized. Model-driven generation (right) starts from a model. Mining and model checking or property extraction are used for generating challenge properties and expected verdicts. The model is expanded through semantics-preserving transformations. Code is generated from the model. The desired benchmark profile determines the extent of expansion and the language features used in the code

were made to arrive at more detailed, relevant, and valid profiling of participating approaches.

Competition-based Ranking The competition-based ranking was established to facilitate competition and as a direct comparative evaluation of the capabilities of tools. Participants are free to opt out of this ranking and to only aim at obtaining achievements. For the ranking, a score for the performance of every participating tool is computed. Based on these scores, tools are ranked. Positive points are awarded for correct verdicts. Incorrect verdicts lead to penalties whose heights was a major point of discussion over the years, leading to frequent changes.

The negative impact of incorrect verdicts on a tool's score in the competition-based ranking was originally quite small. In 2012 it was just -1 point, and it was -2 points in 2013 to 2015. In 2016 there was a drastic change in the penalty which became exponential in the number of errors (-2^n). This change has turned out to be too drastic and we are therefore using quadratic penalties ($-n^2$) since 2018.

For RERS 2019, also the points for correct answers were refined (from previously one point per correct answer) to two points for verifiable LTL properties or unreachable errors and one point for refutable LTL properties or reachable errors, accounting for the fact that showing the existence of counterexamples and errors is usually easier than proving their absence.

Achievements To honor the accomplishments of verification tools and methods without the pressure of loosing in a competition despite good results and only in relation to the complexity of the set of benchmark problems, RERS introduced achievements for different nuances of difficulty.

For every category there are three achievements: bronze, silver and gold. An achievement is only awarded if no wrong answers are given in the respective category. For tracks on CTL properties, a participant needs to answer 12 out of 20 properties correctly in order to “solve” an individual problem. If there are n problems within such a track, then a participant needs to answer $\frac{1}{3} \cdot n \cdot 12$ properties correctly for a bronze award, $\frac{2}{3} \cdot n \cdot 12$ for silver and $n \cdot 12$ for gold.

For the remaining tracks of RERS, proving the absence of a property violation is typically much harder than showing such a violation. Taking this into account, 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 benchmark instances, 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* tracks. Only the highest achievement for every category is actually awarded and the thresholds for every category are calculated as follows:

$$\begin{aligned}
\text{bronze} &= \text{\#falsifiable properties of small problem}\\
\text{silver} &= \text{\#bronze + falsifiable prop. of med. problem}\\
\text{gold} &= \text{\#silver + falsifiable prop. of large problem}
\end{aligned}$$

The participant's achievement score within a category is computed from all submitted results (verified or falsified). Let $a_t(C) = n$ be the achievement score of tool t for category C, where n is the number of correct (i.e., reported) verdicts for category C. Now let, e.g.,

$$bronze(C) \leq a_t(C) \text{ and } a_t(C) < silver(C).$$

Then participant t is awarded the Bronze Achievement in category C. It is possible to receive six achievements in the sequential tracks: one for each category (Plain, Arithmetic, Data Structures) in the Reachability and LTL track, respectively. In the parallel tracks, an overall of six achievements can be obtained by participating tools for small, medium, and large problems in the LTL and CTL tracks.

Since achievements are awarded on a per-participant basis, there may be multiple gold-medalists in some category in any particular year of RERS.

Evaluation-based Award To honor creativity and cross-fertilization between different research areas, RERS features jury-based awards. For these awards, category winners are chosen based on the employed (combination of) methods which must not necessarily have scored highest. Submitted descriptions of approaches and solutions are reviewed and ranked by the challenge organizers. Due to the possible variety of methods there may be several winners in this category.

2.6 Impact

In the ten years since its inception, RERS has had an impact in different dimensions.

Scientific Contributions. First of all, RERS has facilitated a number of scientific advances by challenge participants. Some examples are presented in [1,7,10,11,15–17,30,34,37,44,45,47,48,50,51,56,57,59,69–71].

Benchmark Generation. Organizing RERS required the generation of benchmarks. Over the past decade, we have developed multiple approaches for generating scalable and realistic benchmarks with known properties. Benchmark generation required integration of a diverse set of formal methods and RERS benchmarks have been integrated by other verification competitions (e.g., SV-COMP) into their sets of benchmark programs.

Combination of Methods. Over the years, RERS has facilitated a number of promising combinations of methods, e.g. [44]. In the latest instance, participants of RERS 2019 notably used diverse combinations of tools to produce their answers to the given verification tasks. As an example for this diversity, one of the participating teams combined verification based on grey-box fuzzing and traditional compiler-based interval analysis. Another team employed three different available verification tools to generate their submission and thereby profiled and utilized the individual strengths and weaknesses of these tools.

In summary, one can argue that instead of submitting an executable tool that computes a single verdict automatically as commonly required in verification competitions such as SV-COMP or MCC, participants of RERS make use of the freedom from resource constraints by employing an entire toolkit to solve the given verification tasks. RERS allows manual comparison of the output of tools and gives room for final judgment made by humans on the verdicts of a bouquet of verifiers and approaches, whereas other competitions enforce completely automated decisions by tools. This plethora of approaches provides evidence that RERS achieves one of its main goals, namely to motivate the comparison of different approaches and technologies (see Sect. 2.1).

3 Guaranteeing hardness of benchmarks

In this section, we sketch our most recent approach to tailor benchmark problems according to hardness: the generation of benchmark problems which are known to have no evidence for a counterexample that is shorter than a given threshold, but which are also guaranteed to have such evidence for an additionally provided upper bound. This allows the production of benchmarks with a designed distribution of depths to which the programs have to be investigated in order to find all errors. In particular, this gives benchmark designers a methodology for challenging contributors that are claiming satisfaction without having a proper proof.

3.1 Preliminaries

Fundamental to our approach are the notions related to words and languages:

Definition 1 (*Words*) Given a finite alphabet Σ, a *word* over Σ is a (possibly empty or infinite) sequence of symbols from Σ. Given an integer $n \in \mathbb{N}$ and a finite word $w = \sigma_1\sigma_2\ldots\sigma_n$, $|w|$ denotes the *length* n of w. Any infinite word w has the length $|w| = \infty$. Given any word $w = \sigma_1\sigma_2\ldots$ and any integer $i \in \mathbb{N}$ such that $i \leq |w|$, $w_{\leq i}$ denotes the *prefix* of w of length i.

Definition 2 (*Languages*) Given a finite alphabet Σ, a language (over Σ) is a set of words over Σ. For a given $n \in \mathbb{N}$, the language Σ^n consists of all words $w = \sigma_1\sigma_2,\ldots\sigma_n$ of length $|w| = n$ such that $\sigma_i \in \Sigma$ for all $i \in 1 \mathinner{..} n$.

For any $n \in \mathbb{N}$, we define $\Sigma^{\leq n} := \bigcup_{i=1}^{n} \Sigma^i$, and additionally $\Sigma^* := \bigcup_{i\in\mathbb{N}} \Sigma^i$. A language L is finite iff $|L| \in \mathbb{N}$ and infinite otherwise. Σ^ω denotes the infinite language that contains all infinite words over Σ. Moreover, L is a language of finite words iff $L \subseteq \Sigma^*$, and a language of infinite words iff $L \subseteq \Sigma^\omega$. The concatenation of symbols extends naturally to languages: Given a language $L \subseteq \Sigma^*$ and any language L', we have

$$LL' := \{ww' \mid w \in L \land w' \in L'\}$$

Our approach to benchmark generation (cf. Sect. 3.3) is based on the automatic generation of Büchi automata [13].

Definition 3 (*Büchi Automaton*) Let $B = (S, \Sigma, \Delta, s_0, F)$ be a finite automaton with a set S of states and an alphabet Σ. State $s_0 \in S$ represents the initial state and $F \subseteq S$ a set of accepting states. The relation $\Delta \subseteq (S \times \Sigma \times S)$ represents transitions between states in S. We also write $p \overset{\sigma}{\to} q$ to denote $(p, \sigma, q) \in \Delta$.

A *path* p in B is a sequence of transitions $u_i \overset{\sigma_i}{\to} u_{i+1}$ with i ranging from 1 to either a fixed integer n or infinity. Path p *spells* the word $w = \sigma_1\sigma_2 \ldots$.

Given these definitions, B is called a *Büchi automaton* if it adheres to Büchi acceptance, meaning that it accepts infinite words $w \in \Sigma^\omega$ based on the following criteria:

1. There exists a path p in B that starts in s_0 and spells w
2. This path p visits a state in F infinitely often

The set $\mathcal{L}(B) := \{w \in \Sigma^\omega \mid B \text{ accepts } w\}$ defines the language of B.

The following definitions specify (propositional) linear temporal logic (LTL) [54] which we use to specify properties and as a basis for synthesizing Büchi automata. In essence, LTL is an extension of propositional logic that includes additional temporal operators. Its syntax is defined as follows [3]:

Definition 4 (*Syntax of Linear Temporal Logic*) Let AP be a set of atomic propositions and $a \in$ AP. The syntax of propositional linear temporal logic (LTL) is defined by the following grammar in Backus-Naur form:

$$\varphi :: = \top \mid a \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbf{X}\,\varphi \mid (\varphi\ \mathbf{U}\ \varphi)$$

The operator $\mathbf{X}$ (or “next”) describes behavior that has to hold at the next time step. A formula $(\varphi_1\ \mathbf{U}\ \varphi_2)$ describes that φ_2 has to occur eventually and that φ_1 has to hold until φ_2 occurs in a sequence. The formal semantics of LTL is based on a satisfaction relation between infinite words and LTL formulas [3]:

Definition 5 (*Semantics of LTL*) Let AP be an alphabet of atomic propositions and let $(2^{\mathrm{AP}})^\omega$ denote infinite sequences over sets $A \subseteq$ AP. For any sequence $w = (A_1, A_2, \ldots) \in (2^{AP})^\omega$ and any $i \in \mathbb{N}$, let $w_i = A_i$ be the i-th element of w and $w_{\geq i} = (A_i, A_{i+1}, \ldots)$ be the suffix of w starting at index i.

Then the satisfaction relation $\models \subseteq ((2^{\mathrm{AP}})^\omega \times \text{LTL})$ is defined as the relation that adheres to the following rules:

$$\begin{array}{lll}
w \models \top & & \\
w \models a & \text{iff} & a \in w_1 \\
w \models (\varphi \wedge \psi) & \text{iff} & w \models \varphi \text{ and } w \models \psi \\
w \models \neg\varphi & \text{iff} & w \not\models \varphi \\
w \models \mathbf{X}\,\varphi & \text{iff} & w_{\geq 2} \models \varphi \\
w \models (\varphi\ \mathbf{U}\ \psi) & \text{iff} & \exists k \in \mathbb{N} : w_{\geq k} \models \psi \text{ and} \\
& & \forall i \in \mathbb{N}_{<k} : w_{\geq i} \models \varphi
\end{array}$$

where $w \in (2^{\mathrm{AP}})^\omega$ and $\varphi, \psi \in$ LTL.

Given a language $L \subseteq \Sigma^\omega$, we define

$$L \models \varphi \quad \text{iff} \quad \forall w \in L.\ w \models \varphi,$$

and given a Büchi automaton B, we further define

$$B \models \varphi \quad \text{iff} \quad \mathcal{L}(B) \models \varphi.$$

For any $\varphi \in$ LTL, the semantics $[\![\varphi]\!]$ of φ is given by

$$[\![\varphi]\!] := \{w \in (2^{\mathrm{AP}})^\omega \mid w \models \varphi\}.$$

Büchi automata are strictly more expressive than LTL [72]. One can synthesize a Büchi automaton B from an LTL property φ such that $\mathcal{L}(B) = [\![\varphi]\!]$ holds [55].

Using the basic set of operators in Definition 4, abbreviations for commonly described constraints can be introduced. Popular ones include $\mathbf{F}(\varphi) := (\top\ \mathbf{U}\ \varphi)$ which expresses that φ will eventually become true and its dual operator $\mathbf{G}(\varphi) := \neg\mathbf{F}(\neg\varphi)$ which claims that φ is always true. A later example also utilizes the weak-until operator $(\varphi\ \mathbf{W}\ \psi) := (\varphi\ \mathbf{U}\ \psi) \vee \mathbf{G}(\varphi)$.

In the following, we introduce our approach to specify languages such that a given verification property $\varphi \in$ LTL is violated, however in a way such that all counterexamples that witness this violation have a guaranteed minimal length.

3.2 Guaranteeing deep LTL counterexamples

In this section we show how to construct $(m, n]$-*hard* verification tasks. Here, hardness is based on an integer interval $(m, n]$ of prefix lengths that means the following: looking at prefixes of words $w \in L$ of length at most m does *not* suffice to explain the property violation, however there exists such a violating prefix of length at most n. In other words, every prefix of length smaller or equal to m can be extended to a word that satisfies φ, but this is not the case for all prefixes of length up to n. We aim for verification tasks (L, φ) such that

1. $L \subseteq \Sigma^\omega$ and
2. φ is an LTL formula satisfying that
3. (L, φ) is $(m, n]$-hard.

In the following, we only synthesize reactive programs and LTL properties for reasoning about non-terminating paths. Our construction then works by constructing a maximal sublanguage $L' \subseteq L$ that is $(m, n]$-hard w.r.t. φ (see. Sec. 3.3 for our realization based on Büchi automata). In general, L' may well be empty, a phenomenon that we deal with in a heuristic fashion.

The following notion of *violating prefix* is important:

Definition 6 (*Violating Prefix*) Let $w \in \Sigma^*$. Then w violates φ iff the following holds:

$$\forall w' \in \Sigma^\omega .\ ww' \not\models \varphi$$

An infinite word $w \in \Sigma^\omega$ k-violates φ iff its prefix $w_{\leq k}$ violates φ. A language $L' \subseteq \Sigma^\omega$ k-violates φ iff there exists a word $w \in L'$ such that w k-violates φ.

Intuitively speaking, a finite word violates φ if it cannot be extended to a word that satisfies φ. The following lemma follows straightforwardly:

Lemma 1 (Monotonicity) *If a word $w \in \Sigma^\omega$ k-violates φ, then for all $k' \in \mathbb{N}$ with $k' \geq k$, w also k'-violates φ.*

This monotonicity property allows us to specify $(m, n]$-hardness simply based on the boundaries of this integer interval.

Definition 7 (*Hardness*) A language $L' \subseteq \Sigma^\omega$ is called $(m, n]$-hard w.r.t. φ iff the following hold:

1. L' does *not* m-violate φ
2. L' n-violates φ

Based on this hardness definition, we can deduce a constructive approach to generate the maximal sub-language of L that is $(m, n]$-hard w.r.t. φ. We simply construct the maximal sub-language L^m_φ of L that does *not* m-violate φ and then check whether or not L^m_φ n-violates φ. If it does, (L^m_φ, φ) is an $(m, n]$-hard verification task. Otherwise, we know that no $(m, n]$-hard verification task exists for L and φ, and we continue by heuristically modifying the parameters.

The remainder of this section is therefore dedicated to the construction of L^m_φ and the subsequent check whether it n-violates φ.

Definition 8 (*Violating Prefixes*) Let $L' \subseteq \Sigma^\omega$ and $k \in \mathbb{N}$. We denote the set of prefixes of L' with length at most k by

$$L'_{\leq k} := \{w_{\leq i} \mid w \in L' \ \wedge \ i \leq k\}.$$

Given a $\varphi \in$ LTL, we call

$$\mathrm{VP}(L, \varphi, k) := L_{\leq k} \backslash [\![\varphi]\!]_{\leq k}$$

the *violating prefixes* of φ in L with length at most k.

The following lemma is straightforward to prove:

Lemma 2 *Let $k \in \mathbb{N}$. Then $\mathrm{VP}(L, \varphi, k)$ consists of all words $w \in L_{\leq k}$ that violate φ.*

The following theorems follow straightforwardly from Lemmas 1 and 2:

Theorem 1 $L^m_\varphi = L \backslash (\mathrm{VP}(L, \varphi, m)\Sigma^\omega)$

and

Theorem 2

$$L \subseteq \Sigma^\omega \ n\text{-}violates\ \varphi \quad iff \quad \mathrm{VP}(L, \varphi, n) \neq \emptyset$$

Complementation of Büchi automata is a very expensive operation. The following theorem guarantees that this operation can be avoided and instead replaced by one that executes in quadratic time:

Theorem 3

$$L^m_\varphi = L \cap ([\![\varphi]\!]_{\leq m} \Sigma^\omega)$$

Proof We show the two inclusions between L^m_φ and $L' := L \cap ([\![\varphi]\!]_{\leq m} \Sigma^\omega)$.

Every word $w \in L'$ lies in $[\![\varphi]\!]_{\leq m} \Sigma^\omega$ which excludes that it m-violates φ. Thus we have as desired $L \cap ([\![\varphi]\!]_{\leq m} \Sigma^\omega) \subseteq L^m_\varphi$.

For the converse inclusion let $w \in L^m_\varphi$. According to Def. 6, this means that there exists a word $w' \in \Sigma^\omega$ such that $w_{\leq m} w'$ satisfies φ which yields $w_{\leq m} \in [\![\varphi]\!]_{\leq m}$ and therefore in particular $w \in [\![\varphi]\!]_{\leq m} \Sigma^\omega$. On the other hand, $L^m_\varphi \subseteq L$. Together this guarantees that $w \in L'$. □

The next section presents the Büchi automaton-based realization of L^m_φ in the way that it is used for our RERS benchmarks.

3.3 Realization based on Büchi automata

RERS' benchmark generation follows the idea of requirement-driven system generation. More precisely, starting point for RERS benchmarks is a set of *structural LTL properties Φ* which are meant to impose realistic benchmarks structures. Thus, the initial languages L we consider in the rest of this paper are of the form $L = [\![\Phi]\!]$, and the goal is to construct $L^m_\varphi = [\![\Phi]\!]^m_\varphi$. According to Theorem 3 this means that we have to compute

$$L' = [\![\Phi]\!] \cap ([\![\varphi]\!]_{\leq m} \Sigma^\omega).$$

This can be done by means of well-known technology for Büchi automata as follows:

1. Compute $L = [\![\Phi]\!]$ and $[\![\varphi]\!]$. We use the Spot library [18] for this purpose. Please note that we need to constrain the construction of $L = [\![\Phi]\!]$ such that all transitions within the resulting Büchi automaton are labeled with a single atomic proposition. This can be accomplished by enforcing an according invariant Ω in LTL (cf. [35]).
2. Concatenate the prefix tree of depth m for $[\![\varphi]\!]$ with Σ^ω to obtain a Büchi automaton for $[\![\varphi]\!]_{\leq m} \Sigma^\omega$. Essentially, this means to add an accepting Σ^ω-loop at the end of each leaf of this prefix tree.

3. Compute the intersection of the two Büchi automata constructed in steps 1 and 2. This is again accomplished using the Spot library.
4. Heuristically minimize the Büchi automaton that results from step 3, again based on the Spot library. This is important for the scalability of later transformation steps in our overall approach, and it helps to obfuscate the tree expansion in step 2.

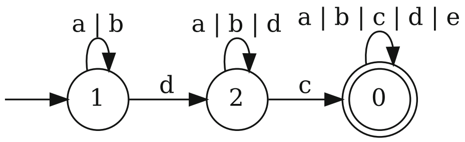

Fig. 2 Büchi automaton B for $[\![\Phi]\!]$

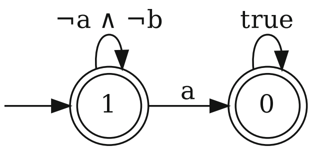

Fig. 3 Büchi automaton for $[\![\varphi]\!]$

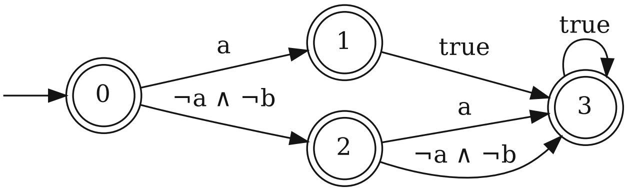

Fig. 4 Büchi automaton B' for $[\![\varphi]\!]_{\leq m}\Sigma^{\omega}$ for $m = 2$

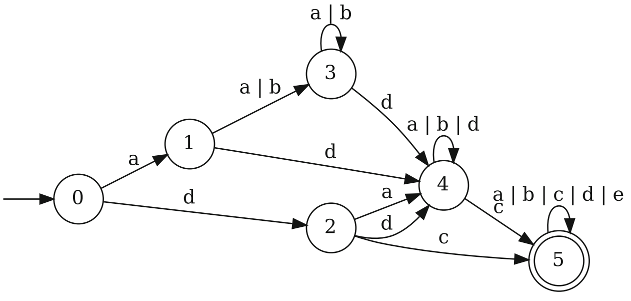

Fig. 5 Büchi automaton $B_{res} = B \cap B'$

In order to be sure that (L', φ) is indeed an $(m, n]$-hard verification task, it remains to be shown that L' n-violates φ (cf. Def. 7). This can be done simply by means of an emptiness check for

$$L' \backslash ([\![\varphi]\!]_{\leq n}\Sigma^{\omega})$$

If it fails, we are guaranteed to have a violating prefix that is longer than m but shorter than or equal to n. Otherwise, we know that no $(m, n]$-hard verification task exists for $[\![\Phi]\!]$ and φ, and we continue by heuristically modifying the parameters.

3.3.1 Example

The following example illustrates each step of realizing

$$L' = [\![\Phi]\!] \cap ([\![\varphi]\!]_{\leq m}\Sigma^{\omega})$$

for $m = 2$,

$$\Phi = \{\neg(c \vee e)\mathbf{W}d,\ \neg e\mathbf{U}c,\ \Omega\}$$

and

$$\varphi = \neg b\mathbf{W}a$$

where Ω is the above-mentioned invariant that ensures that every Büchi automaton transition is labeled with exactly one atomic proposition (cf. [35]). In order to ease readability when this invariant is enforced, we abbreviate n transitions labeled $b_1, \ldots, b_n$ that share their sources and targets by a single transition labeled "$b_1|\ldots|b_n$".

Figures 2 and 3 display the Büchi automata for $[\![\Phi]\!]$ and $[\![\varphi]\!]$, respectively, whereas Fig. 4 shows the Büchi automaton for the language that guarantees that there are no violating prefixes of length smaller or equal to m (cf. step 2 above). As $[\![\varphi]\!]$ and $[\![\varphi]\!]_{\leq m}\Sigma^{\omega}$ do *not* feature a singleton invariant, they are an exception to our simplified representation. Spot [18], the library that we use for Büchi synthesis, uses a BDD representation for Büchi automaton transitions. Therefore, all labels within Figs. 3 and 4 have to be interpreted as BDDs and *not* in our simplified manner presented above. Note that because B' (Fig. 4) is afterwards intersected with B (Fig. 2), the self loop at state 3 of the former does not need to specify the exact alphabet Σ. The Büchi automaton B_{res} shown in Fig. 5 specifies already the desired language (cf. step 3 above), but it needs to be minimized to obfuscate the tree expansion step and to achieve scalability of subsequent transformations (cf. Fig. 6).

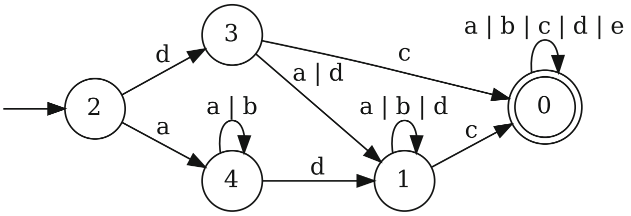

Fig. 6 B_{res} after minimization

3.3.2 Experiments

To provide an impression of the scalability of our approach, we analyzed its execution time and the occurring numbers of states with Φ and φ given as in the previous section, but with increasing lower hardness bounds. This means in particular that B (cf. Fig. 2) and B_φ (cf. Fig. 3) are maintained during our experiments.

The first column of Table 3 shows the hardness bound m which was set to 2 during the discussion of the previous section, while columns two and three summarize the number of states of the resulting automata before minimization (corresponding to Fig. 5) and after heuristic minimization (corresponding to Fig. 6). The fourth column provides the wall-clock execution time for computing the final heuristically minimized Büchi automata as well as the proportion of execution time which is needed for that minimization.

As one can see, the numbers are strictly increasing. This seems to indicate that the corresponding languages are continuously changing, or more precisely, continuously strictly decreasing. This is, however, not guaranteed, because the four-step construction via Spot may well provide two different Büchi automata for the same language. There is no canonicity. Thus, to be sure that one has a a valid $(m, n]$-hard verification task one still has to check whether the languages for n and m are indeed different. In the considered cases, this could always be verified.

Our C++ implementation utilizes the Spot library [18] for synthesizing, modifying, and optimizing Büchi automata. The execution times in Table 3 are based on our implementation that was executed on a machine running Arch Linux 5.5.13-arch2-1 and featuring an AMD Ryzen 3950X processor with 32GiB of RAM.

Table 3 State numbers and execution time (rounded up to two significant digits) of B_{res} and minimized B_{res} for the above example and different values of m

m	$B_{res} =$ $B \cap B'_\varphi$	Minimized B_{res}	Exec. time in ms	
			Minimization	Total
2	6	5	0.04	1.20
3	11	7	0.06	1.30
4	20	9	0.11	1.30
5	31	11	0.16	1.40
6	44	13	0.22	1.50
7	59	15	0.31	1.60
8	76	17	0.40	1.80
9	95	19	0.52	1.90
10	116	21	0.65	2.10
20	436	41	4.60	6.70
30	956	61	18.00	22.00
40	1676	81	46.00	50.00
50	2596	101	99.00	110.00
60	3716	121	190.00	200.00
70	5036	141	330.00	340.00
80	6556	161	540.00	550.00
90	8276	181	830.00	850.00
100	10,196	201	1200.00	1300.00

4 Conclusion and perspective

We have summarized the history of the RERS challenge for the analysis and verification of reactive systems and its objectives in two parts. In the first part, its profile and intentions, its relation to other competitions, and, in particular, its evolution due to the feedback of participants were discussed. This comprised, in particular, the discussion of 'oddities' like over-tuning: some participants tweak their tools to the sometimes concretely known solutions of the competitions' benchmarks, which leads to scores that have little to do with the tools' performance in realistic scenarios. This way, even winning a competition does not necessarily need to be a recommendation for potential users.

The second part presents our latest development with regard to the over-tuning problem: the automatic synthesis of benchmark problems with tailored difficulty in a requirement-driven' fashion. More precisely, since the beginning, the starting point of the RERS benchmark generation are desired structural properties, here formulated in LTL, which are successively transformed via Büchi automata that characterize all satisfying executions to Modal Transitions Systems and, with a few more steps, to code of various implementation languages (cf. Sec. 2). This way, RERS aims at benchmarks that closely resemble realistic code, but can be flexibly tailored in their degree of difficulty.

The contribution of the second part is a way to tailor benchmarks according to the depths to which programs have to be investigated in order to find all errors. This approach gives benchmark designers a method to challenge contributors that try to perform well by excessive guessing, e.g., based on 'inappropriate' side knowledge. Combined with our traditional way of benchmark tailoring concerning the code/model size, the amount of arithmetic, and the used data structures as measure for intricacy, RERS provides benchmark designers with a very powerful engine that we plan to make available open source.

It should be noted that the ideas presented in this paper are not only applicable to the generation of benchmarks that feature sequential programs. Rather, they can also be applied during the generation of parallel benchmark problems. This means that we can provide not only SV-COMP and similar competitions with tailored benchmark problems, but also competitions like MCC.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain types: abstract-domain selection based on variable usage. In: Bertacco, V., Legay, A. (eds.) Hardware and Software: Verification and Testing, pp. 262–278. Springer, Cham (2013)
2. Apt, K.R., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Texts and Monographs in Computer Science. Springer (1991). https://doi.org/10.1007/978-1-4757-4376-0
3. Baier, C., Katoen, J.P., Larsen, K.G.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Barrett, C., de Moura, L., Stump, A.: Smt-comp: satisfiability modulo theories competition. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, pp. 20–23. Springer, Berlin (2005)
5. Bartocci, E., Beyer, D., Black, P.E., Fedyukovich, G., Garavel, H., Hartmanns, A., Huisman, M., Kordon, F., Nagele, J., Sighireanu, M., Steffen, B., Suda, M., Sutcliffe, G., Weber, T., Yamada, A.: Toolympics 2019: an overview of competitions in formal methods. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 3–24. Springer, Cham (2019)
6. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., et al.: First International Competition on Runtime Verification: Rules, Benchmarks, Tools, and Final Results of CRV 2014. STTT pp. 1–40 (2017)
7. Bauer, O., Geske, M., Isberner, M.: Analyzing program behavior through active automata learning. Int. J. Softw. Tools Technol. Transf. **16**(5), 531–542 (2014)
8. Beyer, D.: Competition on software verification. TACAS. LNCS, vol. 7214, pp. 504–524. Springer, Berlin (2012)
9. Beyer, D.: Status Report on Software Verification. In: Proceedings of the TACAS, LNCS 8413, pp. 373–388. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_25
10. Beyer, D., Stahlbauer, A.: Bdd-based software model checking with cpachecker. In: Kučera, A., Henzinger, T.A., Nešetřil, J., Vojnar, T., Antoš, D. (eds.) Mathematical and Engineering Methods in Computer Science, pp. 1–11. Springer, Berlin (2013)
11. Beyer, D., Stahlbauer, A.: BDD-based software verification. Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transf. **16**(5), 507–518 (2014)
12. Briggs, P., Cooper, K.D.: Effective partial redundancy elimination. In: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), pp. 159–170 (1994). https://doi.org/10.1145/773473.178257
13. Büchi, J.R.: Symposium on decision problems: On a decision method in restricted second order arithmetic. In: Logic, Methodology and Philosophy of Science, Studies in Logic and the Foundations of Mathematics, vol. 44, pp. 1 – 11. Elsevier (1966). https://doi.org/10.1016/S0049-237X(09)70564-6
14. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
15. Decker, N., Pirogov, A.: Flat model checking for counting ltl using quantifier-free presburger arithmetic. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation, pp. 513–534. Springer, Cham (2019)
16. Dietsch, D., Heizmann, M., Langenfeld, V., Podelski, A.: Fairness modulo theory: a new approach to ltl software model checking. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 49–66. Springer, Cham (2015)
17. Duan, Z., Tian, C., Duan, Z.: Verifying temporal properties of c programs via lazy abstraction. In: Duan, Z., Ong, L. (eds.) Formal Methods and Software Engineering, pp. 122–139. Springer, Cham (2017)
18. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16), Lecture Notes in Computer Science, vol. 9938, pp. 122–129. Springer (2016). https://doi.org/10.1007/978-3-319-46520-3_8
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 411–420 (1999). https://doi.org/10.1145/302405.302672
20. Garavel, H.: Nested-unit petri nets. J. Log. Algebraic Methods Program. **104**, 60–85 (2019). https://doi.org/10.1016/j.jlamp.2018.11.005
21. Geske, M., Isberner, M., Steffen, B.: Rigorous examination of reactive systems. In: Bartocci, E., Majumdar, R. (eds.) Runtime Verification, pp. 423–429. Springer, Cham (2015)
22. Geske, M., Jasper, M., Steffen, B., Howar, F., Schordan, M., van de Pol, J.: RERS 2016: parallel and sequential benchmarks with focus on LTL verification. In: ISoLA. LNCS, vol 9953, pp. 787–803. Springer (2016)
23. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of ltl formulae to büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) Formal Techniques for Networked and Distributed Sytems—FORTE 2002, pp. 308–326. Springer, Berlin (2002)
24. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2011)
25. Holzmann, G.J., Smith, M.H.: Software model checking: extracting verification models from source code. Softw. Test. Verif. Reliab. (2001). https://doi.org/10.1002/stvr.228
26. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The rers grey-box challenge 2012: analysis of event-condition-action systems. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 608–614. Springer, Berlin (2012)
27. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. STTT **16**(5), 457–464 (2014)
28. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification, and Validation, pp. 687–704. Springer, Berlin (2010)

29. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012. STTT **17**(6), 647–657 (2015)
30. Jasper, M.: Counterexample-guided prefix refinement analysis for program verification. In: Lamprecht, A.L. (ed.) Leveraging Applications of Formal Methods, Verification, and Validation, pp. 143–155. Springer, Cham (2016)
31. Jasper, M., Fecke, M., Steffen, B., Schordan, M., Meijer, J., Pol, J.v.d., Howar, F., Siegel, S.F.: The RERS 2017 challenge and workshop (invited paper). In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, SPIN 2017, pp. 11–20. ACM (2017). https://doi.org/10.1145/3092282.3098206
32. Jasper, M., Mues, M., Murtovi, A., Schlüter, M., Howar, F., Steffen, B., Schordan, M., Hendriks, D., Schiffelers, R., Kuppens, H., Vaandrager, F.W.: Rers 2019: combining synthesis with real-world models. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 101–115. Springer, Cham (2019)
33. Jasper, M., Mues, M., Schlüter, M., Steffen, B., Howar, F.: Rers 2018: Ctl, ltl, and reachability. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification, pp. 433–447. Springer, Cham (2018)
34. Jasper, M., Schordan, M.: Multi-core model checking of large-scale reactive systems using different state representations. In: ISoLA. LNCS, vol 9952, pp. 212–226. Springer (2016)
35. Jasper, M., Steffen, B.: Synthesizing subtle bugs with known witnesses. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification, pp. 235–257. Springer, Cham (2018)
36. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. AI Mag. **33**(1), 89–92 (2012). https://doi.org/10.1609/aimag.v33i1.2395
37. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: Ltsmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 692–707. Springer, Berlin (2015)
38. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
39. Knoop, J., Rüthing, O., Steffen, B.: Lazy code motion. In: Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), pp. 224–234. ACM (1992). https://doi.org/10.1145/143095.143136
40. Knoop, J., Rüthing, O., Steffen, B.: Lazy strength reduction. J. Program. Lang. **1**, 71–91 (1993)
41. Knoop, J., Rüthing, O., Steffen, B.: Partial dead code elimination. In: Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), pp. 147–158. ACM (1994). https://doi.org/10.1145/178243.178256
42. Knoop, J., Rüthing, O., Steffen, B.: Expansion-based removal of semantic partial redundancies. In: Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22–28 March, 1999, Proceedings, LNCS, vol. 1575, pp. 91–106. Springer (1999). https://doi.org/10.1007/b72146
43. Kordon, F., Linard, A., Buchs, D., Colange, M., Evangelista, S., Lampka, K., Lohmann, N., Paviot-Adet, E., Thierry-Mieg, Y., Wimmel, H.: Report on the model checking contest at petri nets 2011. In: Transactions on Petri Nets and Other Models of Concurrency VI. LNCS, vol 7400, pp. 169–196. Springer (2012)
44. Lang, F., Mateescu, R., Mazzanti, F.: Compositional verification of concurrent systems by combining bisimulations. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods—The Next 30 Years, pp. 196–213. Springer, Cham (2019)
45. Lang, F., Mateescu, R., Mazzanti, F.: Sharp congruences adequate with temporal logics combining weak and strong modalities. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, pp. 57–76. Springer, Cham (2020)
46. Larsen, K.G.: Modal specifications. In: CAV. LNCS, vol 407, pp. 232–246. Springer (1989)
47. Meijer, J.: Efficient Learning and Analysis of System Behavior. Ph.D. thesis, University of Twente, Netherlands (2019). https://doi.org/10.3990/1.9789036548441
48. Meijer, J., van de Pol, J.: Sound black-box checking in the learnlib. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NASA Formal Methods, pp. 349–366. Springer, Cham (2018)
49. Morel, E., Renvoise, C.: Global optimization by suppression of partial redundancies. Commun. ACM **22**(2), 96–103 (1979). https://doi.org/10.1145/359060.359069
50. Morse, J.: Expressive and Efficient Bounded Model Checking of Concurrent Software. Ph.D. thesis, University of Southampton (2015). http://eprints.soton.ac.uk/id/eprint/379284
51. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Applying symbolic bounded model checking to the 2012 RERS Greybox challenge. Int. J. Softw. Tools Technol. Transf. **16**(5), 519–529 (2014)
52. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (1999)
53. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Hoboken (1981)
54. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), pp. 46–57 (1977). https://doi.org/10.1109/SFCS.1977.32
55. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '89, pp. 179–190. ACM (1989). https://doi.org/10.1145/75277.75293
56. van de Pol, J., Meijer, J.: Synchronous or Alternating?, pp. 417–430. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22348-9_24
57. van de Pol, J., Ruys, T.C., te Brinke, S.: Thoughtful Brute-force attack of the RERS 2012 and 2013 challenges. Int. J. Softw. Tools Technol. Transf. **16**(5), 481–491 (2014). https://doi.org/10.1007/s10009-014-0324-3
58. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM (1988). https://doi.org/10.1145/73560.73562
59. Schordan, M., Prantl, A.: Combining static analysis and state transition graphs for verification of event-condition-action systems in the RERS 2012 and 2013 challenges. Int. J. Softw. Tools Technol. Transf. **16**(5), 493–505 (2014). https://doi.org/10.1007/s10009-014-0338-x
60. Steffen, B.: Property-oriented expansion. In: Cousot, R., Schmidt, D.A. (eds.) Static Analysis, pp. 22–41. Springer, Berlin (1996)
61. Steffen, B., Howar, F., Isberner, M., Naujokat, S., Margaria, T.: Tailored generation of concurrent benchmarks. STTT **16**(5), 543–558 (2014)
62. Steffen, B., Howar, F., Merten, M.: Introduction to Active Automata Learning from a Practical Perspective, pp. 256–296. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-21455-4_8
63. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation. In: Bartocci, E., Ramakrishnan, C.R. (eds.) Model Checking Software, pp. 341–357. Springer, Berlin (2013)
64. Steffen, B., Isberner, M., Naujokat, S., Margaria, T., Geske, M.: Property-driven benchmark generation: synthesizing programs of realistic structure. STTT **16**(5), 465–479 (2014)
65. Steffen, B., Jasper, M.: Property-preserving parallel decomposition. In: Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 125–145. Springer (2017)

66. Steffen, B., Jasper, M.: Generating Hard Benchmark Problems for Weak Bisimulation, pp. 126–145. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31514-6_8
67. Steffen, B., Jasper, M., Meijer, J., van de Pol, J.: Property-preserving generation of tailored benchmark petri nets. In: 17th International Conference on Application of Concurrency to System Design (ACSD), pp. 1–8 (2017). https://doi.org/10.1109/ACSD.2017.24
68. Steffen, B., Knoop, J.: Finite Constants: Characterizations of a New Decidable Set of Constants. In: Kreczmar, A., Mirkowska, G. (eds.) Mathematical Foundations of Computer Science (MFCS'89), LNCS, vol. 379, pp. 481–491. Springer (1989). https://doi.org/10.1007/3-540-51486-4_94
69. Wang, M., Tian, C., Duan, Z.: Full regular temporal property verification as dynamic program execution. In: IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pp. 226–228 (2017). https://doi.org/10.1109/ICSE-C.2017.98
70. Wang, M., Tian, C., Zhang, N., Duan, Z.: Verifying full regular temporal properties of programs via dynamic program execution. IEEE Trans. Reliab. **68**(3), 1101–1116 (2019). https://doi.org/10.1109/TR.2018.2876333
71. Wang, M., Tian, C., Zhang, N., Duan, Z., Yao, C.: Translating Xd-C programs to MSVL programs. Theor. Comput. Sci. **809**, 430–465 (2020). https://doi.org/10.1016/j.tcs.2019.12.038
72. Wolper, P.: Temporal logic can be more expressive. Inf. Control **56**(1), 72–99 (1983). https://doi.org/10.1016/S0019-9958(83)80051-5