Kalpa Publications in Computing
Volume 3, 2017, Pages 157–163
RV-CuBES 2017. An International Workshop on
Competitions, Usability, Benchmarks, Evaluation,
and Standardisation for Runtime Verification Tools

Online Runtime Verification Competitions: How To Possibly Deal With Their Issues (Position Paper)

Julien Signoles

CEA, LIST, Software Reliability and Security Lab,
91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Abstract

This short paper presents a compilation of feedback about online runtime verification competitions from an active contestant. In particular, it points out several issues and how they could possibly be fixed.

Keywords: online runtime verification, tool competitions, benchmarks

1 Introduction

CRV, the runtime verification competition, was created in 2014 [3], then organized again in 2015 [9] and 2016 [19]. Each year, the competition was split in three categories: the *C* track, the *Java* track and the offline track. The number of competitors used to be low in each track. The *C* track was even cancelled in 2016 because the *E-ACSL* tool [16] was eventually the only contestant. Then the organizers decided to take a break and to not organize the competition this year (2017). I think that it is an appropriate moment to summarize what could be learned from the first editions in order to try to improve the next ones. This paper is an attempt in that direction for the online tracks (*C* and *Java*). Yet some ideas could also apply to the offline track. It is based on my own experience from my participation in the *C* track with the *E-ACSL* tool in 2014 (2nd place) and 2015 (winner). It is also based on informal discussions with a few other contestants and organizers. The main goal of this paper is to encourage discussions between the competition organizers and the tool developers in order to improve the next editions of any online runtime verification competition.

2 Areas of Improvement

This section introduces several areas of improvement. First, Section 2.1 introduces the competition goals, while Section 2.2 discusses the scoring method. Then, Section 2.3 and Section 2.4 respectively describe the wide variety of properties and input languages targeted by CRV. Next, Section 2.5 details benchmark issues. Finally, Section 2.6 deals with a few organization concerns.

2.1 Competition Goals

The official goals of the competition organizers are multiple [3]:

- stimulate the runtime verification tool development;
- produce shared runtime verification benchmarks;
- discuss measures for comparing tools;
- compare different aspects of tools with different benchmarks and criteria;
- enhance runtime verification tool visibility.

All these goals are perfectly fine and were actually shared by almost all contestants, even if possibly with different priorities. For instance some of them were more interested in measuring runtime and memory overhead, some others preferred to focus on tool expressiveness, while others would like to experiment with new benchmarks. However it is often difficult to kill several birds with a single stone and the risk is high to kill no bird at all. In particular, organizers and contestants have only a limited amount of time to devote to the competition. For instance if contestants spend a lot of time to create new benchmarks, they have almost no time left to run benchmarks from the other contestants. Indeed it is always important to remember that tool developers do not participate if it requires too much efforts for not enough benefits. However, if they do not invest enough effort, then the organization team may quickly be demotivated and would not organize the competition anymore. Though removing motivating goals is always annoying. A workaround could be to just fix priorities between them, so organizers and contestants would reach the most important goals but could just skip a few others according to their available time.

2.2 Scoring

Fixing priorities could also help to improve tool evaluation. Indeed runtime verification tools are usually evaluated according to three main criteria:

- *efficiency:* what is the time and memory overhead of the instrumented program with respect to the input program?
- *expressiveness:* what properties is the tool able to verify?
- *correctness:* does the tool solve correctly the input problem?

The current scoring methodology of CRV tries to take into account these three criteria by including a correctness score, a time overhead score and a memory score in a same formula [3], while correctness is measured by a score of 0 for every unanswered problem. It leads to a final score which mixes all these criteria. However it is difficult (or even impossible) to interpret such a result: is tool A better than tool B because it is more time efficient or more expressive? We just do not know. A practical solution could be to compute two different scores, one for expressiveness and another one for efficiency, while unsoundness should result in any case to very low (or even negative) scores.

2.3 Properties

The above-mentioned question of expressiveness is particularly important because runtime verification tools may verify very broad kinds of properties, including but not limited to various timed or temporal properties, informal flow properties, concurrency properties, integer and memory overflows, memory errors and functional properties. It seems unlikely that a single tool is able to tackle all of them. It usually focuses on one single class of properties even if it could possibly encode a few more. But comparing tools which verify different properties is dubious, even if the target programming language is the same. Consequently creating several categories corresponding to different properties is necessary. However, the usual low number of contestants is probably an evidence that there are not enough tools to create numerous categories. That is certainly a tough issue which should be carefully dealt with. One workaround could be to split the online tracks into two or three big families of properties (*e.g.*, for *C* programs, temporal and timed properties on one hand, and undefined behaviors in another hand). The properties which do not belong to any track could not be addressed but they could possibly be grouped together in a free unrated categories to make known that some tools are able to deal with them.

2.4 Input Languages

Online runtime verification tools analyze at runtime a program written in a programming language (either *C* or *Java* in the first CRV editions) in order to check the validity of properties written in a specification language. Sometimes these properties are even left implicit, for instance when checking the absence of *C* undefined behaviors.

Specifying the programming language is necessary but usually easy: it is enough to indicate the exact supported version(s) and add a pointer to a reference document (*e.g* the norm ISO C99 [14] for *C* or the Java Language Specification for *Java* SE 7 [11]).

Specifying the specification language is more complicated because there is nowadays no *de facto* standard for most classes of properties, apart from *JML* [17] for *Java* and *ACSL* [4] for *C* when interested in functional properties. For instance, there are many temporal logic languages and, even for one particular language like LTL [18], there are several input languages expressing properties as logic formulæ or Büchi automata. Currently most tools support their own input specification language and there are only a few translators from one language to the other. Given that situation, I think that the present CRV choice is fine. It consists in providing both an informal specification and a formal one expressed in any specification language as long as the expected property is clear enough.

2.5 Benchmarks

There is no competition without benchmarks. Though there is currently no widespread benchmarks for runtime verification tools. One of the underlying goal of CRV was actually to create them. Consequently, taking inspiration from the software verification competition (SV-COMP) [5], CRV was split in three different phases:

1. collecting benchmarks;
2. training tools and submitting monitors;
3. evaluating results.

The two first steps were performed by the contestants while the latter was performed by the organizers. In particular, the contestants should provide the benchmarks of their choice (either 3, or 5 benchmarks, depending on the competition edition). Each benchmark must contain a program package including the source code and the way to compile and run it, as well as a specification package including an English description of the property to be verified, one logical representation in any specification language, instrumentation information based on a notion of events and traces and the expected verdict [3]. Depending on the edition, the submitted results consist in an instrumented code (CRV 2014) or a way to generate it (CRV 2015 and 2016). Even if it is globally fine, I see some issues with this approach.

Too Strong Assumptions

First, it makes some implicit assumptions about runtime verification tool design which are not always true. Indeed it assumes

1. explicit notions of events and traces; and

2. a clear separation between the monitor and the instrumented system.

While events and traces are at the heart of runtime verification tools [8, 20], they are not necessarily made explicit. In particular, runtime assertion checkers and memory debuggers usually consider program states as events and the successive states of the program at runtime as traces, but they almost never exhibit them. Also, automatic pruning of states in order to observe only a few memory locations and make the verification efficient may be seen as part of the tool's necessary features [15, Section 5]. Consequently, the events of interest (that is, the set of memory locations to be observed) should not be part of the inputs to be provided and property 1 should not be assumed.

Also assumption 2 does usually not hold for inline verification tools which weave the monitor directly in the input program. Indeed, in that case, the instrumented system contains both the input program and the inline monitor. Consequently the input program should not be pre-instrumented. It is also much better to produce as result the ways to generate the output programs, otherwise it may eventually lead to some manual modifications of the monitor or the instrumented system before submission.

Benchmark Origins

A second issue of this competition is that the contestants must provide all the benchmarks. That is indeed very time consuming and Section 2.1 already explained what are the probable consequences in term of team participation. It may also lower benchmark quality since contestants have possibly not enough time to prepare them accurately. They could also contain bias in order to match the tool capabilities of the benchmark author. Still a competition needs benchmarks and, when there is none, they must be created at some point. It would certainly be too much effort to let the organization committee deal with their creation alone. A potential trade-off could be to have a few benchmarks proposed by the organizers, some proposed by the contestants and some others coming from past editions. In addition to solving the time issue, the benchmarks of lower quality would disappear from edition to edition. It would also make visible the progress of each tool by illustrating which benchmarks are successfully/efficiently tackled on year y which were only imperfectly/slowly handled in year $y - 1$.

2.6 Organization

As already pointed out, one of the major issue of such competitions is the lack of time of both organizers and tool developers. It causes unexpected delays, a low number of submitted benchmarks of various degrees of quality, and an even lower number of final submissions. That eventually leads to demotivating both organizers and contestants. A few solutions have been proposed in the previous sections. However, if implemented, some work previously done by contestants would now be done by organizers. For instance, from Section 2.5, the organizers must provide new benchmarks and also select additional ones from past editions. It would require extra work for the organization committee. To solve this drawback, a few contestants could be partly involved in the competition organization. Also one could consider to split the organization in different parts. This way, an organizer could be only partially involved and so (s)he would have more free time to do something else, including becoming a competition contestant. Below is one organization proposal.

- The *steering committee* defines the competition policy, frequency and goals, follows the competition from year to year and provides advice. It also chooses the organization committee of the next edition.
- The *organization committee* leads one particular edition. In particular, it sets the exact rules and evaluates the submissions.
- The *track committee* for every track (or a single *competition committee* shared by all tracks) is composed by one contestant of each registered team and a subset of the organization committee. It chooses benchmarks (in addition to the contestant benchmarks), either from past editions or newly created. It also validates contestant benchmarks, asks for clarification and fixes contentions.

3 Related Work

Tool competitions have been organized in several computational domains for a long time. We limit our related work to competitions related to program analysis and provers. CRV is directly inspired by SV-COMP [6] organized since 2012. SV-COMP aims at a comparative evaluation of fully-automated software verifiers, mainly model checkers. SV-COMP 2017 targeted *C* programs only and was split into several categories and sub-categories. Categories tackled different properties (*e.g.* termination or overflow), while sub-categories reflect different types of programs. Having many categories and sub-categories is certainly possible because of the large number of contestants (32 in 2017). The 2017 edition also required contestants to provide witness automata as proof of validity or invalidity.

Similarly to SV-COMP, termCOMP [10] has also different categories depending on the type of properties to be verified (worst case execution time, certified properties or termination) and sub-categories depending on the target systems (reactive systems, *Java* or *C* programs). In 2017, 17 tools entered into the competition which contained 4 categories and 20 sub-categories (with between 2 and 5 contestants in each of them).

VerifyThis [13] is a 2-day event organized during international conferences or workshops that focuses on semi-automatic mathematical proof guided by tools (instead of fully-automatic verification). The competition is more about comparing teams' skills than evaluating tools. Accordingly, the criteria of evaluation are correctness, completeness and elegance of solutions, which are fully different from the current CRV criteria.

The RERS grey-box challenge [12] focuses on verifying specific properties (reachability and LTL properties) on specific program patterns. Consequently they only address a few programs (for instance, *C* programs whose all variables are of type `int`). It is worth noting that this competition has two different rankings, one numerical ranking and one conceptual ranking. The latter emphasizes the challenge character of the employed methods and is manually computed by the organizers.

The SAT [1], SMT-COMP [2] and CASC [21] competitions respectively focus on comparing SAT solvers, SMT solvers and automated theorem provers (ATPs). Each of them takes benefit of uniform common formats supported by every contestant tool (*e.g.* TPTP for ATPs). This way, comparison of tool expressiveness is made easier.

Cuoq *et al.* [7] explain how static analyzers should be benchmarked while highlighting issues that competition benchmarks should avoid. Some of their conclusions are similar to ours.

4 Conclusion

CRV, the runtime verification competition, was created in 2014. It suffers from a lack of active contestants and has not been organized this year. This paper explains a few of its issues regarding the online runtime verification tracks. It also proposes possible solutions. While I do not pretend that it would instantaneously solve all of them, I hope that this *discussion paper* provides a point of view from an active contestant that could be useful for competition organizers.

Acknowledgment

I would like to thank Zakaria Chihani for his feedback, as well as the anonymous reviewers from their helpful advice. They significantly contribute to improve the quality of this paper. This work has received funding for the S3P project from French DGE and BPIFrance.

References

[1] Balyo, T., Heule, M.J., Järvisalo, M.: Sat competition 2016: Recent developments. In: Conference on Artificial Intelligence (AAAI 2017) (Feb 2017)

[2] Barrett, C., Deters, M., de Moura, L.M., Oliveras, A., Stump, A.: 6 years of SMT-COMP. Journal of Automated Reasoning (Mar 2013)

[3] Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international Competition on Runtime Verification: rules, benchmarks, tools, and final results of CRV 2014. International Journal on Software Tools for Technology Transfer (Apr 2017)

[4] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html

[5] Beyer, D.: Competition on software verification. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). pp. 504–524 (Mar 2012)

[6] Beyer, D.: Software Verification with Validation of Results (Report on SV-COMP 2017). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). pp. 331–349 (Apr 2017)

[7] Cuoq, P., Kirchner, F., Yakobowski, B.: Benchmarking static analyzers. In: International Workshop on Comparative Empirical Evaluation of Reasoning Systems. pp. 32–35 (Jun 2012)

[8] Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, pp. 141–175 (Jun 2013)

[9] Falcone, Y., Ničković, D., Reger, G., Thoma, D.: Second international competition on runtime verification. In: Bartocci, E., Majumdar, R. (eds.) International Conference on Runtime Verification. pp. 405–422 (2015)

[10] Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination Competition (termCOMP 2015). In: International Conference on Automated Deduction (CADE 2015). pp. 105–108 (Aug 2015)

[11] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification, Java SE 7 Edition (Feb 2013), http://docs.oracle.com/javase/specs/jls/se7/html/index.html

[12] Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D.: The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In: International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2012) (Oct 2012)

[13] Huisman, M., Klebanov, V., Monahan, R.: Verifythis 2012 - A program verification competition. International Journal on Software Tools for Technology Transfer 17(6), 647–657 (2015)

[14] ISO: ISO C Standard 1999 (1999), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, iSO/IEC 9899:1999 draft

[15] Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a Shadow, Expressive as a Tree: Optimized Memory Monitoring for C. Science of Computer Programming pp. 226–246 (Oct 2016)

[16] Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: International Conference on Runtime Verification (RV 2013). LNCS, vol. 8174, pp. 386–399. Springer (Sep 2013)

[17] Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design, chap. 12, pp. 175–188. Springer (Oct 1999)

[18] Pnueli, A.: The temporal logic of programs. In: Symposium on Foundations of Computer Science (FCS'77) (Nov 1977)

[19] Reger, G., Hallé, S., Falcone, Y.: Third international competition on runtime verification. In: Falcone, Y., Sánchez, C. (eds.) International Conference on Runtime Verification (2016)

[20] Reger, G., Havelund, K.: What Is a Trace? A Runtime Verification Perspective. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16) (Oct 2016)

[21] Sutcliffe, G.: The CADE ATP System Competition - CASC. AI Magazine 37(2), 99–101 (2016)