RERS 2019: Combining Synthesis with Real-World Models

Marc Jasper[1], Malte Mues[1], Alnis Murtovi[1], Maximilian Schlüter[1], Falk Howar[1], Bernhard Steffen[1], Markus Schordan[2(✉)], Dennis Hendriks[3], Ramon Schiffelers[4], Harco Kuppens[5], and Frits W. Vaandrager[5]

[1] TU Dortmund University, Dortmund, Germany
{marc.jasper,malte.mues,alnis.murtovi,maximilian.schlueter,falk.howar,bernhard.steffen}@tu-dortmund.de
[2] Lawrence Livermore National Laboratory, Livermore, CA, USA
schordan1@llnl.gov
[3] ESI (TNO), Eindhoven, The Netherlands
dennis.hendriks@tno.nl
[4] ASML and Eindhoven University of Technology, Veldhoven/Eindhoven, The Netherlands
ramon.schiffelers@asml.com
[5] Radboud University, Nijmegen, The Netherlands
{H.Kuppens,F.Vaandrager}@cs.ru.nl

Abstract. This paper covers the Rigorous Examination of Reactive Systems (RERS) Challenge 2019. For the first time in the history of RERS, the challenge features industrial tracks where benchmark programs that participants need to analyze are synthesized from real-world models. These new tracks comprise LTL, CTL, and Reachability properties. In addition, we have further improved our benchmark generation infrastructure for parallel programs towards a full automation. RERS 2019 is part of TOOLympics, an event that hosts several popular challenges and competitions. In this paper, we highlight the newly added industrial tracks and our changes in response to the discussions at and results of the last RERS Challenge in Cyprus.

Keywords: Benchmark generation · Program verification · Temporal logics · LTL · CTL · Property-preservation · Obfuscation · Synthesis

1 Introduction

The Rigorous Examination of Reactive Systems (RERS) Challenge is an annual event concerned with software verification tasks—called benchmarks—on which participants can test the limits of their tools. In its now 9th iteration, the RERS Challenge continues to expand both its underlying benchmark generator infrastructure and the variety of its tracks. This year, RERS is part of

D. Beyer et al. (Eds.): TACAS 2019, Part III, LNCS 11429, pp. 101–115, 2019.
https://doi.org/10.1007/978-3-030-17502-3_7

TOOLympics [2]. As during previous years [9,12,13], RERS 2019 features tracks on sequential and parallel programs in programming/specification languages such as Java, C99, Promela [11], and (Nested-Unit) Petri nets [8,19]. Properties that participants have to analyze range from reachability queries over linear temporal logic (LTL) formulae [20] to computational tree logic (CTL) properties [6]. Participants only need to submit their "true"/"false" answers to these tasks. As a new addition in 2019, we enrich RERS with industrial tracks in which benchmarks are based on real-world models.

The main goals of RERS[1] are:

1. Encourage the combination of methods from different (and usually disconnected) research fields for better software verification results.
2. Provide a framework for an automated comparison based on differently tailored benchmarks that reveal the strengths and weaknesses of specific approaches.
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.

One aspect that makes RERS unique in comparison to other competitions or challenges on software verification is its automated benchmark synthesis: The RERS generator infrastructure allows the organizers to distribute new and challenging verification tasks each year while knowing the correct solution to these tasks. Contrarily, in similar events such as the Software Verification Competition (SV-COMP) [3] which focuses on programs written in C and reachability queries, benchmarks are hand-selected by a committee and most of them are used again for subsequent challenge iterations. That the solutions to these problems are already known does not harm because, e.g. SV-COMP, does not merely focus on the answers to the posed problems, but also on details of how they are achieved. To attain this, SV-COMP features a centralized evaluation approach along with resource constraints where participants submit their tools instead of just their answers to the verification tasks. During this evaluation phase, which builds on quite an elaborate competition infrastructure, obtained counterexample traces are also evaluated automatically [4].

The situation is quite different for the Model Checking Contest (MCC) [16], a verification competition that is concerned with the analysis of Petri nets, where the correct solutions to the selected verification tasks are not always known to the competition organizers. In such cases, the MCC evaluation is often based on majority voting concerning the submissions by participants, an approach also followed by a number of other competitions despite the fact that this may penalize tools of exceptional analysis power. In contrast, the synthesis procedure of verification tasks for RERS also generates the corresponding provably correct solutions using a correctness-by-construction approach. Both SV-COMP and MCC have therefore added RERS benchmarks to their problem portfolio.

[1] As stated online at http://www.rers-challenge.org/2019/.

As stated above, RERS aims to foster the combination of different methods, and this includes the combination of different tools. During last year's RERS Challenge for example, one participant applied three different available tools in order to generate his submission[2] and thereby won the Method Combination Award within RERS[3]. In order to host an unmonitored and free-style challenge such as RERS on a regular basis—one where just the "true"/"false" answers need to be submitted—an automated benchmark synthesis is a must.

Potential criticism of such a synthesis approach might be that the generated verification tasks are not directly connected to any real-world problem: Their size might be realistic, however their inherent structure might be not. This criticism very much reflects a perspective where RERS benchmarks are structurally compared to handwritten code. On the other hand, being synthesized from temporal constraints, RERS benchmarks very much reflect the structure that arises in generative or requirements-driven programming. In order to be close to industrial practice, RERS 2019 also provides benchmarks via a combination of synthesis with real-world models. For this endeavor, we collaborated with ASML, a large Dutch semiconductor company.

When developing controller software, over time updates and version changes inevitably turn originally well-documented solutions into legacy software, which typically should preserve the original controller behavior. RERS 2019 addresses this phenomenon by generating legacy-like software from models via a number of property-preserving transformations that are provided by the RERS infrastructure [22]. This results in correct 'obfuscated' (legacy) implementations of the real-world models provided by ASML.

The parallel benchmarks of the last RERS challenge were built on top of well-known initial systems, dining philosophers of various sizes. As a next step towards a fully automated benchmark generation process, we created the initial system in a randomized fashion this year. The subsequent property-preserving parallel decomposition process, which may result in benchmarks of arbitrary degrees of parallelism, remained untouched [23]. For RERS 2020 we plan to use the more involved synthesis approach presented in [15] in order to be able to also guarantee benchmark hardness.

Moreover, in response to participants' requests, we implemented a generator that creates candidates for branching time properties for the parallel benchmarks. The idea is to syntactically transform available LTL properties into semantically 'close' CTL formulae. This turns out to provide interesting CTL formulae for the benchmarks systems. These formulae's validity has, of course, to be validated via model checking as the generation process is not (cannot be) semantics preserving.

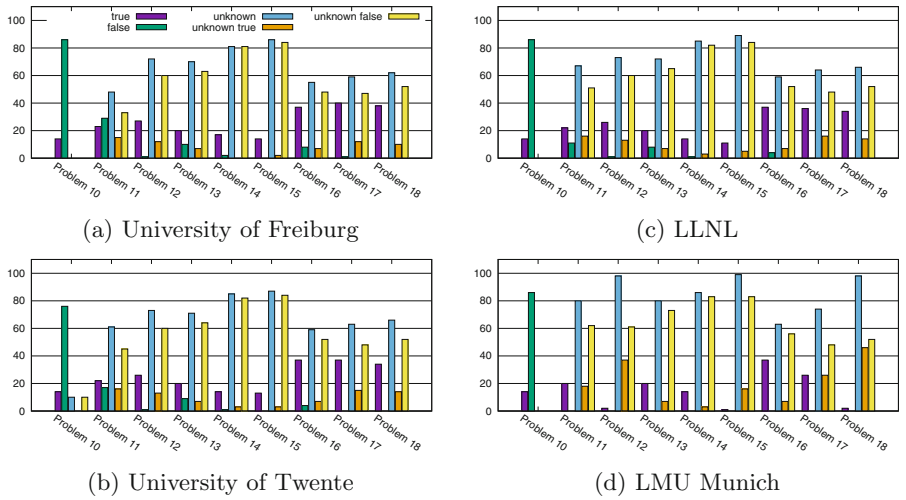
In the following, the detailed observations from RERS 2018 are described in Sect. 2. Section 3 then summarizes improvements within the parallel tracks of RERS that we implemented for the 2019 challenge, before Sect. 4 introduces

[2] Details at http://www.rers-challenge.org/2018/index.php?page=results.
[3] The reward structure of RERS is described in previous papers such as [12].

the new industrial tracks with their dedicated benchmark construction. Our conclusions and outlook to future work can be found in Sect. 5.

2 Lessons Learned: The Sequential Tracks of RERS 2018

For RERS 2018, we received four contributions to the Sequential Reachability track and two contributions to the Sequential LTL track. Detailed results are published online.[4] The tools that participants used for the challenge are quite heterogeneous: Their profiles range from explicit-state model checking over trace abstraction techniques to a combination of active automata learning with model checking [5,10,14,18,25]. During the preparations for the new sequential and industrial tracks, we started a closer investigation on lessons we might learn from the results of the RERS 2018 challenge in addition to the valuable feedback collected during the RERS 2018 meeting in Limassol.



(a) University of Freiburg (c) LLNL

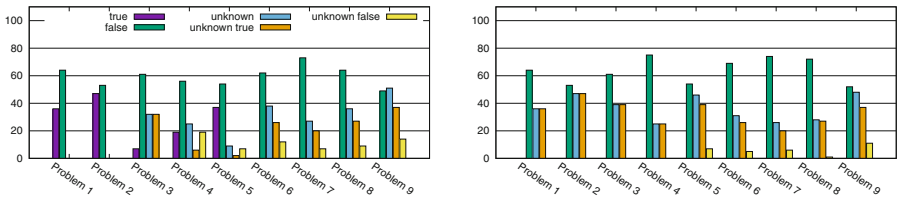
(b) University of Twente (d) LMU Munich

Fig. 1. Reachability results. (Color figure online)

In Fig. 1, the results of the participants of the reachability challenge are visualized. The blue bars indicate how many properties have not been addressed by the respective participant for a problem. Hence, these blue bars point to potential opportunities for achieving better challenge results for each tool. It is observable that the amount of green bars is decreasing with increasing problem size and difficulty. This shows that less unreachable errors are detected with increased problem size. In contrast, the purple bars still show a fair number of results for reachable errors.

[4] http://www.rers-challenge.org/2018/index.php?page=results.

It is obvious that showing the absence of a certain error requires a more complicated proof than demonstrating that it is reachable. Therefore, the observed result is not unexpected. To investigate this further, the blue bars are split up into the corresponding categories from which the unsolved properties originate. An orange bar shows the number of unreported reachable errors. A yellow bar shows the number of unreported unreachable errors. In most cases, the yellow bar is comparable in size to the blue bar for a problem. On the one hand, this is evidence which demonstrates that proving unreachable errors is still a hard challenge no matter which approach has been applied. On the other hand, the charts indicate that participating tools scale quite well also on the larger problems for demonstrating the existence of errors.



(a) LLNL (b) University of Twente

Fig. 2. LTL results. (Color figure online)

We found a similar situation in the LTL track results reported in Fig. 2. In this figure a purple bar indicates that a LTL formula holds. This proof requires a deep understanding of major parts of the complete execution graph. This is therefore the counterpart for proving an error unreachable. As expected, it appears to be much easier for tools to disprove an LTL formula on the given examples the same way as it seems significantly easier to prove error reachability. With a few exceptions, the blue bars indicating unreported properties for a given problem are comparable in height with the orange bars for LTL formulae expected to hold on the given instance. We want to highlight that the tools which participated in RERS 2018 demonstrated a good scalability for disproving LTL formulae across the different problem sizes.

Based on the results handed in to RERS 2018, we observe some maturity in tools disproving LTL formulae and finding errors, which are both characterized by having single paths as witnesses. We appreciate this trend because a lacking scalability of verification tools was a major motivation to start the RERS challenge.

As a next step, we intend to motivate future participants to further investigate the direction of proving LTL formulae and error unreachability on systems. These properties require more complex proofs as it is not possible to verify the answer with a single violating execution path. Instead it is required to create a deeper understanding of all possible execution paths in order to give a sound answer. There is a higher chance to make a mistake and give a wrong answer resulting in a penalty.

With RERS 2019, we therefore want to encourage people to invest into corresponding verification tools by valuing that verifiable properties are more complicated to analyze than refutable ones. In the future we will award two points for each correct report of an unreachable error or a satisfied LTL formula in the competition-based ranking. The achievement reward system remains unchanged.

3 Improvements in the Parallel Tracks for RERS 2019

The initial model used for the RERS 2018 tracks on parallel programs was chosen to be the Dining Philosophers Problem in order to feature a well-known system [13]. With the goal to reflect the properties of this system as best as possible, the corresponding LTL and CTL properties were designed manually. To streamline our generation approach and minimize the amount of manual work involved, we decided to further automate these steps for RERS 2019.

In [15], a new workflow for the generation of parallel benchmarks was presented that fully automates the generation process while ensuring certain hardness guarantees of the corresponding verification/refutation tasks. Due to time constraints, we could not fully integrate this new approach into our generation pipeline for RERS 2019. Instead, we combined new and existing approaches to achieve a full automation (Fig. 3). Our workflow for RERS'19 therefore does not yet guarantee the formal hardness properties presented in [15]. On the other hand, it integrates the generation of CTL properties, an aspect that was not discussed in [15].

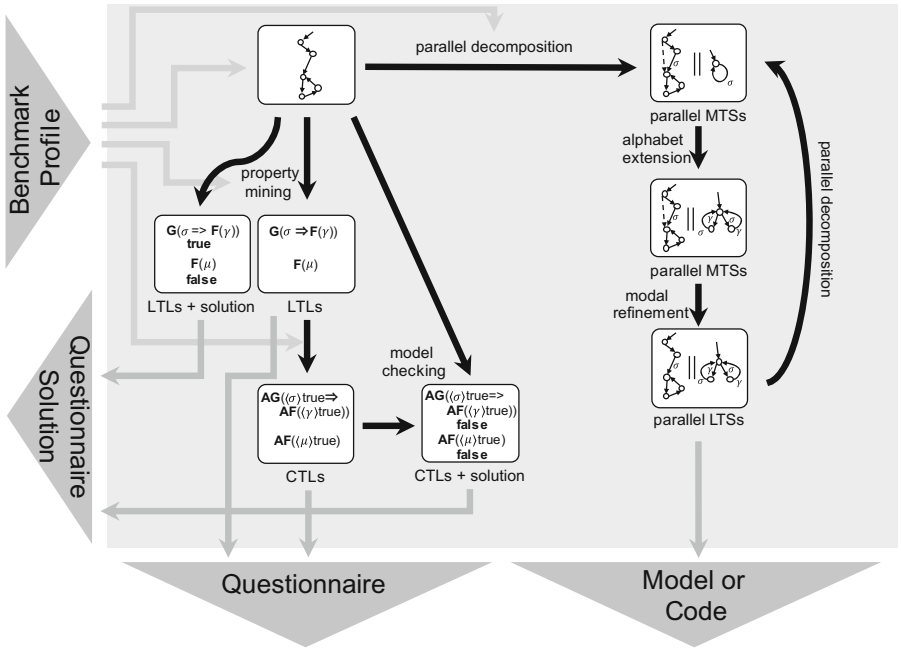


Fig. 3. Workflow of the benchmark generation for the RERS'19 parallel programs.

Input to the overall workflow (Fig. 3) is a benchmark profile that contains metadata such as the number of desired verifiable/refutable LTL/CTL properties, number of parallel components in the final code, and similar characteristics. The generation of a parallel benchmark starts with a labeled transition system (LTS). We chose to randomly generate these for RERS'19, based on parameters in the input benchmark profile. Alternatively, one could choose an existing system modeled as an LTS if its size still permits to model check it efficiently.

3.1 Property Generation

Given the initial LTS, we randomly select verifiable and refutable properties based on certain LTL patterns. This process is called property mining in Fig. 3 and was previously used to generate the parallel benchmarks of RERS'16 [9] and some of RERS'17 [12].

As a new addition to the automated workflow, we implemented a generation of CTL formulae based on the following idea:

- Syntactically transform an LTL formula ϕ_l to a CTL formula ϕ_c. This yields structurally interesting CTL properties but is not guaranteed to preserve the semantics.
- Check ϕ_c on the input model. This step compensates for the lack of property preservation of the first step.
- Possibly negate ϕ_c and then apply de Morgan-like rules to eliminate the leading negation operator in case the ratio of satisfied and violated properties does not match the desired characteristics. This works for CTL, as in contrast to LTL, formulae or their negations are guaranteed to hold (law of excluded middle).

We realized the transformation from an LTL formula to a corresponding CTL formula by prepending an **A** ('always') to every LTL operator which requires the formula to hold on every successor state. For a state to satisfy $\mathbf{AG}\,\phi$ for example, ϕ has to hold in every state on every path starting in the given state. Additionally, we introduced a diamond operator for every transition label that is not negated in the LTL formula and a box for every negated label as detailed below. The transformation was implemented as follows where the LTL formula to the left of the arrow is replaced by the CTL formula to the right of the arrow.[5]

$$\begin{aligned}
\mathbf{G}\,\phi &\rightarrow \mathbf{AG}\,\phi \\
\mathbf{F}\,\phi &\rightarrow \mathbf{AF}\,\phi \\
\phi\,\mathbf{U}\,\psi &\rightarrow \mathbf{A}(\phi\,\mathbf{U}\,\psi) \\
\phi\,\mathbf{W}\,\psi &\rightarrow \mathbf{A}(\phi\,\mathbf{W}\,\psi) \\
a &\rightarrow \langle a\rangle \mathrm{true} \\
\neg a &\rightarrow [a]\mathrm{false}
\end{aligned}$$

[5] For more details on the syntax of the LTL and CTL properties, see http://rers-challenge.org/2019/index.php?page=problemDescP.

The diamond operator $\langle a \rangle \phi$ holds in a state iff the state has at least one outgoing transition labeled with an a whose target state satisfies ϕ. In this case $\langle a \rangle$true holds in a state if it has an outgoing transition labeled with a because every state satisfies 'true'. The box operator $[a]\phi$ holds in a state iff every outgoing transition labeled with an a satisfies ϕ. The negation of an atomic proposition a was replaced by $[a]$false which is only satisfied by a state which has no outgoing transitions labeled with an a.

Based on the previously mentioned steps, we can automatically generate LTL and CTL properties that are given to participants of the challenge as a questionnaire (see Fig. 3). Similarly, the corresponding solution is extracted and kept secret by the challenge organizers until the submission deadline has passed and the results of the challenge are announced.

3.2 Expansion and Translation of the Input Model

In order to synthesize challenging verification tasks and provide parallel programs, we expand the initial LTS based on property-preserving parallel decompositions [23] (see top and right-hand side of Fig. 3). The corresponding procedure works on modal transition systems (MTSs) [17], an extension of LTSs. This parallel decomposition can be iterated. During this expansion procedure, the alphabet of the initial system is extended by artificial transition labels. More details including examples can be found in [13,21].

As a last step, the final model of the now parallel program is encoded in different target languages such as Promela or as a Nested-Unit Petri net [8] in the standard PNML format[6]. The final code or model specification is presented to participants of the challenge along with the questionnaire that contains the corresponding LTL/CTL properties.

Please note the charm of verifying branching time properties: As CTL is closed under negation, proving whether a formula is satisfied or violated can in both cases be accomplished using standard model checking, and in both cases one can construct witnesses in terms of winning strategies. Thus there is not such a strong discrepancy between proving and refuting properties as in LTL.

4 Industrial Tracks

RERS 2019 includes tracks that are based on industrial embedded control systems provided by ASML. ASML is the world's leading provider of lithography systems for the semiconductor industry. Lithography systems are very complex high-tech systems that are designed to expose patterns on silicon wafers. This processing must not only be able to deliver exceptionally reliable results with an extremely high output on a 24/7 basis, it must do so while also being extremely precise. With patterns becoming smaller and smaller, ASML TWINSCAN lithography systems incorporate an increasing amount of control software to compensate for nano-scale physical effects.

[6] ISO/IEC 15909-2: https://www.iso.org/standard/43538.html.

To deal with the increasing amount of software, ASML employs a component-based software architecture. It consists of components that interact via explicitly specified interfaces, establishing a formalized contract between those components. Such formal interface specifications not only include syntactic signatures of the functions of an interface, but also their behavioural constraints in terms of interface protocols. Furthermore, non-functional aspects, such as timing, can be described.

Formal interface specifications enable the full potential of a component-based software architecture. They allow components to be developed, analyzed, deployed and maintained in isolation. This is achieved using enabling techniques, among which are model checking (to prove interface compliance), observers (to check interface compliance), armoring (to separate error handling from component logic) and test generation (to increase test coverage).

For newly developed components, ASML specifies the corresponding interface protocols. However, components developed in the past often do not have such interface protocol specification yet. ASML aims to obtain behavioral interface specifications for such components. Model inference techniques help to obtain such specifications in an effective way [1]. Such techniques include, for instance, static analysis exploiting information in the source code, passive learning based on execution logs, active automata learning querying the running component, and combinations of these techniques.

ASML collaborates with ESI[7] in a research project on the development of an integrated tool suite for model inference to (semi-automatically) infer interface protocols from existing software components. This tool suite is applied and validated in the industrial context of ASML. Recently, this tool suite has been applied to 218 control software components of ASML's TWINSCAN lithography machines [26]. 118 components could be learned in an hour or less. The techniques failed to successfully infer the interface protocols of the remaining 100 components.

Obtaining the best performing techniques to infer behavioral models for these components is the goal of the ASML-based industrial tracks of RERS 2019. Any model inference technique, including source code analyzers, passive learning, (model-based) testers and (test-based) modelers including active automata learning, and free-style approaches, or combinations of techniques can be used. The best submissions to the challenge might be used by ASML and ESI and incorporated into their tool suite.

4.1 ASML Components for RERS

For the RERS challenge, ASML disclosed information about roughly a hundred TWINSCAN components. We decided to select 30 among them to generate challenging benchmark problems for RERS 2019, and three additional ones that are used for training problems. Using these components allows participants to

[7] ESI is a TNO Joint Innovation Centre, a collaboration between the Netherlands Organisation for Applied Scientific Research (TNO), industry, and academia.

apply their tools and techniques on components of industrial size and complexity, evaluating their real-world applicability and performance.

For the disclosed components, Mealy machine (MM) models and (generated) Java and C++ source-code exist. The generation of benchmarks for the RERS challenge is based on the MM models. This allows us to open the industrial tracks also to tools that analyze C programs. The Java code of the challenge is generated by the organizers as described later in Sect. 4.4 and does not represent the originally generated Java code provided by ASML. This prevents participants from exploiting potential structural patterns in this original Java code (such structural information does not exist in legacy components). Furthermore, an execution log is provided for each component. Each execution log contains a selected number of logged traces, provided by ASML, representing behavior exhibited by either a unit or integration test.

The remainder of this section provides a brief overview of how properties are generated for these benchmarks and how code is generated using the obfuscation infrastructure from previous sequential RERS tracks. Figure 4 presents an overview of the corresponding benchmark generation workflow that is described in the following.

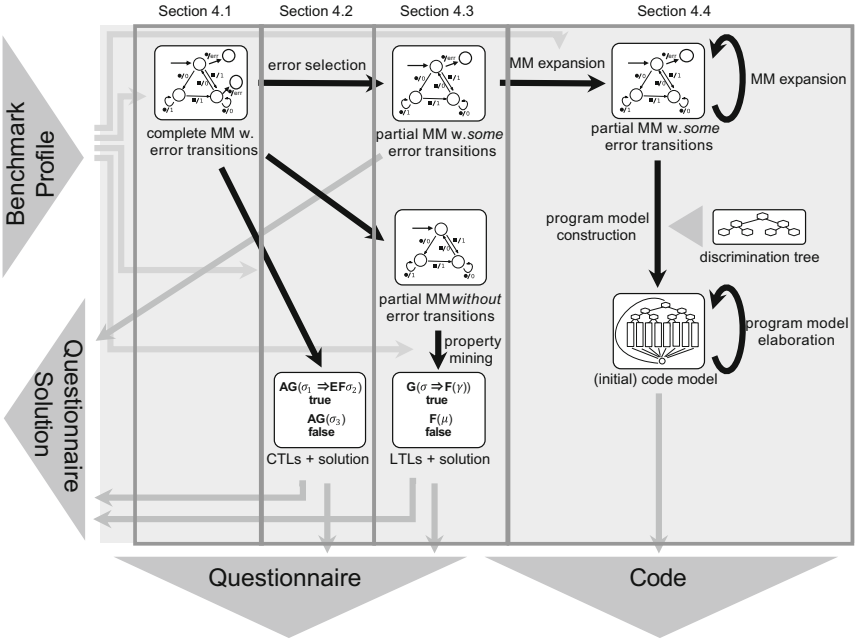


Fig. 4. Workflow of the benchmark generation for the new industrial tracks.

4.2 Generation of CTL Properties

We compute CTL formulae from Mealy machines using conformance testing algorithms. We generate a small set of traces that characterizes each state.

Using this, we can define for each state q a CTL state formula σ_q that characterizes part of its behavior. If $i_1/o_1, i_2/o_2$ is an IO sequence of state q, then formula σ_q takes the form

$$EX(i_1 \wedge EX(o_1 \wedge EX(i_2 \wedge EXo_2))).$$

These characterizing formulae are the basis for CTL properties, e.g., of the form

$$\begin{aligned} &AG(\sigma_1 \vee \sigma_2 \vee \ldots \vee \sigma_n), \\ &AG(\sigma_1 \Rightarrow EX(i \wedge EXo \wedge \sigma_2)), \text{ or} \\ &AG(\sigma_1 \Rightarrow EF\sigma_3), \end{aligned}$$

where i and o denote symbols from the set of inputs and outputs of the Mealy machine model, respectively. Additionally, we generate CTL formulae that do not hold in the model using the same approach.

4.3 LTL and Reachability Properties

Regarding the new ASML-based benchmarks, we used a property mining approach for the generation of LTL properties. By mining we mean that properties are extracted from the model without altering this model. As a first step, we temporarily discard all error transitions from the input Mealy machine (MM) (see Fig. 4): In line with the benchmark definition used in former editions of the RERS tracks on sequential programs, our LTL properties only constrain infinite paths. This nicely reflects the fact that controllers or protocols are typically meant to continuously run in order to react on arising input.

Having discarded all error transitions, our approach first generates random properties from relevant patterns according to [7]. A model checker is then used to determine whether or not the generated properties hold on the given input model. We iterate this process until we find a desired ratio between satisfied and violated properties. This mining approach is very similar to the previous LTL generation in RERS (cf. [22]), with the exception that no properties are used for synthesizing a MM. Because we have never altered the original MM with regard to its infinite paths, all extracted LTL properties that are satisfied characterize the input/output behavior of the given real-world model.

Similar to the former editions of RERS, the new industrial tracks also provides reachability tasks ("Is the error labeled x reachable?"). This generation process is disjoint with the LTL track generation. While we discarded all error transitions from the input model during conversion from the input model to code in the LTL track generation, we select real errors from the given input model and map them to unique error states before code generation during the reachability task generation. This way, all included errors are taken from a real system and are not synthetical. The same input models are used for the generation of the benchmarks for the reachability track and the LTL track. There are just slightly different pre-processes in place that address the handling of error transition during generation. Using real error paths is again in contrast to the benchmark

synthesis of RERS that was applied during previous years where reachability tasks were artificially added to (already artificial) input models.

As depicted in the top-left corner of Fig. 4, the initial MM model is complete, meaning that each input symbol that is not supported in a certain state is represented by an error transition leading to a sink state. We randomly select some of those error transitions and reroute them so that they each lead to a distinct error state. At the same time, we introduce unreachable error states to the MM and enumerate both the reachable and unreachable error states. The resulting reachability vector is reported back to the challenge organizer as part of the Questionnaire Solution (Fig. 4). The error states are then rendered as guarded "verifier errors" in the final program (see Sect. 4.4). Unsupported transitions that were not selected for the reachability tasks are rendered as "invalid input", in line with the previous RERS tracks on sequential programs.

4.4 Obfuscation and Code Generation

The obfuscation and code generation steps are reused from the existing RERS benchmark generator of the sequential tracks. As described in Fig. 4 and in Section 11 and Section 12 of [22], the translation from the initial MM to the final code is divided into smaller steps, which are implemented as individual modules.

As shown in the right-hand side of Fig. 4, the partial MM is first expanded as done before. Additional states which are clones of existing states are added such that they are unreachable. Next, a discrimination tree is constructed using different kinds of variables as properties on the nodes and constraints on these variables on the outgoing edges of the decision tree. Based on the choice of these variables, the current complexity of the synthetic RERS benchmarks is controllable. It may range from plain encodings using only integer variables to encode the subtrees, to options with string variables, arithmetic operations and array variables in the same fashion as it was done in the past for RERS.

Next, the automaton is randomly mapped to the leaf nodes of the decision tree. The constraints collected along a path from the decision tree root to a leaf is used to encode a state of the automaton associated with that leaf. The automaton transitions are encoded based on the decision tree encoding. The now completely encoded problem is translated into the target language. While ASML normally generates C++ code from its automaton models, we decided to maintain the old RERS tradition of providing a Java and a C encoding for each problem. The underlying MM is maintained during this obfuscation and encoding step as it has been in the previous editions of RERS.

5 Conclusion and Outlook

With the addition of industrial tracks where benchmarks are based on real-world models, RERS 2019 combined the strength of automated synthesis with the relevance of actively used software. Due to these new tracks based on a collaboration with the company ASML, the variety of different tasks that participants of

RERS can address has again expanded. Independently of this new addition, we further improved our generation infrastructure and realized a fully-automated synthesis of parallel programs that feature intricate dependencies between their components.

In the future, we intend to fully integrate the approach presented in [15] into our infrastructure in order to guarantee formal hardness properties also for violated formulae. Future work might include equivalence-checking tasks between a model and its implementation, for example based on the systems provided by ASML. Furthermore, we intend to provide benchmark problems for weak bisimulation checking [24] for the RERS 2020 challenge. As a longer-term goal, we continue our work towards an open-source generator infrastructure that allows tool developers to generate their own benchmarks.

Acknowledgments. This work was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, and was supported by the LLNL-LDRD Program under Project No. 17-ERD-023. IM Release Nr. LLNL-CONF-766478.

References

1. Aslam, K., Luo, Y., Schiffelers, R.R.H., van den Brand, M.: Interface protocol inference to aid understanding legacy software components. In: Proceedings of MODELS 2018, co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, pp. 6–11 (2018)
2. Bartocci, E., et al.: TOOLympics 2019: an overview of competitions in formal methods. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. xx–yy. Springer, Cham (2019)
3. Beyer, D.: Competition on software verification (SV-COMP). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_38
4. Beyer, D.: Software verification and verifiable witnesses. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 401–416. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_31
5. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_31
6. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982). https://doi.org/10.1007/BFb0025774
7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No. 99CB37002), pp. 411–420, May 1999
8. Garavel, H.: Nested-unit Petri nets. J. Log. Algebraic Methods Program. **104**, 60–85 (2019)

9. 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: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 787–803. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47169-3_59
10. Heizmann, M., et al.: Ultimate Automizer with SMTInterpol. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 641–643. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_53
11. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2011)
12. Jasper, M., et al.: 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)
13. Jasper, M., Mues, M., Schlüter, M., Steffen, B., Howar, F.: RERS 2018: CTL, LTL, and reachability. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 433–447. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_27
14. Jasper, M., Schordan, M.: Multi-core model checking of large-scale reactive systems using different state representations. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 212–226. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_15
15. Jasper, M., Steffen, B.: Synthesizing subtle bugs with known witnesses. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 235–257. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_16
16. Kordon, F., et al.: Report on the model checking contest at Petri nets 2011. In: Jensen, K., van der Aalst, W.M., Ajmone Marsan, M., Franceschinis, G., Kleijn, J., Kristensen, L.M. (eds.) Transactions on Petri Nets and Other Models of Concurrency VI. LNCS, vol. 7400, pp. 169–196. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35179-2_8
17. Larsen, K.G.: Modal specifications. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 232–246. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-52148-8_19
18. Meijer, J., van de Pol, J.: Sound black-box checking in the LearnLib. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NFM 2018. LNCS, vol. 10811, pp. 349–366. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77935-5_24
19. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River (1981)
20. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (SFCS 1977), pp. 46–57, October 1977
21. 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, June 2017
22. 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)
23. Steffen, B., Jasper, M.: Property-preserving parallel decomposition. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 125–145. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_7
24. Steffen, B., Jasper, M.: Generating hard benchmark problems for weak bisimulation. LNCS. Springer (2019, to appear)

25. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 332–347. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_24
26. Yang, N., et al.: Improving model inference in industry by combining active and passive learning. In: IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER) (2019, to appear)