Second International Competition on Runtime Verification

Yliès Falcone, Dejan Nickovic, Giles Reger, Daniel Thoma

Second International Competition on Runtime Verification

CRV 2015

Yliès Falcone[1], Dejan Nickovic[2], Giles Reger[3], and Daniel Thoma[4]

[1] Univ. Grenoble Alpes, Inria, LIG, F-38000 Grenoble, France
ylies.falcone@imag.fr
[2] AIT Austrian Institute of Technology GmbH, Austria Dejan.Nickovic@ait.ac.at
[3] University of Manchester, UK giles.reger@manchester.ac.uk
[4] University of Lübeck, Germany thoma@isp.uni-luebeck.de

Abstract. We report on the Second International Competition on Runtime Verification (CRV-2015). The competition was held as a satellite event of the 15th International Conference on Runtime Verification (RV'15). The competition consisted of three tracks: offline monitoring, online monitoring of C programs, and online monitoring of Java programs. This report describes the format of the competition, the participating teams and submitted benchmarks. We give an example illustrating the two main inputs expected from the participating teams, namely a benchmark (i.e., a program and a property on this program) and a monitor for this benchmark. We also propose some reflection based on the lessons learned.

1 Introduction

Runtime Verification (RV) [8,14] is a lightweight yet powerful formal specification-based technique for offline analysis (e.g., for testing) as well as runtime monitoring of system. RV is based on extracting information from a running system and checking if the observed behavior satisfies or violates the properties of interest. During the last decade, many important tools and techniques have been developed and successfully employed. However, it has been observed that there is a general lack of standard benchmark suites and evaluation methods for comparing different aspects of existing tools and techniques. For this reason, and inspired by the success of similar events in other areas of computer-aided verification (e.g., SV-COMP, SAT, SMT), the First Internal Competition on Software for Runtime Verification (CSRV-2014) was established [2]. This is the second edition of the competition and the general aims remain the same:

- 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 metrics 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 monitoring.

CRV-2015 was held between January and August 2015 with the results presented in September 2015 in Vienna, Austria, as a satellite event of the 15th International Conference on Runtime Verification (RV'15). This report was produced in June 2015, prior to the evaluation stage of the competition described later. Therefore, results of the competition have not been included and will be available on the competition website.[5]

This report will begin (Sec. 2) by looking at the changes to the competition between this and the previous edition. There is then a discussion of the format of the competition (Sec. 3) where the three tracks (Offline, Java, and C) and the three stages (Benchmark Submission, Monitor Submission, and Evaluation) are discussed. To illustrate this process, we provide an example using data from the competition (Sec. 4). We then present and briefly describe the participants to each track (Sec. 5), followed by an overview of the benchmarks submitted in each track (Sec. 6). Finally, we reflect on the challenges faced and give recommendations to future editions of the competition (Sec. 7) before making some concluding remarks (Sec. 8).

2 Changes from CSRV 2014

In this section, we highlight and discuss the changes from the first edition of the competition - CSRV 2014.[6]

2.1 Towards Standardisation

One of the major difficulties faced when organising a competition is designing it so that tools can compete on the same benchmarks. For this to work it is convenient to conform to certain common standards. Prior to the previous competition there was very little focus on standardisation in the Runtime Verification community and it is still an area requiring much work.

The previous competition introduced a standard format for traces used in Offline Monitoring and these have been updated this year, as described in Sec. 3.2 on page 6. The formats have been changed to conform with general formats for CSV, JSON and XML files (see Sec. 3.2).

[5] See `http://rv2015.conf.tuwien.ac.at` - CRV-2015 Competition.

[6] The steering committee of the competition decided to change the name of the competition from CSRV (Competition on Software for Runtime Verification) to CRV (Competition on Runtime Verification) to reflect the intended broader scope of the competition.

Currently, there are two important aspects of the runtime verification process that have not yet been standardised. Firstly, there are no standard specification languages for Runtime Verification, and most tools use their own language. Secondly, there is no standard instrumentation format for Online Monitoring, although for `Java` programs `AspectJ` is becoming a de-facto standard, although there are also other instrumentation techniques often used. There exists a working group [7] looking at these issues and it is hoped that standards in this area will be available for the future editions of the competition.

2.2 Providing a Resource for the Community

As discussed previously, there is a lack of benchmarks for comparing Runtime Verification techniques. To make the submitted benchmarks accessible and usable, the organizers have used a Wiki [8] to host the benchmarks and submissions this year.

Each benchmark has its own page containing three main sections:

- **Benchmark Data.** Describing the property to be monitored (formally and informally) and the artefact to be monitored i.e. trace or program.
- **Clarification Requests.** A space for benchmark clarifications (see below).
- **Submitted Specifications.** The specification used to capture the monitored property from each team participating on the benchmark.

As well as providing benchmarks for evaluation, this information can provide an insight into how different specification languages can be used to express the same property. For further information, see the details about the format of the competition in Sec. 3.

2.3 Transparency and Communication

One of the observations made after the previous competition was that communication often occurred in emails between the chairs and participants without all participants necessarily being involved. This had two disadvantages:

1. Clarifications and instructions can be spread across many email threads with the most recent version being difficult to find.
2. Information relevant to all participants is not necessarily received by all participants, or participants may receive information at different times.

For CRV 15, there has been an effort to ensure that all communication between chairs and participants, and between participants themselves, was conducted via the Wiki in either a separate *Rules* page or the *Clarification Request* section of a benchmark.

[7] `https://www.cost-arvi.eu/`
[8] `https://forge.imag.fr/plugins/mediawiki/wiki/crv15/index.php/Main_Page`

3 Format of the Competition

In this section we describe in detail all the phases of the competition.

3.1 Declaration of Intent

The competition was announced in relevant mailing lists starting from November 2014. Potential participants were requested to declare their intent to participate to CRV 15 via email[9]. The deadline was January 30, 2015. The information requested from the participants included: institute(s), contact person and email, alternate contact persons and emails, tool home page, references to the tool, programming language of the tool, specification language(s), features of the specification languages handled by the tool (logical/dense time, propositional/-parametric, etc), and a reference to the specification language. This information was purposed to let the chairs and participants having early information about the competing tools and to get familiar with the tools and their supported specifications languages. Participants had to also indicate the track(s) in which they intended to participate in. Identification numbers (ids) were assigned to participants. As indicated in Sec. 2, a Wiki was created for the competition and participants had to register to it. For each of the three tracks (Offline, C and Java), the teams participating in the competition are listed in alphabetical order in Tables 1, 2, and 3, respectively. See Sec. 5 for further details on participants.

3.2 Submission of Benchmarks and Specifications

In this phase, participants were asked to prepare benchmark/specification sets. The deadline was March 15, 2015. The benchmarks and specifications were collected in a shared repository[10]. The repository was made accessible through SFTP and SSH protocols to facilitate the upload of benchmarks by allowing easy transfer and Unix commands to participants. The benchmarks were collected and classified into a hierarchy of directories. The hierarchy of directories has been arranged according to tracks and teams, following their ids. The hierarchy was the following:

```
falcone@lig-crv15:/work$ ls *
    C_track:
1_MarQ 2_E-ACSL 3_RiTHM 4_RV-Monitor 5_TimeSquareTrace 6_RTC
    Java_track:
1_MarQ 2_TJT 3_Java-MOP 4_Mufin
    Offline_track:
1_MarQ 2_RiTHM 3_OCLR-Check 4_RV-Monitor 5_OptySim 6_AgMon
 7_Breach 8_LogFire
```

Each of these directories had 5 sub-directories:

[9] crv15.chairs@imag.fr

[10] crv15.imag.fr

`Benchmark1 Benchmark2 Benchmark3 Benchmark4 Benchmark5.`

Each track directory contained a directory per participating team. Each team directory contained a directory per benchmark. Files had to be placed in the directory relevant to the track, team, and benchmark.

In this hierarchy, participants had reading rights to all directories and writing rights in their directories only.

Online monitoring of Java and C programs tracks. In the case of Java and C tracks, each benchmark contribution was required to be structured as follows:

- *Program package* containing the program source code, a script to compile it, a script to run the executable, and an English description of the functionality of the program.
- *Specification package* containing the associated property, the instrumentation setup, and some explanations. The property description had to contain a formal description of it in the team's specification language, informal explanations, 6 short traces demonstrating valid and invalid behaviors (3 of each), and the expected verdict (the evaluation of the property on the program).

Each specification consisted of a list of properties, with instrumentation information, and explanations. The instrumentation information mapped the events referred to in the properties to concrete program events. A property consisted of a formalization (automata, formula, etc), an informal description, and the expected verdict (indicating whether the program satisfies the property or not). Instrumentation was a mapping from concrete events (in the program) to abstract events (in the specification). For instance, considering the classical HasNext property on iterators, the mapping should have indicated that the `hasNext()` event in the property refers to a call to the `hasNext()` method on an Iterator object. Several concrete events could be associated to one abstract event.

Remark 1. The following additional guidelines were conveyed to participants:

- Too comprehensive properties should have been avoided, in order not to refrain other participants from competing on such properties.
- Programs exhibiting non-deterministic behaviors were prohibited in order to avoid interference with verdict detection.
- Benchmarks were requested to be standalone, not depending on any third-party program.

Offline monitoring track. In the case of offline track, each benchmark contribution was required to be structured as follows:

- a *trace* in either XML, CSV, or JSON format, with, for each event appearing in the trace, its number of occurrences;
- a *specification package* containing the formal representation of the property, informal explanation and the expected verdict (the evaluation of the property on the program), and informal explanations; *instrumentation information* indicating the mapping from concrete events (in the trace) to abstract events (in the specification).

At an abstract level, we defined traces as sequences of named records of the form:

```
NAME{
  field1 : value1,
  ...
  fieldn : valuen
}
```

We defined an event as an entity that has a name and arguments each of which has a name and a value.

Below, we present some example traces illustrating the three formats accepted for traces, where `an_event_name` ranges over the set of possible event names, `a_field_name` ranges over the set of possible field names, `a_value` ranges over the set of possible runtime values.

- In XML 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>
</log>
```

- In CVS format (following the standard `http://www.ietf.org/rfc/rfc4180.txt`), where the spaces are intended and required. Not the required header:

```
event, a_field_name, a_field_name, a_field_name
an_event_name, a_field_value, a_field_value, a_field_value
```

- In JSON format (following the standard `https://tools.ietf.org/html/rfc7159`):

```
an_event_name: {
a_field_name: a_value,
a_field_name: a_value
}
```

A tool was also provided to translate traces between the different formats.

3.3 Sanity-Check Phase

After the benchmark and specification phase ended on March 15, 2015, the organizers performed a sanity check over the submitted benchmarks and specifications. The purposes of the sanity check were to ensure that i) the benchmarks and specifications were complete and followed the required formats, ii) sufficient and unambiguous explanations of the specifications were provided. The sanity check resulted in clarification requests made to participants using the Wiki. Clarification requests were made on the benchmark page of the participants and were publicly available, for communication and transparency purposes. The sanity check phase ended on March 30, 2015.

3.4 Training Phase and Monitor Collection phase

The training phase started on March 30, 2015. During this phase, all participants were supposed to train their tools with all the available benchmarks in the repository. This phase was scheduled to be completed by June 10, 2015, when the participants would submit the monitored versions of benchmarks.

In this phase, competitors provided monitors for benchmarks. Participants decide to compete on a benchmark described by a pair (team_id, benchmark_number) where the team_id is the id of the team who has provided the benchmark and benchmark_number is the number of the benchmark provided by the team. That is, a contribution is related to a benchmark and contains monitors for the properties of this benchmark. Each monitor is related to one property.

More precisely, a contribution takes one of the two forms below depending on whether a program or a trace is monitored.

Java and C tracks. In the Java and C tracks, each contribution should contain the following elements:

1. The monitor given by two scripts to build and run the monitored program.
2. The property from which the monitor has been synthesized, where the property is described by:
   (a) a formal definition of the property in a well-defined specification language;
   (b) a reference to the specification language;
   (c) an informal explanation of the property.
3. The source code for instrumentation (e.g., AspectJ file for the Java track).
4. The source code of the monitoring code.

A monitor consists of two scripts, one for building the monitored version of the program, one for running it. The actions performed by the script should be documented. The description of the property should contain a formal definition of the property in the specification language chosen by the participants. References to the specification language should be given. An informal description of the property should be provided to help understanding the formalization. If the property that was used to synthesize the monitor has been expressed in

a different specification language than the one used to define the benchmark, explanations should be given as of why the submitted specification indeed corresponds to the one in the benchmark. A contribution also contains the source of the code for monitoring.

Offline track. Similarly, for the offline track, each contribution should take the following form:

1. The monitor given by two scripts to build and run the monitor over the trace.
2. The property from which the monitor has been synthesized, where the property is described by:
   (a) a formal definition of the property in a well-defined specification language;
   (b) a reference to the specification language;
   (c) an informal explanation of the property.
3. The code that is used to build concrete events out of the log entries.

The above elements are supposed to follow the same constraints as in the Java and C tracks.

3.5 Benchmark-Evaluation Phase

The competition experiments for evaluation are performed on DataMill (`http://datamill.uwaterloo.ca`), 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 on real hardware and incorporates results from existing research on how to setup experiments and hidden factors.

Each participant had the possibility to setup and try directly their tool using DataMill or by using the virtual machine provided by DataMill. The final evaluation will be performed by the competition organizers.

Computing scores. For CSRV 2014, the organizers designed an algorithm to calculate the final score for each tool. We do not want to reiterate the description of the algorithm but give an overview of the algorithm below and refer to [2] for more details.

Essentially, the final score of each team is obtained by summing the score obtained by this team on each available pair of benchmark and property on which the team has competed. The score of a team on a benchmark consists of three subscores: the first one for correctness, the second one for time overhead, and the third one for memory overhead. The correctness score assesses whether the tool produces the expected verdict for the property on the benchmark. A penalty is applied in case of an incorrect verdict reported or in case if the tool crashes. The scores for time and memory overheads assess how better is the

overhead obtained by the tool compared to the other tools. The score of each team is influenced not only by the overhead of the team but also by the factor by which it is better or worse than the average overhead obtained by the teams competing on this benchmark.

4 Illustrative Example

We use a concrete benchmark submitted to the Java track to illustrate how a benchmark is submitted by a team and how a specification and monitor are submitted for that benchmark by a different team.

4.1 Benchmark Submission

As an example benchmark we consider the first benchmark submitted by team 3 in the Java track i.e. MUFIN (see Sec. 5). The following description of the specification to be monitored was uploaded to the Wiki.

> It should be verified that no iterator object is used (by invoking the method `Iterator.next()`) after the corresponding collection has changed (by an invocation of `Collection.add()`). The property could be stated, for example, in Linear Temporal Logic enriched with predicates and quantification over object identities (cf. [4]) as
>
> $$\forall c \forall i : G(\texttt{create}(c, i) \rightarrow G(\texttt{modify}(c) \rightarrow G\neg\texttt{next}(i)))$$
>
> where $\texttt{create}(c, i)$ holds iff the method Collection.iterator() is invoked on some collection c instantiating an iterator i. The predicate $\texttt{modify}(c)$ holds at those positions in the program execution trace where `Collection.add()` is invoked on some collection c and $\texttt{next}(i)$ is true whenever `Iterator.next()` is called on an iterator i. The resulting symbolic monitor is the following:
>
> - State space $\mathrm{Q} = \{1, 2, 3, 4\}$
> - Quantification: $\forall c, \forall i$
> - Transition function δ
>   - $\delta(1, \neg\texttt{create}(c, i)) = 1, \delta(1, \texttt{create}(c, i)) = 2$
>   - $\delta(2, \neg\texttt{modify}(c)) = 2, \delta(2, \texttt{modify}(c)) = 3$
>   - $\delta(3, \neg\texttt{next}(i)) = 3, \delta(3, \texttt{next}(i)) = 4$
>   - $\delta(4, true) = 4$
> - Accepting states (with output) $F = \{1, 2, 3\}$

Importantly, this description contained an informal description of the property being monitored and a formal specification in a well-defined specification language [4]. The program to be monitored in this benchmark was uploaded to the repository and described on the Wiki, including metadata such as the number of each kind of event. In this case there are 2,000,001 `create` events, 10 `modify` events and 1 `next` event. It is indicated that the program is expected to violate

the property. Additionally, team 3 provided `AspectJ` pointcuts to connect the specification and monitored program. For example, the `create` event is associated with the following pointcut and advice:

```
public pointcut iteratorCreate(List l) :
    call(Iterator List.iterator(..)) && target(l);
after(List l) returning(Iterator i): iteratorCreate(l) { ... }
```

Finally, a number of short traces are given to illustrate valid and invalid behaviour. For example the trace

$$\texttt{create}(1,2).\texttt{create}(3,4).\texttt{next}(2).\texttt{modify}(1).\texttt{next}(4)$$

is given as a valid behaviour and the trace

$$\texttt{create}(1,2).\texttt{next}(2).\texttt{modify}(1)\texttt{next}(2)$$

is given as invalid behaviour.

4.2 Clarifications

After the benchmark has been submitted there is a time for sanity checking where clarification requests can be made. In the case of this benchmark only a few requests were made with respect to presentation.

4.3 Specification/Monitor Submission

We now consider how team 1 in the Java track (i.e. MarQ, see Sec. 5) submitted a specification and monitor for this benchmark. The first step involved placing a specification of the property on the Wiki as follows.

The property can be captured in the QEA language of MarQ as follows:

```
qea(unsafeIter){
  Forall(c,i)
  accept skip(start){  create(c,i) -> created }
  accept skip(created){  modify(m) -> modified }
  accept skip(modified){ next(i) -> failure }
}
```

This QEA quantifies universally over c and i. Note that the domain of quantification for QEA is defined by matching symbolic events in the specification against concrete events from the trace. The event automaton uses four states and three transitions to capture the path to a failure. The skip annotation on states indicates that events that do not match a transition are implicitly skipped.

The next step involved uploading the two relevant scripts to the FTP. The first script must compile the monitored program. For this team 1 submitted an `AspectJ` file and a script to weave this into the provided source code. The `AspectJ` code uses an API to construct the `QEA` given above and a monitor from that specification.

```
public void init(){
  QEABuilder b = new QEABuilder(``unsafeIter'');
  //Quantified Variables
  int c = -1; int i = -2;
  b.addQuantification(FORALL,c,i);
  //Transitions
  b.addTransition(1,CREATE,c,i,2);
  b.addTransition(2,MODIFY,c,3);
  b.addTransition(3,NEXT,i,4);
  //Sate modifiers
  b.setAllSkipStates();
  b.addFinalStates(1,2,3);
  //Create monitor
  monitor = MonitorFactory.create(b.make(),GarbageMode.LAZY);
}
```

The pointcuts provided by team 3 are then used to submit events to the monitor and check that the verdict is safe - reporting an error if not. An example of advice submitting an event to the monitor is given below.

```
after(List l) returning(Iterator i): iteratorCreate(l) {
  synchronized(LOCK){
    check(monitor.step(CREATE,l,i));
  }
}
```

As the MarQ tool relies on a number of libraries (such as `aspectjrt.jar`) the installation script also downloads these before weaving the source code using the command:

```
java -cp "lib/*" org.aspectj.tools.ajc.Main -source 1.7 -d bin -sourceroots src
```

The running script then runs the instrumented program using the command:

```
java -cp "lib/*:bin"
de.uni_luebeck.isp.rvwithunionfind.benchmarks.benchmark1.MainBad
```

5 Participating Teams

In this section, for each track, we report on the teams and tools that participated in CRV'15. Table 1 (resp.2, 3) gives a summary of the teams participating in the C (resp. Java, Offline) track. In the following of this section, we provide a short overview of the tools involved in the competition.

MarQ. MarQ [18] (Monitoring at runtime with QEA) monitors specifications written as Quantified Event Automata [1] (QEA). QEA is based on the notion of trace-slicing, extended with existential quantification and free variables. The MarQ tool is written in Java.

Tool	Ref.	Contact person	Affiliation
E-ACSL	[5]	J. Signoles	CEA LIST, France
MarQ	[18]	G. Reger	University of Manchester, UK
RiTHM-v2.0	[16]	Y. Joshi	McMaster Univ. and U. Waterloo, Canada
RV-Monitor	[15]	P. Daian	Runtime Verification Inc., Urbana, IL
RTC		R. Milewicz	University of Alabama at Birmingham, USA
TimeSquare	[3]	F. Mallet	Univ. Nice Sophia Antipolis

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

Tool	Ref.	Contact person	Affiliation
JavaMop	[11]	Y. Zhang	U. of Illinois at Urbana Champaign, USA
MarQ	[18]	G. Reger	University of Manchester, UK
Mufin		D. Thoma	University of Lübeck, Germany

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

OCLR-Check. OCLR-Check [7] is a toolset for performing offline checking of OCLR properties on system execution traces. OCLR is a temporal extension of OCL (Object Constraint Language) which allows users to express temporal properties using property specification patterns.

RV-Monitor. RV-Monitor [15] is a runtime verification tool developed by Runtime Verification Inc. (`http://runtimeverification.com`), capable of online and offline monitoring of properties written in a variety of formalisms ("logic plugins"). RV-Monitor separates instrumentation and library generation.

RiTHM-v2.0. RiTHM-v2.0 [16] takes a C program under inspection and a set of First Order Linear Temporal Logic properties as input and generates an instrumented C program that is verified at run time by a time-triggered monitor. RiTHM-v2.0 provides two techniques based on static analysis and control theory to minimize instrumentation of the input C program and monitoring intervention.

OptySim. OptySim [6] is a tool for the analysis and optimization of heterogeneous systems whose behaviour can be observed as execution traces. OptySim is based on the Spin model checker and analyzes systems observed as execution traces. OptySim supports Linear Temporal Logic specifications.

AgMon. AgMon is a monitoring framework and tool for the offline monitoring of temporal formulae expressed in a bounded variant of MTL. The monitoring strategy is based on sampling, i.e., the events in the trace are time-triggered. AgMon takes traces expressed in the CSV format as input.

Breach. Breach [12] is a Matlab toolbox supporting quantitative monitoring of Signal Temporal Logic (STL) properties. Breach provides a set of simulation-based techniques aimed at the analysis of deterministic models of hybrid dynamical systems.

Tool	Ref.	Contact person	Affiliation
AgMon	[13]	A. Kane	Carnegie Mellon University, USA
Breach	[12]	A. Donzé	University of California, Berkeley, USA
LogFire	[10]	K. Havelund	NASA JPL, USA
MarQ	[18]	G. Reger	University of Manchester, UK
OCLR-Check	[7]	W. Dou	University of Luxembourg, Luxembourg
OptySim	[6]	A. Salmerón	University of Mlaga, Spain
RiTHM-v2.0	[16]	Y. Joshi	University of Waterloo, Canada
RV-Monitor	[15]	H. Xiao	University of Illinois at Urbana Champaign, USA

Table 3. Tools participating in the offline monitoring track.

LogFire. LogFire is a rule-based runtime verification tool. It is based on the RETE [9] algorithm, and is built as an API in the Scala programming language. A monitor is an instance of a monitor class. Specifically a monitor is a user-defined `Scala` class that extends a pre-defined Monitor class defined in the LogFire API.

JavaMop. Monitoring-Oriented Programming (MOP), is a software development and analysis framework which aims to reduce the gap between formal specification and implementation by allowing them together to form a system. In MOP, monitors are automatically synthesized from specified properties and integrated with the original system to check its dynamic behaviors during execution. JavaMOP [11] is an instance of MOP for Java.

Mufin. Mufin (Monitoring with Union-Find) is a framework for monitoring Java programs. (Finite or infinite) monitors are defined using a simple API that allows to manage multiple instances of monitors. Internally Mufin uses hash-tables and union-find-structures as well as additional fields injected into application classes to lookup these monitor instances efficiently. The main aim of Mufin is to monitor properties involving large numbers of objects efficiently.

E-ACSL. E-ACSL [5] is both a formal specification language and a Frama-C plug-in. The formal specification language is a behavioral first-order typed specification language which supports in particular function contracts, assertions and built-in predicates. An example of a built-in predicate is `\valid(p)` which indicates that the pointer p points to a memory location that the program can write and read.

TimeSquare. TimeSquare [3] is an MDK (Model Development Kit) provided as a set of Eclipse plugins that can be downloaded or installed over an existing Eclipse. TimeSquare is based on the formal Clock Constraint Specification Language (CCSL), which allows the manipulation of logical time. Logical time is a relaxed form of time where any events can be taken as a reference for counting (e.g. do something every 30 openings of the door). It can be used for specifying classical and multiform real-time requirements as well as formally specifying constraints on the behavior of a model (either a UML-based or a DSL model).

RTC. RTC (Run-Time error check 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 [17]. The tool finds memory bugs and arithmetic overflows and underflows, and run-time type violations. Most of the instrumentation code is directly added to the source code and only requires a minimal runtime system.

6 Benchmarks

We give a brief overview of the benchmarks submitted to each track.

6.1 Offline Track

There were 30 benchmarks submitted to the offline track by 6 teams - MarQ, RiTHM-v2.0, OCLR-Check, RV-Monitor, Breach and LogFire. Generally each team submitted benchmarks from a particular domain:

- 4 benchmarks on `Java` API properties from RV-Monitor
- 5 benchmarks on resource management from LogFire
- 5 abstract benchmarks using letters from OCLR-Check and 1 from RV-Monitor
- 5 concurrency benchmarks from RiTHM-v2.0
- From MarQ 1 benchmark on security, 1 on programming and 3 on abstractions of online systems

The benchmarks varied in complexity of specification and length of log file. Some log files consisted of a few hundred events whilst others contained tens of millions of events. Most events were relatively simple, consisting of a small number (one or two) of parameters and a small number (two to four) of different event names. The specification languages had a wide range of features leading to a distinctive collection of specifications. Some of these features (e.g. the second-order numeric constraints on quantifiers used by RiTHM-v2.0 and scoping modifiers used by OCLR-Check) led to the modification of specification languages used by other tools.

6.2 Java Track

There were 13 benchmarks submitted to the `Java` track by 3 teams - MarQ, Java-MOP and Mufin. All three teams used AspectJ as an instrumentation tool, allowing for easy reuse of instrumentation code.

Specifications of `Java` programs in the literature have tended to focus on properties of `Java` API properties. However, this year saw a wide range of domains covered by submitted benchmarks. Five benchmarks are concerned with properties of data structures. The rest were from the following varied domains:

- A protocol property about communicating nodes

– A property about the lifetime of channels in multiplexer usage
– A property about a modal device and the correct usage of actions in modes
– A property about resource usage
– A property capturing an abstract notion of SQL injection
– A marking policy property for an abstract exam system

Whilst there were a variety of domains represented in the properties being monitored, most programs had been written for the competition as short programs that captured the desired behaviour (or not). This raises the question as to whether the results are reflective of monitor usage in the real world.

Finally, one submitted property aimed at exposing the different ways monitors for `Java` programs treated equality i.e. either semantically via `equals` or referentially by `==`. Benchmarks that test the expressiveness and usability of tools in this way are helpful in a competition and should be encouraged.

6.3 C Track

There were 18 benchmarks submitted to the `C` track by 4 teams - E-ACSL, RiTHM-v2.0, RV-Monitorand RTC. Thirteen of the benchmarks are concerned with C-specific properties such as:

– Out of bounds array access
– Signed overflow
– Memory safety i.e. invalid memory deallocation or reallocation
– Heap-based buffer overflow
– Correct calling of functions such as `strcat`, `strcopy`, `memcpy`

Some of these also incorporate semantic properties such as sortedness of arrays. Many of the benchmarks were modified versions of those used in static analysis and two of the tools have this background (E-ACSL and RTC).

The five benchmarks from RiTHMv-2.0 are semantic properties related to the usage of sockets and threads (and are the online version of the offline concurrency benchmarks).

7 Reflection

As would be expected in an endeavour of this kind, we have encountered considerable challenges. Here we reflect on this challenges and suggest how they could be tackled in future iterations of the competition.

7.1 Participant Engagement

The benefit of the competition relies on engagement from participants so that it can be presented as a reflection of the current status in the field. Therefore it is necessary to consider how best to encourage participant engagement.

There are two factors that influence a participant's likeliness to engage with the competition: a *low cost* to entry and a *benefit* to entry.

Low cost of entry. This has been a considerable challenge in both years the competition has run. The burden on participants is relatively high, especially when compared to competitions such as SAT, SMT and CASC. The effort required is more similar to that required in SV-COMP. Below we discuss why this level of effort is required in a competition of this maturity i.e. lack of standardisation.

One key issue is that entering the competition requires the participant to carry out more tasks than just submitting a monitoring tool. Due to a lack of standard specification languages, it also involves understanding and translating specifications written in another language to the participant's own specification language. Additionally, due to a lack of standard notions of how monitoring tools should be executed, it is necessary for participants to write and submit scripts allowing their tools to be run. In the case of online monitoring requiring instrumentation, it may be necessary to write such a script for each benchmark. Addressing these issues of standardisation is key to the future of the competition.

Finally, the organizers of the competition have been attempting to move towards more automated methods of evaluation by scripting processes that would otherwise be manual. Continuing this automation is necessary to lift the burden on both the participants and organizers.

Benefit of entry. Whilst it may be obvious that this competition is important for the runtime verification community in general, it is also important to ensure that participants receive some benefit from entering. Only one tool can claim to be a winner in each track, and in some cases tools may be confident that they will not win before entry. Therefore, a benefit beyond the chance of winning is required.

One suggestion is that the future editions of the competition invite participants to submit a short (2-page) *system description* papers that are included in the proceedings of the conference. This is a practice taken by some other competitions and acts as an obvious benefit to entry.

7.2 Engaging with Static Analysis Based Tools

In recent years, the runtime verification community has made a special effort to engage with the static analysis community and this effort has been successful. It is an exciting result that we are seeing tools with roots in static analysis adopting runtime verification techniques and participating in the competition. However, it is therefore necessary to consider how this difference in viewpoint effects the design of the competition.

One point that was raised during the competition was that tools that perform static analysis do not typically deal with a concepts of *events* and *traces* and extensions to dynamic analysis typically involve introducing runtime assertions and additional code to track data values, rather than extracting events.

7.3 The C Track

Whilst the areas of runtime verification for log file analysis and monitoring `Java` programs have received a reasonable amount of attention in the literature, there

has not been as much focus on monitoring `C` programs. Additionally, `C` programs are more likely to be targeted by tools coming from the domain of static analysis, as mentioned above. Consequently, there continues to be issues surrounding the definition of benchmarks and monitors in the C track.

8 Concluding Remarks

This report was written during the training phase. Once this phase is complete, the organizers will evaluate all the submitted monitors using the scoring mechanism introduced in [2] and outlined in Sec. 3.5. The results of the competition are expected to be announced during the RV 2015 conference in Vienna, Austria. This report is published to assist future organizers of CRV to build on the efforts made to organize CSRV 2014 and CRV 2015.

Acknowledgment. The organizers would like to thank Christian Seguy from the IT team of Laboratoire d'Informatique de Grenoble for his help on setting up the the repository hosting the benchmarks. The organizers are also grateful to Yuguang Zhang from the DataMill team for setting up a convenient and powerful evaluation infrastructure on DataMill.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., M., D. (eds.) FM 2012: Formal Methods, Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer Berlin Heidelberg (2012)
2. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 1–9. Springer (2014)
3. Deantoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: Carlo A. Furia, S.N. (ed.) TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012. vol. 7304, pp. 34–41. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer, Prague, Czech Republic (May 2012), `https://hal.inria.fr/hal-00688590`
4. Decker, N., Leucker, M., Thoma, D.: Monitoring modulo theories. In: Ábrahám, E., Havelund, K. (eds.) TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 341–356. Springer (2014)
5. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of c programs. In: Proceedings of SAC '13: the 28th Annual ACM Symposium on Applied Computing. pp. 1230–1235. ACM (2013)
6. Díaz, A., Merino, P., Salmeron, A.: Obtaining models for realistic mobile network simulations using real traces. IEEE Communications Letters 15(7), 782–784 (2011)
7. Dou, W., Bianculli, D., Briand, L.: A model-driven approach to offline trace checking of temporal properties with ocl. Tech. Rep. SnT-TR-2014-5, Interdisciplinary Centre for Security, Reliability and Trust (2014), `http://hdl.handle.net/10993/16112`

8. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D. (eds.) Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear. IOS Press (2013)
9. Forgy, C.: Rete: A fast algorithm for the many patterns/many objects match problem. Artif. Intell. 19(1), 17–37 (1982)
10. Havelund, K.: Rule-based Runtime Verification Revisited. International Journal on Software Tools for Technology Transfer (STTT) p. To appear (2014)
11. Jin, D., Meredith, P.O., Lee, C., Roşu, G.: JavaMOP: Efficient Parametric Runtime Monitoring Framework. In: Proceedings of ICSE 2012: THE 34th International Conference on Software Engineering, Zurich, Switzerland, June 2-9. pp. 1427–1430. IEEE Press (2012)
12. Juniwal, G., Donzé, A., Jensen, J.C., Seshia, S.A.: Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In: Mitra, T., Reineke, J. (eds.) 2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014. pp. 24:1–24:10. ACM (2014)
13. Kane, A., Fuhrman, T.E., Koopman, P.: Monitor based oracles for cyber-physical system testing: Practical experience report. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014. pp. 148–155. IEEE (2014)
14. Leucker, M., Schallhart, C.: A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (may/june 2008), http://dx.doi.org/10.1016/j.jlap.2008.08.004
15. Luo, Q., Zhang, Y., Lee, C., Jin, D., Meredith, P.O., Serbanuta, T., Rosu, G.: Rv-monitor: Efficient parametric runtime verification with simultaneous properties. In: Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. pp. 285–300 (2014)
16. Navabpour, S., Joshi, Y., Wu, C.W.W., Berkovich, S., Medhat, R., Bonakdarpour, B., Fischmeister, S.: RiTHM: a tool for enabling time-triggered runtime verification for c programs. In: ACM Symposium on the Foundations of Software Engineering (FSE). pp. 603–606 (2013)
17. Quinlan, D.J., Schordan, M., Miller, B., Kowarschik, M.: Parallel object-oriented framework optimization. Concurrency and Computation: Practice and Experience 16(2-3), 293–302 (2004)
18. Reger, G., Cruz, H.C., Rydeheard, D.E.: Marq: Monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9035, pp. 596–610. Springer (2015), http://dx.doi.org/10.1007/978-3-662-46681-0