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, NASA, 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] [51,66,45,82,46,7], 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*.

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 [74,71,10,67,75,41, 70,34,38,36,44,6,28,42,9,5,50]. 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 [69,46]. 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

Send offprint requests to:

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

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 [83, 8, 62].

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 [59], SV-COMP [20], SMT [2], RERS [53, 54]), 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 [4].

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 4 presents the benchmarks used in all the three tracks of the competition. Section 5 defines the method used to compute the score. Section 6 reports on the results. Section 7 discusses lessons learned. Finally, Section 8 concludes the paper.

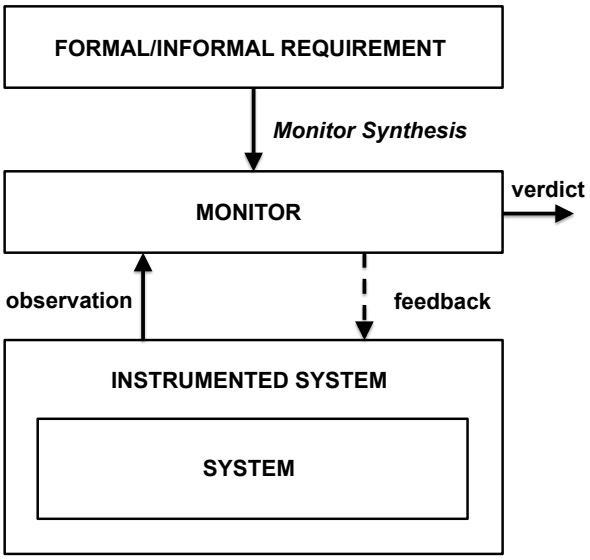

Fig. 1. Runtime verification main phases.

2 Phases and Rules of the Competition

Taking inspiration from the software verification competition (SVCOMP) started in 2012 [19] 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] [73], 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) [71] 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 [68]. RITHM monitor can be run on Graphics Processing Units or multicore Central Processing Units [68] for accelerating the verification of an execution trace [18].

3.1.2 E-ACSL

E-ACSL [41] (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 [63], 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 [16] and is designed in a way that each annotation can be verified at runtime [80]. 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 [81] 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) [64, 56].

3.1.3 RTC

RTC [70] (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 [34] is a Java and AspectJ-based RV tool whose specification language (DATEs [33]) 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 [32] 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 [29]) 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 [31].

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

jUnit$^{\mathrm{RV}}$ [38] 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 [37]. The framework is described in detail in [39, 40].

3.2.3 JAVAMOP

JAVAMOP, with its core component RV-Monitor [67], 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 ([30,61,67]) 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 [75] monitors specifications written in the Quantified Event Automata (QEAs) [3] 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 [71], 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 [68].

3.3.2 MONPOLY

MONPOLY [10] 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 [12, 15, 13]. MONPOLY has been used in real-world case studies, in collaboration with Nokia Research Lausanne [11] and with Google Zurich [14]. Further performance evaluation and comparison with alternative approaches can be found in [12] and [15].

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. [36]. 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 Benchmarks for the Monitoring Competition

In this section, we provide a description of the benchmarks provided by participants.

The benchmarks can be dowloaded by cloning the repository and following the instructions available at:

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

[4] `www.isp.uni-luebeck.de`

[5] `www.scala-lang.org`

Tool	Ref.	Contact person	Affiliation
RITHM	[71]	B. Bonakdarpour	McMaster Univ. and U. Waterloo, Canada
E-ACSL	[41]	J. Signoles	CEA LIST, France
RTC	[70]	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	[34]	C. Colombo	University of Malta, Malta
JUNIT^{RV}	[38,39]	D. Thoma	ISP, University of Lübeck, Germany
JAVAMOP	[60]	G. Roşu	U. of Illinois at Urbana Champaign, USA
QEA,MARQ	[3]	G. Reger	University of Manchester, UK

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

Tool	Ref.	Contact person	Affiliation
RITHM2	[71]	B. Bonakdarpour	McMaster Univ. and U. Waterloo, Canada
MONPOLY	[10]	E. Zălinescu	ETH Zurich, Switzerland
STEPR		N. Decker	ISP, University of Lübeck, Germany
QEA, MARQ	[3]	G. Reger	University of Manchester, UK

Table 3. Tools participating in the offline monitoring track.

In the following, for each benchmark, we describe the related program and property.

4.1 C Track

4.1.1 Maximum Chunk Size in Dropbox Connections

This benchmark is provided by RiTHM team.

Description of the monitored program. The program simulates Dropbox connections. The program uses the dataset described in [43] to run the simulation.

Description of the property. The property states that for all *connections*, it is *always* the case that *chunk size* (used to split files) is less than or equal to 999 999. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall connection : \mathrm{G}\left(chunksize\left(connection\right) \leq 999\,999\right).$$

4.1.2 Changes in the Chunk Size of Dropbox Connections

This benchmark is provided by RiTHM team.

Description of the monitored program. The benchmark uses the same program as the one in the benchmark described in Section 4.1.1.

Description of the property. The property states that for all *connections*, it is always the case that when the *chunk size* becomes strictly larger than 10 000, its value eventually becomes less than or equal to 10 000. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall connection : \mathrm{G}(chunksize\left(connection\right) > 10\,000 \implies \mathrm{F}\ chunksize(connection) \leq 10\,000).$$

4.1.3 Maximum Bandwidth of Youtube Connections

Description of the monitored program. The program simulates Youtube connections. The program uses the dataset described in [85] to run the simulation.

Description of the property. The property states that for all connections, it is *always* the case that the bandwidth is less than or equal to 100 000. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall connection : \mathrm{G}\left(bandwidth(connection) \leq 100\,000\right).$$

4.1.4 Changes in the Bandwidth of Youtube Connections

Description of the monitored program. This benchmark uses the same program as the one in Section 4.1.3

Description of the property. The property states that, for all connections, it is always the case that when the *bandwidth* is strictly larger than 10 000, it *eventually* becomes less than or equal to 10 000 The bandwidth is a parameter of the execution trace. It is calculated by using *size_in_bytes* attribute which provides the number of bytes transferred, and *duration* which provides the time in seconds for the transfer. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall connection : \mathrm{G}\left(bandwidth(connection) > 10\,000 \implies \mathrm{F}\ bandwidth(connection) \leq 10\,000\right).$$

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.

	Input Requirement Specification									Instrumentation			Monitored Systems			Monitoring Mode	
Participating Tool	**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**
RITHM	✓		✓	✓		✓		✓		✓		✓	✓		✓	✓	✓
E-ACSL	✓	✓	✓	✓		✓				✓			✓				✓
RTC		✓								✓			✓				✓
LARVA	✓		✓	✓							✓			✓		✓	✓
JUNIT^{RV}	✓		✓	✓	✓				✓	✓		✓		✓			✓
JAVAMOP	✓		✓	✓	✓	✓	✓	✓			✓			✓			✓
MONPOLY	✓		✓	✓		✓		✓	✓						✓		✓
STEPR	✓		✓	✓											✓		✓
MARQ	✓		✓	✓	✓			✓			✓			✓	✓		✓

Table 5. Summary of features of the tools.

4.1.5 Allowed Operations on Files and Sockets

Description of the monitored program. The program simulates I/O operations by multiple processes on files and sockets. The output of this program is similar to that of running the `strace` utility in Linux to produce a system call trace for a set of processes.

Description of the property. The property states that for all the processes and all the files, it is always the case that if a file (resp. a socket) is opened then it is eventually closed by the process. The property is formalized using a fragment of first-order LTL [68] as the conjunction of the two following formulae:

$$\forall process, \forall file : \\ \mathrm{G}\,(open\,(process, file) \implies \mathrm{F}\,close\,(process, file))$$

$$\forall process, \forall socket : \\ \mathrm{G}\,(accept\,(process, socket) \implies \mathrm{F}\,close\,(process, socket))$$

4.1.6 Binary Search

This benchmark is provided by the E-ACSL team.

Description of the monitored program. This benchmark consists in monitoring the standard binary search function defined below. It searches some `key` in a sorted array `a` of a given `length`.

```
int binary_search
  (int* a, int length, int key)
{
  int low = 0, high = length - 1;
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (a[mid] == key) return mid;
    if (a[mid] < key) low = mid + 1;
    else high = mid - 1;
  }
```

```
  return -1;
}
```

The `main` function of the program calls this function 3 times on an array of 2 000 000 elements in order to:

- search an existing key and check that the return index is correct;
- search an unknown key and check that the function returns `-1`;
- search an existing well-chosen key in a wrongly sorted array (1 element is misplaced) and check that the function incorrectly returns `-1`.

Description of the property. The monitored program must verify the following function contract of `binary_search`:

- it takes as input a positive `length` and a fully-allocated sorted array of at least `length` elements;
- it returns either an index `idx` such that `a[idx] == key`; or `-1` if there is no such index.

In the formal specification language E-ACSL [41] based on behavioral first-order logic, this specification may be described by the following function contract:

```
/*@ requires \valid(a+(0..length-1));
    requires \forall integer i;
      0 <= i < length-1 ==> a[i] <= a[i+1];
    requires length >=0;

    behavior exists:
      assumes \exists integer i;
        0 <= i < length && a[i] == key;
      ensures 0 <= \result < length
      ensures a[\result] == key;

    behavior not_exists:
      assumes \forall integer i;
        0 <= i < length ==> a[i] != key;
      ensures \result == -1;
*/
int binary_search
  (int* a, int length, int key);
```

The first `requires` clause states that each cell of the array must be correctly allocated, the second one states that the array must be sorted and the third one indicates that the length must be positive. Then, the first behavior says that if the searched `key` exists in the array, the result of the function must be an array index corresponding to this key, while the second behavior says that the function returns `-1` if there is no such index.

E-ACSL reports that the second requirement is violated when calling this function on the wrongly sorted array. Here is the result of the execution:

```
Precondition failed at line 18 in
function binary_search.
The failing predicate is:
\forall integer i;
  0 <= i < length-1 ==> *(a+i) <= *(a+(i+1)).
```

4.1.7 Merging Arrays

This benchmark is provided by the E-ACSL team.

Description of the monitored program. This benchmark provides two different implementations of a merging algorithm which merges two sorted arrays into a third one in a way that the resulting array is also sorted. The first implementation is assumed to be correct and is provided below, while the second one introduces an error by removing the marked instruction.

```
void merge
  (int *t1,int *t2,int *t3,int l1,int l2)
{
  int i = 0, j = 0, k = 0 ;
  while (i < l1 && j < l2) {
    if (t1[i] < t2[j]) {
      t3[k] = t1[i];
      i++;
    } else {
      t3[k] = t2[j];
      j++;
    }
    k++;
  }
  while (i < l1) {
    t3[k] = t1[i];
    i++;
    k++; // removed instruction
  }
  while (j < l2) {
    t3[k] = t2[j];
    j++;
    k++;
  }
}
```

The `main` function of the program calls both functions with one array of 6000 elements and another one of 4000 elements.

Description of the properties. The monitored program must check that each call to both functions:

- takes as input two positive lengths `l1` and `l2` and two sorted arrays `t1` and `t2` of length `l1` and `l2` respectively;
- modifies `t3` such that it is a sorted array of length `l1+l2` where each element belongs to either `t1` or `t2`. Reciprocally, each element of `t1` and `t2` must belong to `t3`.

In the formal specification language E-ACSL [41] based on behavioral first-order logic, this specification may be described by the following function contract:

```
/*@ requires l1 >= 0;
    requires l2 >= 0;
    requires \forall integer i; 0<=i<l1-1
      ==> t1[i] <= t1[i+1];
    requires \forall integer i; 0<=i<l2-1
      ==> t2[i] <= t2[i+1];
    ensures \forall integer i; 0<=i<l1+l2-1
```

```
      ==> t3[i] <= t3[i+1];
    ensures \forall integer i; 0<=i<l1
      ==> \exists integer j; 0<=j<l1+l2
          && t1[i] == t3[j];
    ensures \forall integer i; 0<=i<l2
      ==> \exists integer j; 0<=j<l1+l2
          && t2[i] == t3[j];
    ensures \forall integer i; 0<=i<l1+l2
      ==> ((\exists integer j; 0<=j<l1
            && t3[i] == t1[j])
          || \exists integer j; 0<=j<l2
            && t3[i] == t2[j]);
*/
void merge
  (int *t1,int *t2,int *t3,int l1,int l2);
```

The two first requirements indicate that the lengths `l1` and `l2` of the input arrays `t1` and `t2` must be positive, while the two other requirements say that these arrays must be sorted. If the requirements of the function are satisfied, it must ensure 4 postcondition provided by the `ensures` clause. The first one indicates that the output array `t3` must be sorted. The second (resp. third) postconditions indicates that each element of `t1` (resp. `t2`) must also belong to `t2` while the last postcondition states the reverse condition which is that each element of `t3` belongs to either `t1` or `t2`.

E-ACSL reports that the first postcondition is violated when calling the incorrect implementation. It states that the output array is sorted. Indeed, because of the missing instruction, the end of the output array contains (unsorted) garbage. Here is the result of the execution.

```
Postcondition failed at line 59 in
function wrong_merge.
The failing predicate is:
\forall integer i;
  0 <= i < (\old(l1)+\old(l2))-1
  ==> *(\old(t3)+i) <= *(\old(t3)+(i+1)).
```

4.1.8 Quicksort

This benchmark is provided by the E-ACSL team.

Description of the monitored program This benchmark provides an implementation of the standard quicksort algorithm, which sorts a given `array` between two indexes `left` and `right`:

```
void quicksort
  (int *array, int left, int right)
 {
  if(left < right) {
    int idx = (left+right)/2;
    int new_idx =
      partition(array, left, right, idx);
    quicksort(array, left, new_idx-1);
    quicksort(array, new_idx+1, right);
  }
}
```

This implementation uses two helper functions `partition` and `swap`. Only the first one is given below. The second one which swaps the contents of two array cells is straightforward.

```
int partition
  (int *array,int left,int right,int idx)
{
  int val = array[idx], store = left, i;
  swap(array, idx, right);
  for(i = left; i < right; i++) {
    if(array[i] <= val) {
      swap(array, i, store);
      store++;
    }
  }
  swap(array, store, right);
  return store;
}
```

The `main` function of the program contains 2 calls to the `quicksort` function on an unsorted array of 10 000 elements. The first call is a correct one, but the second call gives 10 001 as the right bound instead of 10 000 (at most).

Description of the properties The only property that must be checked by this benchmark is that `quicksort` is called on a fully-allocated array up to the given `length`.

In the formal specification language E-ACSL [41] based on behavioral first order logic, this specification may be described by the following function contract which formally expresses, that each cell of the array between `left` and `right` must be valid (non-null and points to a memory location that the program is allowed to write).

```
/*@ requires \forall integer j;
      left <= j <= right
      ==> \valid(array+j); */
void quicksort
  (int* array, int left, int right);
```

E-ACSL reports that this property is violated on the second function call since the function tries to access to the invalid index 10 001 of the array. Here is the result of the execution.

```
Precondition failed at line 35 in
function quicksort.
The failing predicate is:
\forall integer j;
  left <= j <= right ==> \valid(array+j).
```

4.1.9 Accesses to Arrays without Off-by-one nor Out-of-bounds

This benchmark is provided by the RTC team.

Description of the monitored program. The program is a test case from the Juliet test suite [23], a collection of test cases in the C/C++ language produced by the NIST. This program is a minimal example of a violation of the property described below. It does an array access from a statically allocated array of size 1024 at index 1024.

Description of the property. The property states that there should not be off-by-one errors (see item 193 in the Common Weakness Enumeration[6] list) nor out-of-bounds read (see item 125 in the Common Weakness Enumeration[7] list) on a global array.

4.1.10 Absence of Buffer Overflow in a Palindrome Generator

This benchmark is provided by the RTC team.

Description of the monitored program. The program finds the next highest number that is a palindrome after a number provided as input. The program has been modified by adding a heap-based buffer overflow when reading the input number. The number is written with an unbounded string copy to the heap allocated buffer 'k' of size MAX_LEN (300). The program contains a call to strcpy() that copies a single character from the input string to the buffer, and this line is reached multiple times as the program scans over the input string. The program is a test case of the STONESOUP test suite [17], a collection of test cases in the C and Java languages produced by the NIST.

Description of the property. The property states the absence of buffer overflow. The vulnerability could allow an attacker to execute code by writing past the buffer and overwriting function pointers that exist in memory (see item 122 in the Common Weakness Enumeration[8] list).

4.1.11 Absence of Negative Pointers in Function Calls

This benchmark is provided by the RTC team.

Description of the monitored program. The program attempts to call a standard library function, strcat(), with a negative pointer index. The program is a test case from the Juliet test suite [23], a collection of test cases in the C/C++ language produced by the NIST. The program contains an array access with a negative pointer index in the arguments to strcat().

Description of the property. The property states that there should not be pointers referring to negative values passed as arguments to function.

4.2 Java Track

4.2.1 Gold Users of the Financial Transaction System

The benchmark is provided by the LARVA team. The LARVA team provided five benchmarks (described also in Sections 4.2.2, 4.2.3, 4.2.4, and 4.2.5). Common to all the benchmarks is that the considered properties concern a financial transaction system described in the following.

Description of the monitored program. FiTS (Financial Transaction System) is a cut-down version of a financial transaction system aimed at providing basic functionality to ensure that the focus is on the verification techniques and not on understanding the underlying system. Based on FiTS, a number of properties inspired from real-life case studies are specified. FiTS is a barebone system mocking the behaviour of a financial transaction system. It emulates a virtual banking system in which users may open accounts from which they may perform money transfers. The system has two types of users: administrators and clients. The former have more rights than normal clients enabling them to perform certain actions such as approving the opening of an account, enabling a new user and registering new users. A client can invoke a number of actions to access their (money) accounts which they have registered on FiTS — actions which they may invoke through an online interface. Information about each registered client is stored in a database, including information such as the client's name and country of origin, and client classification information.

Under FiTS, each client may be associated with a number of money accounts belonging to him. Clients may request the creation of a new account or close down one of their accounts at any point in time. To access accounts, an existing client may open a login session on FiTS to make transfers or manage their money accounts. A user may have multiple sessions open at the same time.

The following properties are specified in a guarded-command language (GCL) as a series of Java-based rules of the form *event* | *condition* → *action*.

Description of the property. The property states that only users based in Argentina can be Gold users. The property can be expressed as follows:

```
*.makeGoldUser(..) target (UserInfo u)
| !(u.getCountry().equals("Argentina"))
  → Verification.fail("P1 violated");
```

Informally, upon giving the status of Gold user to a client, it must be ensured that he or she is from Argentina. Thus, any trace containing a call to makeGoldUser on a user whose country is not Argentina violates the property and vice-versa.

4.2.2 Initialization in the Financial Transaction System

The benchmark is provided by the LARVA team.

Description of the monitored program. The program of this benchmark is described in Section 4.2.1.

Description of the property. The property states that the transaction system must be initialized before any user logs in. The property can be expressed as follows:

```
*.initialise(..) | → Verification.initialized=true;
UserInfo.openSession(..) | !Verification.initialized
  → Verification.fail("P2 violated");
```

[6] https://cwe.mitre.org/data/definitions/193.html
[7] https://cwe.mitre.org/data/definitions/125.html
[8] https://cwe.mitre.org/data/definitions/122.html

Informally, set a *initialized* flag to true when the system is initialized and check this flag upon the start of any user session. Hence a trace containing a call to *openSession* before *initialize* violates the property while any other trace satisfies it.

4.2.3 Negative Balance in the Financial Transaction System

The benchmark is provided by the LARVA team.

Description of the monitored program. The program of this benchmark is described in Section 4.2.1.

Description of the property. The property states that no account may end up with a negative balance after being accessed. The property can be expressed as follows:

*.withdraw(..) target (UserAccount a) | a.getBalance() $< 0 \rightarrow$
Verification.fail("P3 violated");
*.deposit(..) target (UserAccount a) | a.getBalance() $< 0 \rightarrow$
Verification.fail("P3 violated");

Informally, the rules check the balance of the account after a withdraw or deposit actions to ensure that it is always positive. A trace containing a withdraw or deposit on an account with subsequent negative balance violates the property. Otherwise, the trace satisfies the property.

4.2.4 Unique Account in the Financial Transaction System

The benchmark is provided by the LARVA team.

Description of the monitored program. The program of this benchmark is described in Section 4.2.1.

Description of the property. The property states that an account approved by the administrator may not have the same account number as any other already existing account in the system. The property can be expressed as follows:

*.approveOpenAccount(Integer uid, String acc_number) | $\rightarrow$
Verification.approvedAccounts.add(acc_number);
*.approveOpenAccount(Integer uid, String acc_number)
| Verification.approvedAccounts.contains(acc_number)
$\rightarrow$ Verification.fail("P4 violated");

Informally, a hashmap is kept for all account numbers and each number assigned to a new account is check for membership in this hashmap. A trace containing two calls to *approveOpenAccount* with the same account number violates the property.

4.2.5 Reactivation in the Financial Transaction System

The benchmark is provided by the LARVA team.

Description of the monitored program. The program of this benchmark is described in Section 4.2.1.

Description of the property. The property states that once a user is disabled, he or she may not withdraw from an account until activated again. The property can be specified as follows:

UserInfo.makeDisabled(..) target (UserInfo u)
| $\rightarrow$ Verification.disabledUsers.add(u);
UserInfo.makeActive(..) target (UserInfo u)
| $\rightarrow$ Verification.disabledUsers.remove(u);
UserInfo.withdrawFrom(..) target (UserInfo u)
| (Verification.disabledUsers.contains(u))
$\rightarrow$ Verification.fail("P5 violated");

Informally, the rules keeps track of disabled users by adding the user to a list upon being disabled and removing the user upon activating. Subsequently, upon a withdraw the third rule ensure that the user is not in the list. Thus, a trace containing a call to *makeDisabled* followed by a withdraw on the same user, violates the property. On the contrary, a user performing a withdraw after being disabled but later activated, satisfies the property.

4.2.6 Incrementing a Counter

This benchmark is provided by the JUNIT^{RV} team.

The JUNIT^{RV} team provided five benchmarks (described also in Sections 4.2.7, 4.2.8, 4.2.9, and 4.2.10). Common to all the benchmarks is that the considered programs are simple Java programs that generate random method calls that in turn generate events for the property of the benchmark. Moreover, the properties are given in linear temporal logic over first-order formulae as defined in [40]. Free variables are assumed to be universally quantified at the outermost position. First-order symbols either refer to some fixed theory or to the current system observation (called observation symbols). The benchmark properties only use the theories of IDs, linear equations over integers and linear equations over reals.

In this benchmark, the property holds on the program.

Description of the monitored program. The aim of this benchmark is to test the ability to monitor properties involving counting. The program simply calls a method step(int counter) repeatedly. With each call the value for counter increases by one.

Description of the property. The required property is that the parameter counter of method step has to increase by one with each call. Formally:

$$\forall x \ \mathrm{G}(\mathsf{counter} = x \implies \overline{\mathrm{X}}(\mathsf{counter} = x + 1))$$

Here, x is a globally quantified variable and counter is an integer-valued observation symbol. The observed event is the call to method step and the observation symbol counter is a constant referring to the current value of the corresponding parameter of step.

Example traces. A positive example would be the sequence $\{\mathsf{counter} \mapsto 1\}, \{\mathsf{counter} \mapsto 2\}, \{\mathsf{counter} \mapsto 3\}$, a negative example the sequence $\{\mathsf{counter} \mapsto 1\}, \{\mathsf{counter} \mapsto 2\}, \{\mathsf{counter} \mapsto 1\}$.

4.2.7 Request and Response

This benchmark is provided by the JUNIT^{RV} team. This benchmark uses the common setting where certain service are requested and providers have to respond to such requests. In this benchmark, the property holds on the program.

Description of the monitored program. The program calls a method request (int service) indicating a service with the given id was requested and respond(int service, int provider) to indicate, that a provider responded to a request for the given service.

Description of the property. The required property is that whenever a service is requested, eventually there is a response for that request from some provider. Formally:

$$\forall s \forall p \ \mathrm{G}((\mathsf{request} \wedge \mathsf{service} = s) \implies \mathrm{X}\,\mathrm{F}\,\exists p\,(\mathsf{respond} \wedge \mathsf{service} = s \wedge \mathsf{provider} = p))$$

The observed events are the calls to methods request and respond. The observation symbol service refers to the service id of the current event. The observation symbol provider refers to the provider id of the current event.

Example traces. A positive example would be the sequence

$$\{\mathsf{request}, \mathsf{service} \mapsto 1, \mathsf{provider} \mapsto 2\},$$
$$\{\mathsf{request}, \mathsf{service} \mapsto 3, \mathsf{provider} \mapsto 4\},$$
$$\{\mathsf{respond}, \mathsf{service} \mapsto 1, \mathsf{provider} \mapsto 2\},$$
$$\{\mathsf{respond}, \mathsf{service} \mapsto 3, \mathsf{provider} \mapsto 4\},$$

a negative example the sequence

$$\{\mathsf{request}, \mathsf{service} \mapsto 1, \mathsf{provider} \mapsto 2\},$$
$$\{\mathsf{request}, \mathsf{service} \mapsto 3, \mathsf{provider} \mapsto 4\},$$
$$\{\mathsf{respond}, \mathsf{service} \mapsto 3, \mathsf{provider} \mapsto 4\}.$$

4.2.8 Locking Critical Resources

This benchmark is provided by the JUNIT^{RV} team. For this benchmark the setting of a critical action requiring locking is considered. In this benchmark, the property holds on the program.

Description of the monitored program. The program runs multiple threads in parallel performing some critical action. The critical action must only be performed by one thread at a time. Thus, the threads have to call the method boolean lock() and obtain the return value true before calling action(). The lock is released by calling unlock().

Description of the property. The property consists of two parts: A thread has to call lock (returning true) to acquire the lock before it may call aaction and a call to lock only returns true, if no thread is currently holding the lock. Formally:

$$\forall i \ \mathrm{G}(((\mathsf{run} \vee \mathsf{unlock}) \wedge \mathsf{id} = i) \Rightarrow \neg(\mathsf{action} \wedge \mathsf{id} = i)\,\overline{\mathrm{U}}(\mathsf{lockTrue} \wedge \mathsf{id} = i)) \wedge \mathrm{G}((\mathsf{lock} \wedge \mathsf{id} = i \Rightarrow \mathrm{X}(\neg \mathsf{lockTrue}\,\mathrm{U}\,\mathsf{unlock} \wedge \mathsf{id} = i)))$$

The observed events are the calls to method run() (the main method of a thread), unlock and action and the return of unlock. The symbol lockTrue not only indicates that lock returned, but also that the return value was true. The observation symbol id refers to the the id of the thread performing the current event.

Example traces. A positive example would be the sequence $\{\mathsf{lock}, \mathsf{id} \mapsto 1\}, \{\mathsf{action}, \mathsf{id} \mapsto 1\}, \{\mathsf{unlock}, \mathsf{id} \mapsto 1\}$, a negative example the sequence $\{\mathsf{lock}, \mathsf{id} \mapsto 1\}, \{\mathsf{action}, \mathsf{id} \mapsto 2\}, \{\mathsf{unlock}, \mathsf{id} \mapsto 1\}$.

4.2.9 Velocity of an Object

This benchmark is provided by the JUNIT^{RV} team. This benchmark pertains the ability to express constraints over real numbers. In this benchmark, the property holds on the program.

Description of the monitored program. The program simulates an object moving with changing velocity. The object's position is observed in discrete steps. A call to the method step(double pos, double time) indicates such an observation.

Description of the property. The required property is that the average speed between two observations (i.e., calls to step) must never exceed the maximal velocity 10:

$$\forall s \forall t \ \mathrm{G}((\mathsf{time} = t \wedge \mathsf{pos} = s) \Rightarrow \overline{\mathrm{X}}(\mathsf{pos} - s < 10 \cdot \mathsf{time} - t)))$$

The observed event is the call to method step. The observation symbol time refers to the time parameter of method step. The observation symbol pos refers to the time parameter of method step.

Example traces. A positive example would be the sequence

$$\{\mathsf{time} \mapsto 0.0, \mathsf{pos} \mapsto 0.0\},$$
$$\{\mathsf{time} \mapsto 1.0, \mathsf{pos} \mapsto 5.0\},$$
$$\{\mathsf{time} \mapsto 2.0, \mathsf{pos} \mapsto 10.0\},$$

a negative example the sequence

$$\{\mathsf{time} \mapsto 0.0, \mathsf{pos} \mapsto 0.0\},$$
$$\{\mathsf{time} \mapsto 1.0, \mathsf{pos} \mapsto 50.0\},$$
$$\{\mathsf{time} \mapsto 2.0, \mathsf{pos} \mapsto 100.0\}.$$

4.2.10 Non-crossing Routes

This benchmark is provided by the JUNIT^{RV} team. The aim of this benchmark is to demonstrate the expressiveness of linear temporal logic combined with first-order constraints. It therefore requires a property for which it would be rather difficult to implement a monitor manually. The setting is inspired by the idea of agents in a multi agent system. The agents request a route from one point to another. Blocked routes must not cross.

Description of the monitored program. The program simulates such a scenario by repeatedly blocking and freeing routes. Blocking a route is indicated by a call to block(Route route). Freeing a route is indicated by a call to free(Route route). A route is a simple line segment described by a starting and end point. The mentioned property can be formulated as: If a point p belongs to a route that is being blocked, no other route that contains p can be blocked as long as p has not been freed again. (The point p is freed by freeing a route that contains p.) Formally:

$$\begin{aligned}\forall p\ \mathrm{G}((&\mathsf{block} \wedge \mathsf{onRoute}(p)\\ &\Rightarrow \overline{\mathrm{X}}(\neg(\mathsf{block} \wedge \mathsf{onRoute}(p))\\ &\quad \overline{\mathrm{U}}(\mathsf{free} \wedge \mathsf{onRoute}(p))))\end{aligned}$$

The observed events are the calls to methods block and free. Let q and r be the start and end points, respectively, given as arguments to the called method. The only observation symbol is $\mathsf{onRoute}(p)$ which indicates whether a point p is on the route between q and r. This predicate can be expressed as an equation $\mathsf{onRoute}(p) := \exists_c (p = q + c \cdot (r - q) \wedge 0 \leq c \leq 1)$.

Example traces. A positive example would be the sequence

$$\{\mathsf{block}, \mathsf{onRoute} \mapsto \{p \mid \exists_{c|0\leq c\leq 1}(p = (1,2) + c \cdot (2,2)\}\},$$
$$\{\mathsf{free}, \mathsf{onRoute} \mapsto \{p \mid \exists_{c|0\leq c\leq 1}(p = (1,2) + c \cdot (1,1)\}\},$$

a negative example the sequence

$$\{\mathsf{block}, \mathsf{onRoute} \mapsto \{p \mid \exists_{c|0\leq c\leq 1}(p = (1,2) + c \cdot (2,2)\}\},$$
$$\{\mathsf{block}, \mathsf{onRoute} \mapsto \{p \mid \exists_{c|0\leq c\leq 1}(p = (2,2) + c \cdot (-1,1)\}\}.$$

4.2.11 HasNext on DaCapo Avrora

This benchmark is provided by the JAVA-MOP team. The benchmark considers the (classical) **HasNext** property on one of the so-called DaCapo [22] benchmark. In this benchmark, the property does not hold on the program.

Description of the monitored program. The considered program is the DaCapo Avrora benchmark, which simulates a number of programs running on a grid of AVR microcontrollers. The program is slightly modified to violate the **HasNext** property intentionally.

Description of the Property. This property requires that the hasNext() method is called and returns true before calling next() method for each Iterator object i. It may raise a false positive because one may safely call method next() multiple times after retrieving the actual number of elements. The following LTL describes the **HasNext** property.

$$\forall i\, \mathrm{G}(next(i) \implies \mathrm{Y}\, hasnext\_true(i))$$

In the formula, X^{-1} means previous. Event $next(i)$ corresponds to a call to the method next() on iterator i and event $hasnext\_true(i)$ corresponds to a call to the hasnext() method on the same iterator that returns true.

4.2.12 Safe Usage of Locks

This benchmark is provided by the JAVA-MOP team. In this benchmark, the property does not hold on the program.

Description of the monitored program. One considers a simple program with 6 threads. Give of the threads release the same number of times as they acquire a lock. However, one thread releases one time less than it acquires. Consequently, the program does not terminate.

Description of the property. The property requires a thread to **release** as many times as it **acquires** a lock. Otherwise, it may cause deadlock and the program may not terminate. The following CFG formalises the property.

```
S -> S begin(t) S end(t)
   | S acquire(t,l) S release(t,l)
   | epsilon
```

Events $begin(t)$ and $end(t)$ map to the start and the end of the execution of a thread `t`, respectively. Events $acquire(t, l)$ and `release(t,l)` map respectively to a call to methods lock() and unlock() on lock `l` in a thread `t`.

4.2.13 Calling Methods the Same Number of Times

This benchmark is provided by the JAVA-MOP team. In this benchmark, the property does not hold on the program.

Description of the monitored program. One considers a simple program which calls some dummy methods A(), B(), and C() a certain number of times. These methods are not called the same number of times.

Description of the property. The property states that methods A(), B(), and C() should be called equal times. The following SRS (String Rewriting System) formalizes the property.

```
b a -> a b .
c a -> a c .
c b -> b c .
a b -> E .
E b -> b E .
E a -> a E .
E c -> epsilon .
c E -> epsilon .
^ done -> #succeed .
a done -> #fail .
b done -> #fail.
c done -> #fail.
```

Events `a`, `b` and `c` map to the calls to the methods A(), B(), and C(), respectively. The event `done` corresponds to the end of a program.

4.2.14 Safe Usage of Maps with Iterators

This benchmark is provided by the JAVA-MOP team. In this benchmark, the property does not hold on the program.

Description of the monitored program. The program has 3 iterators for 2 collections which are created from 2 different maps. The usage of one iterator in this program creates an unsafe map iterator.

Description of the property. This property matches the case where a collection created from a map is modified while it is iterated. The following extended regular expression describes the property.

```
createColl.update*.createIter
   .useIter*.update+.useIter
```

The events map to the following pointcuts in a program:

```
event createColl after(Map map)
      returning(Collection c) :
      (call(* Map.values())
      || call(* Map.keySet()))
      && target(map) {}

event createIter after(Collection c)
      returning(Iterator i) :
      call(* Collection.iterator())
      && target(c) {}

event useIter before(Iterator i) :
      call(* Iterator.next())
      && target(i) {}

event update after(Map map) :
      (call(* Map.put*(..))
      || call(* Map.putAll*(..))
      || call(* Map.clear())
      || call(* Map.remove*(..)))
      && target(map) {}
```

4.2.15 HasNext on Full DaCapo

This benchmark is provided by the PRM4J team.

Description of the monitored program. This benchmark considers the full DaCapo [22] benchmark suite for evaluation. This consists of X programs of varying size and complexity.

Description of the property. The property being monitored is the same as that described in Sections 4.2.11 and 4.2.20 i.e. a property about the safe usage of next() on Java Iterators. In PRM4J this property can be specified using the following Java code:

```
public class FSM_HasNext {

  public final Alphabet alphabet = new Alphabet();
  public final Parameter<Iterator> i =
    alphabet.createParameter("i", Iterator.class);

  public final Symbol1<Iterator> hasNext =
    alphabet.createSymbol1("hasNext", i);
  public final Symbol1<Iterator> next =
    alphabet.createSymbol1("next", i);

  public final FSM fsm = new FSM(alphabet);

  public final  MatchHandler matchHandler =
    MatchHandler.NO_OP;

  public final FSMState initial =
    fsm.createInitialState();
  public final FSMState safe =
    fsm.createState();
  public final FSMState error =
    fsm.createAcceptingState(matchHandler);

  public FSM_HasNext() {
    initial.addTransition(hasNext, safe);
    initial.addTransition(next, error);
    safe.addTransition(hasNext, safe);
    safe.addTransition(next, initial);
    error.addTransition(next, error);
    error.addTransition(hasNext, safe);
  }
}
```

4.2.16 SafeSyncCollection on Full DaCapo

This benchmark is provided by the PRM4J team.

Description of the monitored program. This benchmark considers the same programs as that in Section 4.2.15.

Description of the property. The property being monitored relates to the safe usage of collections created using the Collections.synchronized() method. The property is that if a *synchronized* collection is created in this way then there are two invalid actions that should be detected. Firstly, an iterator should not be created for the collection by a thread not holding the collection's lock. Secondly, if an iterator is created correctly (i.e. whilst holding the lock) then it should not be used incorrectly (i.e. without the lock).

In PRM4J this property can be specified using the following finite state machine. The preceding definitions have been omitted for conciseness.

```
public FSM_SafeSyncCollection() {
  initial.addTransition(sync, s1);
  initial.addTransition(asyncCreateIter, initial);
  initial.addTransition(syncCreateIter, initial);
  initial.addTransition(accessIter, initial);
  s1.addTransition(asyncCreateIter, error);
  s1.addTransition(syncCreateIter, s2);
  s2.addTransition(accessIter, error);
}
```

The `sync` event relates to create a synchronized collection, the `asyncCreateIter` and `syncCreateIter` events relate to creating iterators with and without the collection's lock respectively and the `accessIter` event is the 'bad' access without holding the lock.

4.2.17 UnsafeIterator on Full DaCapo

This benchmark is provided by the PRM4J team.

Description of the monitored program. This benchmark considers the same programs as that in Section 4.2.15.

Description of the property. The property being monitored is a common property relating Java Iterators and the collections they are create from. It states that an Iterator i created from a

Collection c is invalid and should be be used after c has been updated. In PRM4J this property can be specified using the following finite state machine. The preceding definitions have been omitted for conciseness.

```
public FSM_UnsafeIterator() {
    initial.addTransition(updateColl, initial);
    initial.addTransition(useIter, initial);
    initial.addTransition(createIter, s1);
    s1.addTransition(useIter, s1);
    s1.addTransition(updateColl, s2);
    s2.addTransition(updateColl, s2);
    s2.addTransition(useIter, error);
}
```

The state machine describes the path to an error state i.e. creating an iterator, updating the collection and then using the iterator.

4.2.18 UnsafeMapIterator on Full DaCapo

This benchmark is provided by the PRM4J team.

Description of the monitored program. This benchmark considers the same programs as that in Section 4.2.15.

Description of the property. The property being monitored is the same as that described in Section 4.2.14. In PRM4J this property can be specified using the following state machine. The preceding definitions have been omitted for conciseness.

```
public FSM_UnsafeMapIterator() {
  initial.addTransition(createColl, s1);
  initial.addTransition(updateMap, initial);
  initial.addTransition(useIter, initial);
  initial.addTransition(createIter, initial);
  s1.addTransition(updateMap, s1);
  s1.addTransition(createIter, s2);
  s2.addTransition(useIter, s2);
  s2.addTransition(updateMap, s3);
  s3.addTransition(updateMap, s3);
  s3.addTransition(useIter, error);
}
```

4.2.19 Combination of Properties on DaCapo Suite

This benchmark is provided by the PRM4J team. The benchmark checks the combination of the properties described in Sections 4.2.15, 4.2.17 and 4.2.18 on the full DaCapo benchmark suite.

4.2.20 HasNext on DaCapo Batik

This benchmark is provided by the QEA team.

Description of the monitored program. The considered program is the DaCapo [22] Batik benchmark, which is a single-threaded benchmark that produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.

Description of the property. This is the standard iterator based property used frequently as an example for runtime verification of Java properties. Informally, the Java API requires that for each Iterator object the hasNext() method be called and return true before the next() method is called. The following QEA captures the HasNext property. There are two states which capture the status of `unsafe` and `safe` iteration respectively i.e., when it is valid to call next() (mapped to `donext` to distinguish it from the state modifier). A `hasnext` event with a `true` result fires a transition from the `unsafe` to `safe` state. A `donext` event in the `unsafe` state leads to **`failure`**.

```
qea {
  Forall(i)
  accept skip(unsafe){
    hasnext(i,r) if [ r = true ] -> safe
    donext(i) -> failure
  }
  accept skip(safe){
    donext(i) -> unsafe
  }
}
```

The `hasnext` and `donext` events relate to the corresponding events in `java.util.Iterator`. For example, the following traces satisfy the property

$$\tau_1 = \texttt{hasnext}(A, \mathit{true}).\texttt{donext}(A).\texttt{hasnext}(A, \mathit{true}).\texttt{donext}(A).\texttt{hasnext}(A, \mathit{false})$$
$$\tau_2 = \texttt{hasnext}(A, \mathit{true}).\texttt{hasnext}(B, \mathit{true}).\texttt{donext}(B).\texttt{hasnext}(B, \mathit{false}).\texttt{donext}(A).\texttt{hasnext}(A, \mathit{false})$$

and the following traces violate the property

$$\tau_3 = \texttt{hasnext}(A, \mathit{true}).\texttt{donext}(A).\texttt{donext}(A)$$
$$\tau_4 = \texttt{hasnext}(A, \mathit{false}).\texttt{donext}(A)$$

4.2.21 Safe Iterators on DaCapo Batik

This benchmark is provided by the QEA team.

Description of the monitored program. The program is the the same program as the one of the benchmark in Section 4.2.20.

Description of the property. The HasNext property often leads to false positives when a collection's size is used to iterate over its contents, as is the case in the following code snippet.

```
int size = collection.size();
Iterator<Object> iter = collection.iterator();
for(int i=0;i<size;i++){
  doStuff(iter.next());
}
```

A more lenient property is to restrict the number of iterations to the size of the underlying collection. The property can be captured by a QEA with two states. The `iterate` state is entered when the iterator is created and a `donext` event can occur as long as the number of previous `donext` events is no more than `csize`. As `iterate` is a **`next`** state, if the transition cannot be taken due to the guard `csize > 0` failing then there is a transition to the **`failure`** state.

```
qea{
  Forall(i)
  accept skip(start){
    iterator(i,csize)  -> iterate
  }
  accept next(iterate) {
    donext(i) if [ csize > 0 ]
            do [ csize-- ] -> iterate
  }
}
```

Note that to record the `iterator` event on a call to java.util.Collection.iterator, a call to size() is required to extract `csize`. For example, the following traces satisfy the property

$$\tau_1 = \texttt{iterator}(A,2).\texttt{donext}(A).\texttt{donext}(A)$$
$$\tau_2 = \texttt{iterator}(A,8).\texttt{donext}(A).\texttt{donext}(A)$$

and the following trace violates the property

$$\tau_3 = \texttt{iterator}(A,1).\texttt{donext}(A).\texttt{donext}(A)$$

4.2.22 Persistent Hashcodes on DaCapo Batik

This benchmark is provided by the QEA team.

Description of the monitored program. The program is the the same program as the one of the benchmark in Section 4.2.20.

Description of the property. Hashing structures such as HashMap and HashSet rely on the property that the `hashCode` of an object remaining the same whilst the object is inside the collection. The property can be captured by the following QEA. The states `in` and `out` indicate whether the quantified object `o` is in a hashing structure. The counter `count` is used to count the number of occurrences. Note that this relies on the fact that these structures are set-like and `o` cannot belong to a collection more than once. Without this restriction the QEA would need to also have a quantification over collections. The `hashCode` on insertion is recorded in `h` and checked later; note the use of a **`next`** state to ensure that only valid transitions are taken in the `out` state.

```
qea{
  Forall(o)
  accept skip(out){
    add(c,o,h) do [ count:=1; ] -> in
  }
  accept next(in){
    add(c,o,h2) if [ h=h2 ]
                do [ count++ ] -> in
    observe(c,o,h2) if [ h=h2 ] -> in
    remove(c,o,h2) if [ count>1 and h=h2 ]
                   do [ count-- ] -> in
    remove(c,o,h2) if [ count=1 and h=h2 ] -> out
  }
}
```

The `add`, `remove`, and `observe` events relate to the *successful* corresponding (add, put, remove, get, contains, containsKey) methods in java.util.HashSet or java.util.HashMap.

For example, the following traces satisfy the property

$$\tau_1 = \texttt{add}(A,O,5).\texttt{add}(B,O,5).\texttt{remove}(B,O,5)$$
$$\tau_2 = \texttt{add}(A,M,5).\texttt{add}(A,N,6).\texttt{observe}(A,M,5)$$
$$\tau_3 = \texttt{add}(A,O,5).\texttt{remove}(A,O,5).\texttt{add}(B,O,8)$$

and the following trace violates the property

$$\tau_4 = \texttt{add}(A,O,5).\texttt{add}(B,O,6).\texttt{remove}(B,O,7)$$
$$\tau_5 = \texttt{add}(A,O,5).\texttt{add}(C,O,5).\texttt{remove}(A,O,5).\texttt{add}(B,O,6).\texttt{observe}(B,O,6).\texttt{remove}(B,O,6)$$

The final trace (τ_5) violates the property as object O is still in the collection C when it added to B with a different hash code. The explanation for other traces should be straightforward.

4.2.23 Lock Ordering on DaCapo Avrora

This benchmark is provided by the QEA team.

Description of the monitored program. The considered program is the DaCapo [22] Avrora benchmark, which is a multi-threaded benchmark that uses the available threads to simulate a number of programs run on a grid of AVR microcontrollers.

Description of the property. A common deadlock avoidance strategy is to *order* locks to ensure that no cycles can exist between pairs of locks. Note there exists a more general version of this property considering cycles of any length. For the competition, this property was specified using a QEA as follows. Two (non-equal) locks were quantified over and the state machine captures a 'path to failure' i.e., the steps required to violate the property.

```
qea{
  Forall(l1,l2)
  Where(l1 != l2)

  accept skip(start){ lock(l1) -> lock1 }
  accept skip(lock1){
    unlock(l1) -> start
    lock(l2) -> lock12
  }
  accept skip(lock12){ lock(l2) -> lock122 }
  accept skip(lock122){
    unlock(l2) -> lock12
    lock(l1) -> failure
  }
}
```

However, after the competition it was pointed out by Klaus Havelund that this formulation is overly restrictive as it does not constrain which threads are taking the locks. A corrected formulation of the property (along with specifications of this property and others from the competition in QEA and LogFire) can be found in [52].

For example, the following trace satisfies the property

$$\tau_1 = \texttt{lock}(A).\texttt{unlock}(A).\texttt{lock}(B).\texttt{unlock}(B)$$

and the following trace violates the property

$$\tau_2 = \texttt{lock}(A).\texttt{lock}(B).\texttt{unlock}(B).\texttt{unlock}(A)\ \texttt{lock}(B).\texttt{lock}(A).\texttt{unlock}(A).\texttt{unlock}(B)$$

note that τ_2 would not violate the the corrected version of the property in [52] if the locks were taken by the same thread.

4.3 Offline Track

4.3.1 Maximum Chunksize of Dropbox Connections

This benchmark is provided by the RITHM team.

Description of the traces. The traces of this benchmark are extracted from a real-world dataset [43]. The dataset contains various attributes of Dropbox connections. The dataset is preprocessed, and the details of *chunksize* used for file-transfer are extracted from the connections.

Description of the property. This property expresses the requirement that for all *connections*, it is *always* the case that *chunksize* is lesser than or equal to 999 999. The property is formalized using a fragment of first order LTL [68] as follows:

$$\forall \mathit{connection} : \mathrm{G}\,(\mathit{chunksize}\,(\mathit{connection}) \leq 999\,999)$$

4.3.2 Evolution of the Chunksize of Dropbox Connections

This benchmark is provided by the RITHM team.

Description of the traces. This benchmark uses the same dataset of Dropbox traces as the benchmark in Section 4.3.1. This property expresses the requirement that for all *connections*, it is *always* the case that when *chunksize* becomes strictly greater than 10 000 , its value *eventually* becomes lesser than or equal to 10 000 .

Description of the property. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall \mathit{connection} : \mathrm{G}\,\big(\mathit{chunksize}\,(\mathit{connection}) > 10\,000 \implies \mathrm{F}\,\mathit{chunksize}\,(\mathit{connection}) \leq 10\,000\big).$$

4.3.3 Maximum Bandwidth for Youtube Connections

This benchmark is provided by the RITHM team.

Description of the traces. For this benchmark, RiTHM processes a real-world dataset [85], which contains logs for Youtube connections on a campus network. The dataset is preprocessed, and the value of *bandwidth* is calculated for each Youtube connection using various parameters in the original data.

Description of the property. This property expresses the idea that for all connections, it is always the case that the *bandwidth* lesser than or equal to 100 000 . The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall \mathit{connection} : \mathrm{G}\,(\mathit{bandwidth}\,(\mathit{connection}) \leq 100\,000)$$

4.3.4 Changes in the Bandwidth of Youtube Connections

This benchmark is provided by the RITHM team.

Description of the traces. This benchmark uses the same dataset of Youtube traces as the benchmark in Section 4.3.3.

Description of the property. The property expresses the requirement that, for all connections, it is always the case that when the bandwidth is strictly greater than 10 000, it eventually becomes lesser than or equal to 10 000. The property is formalized using a fragment of first-order LTL [68] as follows:

$$\forall \mathit{connection} : \mathrm{G}\,\big(\mathit{bandwidth}\,(\mathit{connection}) > 10\,000 \implies \mathrm{F}\,\mathit{bandwidth}\,(\mathit{connection}) \leq 10\,000\big)$$

4.3.5 Closing Opened Files by Processes

This benchmark is provided by the RITHM team.

Description of the traces. The log file is obtained by tracing various system calls made by a set of processes, using `strace` utility in Linux. The log file of `strace` is preprocessed to produce an XML file in the format described in Section 2.1. The entries of the log files contain the process IDs, file descriptors, and the details of the operations performed on the files.

Description of the property. The property expresses requirement that for all processes and for all files, it is always the case that if a file is opened, then it is eventually closed by the process. A similar condition is expressed for sockets. The property is formalized using a fragment of first-order LTL fragment [68] as follows:

$$\forall \mathit{process}, \forall \mathit{file} : \mathrm{G}\,\big(\mathit{open}\,(\mathit{process}, \mathit{file}) \implies \mathrm{F}\,\mathit{close}\,(\mathit{process}, \mathit{file})\big) \wedge \forall \mathit{process}, \forall \mathit{socket} : \mathrm{G}\,(\mathit{accept}(\mathit{process}, \mathit{socket}) \implies \mathrm{F}\,\mathit{close}\,(\mathit{process}, \mathit{socket}))$$

4.3.6 Reporting Financial Transactions of a Banking System

This benchmark is provided by the MONPOLY team. The MONPOLY team provided five benchmarks from different areas (described also in Sections 4.3.7, 4.3.8, 4.3.9, and 4.3.10). Common to all the benchmarks is that the considered properties are policies in IT systems.

In the following, we describe the log format and the property of each benchmark. The provided MFOTL formalizations are close to MONPOLY's input format. In particular, we use a textual representation of the Boolean connectives and the metric temporal operators. MFOTL's semantics follows the standard semantics of first-order logic and the real-time logic MTL. See [12], for details on MONPOLY's specification language.

Description of the log. The log file is syntactically generated and is provided in CSV format. To represent a sequence of timestamped databases in this format, we have used the conventions explained next, illustrated by the following example.

```
trans,tp=10,ts=32,c=Alice,t=132,a=2035
trans,tp=10,ts=32,c=Bob,t=135,a=2100
```

```
trans,tp=11,ts=32,c=Charlie,t=137,a=60
report,tp=12,ts=38,t=132
```

Each line corresponds to one tuple in one of the timestamped databases. The name of the relation to which the tuple belongs to is given by the first, unnamed field. The next two fields are: 1) the index of the database in the sequence, called *time point*, and 2) the timestamp of the database. The other fields provide the tuple's value on each attribute of the corresponding relation. For instance, the first line in the previous log excerpt corresponds to a transaction event of the customer $c = \text{Alice}$ with the transaction number $t = 132$ and the amount $a = 2035$. The event was carried out at time 32 and is at the tenth position of the event stream.

This representation of the MFOTL's semantic models fits reasonably well with the competition's trace format. That is, the first field represents the event's name, and all other fields are the event's parameters. However, events with the same value of the `tp` parameter are unordered from the MFOTL semantics' perspective.

The size of the log is 13 MB. The following property is violated on the log file.

Description of the property. This property formalizes a compliance policy for a banking system that processes customer transactions. It stipulated that executed transactions t of any customer c must be reported within at most five days if the transferred amount a exceeds \$2000. The property's formalization in MFOTL is:

$$\begin{aligned}&\texttt{ALWAYS FORALL } c, t, a.\\ &\quad \mathit{trans}(c, t, a) \texttt{ AND } 2000 < a \texttt{ IMPLIES}\\ &\quad \texttt{EVENTUALLY}[0, 5]\ \mathit{report}(t)\,.\end{aligned}$$

The event $\mathit{trans}(c, t, a)$ denotes that the client c performs the transaction t, transferring the amount a. The event $\mathit{report}(t)$ denotes that the transaction t is reported. The temporal operator `ALWAYS` requires that the policy is satisfied at every time point. The interval attached to the temporal operator `EVENTUALLY` specifies that transactions must be reported within zero to five days.

Assuming that the previous log excerpt represents the complete event stream, we observe that there are two violations. First, the event $\mathit{trans}(\text{Alice}, 132, 2035)$ that occurred on day 32 is reported too late, namely, on day 38. Second, the event $\mathit{trans}(\text{Bob}, 135, 2100)$ is not reported at all.

4.3.7 Authorizing Financial Transactions in a Banking System

This benchmark is provided by the MONPOLY team.

Description of the log. The description of the log format is presented in Section 4.3.6. The size of the log is 13 MB. The following property is violated on the log file.

Description of the property. This property also formalizes a compliance policy for a banking system that processes customer transactions. It stipulates that executed transactions t of any customer c must be authorized by some employee e before they are executed if the transferred amount a exceeds \$ 2000. The property's formalization in MFOTL is:

$$\begin{aligned}&\texttt{ALWAYS FORALL } c, t, a.\\ &\quad \mathit{trans}(c, t, a) \texttt{ AND } 2000 < a \texttt{ IMPLIES}\\ &\quad \texttt{ONCE}[2, 20]\ \texttt{EXISTS } e.\ \mathit{auth}(e, t)\,.\end{aligned}$$

The event $\mathit{trans}(c, t, a)$ is as in Property 1. The event $\mathit{auth}(e, t)$ denotes the authorization of the transaction t by the employee e. Similar to Property 1, the interval $[2, 20]$ attached to the temporal past-time operator `ONCE` specifies the time period in which transactions must be authorized.

4.3.8 Approval Policy of Business Reports within a Company

This benchmark is provided by the MONPOLY team.

Description of the log. The description of the log format is presented in Section 4.3.6. The size of the log is 2 MB. The following property is violated on the log file.aa

Description of the property. This property formalizes an approval policy for publishing business reports within a company. It stipulates that any report must be approved prior to its publication. Furthermore, the person who publishes the report must be an accountant and the person who approves the publication must be the accountant's manager. Finally, the approval must happen within at most ten days before the publication. The property's formalization in MFOTL is:

$$\begin{aligned}&\texttt{ALWAYS FORALL } a, f.\\ &\quad \mathit{publish}(a, f) \texttt{ IMPLIES}\\ &\quad \big(\texttt{NOT } \mathit{acc}_F(a) \texttt{ SINCE } \mathit{acc}_S(a)\big) \texttt{ AND}\\ &\quad \texttt{ONCE}[0, 10]\ \texttt{EXISTS } m.\ \mathit{approve}(m, f) \texttt{ AND}\\ &\qquad\qquad \big(\texttt{NOT } \mathit{mgr}_F(m, a) \texttt{ SINCE } \mathit{mgr}_S(m, a)\big)\,.\end{aligned}$$

The event $\mathit{publish}(a, f)$ denotes the publication of the report f by a. The event $\mathit{approve}(m, f)$ denotes that m approves to publish the report f. The event $\mathit{acc}_S(a)$ marks the time when a starts being an accountant and the event $\mathit{acc}_F(a)$ marks the corresponding finishing time. The subformula `NOT` $\mathit{acc}_F(a)$ `SINCE` $\mathit{acc}_S(a)$ thus specifies when a is an accountant. Analogously, the events $\mathit{mgr}_S(m, a)$ and $\mathit{mgr}_F(m, a)$ mark the starting and finishing times of m being a's manager.

4.3.9 Withdrawals of Users over Time

This benchmark is provided by the MONPOLY team.

Description of the log. The description of the log format is presented in Section 4.3.6. The size of the log is 10 MB. The following property is violated on the log file.

Description of the property. This property is rooted in the domain of fraud detection. It stipulates that the sum of withdrawals of each user in the last 28 days does not exceed the limit of $ 10 000. The property's formalization is:

$$\begin{array}{l}\texttt{ALWAYS FORALL } s, u.\\ \quad \big(s \leftarrow \texttt{SUM } a; u.\ \texttt{ONCE}[0, 28]\ \mathit{withdraw}(u, a)\ \texttt{AND}\ \mathit{tp}(i)\big)\\ \quad \texttt{IMPLIES}\\ \quad s \leq 10000\,.\end{array}$$

The event $\mathit{withdraw}(u, a)$ denotes the withdrawal by the user u of the amount a. The event $\mathit{tp}(i)$ denotes that the current time point is i. This event is used in the formalization to distinguish different events $\mathit{withdraw}(u, a)$ with the same values for u and a in the relevant time window. Each user's withdrawals are accumulated into s by the aggregation operation SUM [15] over the specified time period.

4.3.10 Data Usage in Nokia's Lausanne Data-Collection Campaign

This benchmark is provided by the MONPOLY team.

Description of the log. The description of the log format is presented in Section 4.3.6. The log file in this benchmark is taken from a real-world case study [65] and is publicly available on the MONPOLY's web page in an anynomized form. The log is 14 GB large. The following property is violated on the log file.

Description of the property. This property is taken from the case study [65]. Several policies stipulate restrictions on the usage of data in Nokia's Lausanne Data Collection Campaign [65] in which sensitive data is uploaded by smartphones into the database $\mathrm{db1}$ and propagated to the databases $\mathrm{db2}$ and $\mathrm{db3}$, where it is eventually stored anonymized. The property used for the competition stipulates that data may be inserted into $\mathrm{db3}$ only if it was inserted into $\mathrm{db2}$ within the last minute. The property's formalization in MFOTL is:

$$\begin{array}{l}\texttt{ALWAYS FORALL } u, p, d.\\ \quad \mathit{insert}(u, \mathrm{db3}, p, d)\ \texttt{AND}\ d \neq \mathrm{unknown}\ \texttt{IMPLIES}\\ \quad \texttt{ONCE}[0, 60)\ \texttt{EXISTS}\ u', q.\ \mathit{insert}(u', \mathrm{db2}, q, d)\,.\end{array}$$

The event $\mathit{insert}(u, db, p, d)$ corresponds to a logged SQL insert operation performed on the database db by the database user u, involving the campaign participant p and the data d.

4.3.11 Early Alarm of Machine Operations

This benchmark is provided by the STEPR team.

Description of the log. The log file for this benchmark is an artificially created data set representing the typical shape of event logs produced during machine operations, e.g., during development, testing or productive use. It is inspired by industrial case studies and projects carried out in the safety critical domain. The XML file represents a sequence of events that carry different attributes such as, for example, *alarm*, *start2* and *time* as well as corresponding data values, e.g., *true*, *false* and *201302*, respectively. It includes a number of traces that are separated by events with name `log`. That is, between consecutive traces there is an entry such as the following.

```
<event>
  <name>log</name>
  <field>
    <name>id</name>
    <value>1</value>
  </field>
</event>
```

Single positions within a trace are represented by events with name `step` such as, for example,

```
<event>
  <name>step</name>
  <field>
    <name>alarm</name>
    <value>true</value>
  </field>
  <field>
    <name>init</name>
    <value>0</value>
  </field>
  <field>
    <name>time</name>
    <value>770576</value>
  </field>
</event>
```

The property is to be evaluated *separately* on the traces represented in the log file.

Description of the property. The property expresses that *alarm* events occurring within a time frame of 60 s after a *start2* event constitute an error. That is, the difference of the values of the attribute *time* between any two *start2*/*alarm* events is supposed to be at least 60 000 ms.

Formally, considering the log file as a linearly ordered structure $(T, <)$ of events $e \in T$ the task is to compute a predicate $F \subseteq T$ comprising those positions that exhibit a failure. We define the predicate by

$$F(e) :\Leftrightarrow \left(\begin{array}{l}\mathsf{field}(e, \texttt{alarm}) = \texttt{true}\ \wedge\\ t(e, \texttt{time}) = x \Rightarrow \exists e' < e :\\ \quad t(e', \texttt{start2}) = \texttt{true}\ \wedge\\ \quad x - t(e', \texttt{time}) < 60\,000\end{array}\right)$$

for all events $e \in T$ and all time stamps x where we assume T to give rise to mapping $t : T \times A \to V$ that maps lines and attribute names A to values V.

As an example, consider the sequence:

$start2$	$stop$	$alarm$
200 000	230 000	240 000

where the first line indicates the Boolean attributes that hold and the second line indicates the corresponding values of the

attribute *time*. The last position in the sequence is violating the property because of the early alarm. However, the sequence

$start1$	$alarm$	$start2$	$stop$	$alarm$
100 000	110 000	200 000	230 000	260 000

satisfies the property, i.e., it has no position exhibiting a failure.

4.3.12 Duration of Machine Operations

Description of the log. This benchmark is based on the same log file as the one described in Section 4.3.11.

Description of the property. The benchmark considers processes that are started upon *start1* events. A *stop* event stops all these running processes. Failures are *stop* events that occur more than 30 s after the most recent *start1* event and if since then no stop event has occurred. That is, the difference of the values of the attribute *time* between two consecutive *start1*/*stop* events is supposed to be at most 30 000 ms.

As is the case with the benchmark in Section 4.3.11, the failure predicate F is such that for all $e \in T$, $F(e)$ holds if and only if for all time stamps $x \in V$, the following formula holds:

$$
\begin{aligned}
&t(e, \texttt{stop}) = \texttt{true} \ \wedge \ t(e, \texttt{time}) = x \ \Rightarrow \exists e' < e : \\
&\quad t(e', \texttt{start1}) = \texttt{true} \wedge \\
&\quad x - t(e', \texttt{time}) > 30\,000 \wedge \\
&\quad (\forall e' < e'' < e : t(e'', \texttt{start1}) \neq \texttt{true} \wedge \\
&\qquad\qquad t(e'', \texttt{stop}) \neq \texttt{true})
\end{aligned}
$$

for all events $e \in T$. As an example, consider the sequence

$start1$	$start2$	$stop$	$alarm$
100 000	200 000	230 000	240 000

where at the third position the property is violated because of the late stop event. However, the sequence

$start1$	$stop$	$start2$	$stop$	$alarm$
100 000	110 000	200 000	230 000	260 000

does not violate the property.

4.3.13 Maximal Error Rates of Machines Operations

Description of the log. This benchmark is based on the same log file as the one described in Section 4.3.11.

Description of the property. This quantitative property specifies that the number of errors is at most a fraction $2 \cdot 10^{-5}$ (i.e., 0.002%) of the number of performed operation cycles. The log file provides information about the number of registered errors and the number of performed cycles at every position.

An event $e \in T$ constitutes a failure if and only if

$$
\begin{aligned}
&t(e, \texttt{error\_count}) > 0 \wedge t(e, \texttt{cycle\_count}) > 0 \wedge \\
&t(e, \texttt{error\_count}) > 0.00002 \cdot t(e, \texttt{cycle\_count}).
\end{aligned}
$$

4.3.14 Ordering of Machine Operations

Description of the log. This benchmark is based on the same log file as the one described in Section 4.3.11.

Description of the property. The trace is divided into groups, phases and runs, the latter are parameterised by (process) IDs. All processes x have to proceed the three steps *init*, *run* and *finish* in that order. Then, they may restart and proceed again. They must always finish. While actions of an individual process must be correctly ordered, they can be interleaved with the actions from other processes. For example, valid interleavings are

$init$	run	$init$	$finish$	run	$finish$
1	1	2	1	2	2

and

$init$	$init$	run	$init$	run	run	$finish$	$finish$	$finish$
1	2	1	3	3	2	2	3	1

.

A phase is such an interleaved sequence preceded by an event *phaseStart*. Whenever a phase starts, all running processes started before must have finished. Phases belong to a group. The beginning of a group of phases is indicated by an event *groupStart* and the end of a group is indicated by an event *groupEnd*. Groups cannot interleave and any process event (*init*, *run*, *finish*) and phase start event must happen within a group, i.e., between two consecutive $groupStart$/$groupEnd$ events. The running time of a group must be smaller than 480 seconds, i.e. the difference of the corresponding time stamps must respect that constraint.

To formalise these requirements we split it up into individual constraints:

- P_1: Groups do not overlap
- P_2: Ending groups must have started
- P_3: Phases are included in groups
- P_4: All processes must have finished before the next phase
- P_5: Init before run
- P_6: Run before finish
- P_7: Init after finish
- P_8: Init, run, finish not outside group
- P_9: Group duration is less than 480 s

We formulate these properties in a combination of Linear-time Temporal Logic (LTL) and First-order Logic (FO) as described in [39]. To this end, the log file is interpreted as sequence of FO structures modelling events and attached data. Every event e provides an interpretation for the unary predicate symbols $init$, run, $finish$, the Boolean propositions $groupStart$, $groupEnd$, $phaseStart$ and the constant $time$

by

$$\begin{aligned} \mathit{init}(x) &:= 0 < x = t(e, \texttt{init}) \\ \mathit{run}(x) &:= 0 < x = t(e, \texttt{run}) \\ \mathit{finish}(x) &:= 0 < x = t(e, \texttt{finish}) \\ \mathit{groupStart} &:= t(e, \texttt{group\_start}) = \texttt{true} \\ \mathit{groupEnd} &:= t(e, \texttt{group\_end}) = \texttt{true} \\ \mathit{phaseStart} &:= t(e, \texttt{phase\_start}) = \texttt{true} \\ \mathit{time} &:= t(e, \texttt{time}) \end{aligned}$$

As above, we assume that the function t maps attributes and event representations in the log file to the corresponding values. Whenever a field `group_start`, `group_end` or `phase_start` is not explicitly specified in the log, the respective value is assumed to be false. We now express the constraints above by the formulae

$$\begin{aligned} P_1 &= \mathit{groupStart} \Rightarrow \overline{\mathrm{Y}}(\neg \mathit{groupStart}\, \mathrm{WS}\, \mathit{groupEnd}) \\ P_2 &= \mathit{groupEnd} \Rightarrow \mathrm{Y}(\neg \mathit{groupEnd} S \mathit{groupStart}) \\ P_3 &= \mathit{phaseStart} \Rightarrow \neg \mathit{groupEnd}\, \mathrm{S}\, \mathit{groupStart}) \\ P_4 &= \forall x\ \mathit{phaseStart} \Rightarrow \neg(\mathit{init}(x) \vee \mathit{run}(x))\, \mathrm{WS}\, \mathit{finish}(x) \\ P_5 &= \forall x\ \mathit{run}(x) \implies \mathrm{Y}(\neg \mathit{run}(x)\, \mathrm{S}\, \mathit{init}(x)) \\ P_6 &= \forall x\ \mathit{finish}(x) \implies \mathrm{Y}(\neg \mathit{finish}(x)\, \mathrm{S}\, \mathit{run}(x)) \\ P_7 &= \forall x\ \mathit{init}(x) \implies \overline{\mathrm{Y}}(\neg(\mathit{init}(x) \vee \mathit{run}(x))\, \mathrm{WS}\, \mathit{finish}(x)) \\ P_8 &= \forall x\ (\neg \mathit{groupStart}\, \mathrm{WS}\, \mathit{groupEnd}) \\ &\qquad \implies \neg \mathit{finish}(x) \wedge \neg \mathit{init}(x) \wedge \neg \mathit{run}(x) \\ P_9 &= \forall x\ \mathit{groupEnd} \wedge \\ &\qquad (\neg \mathit{groupStart}\, \mathrm{S}(\mathit{groupStart} \wedge \mathit{time} = x)) \\ &\qquad \implies \mathit{time} - x < 480\,000) \end{aligned}$$

with the past-time temporal operators Y (yesterday), $\overline{\mathrm{Y}}$ (weak yesterday), S (since) and WS (weak since).

The task is to compute all positions in the log file where any of these properties is violated.

4.3.15 Existence of a Leader Rover

This benchmark is provided by the QEA team.

Description of the log. The supplied trace file contains 9756 events in CSV format and satisfies the property.

Description of the property. This property relates to the self-organisation of communicating rovers and captures the situation where (at least) one rover is able to communicate with all other (known) rovers.

The property states that there exists a leader (rover) who has pinged every other (known) rover and received an acknowledgement. The leader does not need to ping itself and communication is bidirectional i.e. any rover can ping any other rover. For example the following trace satisfies the property as B pings A and C and receives an acknowledgement.

$$\texttt{ping}(B, A).\texttt{ping}(B, C).\texttt{ack}(C, B).\texttt{ack}(A, B)$$

The following trace is not correct as B does not ping D.

$$\texttt{ping}(B, A).\texttt{ping}(B, C).\texttt{ack}(C, B).\texttt{ack}(A, B).\texttt{ping}(D, B)$$

The property can be captured by the following QEA. The syntax of QEA was previously described in Section 4.2.20.

The automaton structure of this QEA is simple; it detects the language `ping` followed by `ack`. The quantifications are non-trivial. As expected there is an existential quantification followed by a universal quantification and the constraint that `r1` and `r2` are not equal. The **`Join`** statement captures the fact that the domains of quantification for `r1` and `r2` should be equal. This is important as domains of quantification are extracted from the trace using the alphabet of the automaton and without this declaration the domains may not be equal.

```
qea{
  Exists(r1) Forall(r2) Where(r1!=r2) Join(r1,r2)
  skip(start) { ping(r1,r2) -> pinged  }
  skip(pinged){ ack(r2,r1)  -> success }
}
```

4.3.16 Granting Resources to Tasks

This benchmark is provided by the QEA team.

Description of the log. The supplied trace file contains 1000002 events in CSV format and violates the property. There are two errors: 1) a resource is cancelled by a task not holding it; and 2) a resource is granted to multiple tasks.

Description of the property. This benchmark is related to resource management in a context where resources can be granted to tasks. The property here is that every resource should only be held by at most one task at any one time. If a resource is granted to a task it should be cancelled before being granted to another task. This is therefore a mutual exclusion property.

The property can be captured by the following QEA. This quantifies over resources `r` and uses two free variables `t1` and `t2` to check mutual exclusion of the task holding the resource. Note that in the `granted` state any `grant` event leads to failure and a `cancel` event with a different task will lead to failure as this is a **`next`** state.

```
qea{
  Forall(r)
  accept next(free){ grant(t1,r) -> granted }
  accept next(granted){
    grant(t2,r) -> failure
    cancel(t2,r) if [ t1 = t2 ] -> free
  }
}
```

4.3.17 Nested Command Messages

This benchmark is provided by the QEA team.

Description of the log. The supplied trace file contains 1200 events in CSV format and violates the property.

Description of the property. This benchmark relates to a communication consisting of `command` and `success` messages. The property states that if command with identifier B starts after command with identifier A has started, then command B must succeed before command A succeeds. It can be assumed that every command is started and succeeds exactly once i.e. this is a property that has been checked separately.

The QEA defining this property is described as follows. Two commands `c1` and `c2` are quantified over and the states `none`, `startedOne` and `startedTwo` indicate whether command 1 or 2 has started respectively. Note that the property is symmetric so there are two instances for each pair of commands reflecting the two orderings of commands.

```
qea{
  Forall(c1,c2)
  accept next(none){
    com(c2) -> none; suc(c2) -> none
    com(c1) -> startedOne
  }
  accept next(startedOne){
    com(c2) -> startedTwo
    suc(c2) -> none
  }
  accept next(startedTwo){
    suc(c2) -> startedOne
  }
}
```

It was noted by Klaus Havelund that this QEA could be rewritten to more accurately reflect the assumption about commands starting and succeeding exactly once. As for the benchmark described in Section 4.2.23, a modified version of the property can be found in [52].

4.3.18 Resource Lifecycle

This benchmark is provided by the QEA team.

As the benchmark in Section 4.3.16, this benchmark is related to resource management.

Description of the log. The supplied trace file contains 1 million events in CSV format and violates the property. The violation occurs due to a resource not being cancelled when the trace finishes.

Description of the property. In this case the property concerns the *lifecycle* of a resource. Implicitly this is with respect to a single task i.e., we assume the trace only contains events from a single tasks interaction. A variant of this property appears in [52] where the task is also quantified.

The steps of the lifecycle are as follows:

- A resource may be requested
- A requested resource may be denied or granted
- A granted resource may be rescinded or cancelled
- A resource may only be requested when not currently requested or granted
- A granted resource must eventually be cancelled

This is captured by the following QEA which quantifies over the resource `r` and captures the lifecycle in the automaton structure.

```
qea{
  Forall(r)
  accept next(free){
    request(r) -> requested
  }
  accept next(requested){
    deny(r) -> free
    grant(r) -> granted
  }
  accept next(granted){
    cancel(r) -> free
    rescind(r) -> granted
  }
}
```

4.3.19 Respecting Conflicts of Resources

This benchmark is provided by the QEA team.

As the benchmark in Section 4.3.16, this benchmark is related to resource management.

Description of the log. The supplied trace file contains $1\,002\,954$ events in CSV format and satisfies the property.

Description of the property. This benchmark focuses on *conflicts* between resources. It is assumed that conflicts between resources are declared at the beginning of operation and that after this point resources that are in conflict with each other cannot be granted at the same time. It is assumed a resource cannot be put in conflict with itself.

A QEA for this property is given as follows. It quantifies over two resources `r1` and `r2` and detects a conflict declaration between these two resources. After this point there is a mutual exclusion between the two resources.

```
qea{
  Forall(r1,r2)
  accept skip(start){
    conflict(r1,r2) -> free
    conflict(r2,r1) -> free
  }
  accept skip(free){
    grant(r1) -> granted1
    grant(r2) -> granted2
  }
  accept next(granted1){
    cancel(r1) -> free
  }
  accept next(granted2){
    cancel(r2) -> free
  }
}
```

5 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 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 [49]).

5.1 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).

5.2 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 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.

5.3 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:

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

5.4 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:

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

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.

6 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`.

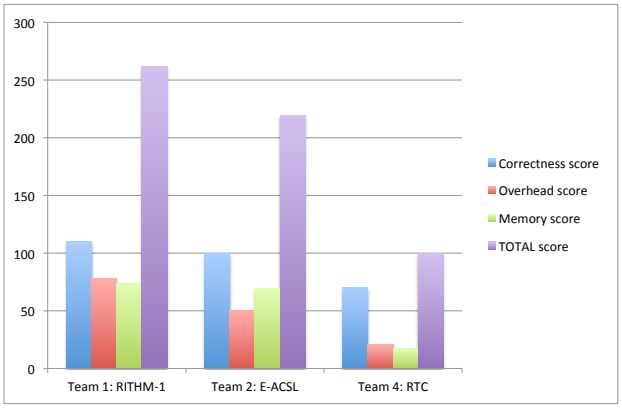

Fig. 5. Graphical representation of the scores for the C track.

For each track, we present the scores obtained in each category and the final scores achieved by each team, as defined in Section 5. In the following tables, teams are ranked according to their total scores.

Let us recall that the experiments were conducted on DataMill [73]. 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.

6.1 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 7.

6.2 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 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

	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 4.1.1	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 4.1.2	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 4.1.3	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 4.1.4	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 4.1.5	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 4.1.6	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 4.1.7	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 4.1.8	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 4.1.9	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 4.1.10	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 4.1.11	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.

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.

6.3 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.

7 Lessons Learned and Discussion

Comparison with other competitions. Over the past fifteen years, the arise of several software tool competitions [84, 2, 54, 59, 55, 21] has deeply contributed to advance the state-of-the-art in the computer-aided verification technology. The international SAT solver competition [59] is a pioneer example with a long track record of editions starting from 2002. The aim of this competition is to determine as quickly as possible

	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 4.2.1	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 4.2.2	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 4.2.3	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 4.2.4	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 4.2.5	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 4.2.6	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 4.2.7	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 4.2.8	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 4.2.9	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 4.2.10	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 4.2.11	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 4.2.12	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 4.2.13	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 4.2.14	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 4.2.15	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 4.2.16	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 4.2.17	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 4.2.18	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 4.2.19	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 4.2.20	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 4.2.21	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 4.2.22	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 4.2.23	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 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 4.3.1	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 4.3.2	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 4.3.3	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 4.3.4	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 4.3.5	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 4.3.6	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 4.3.7	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 4.3.8	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 4.3.9	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 4.3.10	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 4.3.11	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 4.3.12	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 4.3.13	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 4.3.14	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 4.3.15	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 4.3.16	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 4.3.17	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 4.3.18	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 4.3.19	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.

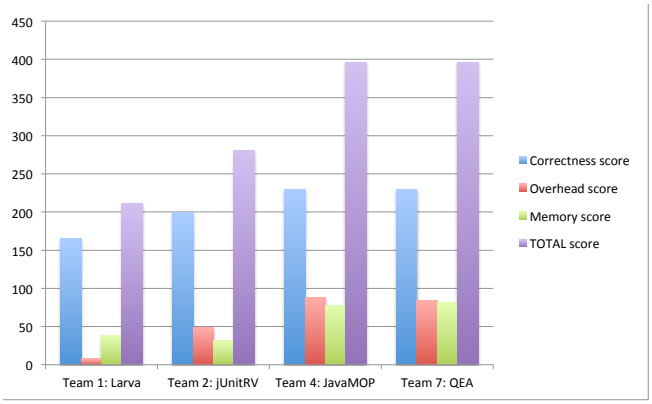

Fig. 6. Graphical representation of the scores for the Java track.

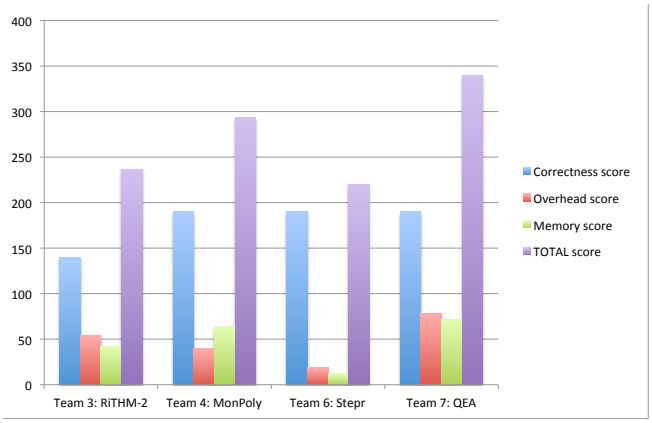

Fig. 7. Graphical representation of the scores for the Offline track.

whether a boolean formula expressed in conjunctive normal form (CNF) is satisfiable or not. If the formula is satisfiable, the tool should return also a correct assignment. If the returned assignment is incorrect the tool is disqualified. The organisers provide three different categories of benchmarks: industrial, crafted and random benchmarks. The performance is evaluated by measuring the CPU time necessary for each tool to return an answer. In the recent editions, the organisers provide also a wall clock time within which a tool can use all the available resources (for example multiple cores) to provide the correct answer. In SAT competition the jury is the responsible of choosing the final benchmarks. This is different in CRV where each team can provide up to five benchmarks to challenge the other teams, highlighting the bottlenecks of the other teams' tools. Furthermore, during the CRV training phase each team has the possibility to improve the development of their tools using new benchmarks that were not considered before.

The success of the SAT competitions has inspired other initiatives such as the Satisfiability Modulo Theories Competitions (SMT-COMP) [2] started in 2005. The challenge of SMT-COMP is to efficiently check the satisfiability of first-order formula modulo a background theory. In this case the chosen theory strictly depends on the nature of the problem to consider (i.e., arrays, bit-vectors, uninterpreted functions, etc.). A major challenge for the SMT community has been to devise a common input language for their tools that could accommodate different theories and to express their syntax and semantics. This goal was achieved in 2004 with the release of SMT-LIB a standard input language that is now used as common format for the selected benchmarks in SMT-COMP.

One important difference of CRV with SMT-COMP and the SAT competition, is the lack of a common input language for the participating tools. In CRV a benchmark unit includes a program or a trace and a property to be monitored. Properties can be expressed using different formal specification languages more or less expressive and computationally complex to be monitored. The interplay between the allowed expressiveness and the monitoring complexity plays an important role in CRV competition. Some tools may result extremely efficient in detecting simple temporal behaviours, but then they may lack the necessary support to detect more complex properties and vice versa. One of the open challenges for CRV remains the possibility to have a common formal specification language that is general enough to express all the other common formal specification languages used in the RV community.

In the area of software verification, there are three related competitions that have been recently introduced: SV-COMP [21], VerifyThis [55] and RERS Challenge [54]. SV-COMP initiated in 2011 within TACAS [1] community with the aim to compare tools for software model checking. Benchmarks are provided as C programs, while the requirements to check are provided in terms of linear temporal logic formulas. SV-COMP targets tools for the exhaustive exploration of all program behaviors. On the contrary, CRV is dedicated to monitoring tools analyzing only a single program's execution using runtime and offline verification techniques. In SV-COMP the memory utilisation is not taken in consideration and the time needed to verify a property does not affect the program execu-

tion itself since the verification process is separated from the program execution. The runtime verification tools competing in CRV introduce instead an overhead for the monitored program and they consume memory resources that could affect the execution of the program itself. This is the reason why CRV assigns a score to both the overhead and the memory utilisation.

VerifyThis [55] is another series of competitions dedicated to program verification and initiated in 2011. In VerifyThis the organisers provide to the participants algorithms in a pseudo-code with an informal specification written in natural language. The challenge for each team is to formalise the requirements, implement a prototype and verify whether the implementation is correct w.r.t. the given requirements. The available time to accomplish this goal is quite short ranging between 45 and 90 minutes. The format of VerifyThis competition differs with CRV format because is problem-centred and focuses more on the skills of the team in formalising and solving the problem rather than on the tool characteristics and performance. For this reason, it is even possible for two different teams to participate with the same tool to the competition.

The *Rigorous Examination of Reactive Systems (RERS)* challenge [54] follows a similar problem-centred approach of VerifyThis in contrast to a more tool-centred approach followed in CRV. The goal of the RERS challenge is to evaluate the effectiveness of various verification and validation approaches on reactive systems (RS), focusing on the analysis of a particular class of RS called *event-condition-action* (ECA) systems. These systems have transitions for input events that are guarded by conditions, operates on the internal state and produce outputs. The RERS challenge consists in verifying a set of properties on ECA systems: properties can be reachability properties or Linear Temporal Logic (LTL) properties. The teams are free to choose the tool and the method they prefer and they can also combine different tools in a toolchain in order to solve the challenge. Another difference with CRV is the selection of the benchmarks: in CRV the benchmarks are provided by the participating teams while in RERS the benchmarks are automatically generated with a procedure discussed in [78].

Positive points. Several positive aspects are to be noted regarding the first edition of CRV competition.

- The competition featured 8 distinct teams participating in the 3 tracks resulting in 11 participating teams in the tracks.
- The organisers have designed a sensible evaluation method. This method has been peer-reviewed and validated by the participating teams before the beginning of the competition. The method has been built upon the research efforts made in the runtime verification community when evaluating runtime verification prototypes.
- Choices needed to be made regarding the classification criteria of tracks. Moreover, different tracks could have been possible: domain of the monitored system, programming language of the monitor, categories of specifications, a track on elegance of the specification. The organisers have arranged the tracks of the competition according to the monitored system: either its programming language in case of monitoring software or traces. This reflects the fact inline monitoring has been so far the most popular RV setting when monitoring software.

Negative points. Several negative aspects are to be noted regarding the first edition of CRV competition.

- Significant delays were observed regarding benchmark submission. These delays were due to the substantial efforts required to convey the exact semantics of the specifications submitted. Indeed, as can be expected, some of the specifications could be interpreted differently by different participants. Moreover, as the participating teams mainly provided specifications in the input language of their tools, participants had also to formalize them in the specification language of their own tool.
- Delays were also observed during the phases where the organizers had to prepare the next phases. For instance, after the benchmark submission phase, a sanity check had to be performed regarding the submissions of some participants. Several iterations were needed to unify the submissions in spite of the provided provided, which was consequently not constraining enough or ambiguous on some aspects. We note that building on this observation, the next edition of the competition learned from this and for instance defined standard formats for traces [47].

Memory measurement in Java. It was not entirely clear how to measure memory usage for the Java benchmarks. It was decided that memory used by the JVM should be excluded and the participants were asked to suggest methods for recording memory usage. The first proposal was to use Java Management eXtensions (JMX) to create a separate Java program that attached to the running benchmark and queried its memory usage. For completeness we include an example Java program utilising this method:

```
public class JMXExample {

  private static final String CONNECTOR_ADDRESS =
    "com.sun.management.jmxremote."+
    "localConnectorAddress";

  public static void main(String[] args)
    throws Exception {
    // attach to target VM
    VirtualMachineDescriptor vmd =
      VirtualMachine.list().get(0);
    VirtualMachine vm = VirtualMachine.attach(vmd);
    JMXConnector jmxc = getLocalConnection(vm);
     MBeanServerConnection mbsc =
       jmxc.getMBeanServerConnection();

    MemoryMXBean memory = ManagementFactory.
      getPlatformMXBean(mbsc, MemoryMXBean.class);
    List<MemoryPoolMXBean> pools = ManagementFactory.
      getPlatformMXBeans(mbsc, MemoryPoolMXBean.class);

    while (true) {
      System.out.println("Used␣Heap:␣" +
        (memory.getHeapMemoryUsage().getUsed() /
```

```
        1000000f) + "mb");
      for (MemoryPoolMXBean pool : pools) {
        System.out.println("Used " + pool.getName() +
          ": " + (pool.getPeakUsage().getUsed() /
          1000000f) + "mb");
      }
      System.out.println();
      Thread.sleep(100);
    }
  }

  private static JMXConnector
    getLocalConnection(VirtualMachine vm)
    throws Exception
  {
    Properties props = vm.getAgentProperties();
    String connectorAddress =
      props.getProperty(CONNECTOR_ADDRESS);
    if (connectorAddress == null) {
      props = vm.getSystemProperties();
      String home = props.getProperty("java.home");
      String agent = home + File.separator + "lib" +
        File.separator + "management-agent.jar";
      vm.loadAgent(agent);
      props = vm.getAgentProperties();
      connectorAddress =
        props.getProperty(CONNECTOR_ADDRESS);
    }

    JMXServiceURL url =
      new JMXServiceURL(connectorAddress);
    return JMXConnectorFactory.connect(url);
  }
}
```

However, this approach required a separate JVM to be started to run this program. As some benchmarks are very short-lived this led to the benchmark program terminating before this method could begin measuring memory utilisation. An alternative method using the `jstat` tool (standing for Java Virtual Machine Statistics Monitoring Tool) was proposed. This method sampled memory usage every 10ms and dumped the output into a file, which was then parsed after the benchmark had run to compute memory utilisation. The script for running a program and recording its memory utilisation is given below:

```
#!/bin/bash

java -cp "lib/*:bin" $1 &> out.log &
pid=$!
jstat -gc $pid 10ms >memory.log

echo "Peak memory was " >> out.log
tail -n +2 memory.log | while read line; do
  count=0
  sum=0.0
  for entry in $line; do
    if [[ count -eq 2 || count -eq 3 ||
          count -eq 5 || count -eq 7 ]]; then
      sum=$(bc -l <<< "$entry + $sum")
    fi
    count=$((count+1))
  done
  int_sum=$(echo $sum | awk '{ print int($1) }')
  if [[ $int_sum -gt $max ]]; then
    max="$int_sum"
    echo "$max"
  fi
  echo $max
done | tail -1 >> out.log
```

The time taken to parse the memory.log file was non-negligible. Therefore, it was necessary to perform separate runs to measure time overhead and memory utilisation. Both approaches make use of the same underlying technology so should produce similar results. Ideally a single method would have been used but both approaches were used by different teams in the competition.

Monitoring hardware. In the last decade, the increasing complexity of the circuit design has been making their verification and validation more convenient to perform using hardware emulation instead of the classical simulation, a task becoming very time consuming and expensive for the industry [57,72].

Hardware emulation has opened new interesting challenges such as how to verify at runtime real-time temporal properties specified in assertion languages and how to synthesise resource efficient monitoring hardware checking these properties. FoCs [35] developed by IBM and MBAC [25,26,27] developed by Zilic and Boulé are important examples of tools for generating synthesizable hardware monitors from Property Specification Language (PSL). In [48], Finkbeiner et al. present a technique to synthesise monitor circuits from LTL formulas with bounded and unbounded future operators. More recently, Reinbacher et. al. introduce in [76,77] synthesizable hardware monitors from different fragments of Metric Temporal Logic (MTL) and Jaksic et al. in [57,58,72,79] propose several practical techniques for generating Field-Programmable Gate Array (FPGA) hardware monitors for Signal Temporal Logic (STL), an extension of MTL handling predicates over the real-values.

The first edition of the CRV competition was entirely dedicated to software runtime verification tools. We are currently exploring the possibility to add a special track for hardware monitoring tools. However, the problem of comparing performances of hardware monitors opens new challenges. In particular, all the aforementioned approaches use not only different specification languages for the property to monitor, but also different hardware and dedicated third-party software for the hardware synthesis, making extremely hard to assess the real merit of the tools for the automatic monitors generation.

Towards a general specification language (for the competition).

- every tool is defined in its own logic that is mathematically defined but have different semantics
- There is no unified specification that the organizers could use to provide specifications
- Even if such a specification language existed some tools would handle only a fragment of such logic, one would have to provide specs in each fragment in such a way that the competition is fair.

On the Challenges of Monitoring C Programs. In spite of its maturity and robust industry support, the C programming language remains a challenging frontier for runtime verification practitioners. As a close-to-the-metal, performance-driven

language, C offers flexible and fine-grained control over memory, and avoids the use of burdensome safety features; the programmer is entrusted with the utmost power and responsibility. As a consequence, however, there is little that can be asserted about the behaviour of a C program other than that which requires deep, potentially expensive analysis. However, because the language has long since passed the threshold of immortality, it is imperative that more scalable and sophisticated tools and techniques be developed to meet the needs of the C programming community. However, this requires that several key challenges be properly addressed.

First, it is necessary to achieve good coverage of the language's features and constructs, and this can a very time-consuming process. Even if one has a highly efficient analysis, insufficient coverage can severely limit the effectiveness of a tool at more than a few handful of 1000 lines of code in size. On the one hand, given the maturity of the language, legacy support becomes a concern. The libraries participating in a mature C project may be staggered chronologically, and when delving into the depths, it is not uncommon to encounter rarely used built-in functions like setjmp, keyworded modifiers such as register, and even the use of the goto statement. To its credit, the C language strives to be parsimonious in its extensions, but this also means that it is not uncommon to find highly tailored and difficult to analyze features such as custom-defined memory allocators. Furthermore, many dialects of C, such as those intended for use in embedded environments, often extend the language with compiler-specific and/or platform-dependent constructs. What is desired is a standard way of monitoring C programs, but it is difficult to conceive of a comprehensive solution. As such, tool developers with novel ideas must either build their work on top of an existing analysis framework or expend considerable time and energy accruing the necessary technological capital. In short, bringing innovative analyses to market can require significant investment.

Second, portability is becoming an increasingly important issue. In years past, it was enough that a tool could function in just two or three environments, but these days we are seeing a proliferation of different kinds of computing environments, including consumer electronics such as smartphones and sophisticated embedded environments such as control systems for avionics and healthcare. In all of these environments, C is either used directly or provides libraries for other languages including Java, Perl, and Python. A common feature of these new environments is a limited tolerance for runtime overhead, which is challenging from a verification standpoint. One possible way of reducing that overhead includes exposing runtime monitoring code in a way that allows for compiler optimizations, but because not all C compilers are available in all environments, many cost-saving measures can actually exacerbate the portability issue. Another way of controlling the run-time overhead of monitoring *C* program is to utilize various parallel algorithms on back-ends such as graphics processing unit (GPU), field-programmable gate array (FPGA), etc. A use of such back-ends requires profiling the monitors for obtaining an optimal performance. Such profiling effort can be prohibitive for scalable runtime verification because runtime verification techniques are expected to automate monitor generation, and a need of manual intervention during the profiling effort goes against the principle of automation. Designing a monitoring framework that is portable, robust in its safety guarantees, and minimally expensive remains an open problem.

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

References

1. Parosh Aziz Abdulla and K. Rustan M. Leino, editors. *Proc. of TACAS 2011: the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*. Springer, 2011.
2. 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.

3. 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.
4. Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In Bonakdarpour and Smolka [24], pages 1–9.
5. 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.
6. 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.
7. 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.
8. 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.
9. Ezio Bartocci and Pietro Liò. Computational modeling, formal analysis, and tools for systems biology. *PLoS Computational Biology*, 12(1), 2016.
10. 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.
11. 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.
12. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.
13. David Basin, Felix Klaedtke, and Eugen Zălinescu. Greedily computing associative aggregations on sliding windows. *Information Processing Letters*, 115(2):186–192, 2015.
14. 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.
15. 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.
16. 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.
17. Azzedine Benameur, Nathan S. Evans, and Matthew C. Elder. MINESTRONE: testing the SOUP. In Chris Kanich and Micah Sherr, editors, *6th Workshop on Cyber Security Experimentation and Test, CSET '13, Washington, D.C., USA, August 12, 2013*. USENIX Association, 2013.
18. 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.
19. 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.
20. 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.
21. Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Proc. of TACAS 2015: the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035, pages 401–416. Springer, 2015.
22. Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 169–190. ACM, 2006.
23. Tim Boland and Paul E. Black. Juliet 1.1 c/c++ and java test suite. *Computer*, 45(10):88–90, 2012.
24. 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.
25. M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proc. of ICCD*, pages 221–228. IEEE Computer Society Press, 2005.
26. M. Boulé and Z. Zilic. Efficient automata-based assertion-checker synthesis of PSL properties. In *Proc. of HLDVT*, pages 69–76. IEEE, 2006.
27. M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Transactions on Design Automation of Electronic Systems*, 13(1), 2008.
28. 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.
29. Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.
30. 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.

31. Christian Colombo, Gordon Pace, and Patrick Abela. Safer asynchronous runtime monitoring using compensations. *Formal Methods in System Design*, 41(3):269–294, 2012.
32. 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.
33. 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.
34. 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.
35. A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamidem, and Y. Lahbib. Combining system level modeling with assertion based verification. In *Proc. of ISQED 2005: Sixth International Symposium on Quality of Electronic Design*, pages 310–315. IEEE, 2005.
36. 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.
37. 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.
38. Normann Decker, Martin Leucker, and Daniel Thoma. jUnit^{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.
39. 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.
40. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *International Journal on Software Tools for Technology Transfer*, pages 1–21, 2015.
41. 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.
42. 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.
43. Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement*, IMC'12, pages 481–494, 2012.
44. 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.
45. 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.
46. 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.
47. Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second international competition on runtime verification - crv 15. In *Runtime Verification - 15th International Conference, RV 2015, Vienna, Austria, 2015. Proceedings*, volume 9333, pages 365–382, 2015.
48. B. Finkbeiner and L. Kuhtz. Monitor circuits for ltl with bounded and unbounded future. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5779 LNCS:60–75, 2009.
49. 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.
50. 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.
51. 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.
52. Klaus Havelund and Giles Reger. Specification of parametric monitors - quantified event automata versus rule systems. In *Formal Modeling and Verification of Cyber-Physical Systems*, 2015.
53. 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.
54. 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.
55. Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. Verifythis 2012 - A program verification competition. *STTT*, 17(6):647–657, 2015.
56. 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.
57. Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Ničković. From signal temporal logic to FPGA monitors. In *Proc. of MEMOCODE 2015: the ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 218–227. IEEE, 2015.
58. Stefan Jaksic, Ezio Bartocci, Radu Grosu, and Dejan Ničković. Quantitative monitoring of STL with edit distance. In *Proc. of RV 2016: the 7th International Conference on Runtime Verification*, LNCS, page to Appear, 2016.
59. Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.
60. 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.
61. 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.
62. 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.
63. 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.
64. 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.
65. Juha L. Laurila, Daniel Gatica-Perez, Imad Aad, Jan Blom, Olivier Bornet, Trinh Minh Tri Do, Olivier Dousse, Julien Eberle, and Markus Miettinen. From big smartphone data to worldwide research: The mobile data challenge. *Pervasive and Mobile Computing*, 9(6):752–771, 2013.
66. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
67. 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 [24], pages 285–300.
68. Ramy Medhat, Yogi Joshi, Borzoo Bonakdarpour, and Sebastian Fischmeister. Accelerated runtime verification of LTL specifications with counting semantics. *CoRR*, abs/1411.2239, 2014.
69. 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.
70. 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.
71. 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.
72. Thang Nguyen, Ezio Bartocci, Dejan Ničković, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In B. Steffen and T. Margaria, editors, *Proc. of ISoLA 2016: 7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 9953 of *LNCS*, pages 371–379. Springer, 2016.
73. 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.
74. 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.
75. 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.
76. T. Reinbacher, M. Függer, and J. Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):230–239, 2014.
77. Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. of TACAS 2014*, volume 8413 of *LNCS*, pages 357–372. Springer-Verlag, 2014.
78. Markus Schordan and Adrian Prantl. Combining static analysis and state transition graphs for verification of event-condition-action systems in the RERS 2012 and 2013 challenges. *STTT*, 16(5):493–505, 2014.
79. Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, Dejan Ničković, and Radu Grosu. Monitoring of MTL Specifications With IBM's Spiking-Neuron Model. In *Proc. of DATE 2016: The 19th Design, Automation and Test in Europe Conference and Exhibition*, pages 924–929. IEEE, 2016.
80. 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`.
81. Julien Signoles. *E-ACSL User Manual*, March 2014. `http://frama-c.com/download/e-acsl/e-acsl-manual.pdf`.
82. Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.
83. 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.

84. Geoff Sutcliffe. The 5th IJCAR automated theorem proving system competition - CASC-J5. *AI Commun.*, 24(1):75–89, 2011.
85. Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Characteristics of youtube network traffic at a campus network - measurements, models, and implications. *Computer Networks*, 53(4):501–514, 2009.