Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

First International Competition on Runtime Verification

Rules, Benchmarks, Tools, and Final Results of CRV 2014

Ezio Bartocci[1], Yliès Falcone[2,3], Borzoo Bonakdarpour[4], Christian Colombo[5], Normann Decker[6], Klaus Havelund[7], Yogi Joshi[8], Felix Klaedtke[9], Reed Milewicz[10], Giles Reger[11], Grigore Rosu[3], Julien Signoles[12], Daniel Thoma[6], Eugen Zalinescu[13], Yi Zhang[3]

[1] Vienna University of Technology, Austria
[2] Univ. Grenoble-Alpes, Inria, CNRS, Laboratoire d'Informatique de Grenoble, F-38000 Grenoble, France
[3] University of Illinois at Urbana-Champaign, USA
[4] McMaster University, Canada
[5] University of Malta, Malta
[6] Lübeck University, Lübeck, Germany
[7] Jet Propulsion Laboratory, California Institute of Technology, USA
[8] University of Waterloo, Canada
[9] NEC Laboratories Europe, Heidelberg, Germany
[10] University of Alabama at Birmingham, USA
[11] University of Manchester, UK
[12] CEA, LIST, Software Security Laboratory, PC 174, 91191 Gif-sur-Yvette, France
[13] ETH Zurich, Switzerland

Received: date / Revised version: date

Abstract. The First International Competition on Runtime Verification (CRV) was held in September 2014, in Toronto, Canada, as a satellite event of the 14th international conference on Runtime Verification (RV'14). The event was organized in three tracks: (1) offline monitoring, (2) online monitoring of C programs, and (3) online monitoring of Java programs. In this paper we report on the phases and rules, a description of the participating teams and their submitted benchmark, the (full) results, as well as the lessons learned from the competition.

1 Introduction

Runtime verification[1] [39,49,35,60,36,6], from here on referred to as RV, refers to a class of lightweight scalable techniques for analysis of execution traces. The core idea is to instrument a program to emit events during its execution, which are then processed by a monitor. This paper focuses specifically on *specification-based* trace analysis, where execution traces are verified against formal specifications written in formal logical systems. Other forms of RV, not treated in this paper, include for example *algorithm-based* trace analysis, such as detecting concurrency issues such as data races and deadlocks; *specification mining* from traces; and *trace visualization*.

Send offprint requests to:

[1] `http://runtime-verification.org`

Specification-based trace analysis is a topic of particular interest due to the many different logics and supporting tools that have been developed over the last decade, including the following to just mention a few [56,54,9,50,57,32, 53,26,29,27,34,5,20,33,8,4,38]. Unlike proof-oriented techniques, such as theorem proving or model checking, that aim to verify exhaustively whether a property is satisfied for all the possible system executions, specification-based RV automatically checks only if a single execution trace is correct, and it therefore does not suffer from the classic manual labor and state-explosion problems, typically associated with theorem proving and model checking. The achieved scalability of course comes at the cost of less coverage.

As illustrated in Fig. 1, an RV process consists of three main steps: *monitor synthesis*, *system instrumentation*, and *monitoring*. In the first step, a monitor is synthesized from a requirement expressed in a formal specification language (e.g., regular expression, automaton, rule set, grammar, or temporal logic formula), or it is programmed directly in a general-purpose programming language [52,36]. A monitor is a program or a device that receives as input a sequence of events (observations) and emits verdicts regarding the satisfaction or violation of the requirement. In the second step, the system is instrumented using event information extracted from requirements. The instrumentation aims at ensuring that the relevant behavior of the system can be observed at runtime. In the third step, the program is executed with the instrumentation activated. In *online monitoring*, the monitor runs in parallel with (or is embedded into) the program, analyzing the event sequence as it is produced. In *offline* monitoring, the event sequence is written to persistent memory, for example a log file, which at a later point in time is analyzed by the monitor. In online monitoring, monitor verdicts can trigger fault protection code. In offline monitoring, verdicts can be summarized and visualized in reports, or trigger the execution of other programs. Instrumentation and monitoring generally increase the memory utilization and introduce a runtime overhead that may alter the timing-related behavior of the system under scrutiny. In real-time applications, overhead control strategies are generally necessary to mitigate the overhead by, for example, using static analysis to minimize instrumentation, or switching on and off the monitor [61,7,46].

During the last decade, many important tools and techniques have been developed. However, due to lack of standard benchmark suites as well as scientific evaluation methods to validate and test new techniques, we believe that the RV community is in pressing need to have an organized venue whose goal is to provide mechanisms for comparing different aspects of existing tools and techniques.

For these reasons, inspired by the success of similar events in other areas of computer-aided verification (e.g., SAT [43], SV-COMP [18], SMT [1], RERS [40,41]), Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone organized the First International Competition on Runtime Verification (CRV 2014) with the aim to foster the process of comparison and evaluation of software runtime verification tools. The objectives of CRV'14 were the following:

– To stimulate the development of new efficient and practical runtime verification tools and the maintenance of the already developed ones.
– To produce benchmark suites for runtime verification tools, by sharing case studies and programs that researchers and developers can use in the future to test and to validate their prototypes.
– To discuss the measures employed for comparing the tools.
– To compare different aspects of the tools running with different benchmarks and evaluating them using different criteria.
– To enhance the visibility of presented tools among different communities (verification, software engineering, distributed computing and cyber security) involved in software monitoring.

CRV'14 was held in September 2014, in Toronto, Canada, as a satellite event of the 14th international conference on Runtime Verification (RV'14). The event was organized in three tracks: (1) offline monitoring, (2) online monitoring of C programs, and (3) online monitoring of Java programs. This paper conveys the experience on the procedures, the rules, the participating teams, the benchmarks, the evaluation process and the results of CRV'14. This paper complements and significantly extends a preliminary report that was written before RV'14 [3].

Paper organization. The rest of this paper is organized as follows: Section 2 gives an overview of the phases and the rules of the competition. Section 3 introduces the participating teams. Section **??** presents the benchmarks used in all the three tracks of the competition. Section 4 defines the method used to compute the score. Section 5 reports on the results. Section **??** discusses lessons learned. Finally, Section 6 concludes the paper.

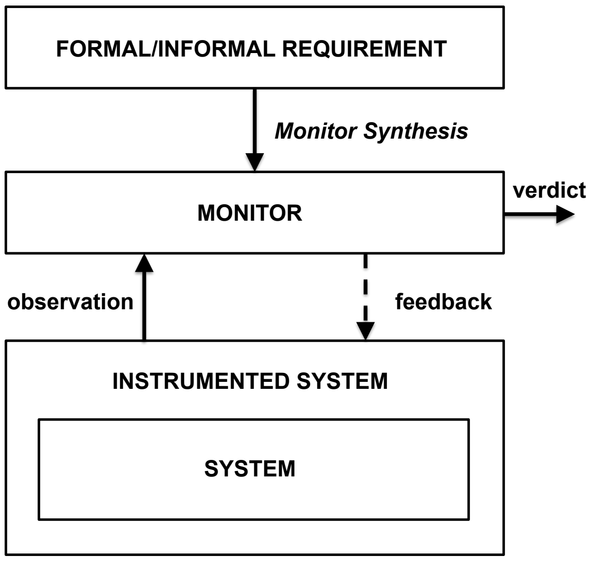

Fig. 1. Runtime verification main phases.

2 Phases and Rules of the Competition

Taking inspiration from the software verification competition (SVCOMP) started in 2012 [17] we have arranged the the overall process along three different phases for each track:

1. *collection of benchmarks* (Section 2.1),
2. *training and monitor submissions* (Section 2.2),
3. *evaluation* (Section 2.3).

The first phase (Dec. 15, 2013 - March 1, 2014) aims to stimulate each team to develop at most five benchmarks per track that may challenge the tools of the other teams. In the second phase (March 2, 2014 - May 30, 2014), the teams have the possibility to further develop and improve their tools using the benchmarks of the adversary teams. This cross-fertilizes new ideas between the teams, since each team is exposed to the same problems and challenges previously faced by the other teams. The goal of the last phase (June 1, 2014 - Sept. 23, 2014) is to provide a framework for a fair and automatic evaluation of the participating tools. In the following we describe the phases in more detail.

2.1 Collection of Benchmarks

In the first phase, the teams participating in each track prepare and upload in a shared repository a set of benchmarks. We now provide a description of the requirements of a benchmark for the online and offline monitoring tracks.

```
an_event_name, a_field_name = a_value, a_field_name = a_value
an_event_name, a_field_name = a_value, a_field_name = a_value
```

Fig. 2. Example of trace in CSV format

```
an_event_name
a_field_name = a_value
a_field_name = a_value

an_event_name
a_field_name = a_value
a_field_name = a_value
```

Fig. 3. Example of trace in custom format

```
<log>
  <event>
   <name>an_event_name</name>
   <field>
     <name>a_field_name</name>
     <value>a_value</value>
   </field>
   <field>
     <name>a_field_name</name>
     <value>a_value</value>
   </field>
 </event>
 <event>
   <name>an_event_name</name>
   <field>
     <name>a_field_name</name>
     <value>a_value</value>
   </field>
   <field>
     <name>a_field_name</name>
     <value>a_value</value>
   </field>
 </event>
</log>
```

Fig. 4. Example of trace in XML format

Online monitoring of C and Java programs tracks. In the case of C and Java tracks, each benchmark contribution is required to contain the following:

– A *program package* containing the program source code (the program to be monitored), a script to compile it, a script to run the executable, and an English description of the functionality of the program.
– A *specification package* containing a collection of files, each describing a property: an English description of the property, a formal representation of it in the logical system supported by the team, instrumentation information, and the expected verdict (the evaluation of the property on the program execution).

The instrumentation information describes of a mapping from concrete events in the program (for example method calls) to the abstract events referred to in the specification. For instance, if one considers the *HasNext* property on Java iterators (that a call of the method next on an iterator should always be preceded by a call of the method hasNext that returns true), the mapping should indicate that the hasNext event in the property refers to a call to the hasNext() method on an Iterator object, and similarly for the next event. Several concrete events can be mapped to the same abstract event.

Offline monitoring track. In the case of offline track, each benchmark contribution is required to contain the following:

– A *trace*, in either CSV, custom, or XML format, and a description of the event kinds contained in the trace. The three trace formats are illustrated in Fig. 2, 3, and 4.
– A *specification package* containing a collection of files describing a property: an English description of the property, a formal representation of it in the logical system supported by the team, and the expected verdict (the evaluation of the property on the trace).

2.2 *Training Phase and Monitor Collection Phase*

During this phase, all participants can apply their tools to all the available benchmarks in the repository, and possibly modify their tools to improve their performance. At the phase end, they submit their contributions as monitors for the benchmarks. A contribution is related one of the benchmarks uploaded in the first phase, and contains a monitor for the property in the benchmark together with two scripts, one for building and one for running the monitor.

2.3 *Benchmark Evaluation Phase*

The evaluation of the teams' contributions is performed on DataMill[2] [55], a distributed infrastructure for computer performance experimentation targeted at scientists that are interested in performance evaluation. DataMill aims to allow the user to easily produce robust and reproducible results at low cost. DataMill executes experiments multiple times, obtaining average values, and generally deploys results from research on how to set up such experiments. Each participant has the possibility to set up and try their tool using DataMill. The final evaluation is performed by the competition organizers.

[2] http://datamill.uwaterloo.ca

3 Participating Teams and Tools

In this section we provide a description of participating teams and tools.

3.1 C Track

Table 1 summarizes the teams and tools participating in the track of online monitoring of C programs. The tools are described in the rest of this subsection.

3.1.1 RiTHM

RiTHM (Runtime Time-triggered Heterogeneous Monitoring) [54] is a tool for runtime verification of *C* programs. RiTHM is developed at the Real-time Embedded Software Group at University of Waterloo, Canada.

RiTHM takes a *C* program and a set of properties expressed in a fragment of first-order LTL as input. RiTHM instruments the *C* program with respect to the definition of predicates supplied along with LTL properties, and it synthesizes an LTL monitor. The program then can be monitored at runtime by the synthesized monitor, where the instrumented program sends events in its execution trace to the monitor. Further, RiTHM monitors a fragment of first order LTL specifications as described in [51]. RiTHM monitor can be run on Graphics Processing Units or multicore Central Processing Units [51] for accelerating the verification of an execution trace [16].

3.1.2 E-ACSL

E-ACSL [32] (Executable ANSI/ISO C Specification Language) is both a formal specification language and a monitoring tool which are designed and developed at CEA LIST, Software Security Labs. They are integrated to the *Frama-C* platform [47], which is an extensible and collaborative platform dedicated to source-code analysis of C software.

The formal specification language is a large subset of the *ACSL* specification language [15] and is designed in a way that each annotation can be verified at runtime [58]. It is a behavioral first-order typed specification language which supports, in particular, function contracts, assertions, user-defined predicates and built-in predicates (such as `\valid(`p`)` which indicates that the pointer p points to a memory location that the program can write and read).

The plug-in E-ACSL [59] automatically converts a *C* program p_1 specified with E-ACSL annotations to another *C* program p_2 which monitors each annotation of p_1 at runtime. More precisely, for each annotation a, p_2 computes the truth value of a and passes it as an argument to the *C* function `e_acsl_assert`. By default, this function stops the program execution with a precise error message if a is 0 (i.e., false) and just continues the execution otherwise. The generated code is linked against a dedicated memory library which can efficiently compute the validity of complex memory-related properties (e.g., use-after-free or initialization of variables) [48, 42].

3.1.3 RTC

RTC [53] (Runtime checking for C programs) is a runtime monitoring tool that instruments unsafe code and monitors the program execution. RTC is built on top of the ROSE compiler infrastructure. RTC finds memory bugs, arithmetic overflows and under-flows, and runtime type violations. Most of the instrumentations are directly added to the source file and only require a minimal runtime system. As a result, the instrumented code remains portable. The team behind RTC consists of researchers from the University of Alabama at Birmingham, North Carolina State University, Lawrence Livermore National Laboratory, and Matlab.

3.2 Java Track

Table 2 summarizes the teams and tools participating in the track of online monitoring of Java programs. The tools are described in the rest of this subsection.

3.2.1 Larva

Larva [26] is a Java and AspectJ-based RV tool whose specification language (DATEs [25]) is a flavour of automata enriched with stopwatches. The automata are symbolic in that they allow the use of local state in terms of Java variables and data structures. Furthermore, Larva allows the full use of Java for the specification of conditions which decide when transitions trigger. Similarly, for each transition, an action can be specified so that when it triggers, the local state can be updated, possibly also carrying out actions on the monitored system, e.g., to handle a detected problem.

The tool design and development has been inspired by case studies in the financial industry [24] where there are frequent soft real-time constraints such as limits on the amount of money spent within a particular period and entity life-cycles such as limiting the kind of operations users are allowed to perform while suspended.

Over the years, a set of tools have been built to support and augment Larva including conversion from other specification languages (such as duration calculus [21]) to Larva specification language, and extensions to support event extraction from databases as well as saving the monitor state to a database when it is not feasible to keep it in memory [23].

3.2.2 jUnit$^{\mathrm{RV}}$

jUnit$^{\mathrm{RV}}$ [29] is a tool extending the unit testing framework jUnit with runtime verification capabilities. Roughly, jUnit$^{\mathrm{RV}}$ provides a new annotation @Monitors listing monitors that are synthesized from temporal specifications. The monitors check whether the currently executed tests satisfy the correctness properties underlying the monitors. As such, jUnit's concept of plain assert-based verification limited to checking properties of single states of a program is extended significantly towards checking properties of complete execution paths.

To support specifications beyond propositional properties jUnit^{RV} uses a generic approach to enhance traditional runtime verification techniques towards first-order theories in order to reason about data. This allows especially for the verification of multi-threaded, object-oriented systems. The framework lifts the monitor synthesis for propositional temporal logics to a temporal logic over structures within some first-order theory. To evaluate such temporal properties, SMT (Satisfiability Modulo Theory) solving and classical monitoring of propositional temporal properties is combined. jUnit^{RV} implements this framework for linear-time temporal logic based on the Z3 SMT solver [28]. The framework is described in detail in [30, 31].

3.2.3 JAVAMOP

JAVAMOP, with its core component RV-Monitor [50], is a formalism-independent RV tool designed to effectively monitor multiple parametric properties simultaneously. It is developed both by University of Illinois at Urbana Champaign and Runtime Verification, Inc.[3].

JAVAMOP specifications support a variety of formalisms such as finite state machine, linear temporal logic, string rewriting systems, etc., which gives users a lot of freedom to express different kinds of properties. At the same time, several optimizations ([22,45,50]) were proposed to make monitors creation, garbage collection, and internal data structure access more efficient. Besides, JAVAMOP can generate a single Java agent out of multiple specifications. The Java agent can be easily attached to the Java virtual machine to run with Java programs. All these efforts make JAVAMOP capable of monitoring multiple properties simultaneously on large Java applications.

3.2.4 Monitoring at Runtime with QEA (MARQ)

The MARQ tool [57] monitors specifications written in the Quantified Event Automata (QEAs) [2] specification language. It has been developed at the University of Manchester by Giles Reger and Helena Cuenca Cruz with input from David Rydeheard.

QEAs combine a quantifier list with an extended finite state machine over parametric events. Trace acceptance is defined via the *trace slicing* approach, extended to allow existential quantification and a notion of free variables.

Syntax of QEA. We give a brief explanation of the syntax used and will not repeat it below. A QEA consists of a quantifier list and a state machine. They can have multiple **Forall** or **Exists** quantifications with an optional **Where** constraint restricting the considered values. States can be **accept** states, indicating that a trace is accepted if *any* path reaches an **accept** state. There are two other state modifiers: **skip** indicates that missing transitions are self-looping; **next** indicates that missing transitions implicitly go to the **failure** state. The failure state is an implicit non-accept state with no outgoing transitions; once the failure state has been reached success (for this binding) is not possible.

The MARQ tool implements an incremental monitoring algorithm for QEAs. A *structural specialisation* module attempts to specialize the algorithm based on structural properties of the specification. Singly-quantified specifications are directly indexed, otherwise a general *symbol-based* indexing approach is used.

For monitoring Java programs, MARQ is designed to be used with AspectJ. It also implements mechanisms for dealing with garbage collection and can either use reference or semantic identity for monitored objects.

3.3 Offline Track

Table 3 summarizes the tools teams and participating in the track of offline monitoring. The tools are described in the rest of this subsection.

3.3.1 RITHM-2

RITHM [54], as previously described, is a tool for runtime verification. In addition to *online* monitoring of *C* programs, it can process execution traces for performing *offline* verification. Further, RITHM was extended to process execution traces in XML and CSV formats as per the schemas described in Section 2.1. RITHM is designed for monitoring specifications described using LTL or a first order fragment of LTL [51].

3.3.2 MONPOLY

MONPOLY [9] is a monitoring tool for checking compliance of IT systems with respect to policies specifying normal or compulsory system behavior. The tool has been developed as part of several research projects on runtime monitoring and enforcement in the Information Security group at ETH Zurich. MONPOLY is open source, written in OCaml.

Policies are given as formulas of an expressive safety fragment of metric first-order temporal logic (MFOTL), including dedicated operators for expressing aggregations on data items. The first-order fragment is well suited for formalizing relations between data items, while the temporal operators are used to express quantitative temporal constraints on the occurrence or non-occurrence of events at different time points. An event streams can be input through a log file or a UNIX pipeline, which MONPOLY processes iteratively, either offline or online. The stream can be seen as a sequence of timestamped databases, each of them consisting of the events that have occurred in the system execution at a point in time. Each tuple in one of the databases' relations represents a system action together with the involved data. For a given event stream and a formula, MONPOLY outputs all the policy violations.

[3] `https://www.runtimeverification.com`

Further details on MFOTL and the tool's underlying monitoring algorithm are given in [11, 14, 12]. MONPOLY has been used in real-world case studies, in collaboration with Nokia Research Lausanne [10] and with Google Zurich [13]. Further performance evaluation and comparison with alternative approaches can be found in [11] and [14].

3.3.3 STEPR

STEPR is a prototype log file analysis tool developed at the Institute for Software Engineering and Programming Languages, University of Lübeck, Germany.[4] It is loosely based on the Lola stream processing verification language proposed by d'Angelo et al. [27]. The log file is considered as an input stream of data and the user can use stream operations to define new streams and combine them in an algebraic fashion. Assertions can be specified on such streams that, once violated, make the program report an error. Streams can further be declared as output streams that are written to report files in various formats and verbosity. They provide additional information on the exact position of the violation and error counts allowing for convenient analysis of the occurred deviations. STEPR is written in the Scala programming language [5] and provides a Scala-internal domain-specific language for specifications. The full power of Scala can be used for specifying further stream operation if needed.

3.3.4 Monitoring at Runtime with QEA (MARQ)

MARQ was previously described in Section 3.2.4 as a tool for monitoring Java programs. Here we give details of how it can be used for offline monitoring.

MARQ can parse trace files in either CSV or XML formats (JSON traces are not supported). The CSV parser has been hand-written to optimise the translation of events into the internal representation. The XML parser makes use of standard Java library features. As a consequence, the XML parser is relatively inefficient compared to the CSV parser. Therefore, we prefer the CSV format and would normally first translate traces into this format.

One can use different events in the specification and the trace when monitoring with MARQ. For example, an abstract event in the specification can have a different name, arity, and parameter order as the corresponding event in the trace. Furthermore, multiple events in the trace can be mapped to an abstract event, and vice-versa. To handle this, MARQ requires the use of so-called *translators* that can translate event names as well as permuting or dropping event parameters. Additionally, translators can be used to *interpret* values i.e., to parse strings into integer objects. Translators are required when a parameter value should be treated as its interpreted value, as is the case with a counter.

3.4 Summary

Table 5 summarizes some of the features of the tools presented in this section. A checkmark sign (✓) indicates a supported feature. Four categories of features are presented.

Input requirement specification. The first category concerns the specification of the input requirement that a tool can monitor. The entry *user-enabled* in Table 5 is ticked when the corresponding tool allows the user to specify the requirement. In this case the tool supports one or more specification languages that allow the user to write flexible requirements to be monitored. The entry *built-in* is ticked when the corresponding tool has a number of built-in specifications that can be checked at runtime without any specification effort by the user. Table 5 list next some of the following common specification language features: *automata-based*, *regular-expressions-based*, and *logic-based*, supporting *logical-time* where only the relative ordering of events is important or *real-time* where the event occurrence times are also relevant. The language can support *propositional events* and/or *parametric events*, depending on whether runtime events cannot carry or, respectively, can carry data values. Generally, more expressive specification languages require more complex monitoring algorithms. The monitoring code can be generated from a high-level specification language or directly implemented in a programming language.

Instrumentation. The entry *own instrumentation* indicates that the tool implements its own instrumentation phase of the RV process. The entry *relies on AspectJ* indicates that the tool uses AspectJ for instrumentation purposes. The entry *relies on another technique* indicates that the tool uses a third-party technique and/or tool different from AspectJ for instrumentation purposes.

Monitored systems. The entries in this category have their expected meaning and indicate the kind of systems that the tool can monitor (C programs, Java programs, or traces).

Monitoring mode. The entry *time triggered* indicates that the stream of observations from the system is obtained through sampling. The entry *event triggered* indicates that the steam of observations is obtained following the execution of events in the system.

4 Evaluation - Calculating Scores

In this section, we present in detail the algorithm to calculate the final score for each tool. Consider one of the three competition tracks (C, Java, and Offline). Let N be the number of teams/tools participating in the considered track and L be the total number of benchmarks provided by all teams. The

[4] www.isp.uni-luebeck.de
[5] www.scala-lang.org

Tool	Ref.	Contact person	Affiliation
RiTHM	[54]	B. Bonakdarpour	McMaster Univ. and U. Waterloo, Canada
E-ACSL	[32]	J. Signoles	CEA LIST, France
RTC	[53]	R. Milewicz	University of Alabama at Birmingham, USA

Table 1. Tools participating in online monitoring of C programs track.

Tool	Ref.	Contact person	Affiliation
Larva	[26]	C. Colombo	University of Malta, Malta
jUnit^{RV}	[29,30]	D. Thoma	ISP, University of Lübeck, Germany
JavaMop	[44]	G. Roşu	U. of Illinois at Urbana Champaign, USA
QEA,MarQ	[2]	G. Reger	University of Manchester, UK

Table 2. Tools participating in online monitoring of Java programs track.

Tool	Ref.	Contact person	Affiliation
RiTHM2	[54]	B. Bonakdarpour	McMaster Univ. and U. Waterloo, Canada
MonPoly	[9]	E. Zălinescu	ETH Zurich, Switzerland
STePr		N. Decker	ISP, University of Lübeck, Germany
QEA, MarQ	[2]	G. Reger	University of Manchester, UK

Table 3. Tools participating in the offline monitoring track.

maximal number of experiments for the track is $N \times L$. That is, each team has the possibility to compete on a benchmark. Then, for each tool T_i ($1 \leq i \leq N$) w.r.t. each benchmark B_j ($1 \leq j \leq L$), we assign three different scores:

- the correctness score $C_{i,j}$,
- the overhead score $O_{i,j}$, and
- the memory utilization score $M_{i,j}$.

In case of online monitoring (Java and C tracks), let E_j be the execution time of benchmark B_j (without monitor). Note, in the following, to simplicity notation, we assume that all participants of a track want to compete on benchmark B_j. Participants can of course decide not to qualify on a benchmark of their track. In this case, the following score definitions can be adapted easily.

Several considerations influenced the scoring principles:

- Since several benchmarks are provided in each track, we wanted to provide participants with the possibility to compete on a benchmark or not. We allocated a maximum number of points that could be gained on a benchmark. In our opinion, it limited the influence of the failure or success on a benchmark and rewarded the overall behavior of tools on the benchmarks in a track.
- We gave an important emphasis on the correctness of monitoring verdicts. As such, the scoring mechanism gives more priority to correctness of verdicts in that performance is evaluated on a benchmark only when a tool provides the correct verdict and negative points are assigned on a benchmark when a tool produces a false verdict or crashes.
- Within a benchmark, scores are assigned to participants/-tools based on how better they perform compared to each other. Moreover, the proportion of points in benchmark assigned to a tool depends on a performance ratio comparing to the average performance of other tools. The average performance of other tools is computed with the geometric mean (because we dealt with normalised numbers [37]).

4.1 Correctness Score

The correctness score $C_{i,j}$ for a tool T_i running a benchmark B_j is (an integer) calculated as follows:

- $C_{i,j} = 0$, if the property associated with benchmark B_j cannot be expressed in the specification language of T_i.
- $C_{i,j} = -10$, if in case of online monitoring, the property can be expressed, but the monitored program crashes.
- $C_{i,j} = -5$, if, in case of online monitoring, the property can be expressed and no verdict is reported after $10 \times E_j$.
- $C_{i,j} = -5$, if, in case of offline monitoring, the property can be expressed, but the monitor crashes.
- $C_{i,j} = -5$, if the property can be expressed, the tool does not crash, and the verification verdict is incorrect.
- $C_{i,j} = 10$, if the tool does not crash, it allows to express the property of interest, and it provides the correct verification verdict.

Note that, in case of a negative correctness score, there is no evaluation w.r.t. the overhead and memory-utilization scores for the pair (T_i, B_j).

4.2 Overhead Score

The overhead score $O_{i,j}$, for a tool T_i running benchmark B_j, is related to the timing performance of the tool for detecting

Tool	Available at (URL)
E-ACSL (ver. 0.4.1)	`http://frama-c.com/download/e-acsl/e-acsl-0.4.1.tar.gz`
JAVAMOP (VER. 4.2)	`http://fsl.cs.illinois.edu/index.php/JavaMOP4`
JUNIT^{RV}	`https://www.isp.uni-luebeck.de/junitrv`
LARVA	`http://www.cs.um.edu.mt/svrg/Tools/LARVA/`
MONPOLY	`http://sourceforge.net/projects/monpoly`
QEA(MARQ)	`https://github.com/selig/qea`
RITHM/RITHM2	`https://uwaterloo.ca/embedded-software-group/projects/rithm`
RTC	`https://github.com/rose-compiler/rose/tree/master/projects/RTC`
STEPR	`http://www.isp.uni-luebeck.de/stepr`

Table 4. URLs where it is possible to download the tools participating to the competition.

	User-enabled	Built-in	Propositional Events	Parametric Events	Automata-based	Logic-based	Regular Expressions-based	Logical-time	Real-time	Own instrumentation	Relies on AspectJ	Relies on another technique	C programs	Java programs	Traces	Time triggered	Event triggered
Participating Tool	Input Requirement Specification									Instrumentation			Monitored Systems			Monitoring Mode	
RITHM	✓		✓	✓		✓		✓		✓		✓	✓		✓	✓	✓
E-ACSL	✓	✓	✓	✓		✓				✓			✓				✓
RTC		✓								✓			✓				✓
LARVA	✓		✓	✓							✓			✓		✓	✓
JUNIT^{RV}	✓		✓	✓	✓				✓	✓		✓		✓			✓
JAVAMOP	✓		✓	✓	✓	✓	✓	✓			✓			✓			✓
MONPOLY	✓		✓	✓		✓		✓	✓						✓		✓
STEPR	✓		✓	✓											✓		✓
MARQ	✓		✓	✓	✓			✓			✓			✓	✓		✓

Table 5. Summary of features of the tools.

the (unique) verdict. For all benchmarks, a fixed total number of points O is allocated when evaluating the tools on a benchmark. Thus, the scoring method for overhead ensures that

$$\sum_{i=1}^{N}\sum_{j=1}^{L} O_{i,j} = O.$$

The overhead score is calculated as follows. First, we compute the *overhead index* $o_{i,j}$, for tool T_i running a benchmark B_j, where the larger the overhead index is, the better.

– In the case of offline monitoring, for the overhead, we consider the elapsed time till the property under scrutiny is either found to be satisfied or violated. If monitoring (with tool T_i) of the trace of benchmark B_j executes in time V_i, then we define the overhead as

$$o_{i,j} = \begin{cases} \dfrac{1}{V_i} & \text{if } C_{i,j} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

– In the case of online monitoring (C or Java), the overhead associated with monitoring is a measure of how much longer a program takes to execute due to runtime monitoring. If the monitored program (with monitor from tool T_i) executes in $V_{i,j}$ time units, we define the overhead index as

$$o_{i,j} = \begin{cases} \dfrac{\sqrt[N]{\prod_{l=1}^{N} V_{l,j}}}{V_{i,j}} & \text{if } C_{i,j} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

In other words, the overhead index for tool T_i evaluated on benchmark B_j is the *geometric mean* of the overheads of the monitored programs with all tools over the overhead of the monitored program with tool T_i.

Then, the overhead score $O_{i,j}$ for a tool T_i w.r.t. benchmark B_j is defined as follows:

$$O_{i,j} = O \times \frac{o_{i,j}}{\sum_{l=1}^{N} o_{l,j}}.$$

For each tool, the overhead score is a harmonization of the overhead index so that the sum of overhead scores is equal to O.

4.3 Memory-Utilization Score

The memory-utilization score $M_{i,j}$ is calculated similarly to the overhead score. For all benchmarks, a fixed total number of points O is allocated when evaluating the tools on a benchmark. Thus the scoring method for memory utilization ensures that:

$$\sum_{i=1}^{N}\sum_{j=1}^{L} M_{i,j} = M.$$

First, we measure the memory utilization index $m_{i,j}$ for tool T_i running a benchmark B_j, where the larger memory utilization index, the better.

- In the case of offline monitoring, we consider the maximum memory allocated during the tool execution. If monitoring (with tool T_i) of the trace of benchmark B_j uses a quantity of memory D_i, then we define the overhead as:

$$m_{i,j} = \begin{cases} \dfrac{1}{D_i} & \text{if } C_{i,j} > 0, \\ 0 & \text{otherwise,} \end{cases}$$

  That is, the memory utilization index for tool T_i evaluated on benchmark B_j is the geometric mean of the memory utilizations of the monitored programs with all tools over the memory utilization of the monitored program with tool T_i.
- In the case of online monitoring (C or Java tracks), memory utilization associated with monitoring is a measure of the extra memory the monitored program needs (due to runtime monitoring). If the monitored program uses D_i, we define the memory utilization as

$$m_{i,j} = \begin{cases} \dfrac{\sqrt[N]{\prod_{l=1}^{N} D_{l,j}}}{D_{i,j}} & \text{if } C_{i,j} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the memory utilization score $M_{i,j}$ for a tool T_i w.r.t. a benchmark B_j is defined as follows:

$$M_{i,j} = M \times \frac{m_{i,j}}{\sum_{l=1}^{N} m_{l,j}}.$$

4.4 Final Score

The final score F_i for tool T_i is then computed as follows:

$$F_i = \sum_{j=1}^{L} S_{i,j}$$

where:

$$S_{i,j} = \begin{cases} C_{i,j} & \text{if } C_{i,j} \le 0, \\ C_{i,j} + O_{i,j} + M_{i,j} & \text{otherwise.} \end{cases}$$

For the results reported in the next section, we set $O = C = M = 10$, giving the same weight to the correctness, overhead, and memory-utilization scores.

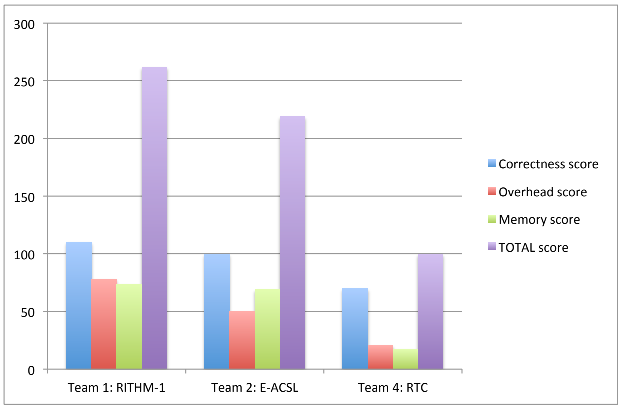

Fig. 5. Graphical representation of the scores for the C track.

5 Results

In this section, we report on the results of the participants. The raw experimental data and the scripts submitted by participants can be obtained by cloning the repository available at:

`https://gitlab.inria.fr/crv14/evaluation`.

For each track, we present the scores obtained in each category and the final scores achieved by each team, as defined in Section 4. In the following tables, teams are ranked according to their total scores.

Let us recall that the experiments were conducted on DataMill [55]. The selected machine was **queen**, which has an Intel(R) Core(TM) i7-2600K CPU @ 3.40 GHz (x86_64 architecture with 8 cores), 7.72 GB of DDR3, and is running on a Gentoo Linux distribution. We have considered the Wall-clock time for our measures. Using DataMill guarantees that each tool had the same execution environment and it was the only running software during each experiment. Tools were allowed to leverage the eight available cores.

5.1 Scores for the C Track

The detailed scores for the C track are presented in Table 6. The final scores of the C track are reported in Table 7 and can be visualized in Fig. 5. The final ranking of the teams is: first is RiTHM, second is E-ACSL, third is RTC.

As one can observe in Table 7, RiTHM made the difference over E-ACSL on the overhead score; whereas RiTHM and E-ACSL have approximately the same correctness and memory-utilization scores. Moreover, there is an important gap between the two first tools in this track (RiTHM and E-ACSL) and RTC. Possible explanations for this discrepancy are discussed in Section **??**.

5.2 Scores for the Java Track

The detailed scores for the Java track are presented in Table 8. The final scores of the Java track are reported in Table 9 and

	RiTHM			**E-ACSL**			**RTC**		
Reference to Benchmark Description	**verdict**	**mem** (MB)	**ovhd** (s)	**verdict**	**mem** (MB)	**ovhd** (s)	**verdict**	**mem** (MB)	**ovhd** (s)
	v score	**m score**	**o score**	**v score**	**m score**	**o score**	**v score**	**m score**	**o score**
Section **??**	F	1 012 756	0.68	N/A	N/A	N/A	N/A	N/A	N/A
	10	10	10	0	0	0	0	0	0
Section **??**	F	1 012 756	0.68	N/A	N/A	N/A	N/A	N/A	N/A
	10	10	10	0	0	0	0	0	0
Section **??**	F	614 168	0.42	N/A	N/A	N/A	N/A	N/A	N/A
	10	10	10	0	0	0	0	0	0
Section **??**	F	614 168	0.42	N/A	N/A	N/A	N/A	N/A	N/A
	10	10	10	0	0	0	0	0	0
Section **??**	F	647 696	0.69	N/A	N/A	N/A	N/A	N/A	N/A
	10	10	10	0	0	0	0	0	0
Section **??**	F	11 916	0.01	F	12 980	0.30	F	37 040	0.19
	10	4.46	9.59	10	4.10	0.16	10	1.44	0.14
Section **??**	F	5388	0.001	F	4320	4.87	F	5984	0.01
	10	3.18	9.09	10	3.96	0	10	2.86	0.29
Section **??**	F	4236	0.01	F	4628	2.66	F	5856	0.19
	10	3.79	9.52	10	3.47	0.03	10	2.74	0.27
Section **??**	F	4212	N/A	F	4312	N/A	F	5792	0.01
	10	3.70	0	10	3.61	0	10	2.69	10
Section **??**	F	4212	N/A	N/A	N/A	N/A	F	1344	N/A
	10	2.42	0	0	0	0	10	7.58	0
Section **??**	F	4216	N/A	F	6804	N/A	F	N/A	0.23
	10	6.17	0	10	3.83	0	10	0	10

Table 6. Detailed scores for the C track.

Rank	Team Name	Correctness Score	Overhead Score	Memory Score	**TOTAL SCORE**
1	RITHM-1	110	78.19	73.72	**261.92**
2	E-ACSL	100	50.19	68.97	**219.16**
3	RTC	70	20.70	17.31	**108.01**

Table 7. Scores for the C track.

can be visualized in Fig. 6.

As one can observe in Table 9, the scores between the two first highest scores are really close, we call it a draw between QEA and JAVA-MOP. Thus, the final ranking of the teams is: firsts are QEA and JAVA-MOP, second is JUNIT^{RV}, third is LARVA. While there is a draw between QEA and JAVA-MOP, one can notice that QEA did slightly better on the memory-utilization score while JAVA-MOP did slightly better on the overhead score. While the scores of the tools do not differ much in terms of correctness, the rankings are due to first the overhead score and then the memory score.

5.3 Scores for the Offline Track

The detailed scores for the Offline track are presented in Table 10. The final scores of the offline track are reported in Table 11 and can be visualized in Fig. 7. The final ranking of the teams is: first is QEA, second is MONPOLY, third is RITHM, fourth is STEPR.

As one can observe in Table 11, there is not much difference in terms of correctness score between the three first tools. There is however a noticeable difference between each of the three first tools in terms of global score. One can also notice that the difference between QEA and MONPOLY was made on the overhead score.

	Larva			**JUnit^{RV}**			**Java-MOP**			**QEA**		
Reference to Benchmark Description	**verdict**	**mem** (MB)	**ovhd** (s)	**verdict**	**mem** (MB)	**ovhd** (s)	**verdict**	**mem** (MB)	**ovhd** (s)	**verdict**	**mem** (MB)	**ovhd** (s)
	v score	**m score**	**o score**	**v score**	**m score**	**o score**	**v score**	**m score**	**o score**	**v score**	**m score**	**o score**
Section **??**	F	0.55	1.54	F	1.92	6.08	F	1.94	0.17	F	2.65	0.20
	10	1.95	2.66	10	2.70	0.14	10	2.68	4.96	10	1.96	4.35
Section **??**	F	0.58	1.56	F	7.75	6.86	F	2.59	0.21	F	2.70	0.18
	10	2.60	3.03	10	1.02	0.13	10	3.04	4.33	10	2.92	4.96
Section **??**	F	0.66	1.56	F	1.92	3.74	F	9.03	0.22	F	5.24	0.24
	10	4.54	2.11	10	4.99	0.28	10	1.06	4.66	10	1.83	4.40
Section **??**	F	0.69	1.56	F	1.92	8.63	F	9.04	0.32	F	2.00	0.18
	10	9.05	0.89	10	4.20	0.12	10	0.89	3.35	10	4.02	5.84
Section **??**	F	0.95	1.57	F	1.92	8.57	F	9.69	0.73	F	2.64	0.22
	10	10.97	0.83	10	4.76	0.17	10	0.94	2.05	10	3.46	6.83
Section **??**	T	0.59	1.57	T	27.94	1.56	T	1.94	0.20	T	3.30	0.23
	10	5.16	1.85	10	0.34	0.59	10	4.92	4.70	10	2.89	4.11
Section **??**	T	0.65	1.58	T	308.67	22.93	T	1.94	0.20	T	2.65	0.25
	10	7.10	1.36	10	0.03	0.04	10	4.97	5.19	10	3.64	4.12
Section **??**	T	0.05	2173.26	T	244.27	49.94	T	32.24	26.27	T	5.85	25.99
	10	162.39	0.29	10	0.19	2.06	10	1.46	3.92	10	8.06	3.97
Section **??**	T	0.64	1.55	T	291.57	24.79	T	1.94	0.20	T	4.59	0.23
	10	5.16	2.08	10	0.04	0.04	10	5.55	4.98	10	2.34	4.34
Section **??**	T	1.13	1.56	T	680.19	100.57	T	2.58	0.24	T	110.33	1.26
	10	7.76	2.45	10	0.03	0.02	10	7.35	7.45	10	0.17	1.40
Section **??**	F	0.10	48.06	F	N/A	2.98	F	5.17	0.79	F	4.53	2.04
	10	40.93	0.56	10	0	1.59	10	4.41	5.99	10	5.04	2.32
Section **??**	N/A	N/A	N/A	F	N/A	0.51	F	5.81	2.23	F	8.41	3.24
	0	0	0	10	0	7.21	10	5.91	1.65	10	4.09	1.14
Section **??**	F	0.10	35.78	F	N/A	0.36	F	7.10	25.22	F	5.20	25.33
	10	3.87	4.37	10	0	9.63	10	2.38	0.14	10	3.25	0.14
Section **??**	F	0.56	1.57	F	N/A	1.61	F	2.58	0.18	F	3.23	0.22
	10	2.58	3.57	10	0	0.55	10	3.57	4.94	10	2.86	3.95
Section **??**	F	0.03	2606.58	F	N/A	7.58	F	647.19	87.00	F	837.29	190.89
	10	841.28	3.03	10	0	8.85	10	3.93	0.77	10	3.04	0.35
Section **??**	F	0.06	15 393.22	T	N/A	N/A	F	1001.69	164.00	F	829.59	217.27
	10	721.39	3.86	-5	0	0	10	2.78	5.66	10	3.36	4.28
Section **??**	T/O	0	N/A	T	717.92	88.35	T	801.15	242.00	T	844.54	288.08
	-5	N/A	0	10	3.64	5.98	10	3.26	2.18	10	3.10	1.83
Section **??**	T/O	0	N/A	T	697.42	113.63	T	795.49	237.00	T	819.92	889.63
	-5	N/A	0	10	3.67	6.22	10	3.21	2.98	10	3.12	0.79
Section **??**	T/O	0	N/A	T	N/A	N/A	F	649.83	94.00	F	820.52	170.31
	-5	N/A	0	-5	0	0	10	5.58	6.44	10	4.42	3.56
Section **??**	F	0.16	27.22	F	N/A	3.68	F	58.27	3.16	F	23.07	0.62
	10	135.98	1.08	10	0	1.22	10	2.53	1.42	10	6.39	7.20
Section **??**	T	0.29	70.12	T	86.04	8.8	T	267.45	5.64	T	142.74	5.29
	10	160.62	2.18	10	4.06	2.30	10	1.31	3.59	10	2.45	3.82
Section **??**	T	0	6882.58	T	300.2	49.9	T	309.00	6.08	T	156.66	5.59
	10	267.43	2.24	10	2.00	0.55	10	1.94	4.53	10	3.82	4.92
Section **??**	N/A	N/A	N/A	F	N/A	2.92	F	39.87	1.58	F	28.71	0.72
	0	0	0	10	0	1.45	10	4.19	2.67	10	5.81	5.88

Table 8. Detailed scores for the Java track.

Rank	Team Name	Correctness Score	Overhead Score	Memory Score	**TOTAL SCORE**
1	QEA	230	84.50	82.01	**396.51**
1	JavaMOP	230	88.56	77.89	**396.45**
2	jUnit^{RV}	200	49.15	31.67	**280.82**
3	Larva	165	7.79	38.43	**211.22**

Table 9. Scores for the Java track.

	RiTHM			**MonPoly**			**StePr**			**QEA**		
Reference to	**verdict**	**mem**	**ovhd**	**verdict**	**mem**	**ovhd**	**verdict**	**mem**	**ovhd**	**verdict**	**mem**	**ovhd**
benchmark		(MB)	(s)		(MB)	(s)		(MB)	(s)		(MB)	(s)
description	**v-score**	**m-score**	**o-score**	**v-score**	**m-score**	**o-score**	**v-score**	**m-score**	**o-score**	**v-score**	**m-score**	**o-score**
Section **??**	F	993	0.60	F	13	0.13	F	29.53	0.90	F	4.535	0.24
	10	0.03	1.12	10	2.31	5.32	10	1.02	0.75	10	6.64	2.81
Section **??**	F	993	0.60	F	1228	8.40	F	645.17	8.87	F	33.30	3.58
	10	0.30	7.67	10	0.24	0.55	10	0.46	0.52	10	8.99	1.28
Section **??**	F	614	0.98	F	13	0.12	F	31.26	0.91	F	4.53	0.19
	10	0.05	0.63	10	2.32	5.41	10	0.97	0.68	10	6.66	3.28
Section **??**	F	614	0.98	F	1696	15.80	F	622.30	21.10	F	517.07	12.22
	10	2.83	8.41	10	1.02	0.52	10	2.79	0.39	10	3.36	0.68
Section **??**	F	628	0.99	F	544	5.09	F	274.29	7.77	F	32.94	3.71
	10	0.43	6.29	10	0.49	1.24	10	0.97	0.80	10	8.11	1.68
Section **??**	N/A	N/A	N/A	F	36.00	5.95	F	645.43	41.73	F	5.19	0.26
	0	0	0	10	1.25	0.41	10	0.07	0.06	10	8.68	9.53
Section **??**	N/A	N/A	N/A	F	20	1.33	F	255.48	96.00	F	4.53	0.25
	0	0	0	10	1.82	1.56	10	0.14	0.02	10	8.04	8.42
Section **??**	N/A	N/A	N/A	F	370	33.51	F	706.26	21.41	F	7.75	0.29
	0	0	0	10	0.20	0.08	10	0.11	0.13	10	9.69	9.79
Section **??**	N/A	N/A	N/A	F	73.00	1.53	F	457.34	3.67	F	552.08	2.58
	0	0	0	10	7.74	4.98	10	1.24	2.07	10	1.02	2.95
Section **??**	N/A	N/A	N/A	F	16.00	330.80	F	721.22	947.00	F	5935.90	1537.30
	0	0	0	10	9.76	6.39	10	0.22	2.23	10	0.03	1.38
Section **??**	T	14.27	5.40	T	13.00	5.05	T	634.99	10.84	T	127.62	4.51
	10	4.48	2.65	10	4.92	2.84	10	0.10	1.32	10	0.50	3.18
Section **??**	F	14.27	0.90	F	13.00	0.80	F	333.01	2.93	F	30.33	0.80
	10	3.83	2.80	10	4.20	3.17	10	0.16	0.86	10	1.80	3.17
Section **??**	F	14.27	7.20	F	13.00	1.04	F	501.91	3.20	F	30.86	1.05
	10	3.86	0.59	10	4.24	4.08	10	0.11	1.32	10	1.79	4.01
Section **??**	F	14.28	2.39	F	13.00	2.0	F	173.37	2.91	F	30.33	0.63
	10	3.77	1.46	10	4.14	1.75	10	0.31	1.20	10	1.78	5.59
Section **??**	T	15	0.04	T	17.00	353.00	T	112.56	2.20	T	29.73	0.59
	10	3.97	9.15	10	3.50	0	10	0.53	0.18	10	2.00	0.67
Section **??**	F	75.00	40.93	F	13.00	432.00	F	631.58	8.46	F	250.30	2.03
	10	1.39	0.38	10	8.03	0.036	10	0.17	1.85	10	0.42	7.73
Section **??**	F	14.04	0.16	F	24.00	3.06	F	36.01	1.19	F	295.52	1.36
	10	4.94	7.67	10	2.89	0.40	10	1.93	1.03	10	0.23	0.90
Section **??**	F	39.79	5.18	F	2675.00	3405.00	F	622.66	9.96	F	234.78	2.04
	10	8.01	2.47	10	0.12	0.375	10	0.51	1.28	10	1.36	6.25
Section **??**	T	14.00	5.14	T	13.00	26.21	T	494.39	9.17	T	239.60	3.54
	10	4.62	3.12	10	4.98	0.61	10	0.13	1.75	10	0.27	4.53

Table 10. Detailed scores for the Offline track.

Rank	Team Name	Correctness Score	Overhead Score	Memory Score	**TOTAL SCORE**
1	QEA	190	77.79	71.36	**339.15**
2	MonPoly	190	39.35	64.19	**293.54**
3	RiTHM-2	140	54.40	42.52	**236.91**
4	StePr	190	18.46	11.93	**220.40**

Table 11. Scores for the Offline track.

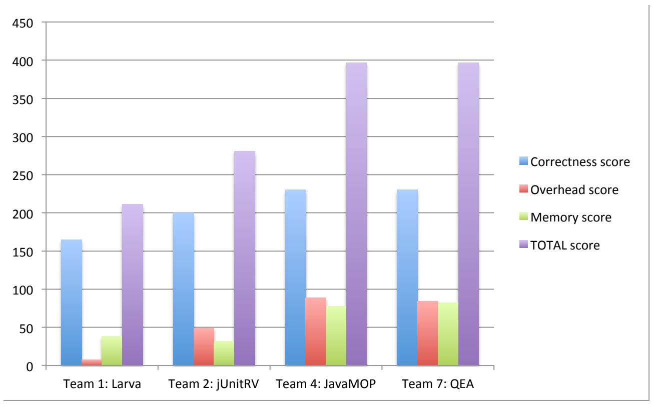

Fig. 6. Graphical representation of the scores for the Java track.

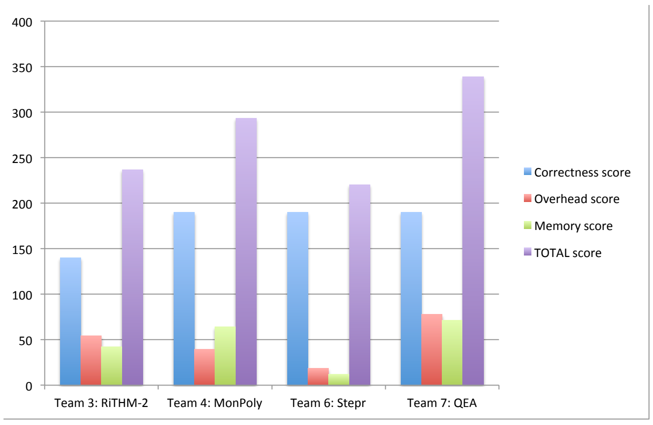

Fig. 7. Graphical representation of the scores for the Offline track.

6 Conclusions

This paper presents the final results of the first international competition on runtime verification. A preliminary presentation of the results have been reported during the RV 2014 conference in Toronto, Canada. This paper provides a comprehensive overview of the teams and their tools, the submitted programs, traces, and specifications, the method used to compute the scores, and the final results for each of the tracks.

We expect this report to help the runtime verification community in several ways. First, this report shall assist the future organizers of the competition to build on the efforts made to organize CRV 2014. Second, the report can also be seen as an entry point to several benchmarks containing non-trivial programs and properties. This shall help developers of tools to assess and experiment with their tools.

Acknowledgements

The competition organizers, E. Bartocci, B. Bonakdarpour, and Y. Falcone, are grateful to many people. The competition organizers would like to warmly thank all participants for their hard work, the members of the runtime verification community who encouraged them to initiate this work, the Laboratoire d'Informatique de Grenoble and its IT team for its support, Inria and its GitLab framework, and finally the DataMill team for providing us with such a nice experimentation platform to run all benchmarks.

All the authors acknowledge the support of the ICT COST Action IC1402 Runtime Verification beyond Monitoring (ARVI). Ezio Bartocci acknowledges also the partial support of the Austrian FFG project HARMONIA (nr. 845631) and the Austrian National Research Network (nr. S 11405-N23) SHiNE funded by the Austrian Science Fund (FWF). The research performed by Klaus Havelund was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

The authors are grateful to the insightful reviewers who helped improving the quality of this paper.

References

1. Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, pages 1–35, 2012.
2. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *Proc. of FM 2012: the 18th International Symposium on Formal Methods*, volume 7436 of *LNCS*, pages 68–84. Springer, 2012.
3. Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In Bonakdarpour and Smolka [19], pages 1–9.
4. Ezio Bartocci, Luca Bortolussi, and Laura Nenzi. A temporal logic approach to modular design of synthetic biological circuits. In *Proc. of CMSB 2013: the 11th International Conferenceon Computational Methods in Systems Biology*, volume 8130 of *LNCS*, pages 164–177. Springer, 2013.

5. Ezio Bartocci, Luca Bortolussi, and Guido Sanguinetti. Data-driven statistical learning of temporal logic properties. In *Proc. of FORMATS 2014: the 12th International Conference on Formal Modeling and Analysis of Timed Systems*, volume 8711 of *LNCS*, pages 23–37, 2014.
6. Ezio Bartocci and Yliès Falcone. Runtime verification and enforcement, the (industrial) application perspective (track introduction). In *Proc. ISoLA 2016: the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications, Part II*, volume 9953 of *LNCS*, pages 333–338, 2016.
7. Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Eretz Zadok, and Justin Seyster. Adaptive runtime verification. In *Proc. of RV 2012: the 3rd International Conference on Runtime Verification*, volume 7687 of *LNCS*, pages 168–182. Springer, 2012.
8. Ezio Bartocci and Pietro Liò. Computational modeling, formal analysis, and tools for systems biology. *PLoS Computational Biology*, 12(1), 2016.
9. David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. MONPOLY: Monitoring usage-control policies. In *Proc. of RV 2011: the 2nd Internat. Conference on Runtime Verification*, volume 7186 of *LNCS*, pages 360–364. Springer, 2012.
10. David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monitoring data usage in distributed systems. *IEEE Transactions on Software Engineering*, 39(10):1403–1426, 2013.
11. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.
12. David Basin, Felix Klaedtke, and Eugen Zălinescu. Greedily computing associative aggregations on sliding windows. *Information Processing Letters*, 115(2):186–192, 2015.
13. David A. Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. *Formal Methods in System Design*, 49(1-2):75–108, 2016.
14. David A. Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.
15. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language. Version 1.8*, March 2014.
16. Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. GPU-based runtime verification. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, pages 1025–1036, 2013.
17. Dirk Beyer. Competition on software verification - (SV-COMP). In *Proc. of TACAS 2012: the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference*, volume 7214 of *LNCS*, pages 504–524. Springer, 2012.
18. Dirk Beyer. Status report on software verification - (competition summary SV-COMP 2014). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2014.
19. Borzoo Bonakdarpour and Scott A. Smolka, editors. *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*, volume 8734 of *Lecture Notes in Computer Science*. Springer, 2014.
20. Sara Bufo, Ezio Bartocci, Guido Sanguinetti, Massimo Borelli, Umberto Lucangelo, and Luca Bortolussi. Temporal logic based monitoring of assisted ventilation in intensive care patients. In B. Steffen and T. Margaria, editors, *Proc. of ISoLA 2014: 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 8803 of *LNCS*, pages 391–403, 2014.
21. Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.
22. Feng Chen, Patrick Meredith, Dongyun Jin, and Grigore Rosu. Efficient formalism-independent monitoring of parametric properties. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, pages 383–394, 2009.
23. Christian Colombo, Gordon Pace, and Patrick Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
24. Christian Colombo and Gordon J. Pace. Fast-forward runtime monitoring - an industrial case study. In *Runtime Verification, Third International Conference, RV 2012*, volume 7687 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2012.
25. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2008.
26. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society.
27. B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *Proceedings of TIME 2005: the 12th International Symposium on Temporal Representation and Reasoning*, pages 166–174, 2005.
28. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
29. Normann Decker, Martin Leucker, and Daniel Thoma. jUnit$^{\text{RV}}$—adding runtime verification to jUnit. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 459–464. Springer, 2013.
30. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2014.
31. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2015.

32. Mickaël Delahaye, Nikolai Kosmatov, and Julien Signoles. Common Specification Language for Static and Dynamic Analysis of C Programs. In *Proceedings of SAC '13: the 28th Annual ACM Symposium on Applied Computing*, pages 1230–1235. ACM, March 2013.
33. Alexandre Donzé, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott A. Smolka. On Temporal Logic and Signal Processing. In Supratik Chakraborty and Madhavan Mukund, editors, *Proc. of ATVA 2012: 10th International Symposium on Automated Technology for Verification and Analysis, Thiruvananthapuram, India, October 3-6*, volume 7561 of *Lecture Notes in Computer Science*, pages 92–106. Springer-Verlag, 2012.
34. Yliès Falcone. You should better enforce than verify. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Proceedings of the 1st international conference on Runtime verification (RV 2010)*, volume 6418 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 2010.
35. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. In *9th International Workshop on Runtime Verification. Selected Papers*, volume 5779, pages 40–59, 2009.
36. Yliès Falcone, Klaus Havelund, and Giles Reger. A tutorial on runtime verification. In Manfred Broy, Doron Peled, and Georg Kalus, editors, *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
37. Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.
38. Ebru Aydin Gol, Ezio Bartocci, and Calin Belta. A formal methods approach to pattern synthesis in reaction diffusion systems. In *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*, pages 108–113. IEEE, 2014.
39. Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer, 2005.
40. Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, and Dirk Beyer. The RERS grey-box challenge 2012: Analysis of event-condition-action systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 608–614. Springer, 2012.
41. Falk Howar, Malte Isberner, Maik Merten, Bernhard Steffen, Dirk Beyer, and Corina S. Pasareanu. Rigorous examination of reactive systems - the RERS challenges 2012 and 2013. *STTT*, 16(5):457–464, 2014.
42. Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Fast as a shadow, expressive as a tree: hybrid memory monitoring for C. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1765–1772. ACM, 2015.
43. Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.
44. D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *Proceedings of ICSE 2012: THE 34th International Conference on Software Engineering, Zurich, Switzerland, June 2-9*, pages 1427–1430. IEEE Press, 2012.
45. Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Roşu. Garbage collection for monitoring parametric properties. In *Programming Language Design and Implementation (PLDI'11)*, pages 415–424. ACM, 2011.
46. Keanan Kalajdzic, Ezio Bartocci, Scott A. Smolka, Scott Stoller, and G. Grosu. Runtime Verification with Particle Filtering. In *Proc. of RV 2013, the fourth International Conference on Runtime Verification, INRIA Rennes, France, 24-27 September, 2013*, volume 8174 of *Lecture Notes in Computer Science*, pages 149–166. Springer, 2013.
47. Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. *Formal Aspects of Computing*, pages 1–37, January 2015.
48. Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *International Conference on Runtime Verification (RV'13)*, volume 8174 of *LNCS*, pages 167–182. Springer, September 2013.
49. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
50. Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian-Florin Serbanuta, and Grigore Rosu. Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In Bonakdarpour and Smolka [19], pages 285–300.
51. Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Accelerated runtime verification of LTL specifications with counting semantics. *CoRR*, abs/1411.2239, 2014.
52. Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
53. Reed Milewicz, Rajesh Vanka, James Tuck, Daniel Quinlan, and Peter Pirkelbauer. Runtime checking c programs. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2107–2114. ACM, 2015.
54. S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. RiTHM: a tool for enabling time-triggered runtime verification for c programs. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 603–606, 2013.
55. Augusto Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: Rigorous performance evaluation made easy. In *Proc. of ICPE 2013: the 4th ACM/SPEC International Conference on Performance Engineering*, pages 137–149. ACM, 2013.
56. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.
57. Giles Reger, Helena Cuenca Cruz, and David Rydeheard. Marq: monitoring at runtime with qea. In *Proceedings of the 21st*

International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), 2015.

58. Julien Signoles. *E-ACSL: Executable ANSI/ISO C Specification Language, version 1.5-4*, March 2014. `frama-c.com/download/e-acsl/e-acsl.pdf`.
59. Julien Signoles. *E-ACSL User Manual*, March 2014. `http://frama-c.com/download/e-acsl/e-acsl-manual.pdf`.
60. Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.
61. Scott D. Stoller, Ezio Bartocci, Justing Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Eretz Zadok. Runtime Verification with State Estimation. In *Proc. of RV 2011, the Second international conference on Runtime verification, San Francisco, CA, USA*, volume 7186 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 2011.