CRV16: Rules and Organisation

Yliès Falcone[1], Sylvain Hallè[2], and Giles Reger[3]

[1] Univ. Grenoble Alpes, Inria, France. ylies.falcone@imag.fr
[2] Universitè du Quèbec à Chicoutimi, Canada. shalle@acm.org
[3] University of Manchester, UK. giles.reger@manchester.ac.uk

Abstract. This document describes the rules and organisation for the Third International Competition on Runtime Verification (CRV16).
This document is related to work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

1 Introduction

This document describes the rules and organisation for the Third International Competition on Runtime Verification (referred to as CRV16). See relevant literature for information about Runtime Verification e.g. [3, 5].

Aims. During the last decade, many important tools and techniques for Runtime Verification have been developed and successfully employed. However, it was 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,CASC), the First Internal Competition on Software for Runtime Verification (CSRV-2014) was established [1]. This year 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.

History. For the previous two iterations of the competition see [1, 2, 4]. The design of the first competition was described in [1] with full results and reflection appearing in [2]. The design of the second competition was described in [4]. In the previous two years partial results were reported at the Runtime Verification conference (held in September) however full results were only reported later for timing reasons.

Organisers. This year the competition will be organised by Yliés Falcone, Sylvain Hallé and Giles Reger. Yliés has helped organise the competition in the previous two years and co-founded the competition with Ezio Bartocci. Giles Reger entered in 2014 and was an organiser in 2015 and Sylvain is a new addition this year.

Contact. Please use the email address `crv2016@crv.liflab.ca` to contact the organisers. If you contact any of the organisers directly they will either forward your email to this address or reply including this address. However, for most queries it is likely that the most appropriate form of communication will be via the Wiki, please see below for details.

This Document. We aim to fully describe all rules and organisation relevant to this year's edition of the competition. The document is structured as follows:

- Section 2 gives an overview of the structure of the competition
- Importantly, Section 2.1 describes the proposed timeline, including deadlines
- The competition is divided into four main phases. Sections 3 to 6 describe these phases in further detail.

2 Overview

The competition will be divided into five parts as follows:

1. **Registration** (described below) collects information about entrants
2. **Benchmark Phase** (Section 3). In this phase entrants can submit benchmarks to be considered for inclusion in the competition. These must conform to the given rules and will be subject to review by the organisers.
3. **Clarification Phase** (Section 4). The benchmarks resulting from the previous phase will be made available to entrants. This phase gives entrants an opportunity to seek clarifications from the authors of each benchmark. Only benchmarks that have had all clarifications dealt with by the end of this phase will be eligible for the next phase.
4. **Monitor Phase** (Section 5). In this phase entrants will be asked to produce monitors for the eligible benchmarks. As described later, these must be runnable via a script on a Linux system and it must be possible to automate any necessary installations. Submissions must conform to the given rules and will be subject to review by the organisers.
5. **Evaluation Phase** (Section 6). Submissions from the previous phase will be collected and executed, collecting relevant data to compute scores as described later. Entrants will have been given an opportunity to test their submissions on the evaluation system so any errors at this point will be scored negatively. The output produced during evaluation will be made available after the competition.

Note that it **is not** necessary to participate in the Benchmark Phase, although doing so may give you an advantage. However, all entrants should take part in the remaining three phases, including the Clarification Phase.

2.1 Timeline

This year the competition will take place over a shorter period of time. This decision is based on the observation that in previous years the reason deadlines were missed was not lack of time, but lack of information and motivation. By keeping the competition shorter (and less work) we hope to maintain a momentum.

The planned periods and deadlines are as follows:

Event	Starts	Ends (Deadline)
Registration	May 1st	June 5th
Benchmark Submission	May 1st	May 29th
Clarifications	June 5th	June 12th
Monitor Submission	June 19th	July 10th
Results		August 1st

There is a very hard deadline of August 8th for all results to be finalised so that they can be included in the proceedings of the Runtime Verification conference. But the competition needs to finish in plenty of time of this date to give the organisers time to prepare this submission.

2.2 Registration

Registration should happen at some point before the start of the Clarification Phase (see Section 2.1 for timings). Entrants wishing to participate in the Benchmark Phase should register earlier. If you are unsure whether you want to participate feel free to contact the organisers informally with any queries you have. It is useful to register as early as you can to aid planning.

During registration entrants will be asked for:

- The tool name
- The names of those participating and their email addresses
- A main contact
- Affiliated Institution(s)
- The track(s) being participated in (see below)
- A tool website (if available)
- The implementation language of the tool
- The specification language of the tool i.e. what it uses as specifications
- Relevant references in Bibtex format
- Links to any other supporting information (i.e. reference manual)

Where possible we encourage tools to be open-source but we do not require it. If there is no publicly available document explaining the specification language already (e.g. a tool paper) we may ask entrants to write a short document explaining their specification language as this information will be required to assess the correctness of submissions.

2.3 Tracks

As in previous years, the competition will consist of three tracks. How these should be treated in the different phases will be discussed in detail in the relevant sections. Here we will give a brief overview of the general scope of what is covered by the competition and then the idea behind each track.

General Scope. There are many activities that could fall under the umbrella term of runtime verification. Here we describe and defend the *current* scope of the competition. Note that we (the general competition community) are open to suggestions for future iterations.

The general activity we consider is that of taking a trace τ and a specification ψ and answering the question whether τ satisfies/is a model for/is accepted by ψ. In some cases the trace τ is taken as a stand-alone artefact and in other cases it is being generated as a program is running. We restrict our attention to *linear* traces (i.e. we do not consider concurrency) and require programs that generate such traces to be (broadly) deterministic.

Note that our formulation precludes the other, related, activity of finding multiple *matches* between the trace and specification describing failure. In all cases it is sufficient to report failure as soon as it is detected. On a similar note, we do not restrict ourselves to safety properties, but (for obvious reasons) require all specification languages to have an interpretation on finite traces (i.e. one could have bounded liveness).

The Offline Track. This covers the scenario where the trace is collected, stored in a log file, and then processed *offline*. We define three acceptable formats for traces (log files) to be used in benchmarks and can give (some) help to entrants in extending their tool with parsing code for these formats. In previous years benchmarks in this track have focussed on parametric, or data-carrying, events. This year we would like to also encourage participation from those interested in purely propositional specifications. Note that this track does not (currently) support notions of time other than as data.

The Online Java Track. This covers a scenario where a Java program is instrumented to produce events that should be handled by a monitor. In the past the majority of instrumentation was carried out via AspectJ. We would like to standardise this where possible. Therefore, benchmark submissions will be required to include AspectJ instrumentation (again, where possible). Entrants may use alternative instrumentation techniques in their submissions but we ask that they justify this.

The Online C Track. This covers a scenario where a C program is run and it is asked whether a specification holds of that run. Starting this year, the C track will consist of two sub-tracks, although this is mainly for organisational reasons and we encourage entrants to participate in both sub-tracks. These are:

1. **Generic Specification.** The C version of the Java track where some instrumentation should abstract the program as a sequence of events to be passed to a monitor. Instrumentation can be automatic or manual (see the relevant section).
2. **Implicit Specification.** This covers implicit properties (such as memory-safety and out of bounds array access). In this case the trace may also be implicit (although we note that it theoretically exists).

2.4 Wiki and Server

Following on from last year[4], all communication (in the form of submissions, clarifications etc) will take place via a Wiki and Server. *The only communication that will occur via email will request entrants to refer to the Wiki.*

The Wiki can be found at `http://crv.liflab.ca/wiki`.

Instructions for how to fill in the Wiki should appear on the Wiki itself and in relevant places throughout this document. We recommend that entrants bookmark the Wiki and visit it with non-zero frequency to ensure that they remain aware of any competition-related progress.

Participants will be contacted by the organisers after registration to be given details of how to log in to the server. If you do not receive such a communication please contact the organisers. The server will be used both as a repository for all submitted material and to run the competition. Therefore, there will be a consistent environment that can be used for submission, testing and evaluation. Each participant will have a directory on the server in which to upload benchmarks and monitors and to install their tool.

3 Benchmark Submission Phase

This section describes the number of benchmarks we are aiming for, the general format benchmarks should take and specific information that should be included for each track. Note that a benchmark exists in a single track only.

3.1 Submitting Benchmarks

Once you have collected together the information for a benchmark submission as described below it is necessary to do two things:

1. Go to the Wiki (see earlier) and follow the instructions for creating a benchmark page (using the appropriate template) in the appropriate track. Then fill in the necessary information.
2. Upload all relevant files to your directory on the sever under a subdirectory with the appropriate name

If you do not do these two things then the benchmark has not been submitted.

[4] This was trailed last year, generally worked, but was not used correctly by everybody.

3.2 Number of Benchmarks

Feedback from previous iterations of the competition indicated that there were too many benchmarks and/or too much effort per benchmark. In this iteration of the competition we are reducing the number of benchmarks with the aim that the effort required of participants goes down.

We will be aiming for roughly 10 benchmarks per track. These will be decided on in the next phase. To keep us below this limit we suggest teams do not submit more than $\frac{10}{T}$ benchmarks where T is the number of teams in the relevant track. This value should be apparent after registration. Please contact the organisers if you are unsure.

We predict that $\frac{10}{T}$ will be at most 3 i.e. we assume we will have at least 3 participants in each track. Therefore, it is a good idea to ensure that benchmarks represent a reasonable level of difficulty in terms of monitoring. Note that the organisers will ensure that no two benchmarks are too similar. Teams are encouraged to check other submissions.

3.3 General Benchmark Format

A benchmark consists of three parts.

1. **The Metadata.** Every benchmark requires:
   - A name
   - A 2-3 sentence description
   - At least 1 domain category to help organise benchmarks
     - Suggested categories are: Aerospace, Artificial, Concurrency, Finance, Medical, Software Engineering. Feel free to use your own.
2. **The Trace Part.** This is track-specific and will be described below (Sec. 3.5 or 3.6)
3. **The Property.** This is a description of the property being monitored and should take the same form for all tracks, with the exception of the **Implicit Specification C subtrack** (see below). The information required for a property description is given below (Sec. 3.4).

3.4 Describing Properties

A specification description consists of the following parts:

1. An *informal* description. This should include
   - The context of the property
   - The relevant events (their names and parameters, if any)
   - The ordering constraints between events that form the property
   - **Any assumptions that are being made**
2. Demonstration traces.
   - These can be in an abstract form i.e. a(1).b(2)
   - At least 6 examples traces (3 that should be accepted, 3 rejected)

- The traces should be explained in terms of the abstract property, not the formal description.
- Ideally these traces will be used to highlight edge cases e.g.
  - Is the empty trace accepted?
  - Does the property assume a certain event can only occur once, what should happen if this assumption is broken (it is often okay to accept a bad trace that breaks an assumption).

3. A *formal* description.
   - The specification language should be well-defined and documented.
   - The specification should be explained in reasonable detail. It is not necessary to say 'there are four states and six transitions' etc. But please draw attention to usage of any non-standard language features.
   - If it is not obvious, please explain why the above demonstration traces work on this specification.
4. (Optional) A description of the property in First-Order Linear Temporal Logic
   - The idea is to have a single general specification language that is neutral and (hopefully) more widely understood than individual team's specific specification languages.
   - This is optional but strongly encouraged. We acknowledge that some properties may be out of the scope of this language.
   - This is the first time we are trying this and welcome feedback.
   - See Appendix A for a description of this logic

An Example. Figure 1 gives an example for what could be written for the commonly used HasNext property. We have used the RuleR language, which has not participated on the competition previously.

Changes for the Implicit Specification C subtrack. In this subtrack there is no notion of trace. Therefore, demonstration traces are not required. Instead, if there is a simple example of snippet code that could be used to demonstrate good/bad behaviour consider including this. Similarly, providing a description in First-Order Linear Temporal Logic is unlikely to be appropriate.

3.5 The Trace Part: Offline Track

The trace part for the Offline Track consists of:

1. A trace file (see below for acceptable formats)
2. Trace statistics including the number of each event in the trace
3. If different, an explanation of how concrete events in the trace relate to abstract events in the property
4. **The status of the trace with respect to the provided property**

Title: **Has Next on Iterators**

Informal Description:
This property considers Iterator objects and their correct usage. The property is drawn from studies considering iterators in the Java standard library (i.e. `java.util.Iterator`) but could apply to similar notions of iterators in other languages. It is assumed that iterators have (at least) the two methods `hasNext`, returning true if there are more elements, and `next`, returning the next element. The property is that, for every iterator object i, if `next` is called on i then there must have been at least one call of `hasNext` on i returning true *since* the last call of `next` or the start of the trace. There are therefore two events of interest: $\texttt{hasNext}(i, r)$ and $\texttt{next}(i)$ where i is an iterator and r the boolean result of the `hasNext` call.
It is assumed that an iterator object can only be created once and that once `hasNext` returns false for the first time it will always return false after this point. Such assumptions could be checked at a separate property.

Demonstration Traces:
The property accepts the empty trace ϵ. Other traces that satisfy the property:

$\texttt{hasNext}(A, \mathit{false})$
$\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{hasNext}(A, \mathit{true}).\texttt{next}(A).\texttt{hasNext}(A, \mathit{false})$
$\texttt{hasNext}(A, \mathit{true}).\texttt{hasNext}(B, \mathit{true}).\texttt{next}(A).\texttt{hasNext}(C, \mathit{true}).\texttt{next}(C).\texttt{next}(B)$

Traces that do not satisfy the property:

$\texttt{next}(A)$
$\texttt{hasNext}(A, \mathit{false}).\texttt{next}(A)$
$\texttt{hasNext}(B, \mathit{true}).\texttt{next}(A)$

Formal Description:
The following is a specification in the RuleR rule-based language [**?**].

```
ruler HasNext{
  observes hasNext(obj,bool), next(obj);
  always Start {
    next(i:obj), !Safe(i) -> Fail;
    hasNext(i:obj,r:bool), r  ->  Safe(i)
  }
  state Safe(i:obj){  % A 'state' rule instance is removed by Ok
    next(i) -> Ok;
  }
  initials Start;
}
```

The above rules ensure that an instance of the rule $\texttt{Safe}(i)$ will be present whenever it is safe to call `next` on iterator i. Therefore, if $\texttt{next}(i)$ is seen and no such instance exists there is a failure.

First-Order LTL description:
The following gives the property firstly using future-time operators and secondly using past-time operators:

$(\forall i : \mathit{iterator})((\neg\texttt{next}(i)\ \mathcal{U}\ \texttt{hasNext}(i, \mathit{true})) \wedge \Box(\texttt{next}(i) \rightarrow \circ(\neg\texttt{next}(i)\ \mathcal{U}\ \texttt{hasNext}(i, \mathit{true}))))$
$(\forall i : \mathit{iterator})\Box(\texttt{next}(i) \rightarrow \bullet(\neg\texttt{next}(i)\ \mathcal{S}\ \texttt{hasNext}(i, \mathit{true})))$

Fig. 1. Property Specification for the HasNext property.

Trace formats. At an abstract level, we define traces as finite sequences of events. An event is an entity that has a name and arguments each of which has a name and a value i.e. it is a named record of the form:

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

The accepted trace formats are XML, CSV, or JSON format (all following the official standards for these formats). Below, we illustrate the three formats accepted for traces, using `an_event_name` to range over the set of possible event names, `a_field_name` to range over the set of possible field names, and `a_value` to range 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[5], where the spaces are intended and required.

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

  Note that we require a single header giving the field name for different events. This can either name fields arbitrarily (i.e. arg1,arg2 etc) or have as many columns as field names, leaving some columns blank for some records.

- In JSON format[6]:

[5] Following the standard `http://www.ietf.org/rfc/rfc4180.txt`
[6] Following the standard `https://tools.ietf.org/html/rfc7159`

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

Whilst JSON supports arbitrarily structured values we do not currently support such events.

Parsing. As these formats comply with official standards it is possible to use standard parsing libraries. The organisers can provide a Java-based parser and a tool for translating between the different trace formats.

3.6 The Trace Part: Online Tracks

The trace part for the Online Track consists of a program that produces the trace. This part in the benchmark should consist of:

1. The source files *without* instrumentation (see below for instrumentation)
2. Two scripts:
   - `compile`: should compile the *unmonitored* program
   - `run`: should run the *unmonitored* program

   Please ensure that these scripts do not include instrumentation information
3. Instrumentation information. This might include
   - An AspectJ file (in the Java track this is preferred)
   - A script to perform automated instrumentation
   - A copy of manually instrumented source files (`diff` should show that only instrumentation has been added)

   Please include an explanation of how the instrumentation relates to the events in the specification
4. Trace statistics. The number of each event that should be observed at runtime. Note that this assumes that the program is deterministic. This is obviously necessary for a fair evaluation.
5. (Optional) An trace log (using one of the formats above) of the observed trace. For short runs this can help ensure that instrumentation has been applied correctly. However, in many cases it will be inappropriate.
6. **The status of the trace captured by the program with respect to the provided property**

Changes for the Implicit Specification C subtrack. For this subtrack no instrumentation information is required. However, the code should be commented at the point a failure, if any, is expected.

4 Clarification Phase

The purpose of this phase is to ensure that all teams are happy with the benchmarks before any effort is spent in preparing for them. During this phase the organisers and teams should check all submitted benchmarks; a checklist appears below. At the end of the phase a decision will be made on each benchmark as to whether it will be included in the next phase.

Checklist. Organisers and teams should check the following things:

- Are all relevant parts present i.e.
  - Informal description
  - Demonstration traces
  - Formal description
  - Trace file or program package
- Are the events involved clear from the informal description?
- Are any non-trivial demonstration traces explained?
- Is the formal description in a well-defined specification language?
- Do the informal and formal descriptions agree?
- If a first-order LTL specification has been provided does it agree with the informal and formal descriptions?
- Is the trace file in an acceptable format (for Offline)?
- Have trace statistics been given, do they agree with the trace (for Offline)?
- Do the compile scripts work (for Online)?
- Is instrumentation information clear (for Online)?
- Is the progam deterministic (for Online)?
- Has a verdict been given?

Further to this list please also consider:

- Does the benchmark make sense?
- Is the benchmark non-trivial?
- Is the benchmark sufficiently dissimilar from other benchmarks?

If the answer is 'no' to any of these questions we may question the suitability of the benchmark.

Clarification Requests. If anything is unclear then a clarification request should be made. There is a section for this at the bottom of each benchmark page on the Wiki. Organisers will monitor these sections and email teams if they have any outstanding clarification requests. It is the submitting teams responsibility to address any requests. Only benchmarks with no outstanding requests at the end of the phase will be considered for the next phase.

Voting. In some cases the organisers may ask teams to vote on whether a benchmark is to be included. This could happen for two reasons.

1. There are too many benchmarks
2. A benchmark is contentious with respect to suitability (not correctness)

We hope that this will not be necessary but will explain the planned process if it is. Each team and each organiser will get one vote (if an organiser is a team they will not get two votes) and voting will **not** be anonymous. A majority will be needed in either direction with the organisers holding the casting votes in the case of a tie. Only those voting will be considered when measuring the majority.

Final Decision. The final decision about any benchmark lies with the organisers. However, they will try as hard as possible to ensure that all submitted benchmarks are deemed suitable by the end of this phase.

5 Monitor Submission Phase

This is the phase where teams create monitors for the benchmarks. At the beginning of the phase a list of the relevant benchmarks will be made available on the Wiki. In this section we describe the output monitors should produce and the processes for testing and submitting the monitors.

Further Clarification Requests. During this phase it is still okay to make clarification requests but the submitting team is under no obligation to answer the requests or change the benchmark as these should have been made in the previous phase. However, the organisers, where possible, will attempt to deal with any such requests, mediating with the submitting team where relevant.

5.1 Required Outputs

Monitors should output the verdict by printing a status line. This should be

- **STATUS: Satisfied** if the property is satisfied
- **STATUS: Violated** if the property is violated
- **STATUS: TimeOut** if the status is not detected within the time limit
- **STATUS: GaveUp** if the monitor fails to find the verdict for any reason

If no status line is printed it will be assumed that the status is TimeOut. Therefore, it is important that these status lines are printed. **If no status line is printed then there will be a penalty even if the correct verdict is found.**

5.2 Optional Outputs

Individual tools may output additional information about the monitoring process, available with command-line options. This is encouraged so that output can be checked and confidence in the results established. The following outputs are encouraged:

- Number of events processed, with a breakdown of event kind (option -events);
- Reason for violations or satisfaction i.e. number of events after which a violation occurs, the event at which the violation occurred, variable bindings that relate to the violation (option -witness).
- The trace of events processed by the monitor (option -trace).

5.3 Testing and Submitting Monitors

The same server that is being used to upload benchmarks and monitors to will be used for evaluation. Therefore, if you test that your submissions work on this machine then they should work during evaluation. After submission any errors will be scored negatively.

As previously noted, the organisers should have sent you login details for the server after registration. If this has not happened please contact them directly.

What needs to be uploaded for benchmarks depends on whether the benchmark belongs to an offline or online track, as explained later.

Tool installation. As evaluation will be carried out on the sever you will need to install your tool here. This may just require uploading the relevant jar file or may involve compiling etc. At the end of this phase the server will be frozen so it is important that the version of the tool you want evaluated is the version that is installed in your directory on the server at the end of this phase.

Offline Benchmark packages. For the offline track we require **a single** script for all benchmarks that takes two inputs:

1. The name of the benchmark. This should be the name given on the Wiki in CamelCase e.g. if the name is "Very interesting property" the name here will be VeryInterestingProperty.
2. The name of the trace file.

The script should then run the monitor for this benchmark on the given trace file. This will allow the organisers to automatically generate the inputs necessary for evaluation.

Online Benchmark packages. For the online tracks (Java and C) a benchmark package should be submitted per benchmark consisting of:

- A script `setup.sh` to setup the benchmark e.g. perform automated instrumentation

– A script `run.sh` to run the monitored program
– Either the original sources or suitably manually instrumented sources

Where possible, the instrumentation described in the benchmark should be used in the submission.

6 Evaluation Phase

In this section we describe the evaluation phase.

6.1 Running Submitted Monitors

Experiments will be run on the shared sever. It is important that all teams have tested their submissions on this server. Anything that fails to run during evaluation will be scored accordingly.

Details of the machine will be provided later. If you have particular requirements it is *very* important that these are made clear as soon as possible.

6.2 Calculating Scores

Let N be the number of teams participating in the considered track and L be the total number of benchmarks in that track. 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.

Correctness Score. The correctness score $C_{i,j}$ for a tool T_i running a benchmark B_j is 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).

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

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

- 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

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

  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.

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:

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

  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

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

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

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

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.

6.3 Announcing Results

Results will be announced initially to the participating teams at the end of the evaluation phase. There will then be a short time for any team to query the results. Timings and outputs from all experiments will be made available.

The final results will then be announced at the Runtime Verification conference.

References

1. Ezio Bartocci, Borzoo Bonakdarpour, and Yliès Falcone. First international competition on software for runtime verification. In 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*, pages 1–9. Springer, 2014.
2. Ezio Bartocci, Borzoo Bonakdarpour, Yliès Falcone, Christian Colombo, Normann Decker, Felix Klaedtke, Klaus Havelund, Yogi Joshi, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 2015 (To appear).
3. Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. In Manfred Broy and Doron Peled, editors, *Summer School Marktoberdorf 2012 - Engineering Dependable Software Systems, to appear*. IOS Press, 2013.
4. Yliès Falcone, Dejan Nickovic, Giles Reger, and Daniel Thoma. Second international competition on runtime verification CRV 2015. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 405–422, 2015.
5. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, may/june 2008.

A First Order Linear Temporal Logic

This year we are asking teams to consider also providing a specification of their properties in first-order linear temporal logic. In this appendix we motivate this decision and describe what we mean by this logic.

Important Note
Whilst this appendix gives a formal description of a general notion of first order linear temporal logic this is just for guidance. We haven't sanity-checked these definitions thoroughly. As we describe below, the main idea is to be able to write something generally understandable to those comfortable with first-order logic and linear temporal logic.

A.1 Brief Motivation

In general within the RV community, and in previous iterations of this competition, it has been observed that there is a lack of coherence in specification languages. Most tools use their own specification language and little is understood about how, in some cases very different, specification languages are related. This has an additional issue that benchmarks used by one tool are not readily usable by another tool. One of the aims of the competition is to bring more coherence to the area.

We note that some may see the development of new specification languages as a major component of runtime verification research. Indeed, activities such as the extension of languages with new features, exploration of notions of usability, adoption of domain-specific concepts, introduction of language-features allowing for optimisation of the monitoring process, and others are all valid research activities. However, we argue that it is still useful to have a common specification language that is sufficiently expressive to capture most of the properties we want to write. It is not clear what such a language should look like. To help address this question we are trying to use a single common language to describe all specifications submitted to this competition. Hopefully this will shed some light on whether this particular language would be suitable, and what issues we need to consider when developing a common language.

Therefore, we propose a general first-order discrete linear-time temporal logic with past and future operators and *finite trace* semantics.

We chose temporal logic as it is commonly used in runtime verification and is abstract enough to allow reasonably concise specifications (compared to, say, state machines). The language had to be first-order to capture parametric or first-order properties. We allow past and future operators to aid readability of specifications. Finite trace semantics are necessary for the nature of runtime verification in general. Note that we do not provide a monitoring algorithm, the aim is to perform specification in this language only.

A.2 The Syntax

We begin by introducing the necessary syntax.

Definition 1 (Signature). *A signature* $(\mathcal{F}, \mathcal{P}, \Sigma, \pi, \mathcal{S}, \gamma_A, \gamma_R, =)$ *consists of*

- *A finite set of function symbols* $\mathcal{F}$*,*
- *A finite set of predicate symbols* $\mathcal{P}$*,*
- *A finite set of event names* Σ
- *An arity function* $\pi : (\mathcal{F} \cup \mathcal{P} \cup \Sigma) \to \mathbb{N}_0$*,*
- *A finite sort of sorts* $\mathcal{S}$*,*
- *Two sorting functions* $\gamma_A : (\mathcal{F} \cup \mathcal{P} \cup \Sigma) \times \mathbb{N} \to \mathcal{S}$ *and* $\gamma_R : \mathcal{F} \to \mathcal{S}$ *with* γ_A *giving sorts to the arguments of functions, predicates and events and* γ_R *giving sorts to the results of a function*
- *A special equality symbols* $=$

A signature is well-formed *if* $\mathcal{F}$*,* $\mathcal{P}$ *and* Σ *are disjoint. This allows the arity function* π*, which assigns arities to each of the symbols, to be well-defined. If a symbol* s *has arity* n *(i.e.* $\sigma(s) = n$*) we call it* n*-ary. We separate event names and predicate names so that we can treat them differently. However they will be used in a similar way. We call zero-arity function symbols* constants *and zero-arity predicates* propositions*.*

We define the syntax of First-Order Linear Temporal logic assuming a signature and an infinite set of variables $\mathcal{V}$ such that variables are distinct from previously introduced symbols.

Definition 2 (Term). *Terms are inductively defined. A term is either a variable* x*, a constant function symbol* c *or an* n*-ary function* f *applied to* n *terms* $t_1, \ldots, t_n$*.*

Definition 3 (Formula). *Formulas are inductively defined as*

$$(\varphi_1) \mid \mathit{true} \mid q \mid p(t_1, \ldots, t_n) \mid t_1 = t_2 \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid (\forall x : s)(\varphi_1) \mid \\ \hat{\circ}\varphi_1 \mid \varphi_1 \mathcal{W} \varphi_2 \mid \hat{\bullet}\varphi_1 \mid \varphi_1 \mathcal{B} \varphi_2 \mid e \mid e(t_1, \ldots t_n)$$

where t_i *are terms,* q *is a proposition,* p *is an* n*-ary predicate symbol,* φ_i *are formulas,* x *is a variable,* s *is a sort, and* e *is an event name. The temporal operator* $\hat{\circ}$ *is read as* weak next*,* $\mathcal{W}$ *is read as* weak-until*,* $\hat{\bullet}$ *is read as* weak last*, and* $\mathcal{B}$ *is read as* back-to*.*

We have chosen to use the weak operators as they make the finite-trace semantics easier to define. As usual we can define derived operators (including the strong versions of temporal operators) allowing for nicer specifications.

Definition 4 (Derived Operators). *We define the usual derived operators*

false	$= \neg \mathit{true}$	
$\varphi_1 \vee \varphi_2$	$= \neg(\neg\varphi_1 \wedge \neg\varphi_2)$	
$\varphi_1 \Rightarrow \varphi_2$	$= \neg\varphi_1 \vee \varphi_2$	
$\varphi_1 \Leftrightarrow \varphi_2$	$= (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1)$	
$(\exists x : s)(\varphi)$	$= \neg(\forall x : s)(\neg\varphi)$	
$\circ\varphi$	$= \neg\hat{\circ}\neg\varphi$	(strong-next) or just (next)
$\bullet\varphi$	$= \neg\hat{\bullet}\neg\varphi$	(strong-last) or just (last)
$\Box\varphi$	$= \varphi \; \mathcal{W} \; \mathit{false}$	(always)
$\Diamond\varphi$	$= \neg \, \Box \, \neg\varphi$	(eventually)
$\blacksquare\varphi$	$= \varphi \; \mathcal{B} \; \mathit{false}$	(always before)
$\blacklozenge\varphi$	$= \neg \, \blacksquare \, \neg\varphi$	(previously)
$\varphi_1 \; \mathcal{U} \; \varphi_2$	$= (\varphi_1 \; \mathcal{W} \; \varphi_2) \wedge \Diamond\varphi_2$	(until)
$\varphi_1 \; \mathcal{S} \; \varphi_2$	$= (\varphi_1 \; \mathcal{B} \; \varphi_2) \wedge \blacklozenge\varphi_2$	(since)
$\varphi_1 \; \mathcal{R} \; \varphi_2$	$= \neg(\neg\varphi_1 \; \mathcal{U} \; \neg\varphi_2)$	(release)

Next we introduce some properties of formulas important for their semantics.

Definition 5 (Closed Formulas). *A variable is bound in a formula if it appears underneath a quantifier binding it. A formula is* closed *if all variables are bound. This is a standard definition.*

Definition 6 (Well-Sorted). *It should be obvious that is possible to define a notion of well-sortedness for closed formulas using the sorting functions γ_A and γ_R and the sort bindings for variables. We do not give this here but assume that all formulas are well-sorted.*

Definition 7 (Precedence). *Unary operators have higher precedence over binary operators. Operators with the same arity have the same precedence, therefore parenthesis are required to remove ambiguity.*

A.3 The Semantics

We introduce the semantics of the formulas.

Definition 8 (Structures and Interpretations). *A structure interprets function and predicate symbols and events. An interpretation $(\mathcal{D}, \tau)$ consists of a* domain function $\mathcal{D}$ *and and a finite sequence of structures τ. The domain function maps sorts from a signature to (finite or infinite) sets of sorted domain elements. Each structure Δ interprets symbols with respect to the domain function.*

- *A function symbol f is interpreted as a function $\Delta(f)$ in $\mathcal{D}(\gamma_A(f,1)) \times \ldots \times \mathcal{D}(\gamma_A(f,\pi(f))) \to \mathcal{D}(\gamma_R(f))$*
- *A predicate symbol p is interpreted as a boolean-valued function $\Delta(p)$ in $\mathcal{D}(\gamma_A(p,1)) \times \ldots \times \mathcal{D}(\gamma_A(p,\pi(p))) \to \{\bot, \top\}$*
- *An event symbol e is interpreted as a boolean-valued function $\Delta(e)$ in $\mathcal{D}(\gamma_A(e,1)) \times \ldots \times \mathcal{D}(\gamma_A(e,\pi(e))) \to \{\bot, \top\}$*

Note that this definition assumes a *fixed domain* for sorts. Alternative semantics would allow a changing domain i.e. a different domain at each point in the sequence. The following definitions would not need to be modified significantly to allow for this generalisation. But we think a *fixed domain assumption* is reasonable.

We use the following sequence notation. If τ is a sequence then τ_i is the ith element of τ where sequences are indexed from 1.

In the following a variable valuation Γ is a finite function (a map) from variables to elements of a domain. We will assume such valuations are well-sorted i.e. $\Gamma(x) \in \mathcal{D}(s)$ where the sort of x is s.

Definition 9 (Interpretation of Terms). *Given a structure Δ and variable valuation Γ we can interpret terms as members of a domain. Let us define $(\Delta, \Gamma)(t)$ in the obvious way i.e. inductively as*

$$\begin{aligned}(\Delta,\Gamma)(x) &= \Gamma(x)\\ (\Delta,\Gamma)(c) &= \Delta(c)\\ (\Delta,\Gamma)(f(t_1,\ldots,t_n)) &= \Delta(f)((\Delta,\Gamma)(t_1),\ldots,(\Delta,\Gamma)(t_n))\end{aligned}$$

Next we define what it means for an interpretation to be a model of a first-order linear temporal formula.

Definition 10 (Semantics (Models)). *An interpretation $(\mathcal{D}, \tau)$ is a model for a first-order linear temporal logic formula φ iff $(\mathcal{D}, \tau), \{\}, 0 \models \varphi$ where the relation $\models$ is inductively defined as follows. For non-temporal operators:*

$(\mathcal{D},\tau),\Gamma,i \models \mathit{true}$	
$(\mathcal{D},\tau),\Gamma,i \models q$	*If* $\tau_i(p)$
$(\mathcal{D},\tau),\Gamma,i \models p(t_1,\ldots,t_n)$	*If* $\tau_i(p)((\tau_i,\Gamma)(t_1),\ldots,(\tau_1,\Gamma)(t_n))$
$(\mathcal{D},\tau),\Gamma,i \models t_1 = t_2$	$(\tau_i,\Gamma)(t_1) = (\tau_i,\Gamma)(t_2)$
$(\mathcal{D},\tau),\Gamma,i \models \neg\varphi$	$(\mathcal{D},\tau),\Gamma,i \not\models \varphi$
$(\mathcal{D},\tau),\Gamma,i \models \varphi_1 \wedge \varphi_2$	$(\mathcal{D},\tau),\Gamma,i \models \varphi_1$ *and* $(\mathcal{D},\Gamma) \models \varphi_2$
$(\mathcal{D},\tau),\Gamma,i \models (\forall x : s)(\varphi)$	*For every* $d \in \mathcal{D}(s)$ *we have* $(\mathcal{D},\tau),\Gamma \cup (x \mapsto d), i \models \varphi$
$(\mathcal{D},\tau),\Gamma,i \models e$	*If* $\tau_i(e)$
$(\mathcal{D},\tau),\Gamma,i \models e(t_1,\ldots,t_n)$	*If* $\tau_i(e)((\tau_i,\Gamma)(t_1),\ldots,(\tau_1,\Gamma)(t_n))$

And for the temporal operators (separated for ease of reading)

$(\mathcal{D},\tau),\Gamma,i \models \hat{\circ}\varphi$	$(\mathcal{D},\tau),\Gamma,i+1 \models \varphi$ *if* $i \leq \lvert\tau\rvert$ *and true otherwise*
$(\mathcal{D},\tau),\Gamma,i \models \varphi_1 \mathcal{W} \varphi_2$	*Either* *a) For all* $k \geq i$ *we have* $(\mathcal{D},\tau),\Gamma,k \models \varphi_1$ *b) Or there is a* $j \geq i$ *such that* $(\mathcal{D},\tau),\Gamma,j \models \varphi_2$ *and for all* $i \leq k < j$ *we have* $(\mathcal{D},\tau),\Gamma,k \models \varphi_1$
$(\mathcal{D},\tau),\Gamma,i \models \hat{\bullet}\varphi$	$(\mathcal{D},\tau),\Gamma,i-1 \models \varphi$ *if* $i > 1$ *and true otherwise*
$(\mathcal{D},\tau),\Gamma,i \models \varphi_1 \mathcal{B} \varphi_2$	*Either* *a) For all* $k \leq i$ *we have* $(\mathcal{D},\tau),\Gamma,k \models \varphi_1$ *b) Or there is a* $j \leq i$ *such that* $(\mathcal{D},\tau),\Gamma,j \models \varphi_2$ *and for all* $j < k \leq i$ *we have* $(\mathcal{D},\tau),\Gamma,k \models \varphi_1$

Finite trace semantics. Note that we have a finite trace semantics, our model is a finite sequence of structures. We have therefore chosen a certain behaviour for the temporal operators at the ends of the trace. This is symmetric for future and past operators. The effect on derived operators is as desired. At the end of a finite trace we want $\Box\varphi$ to be true and $\Diamond\varphi$ to be false, symmetrically at the beginning of the trace $\blacksquare\varphi$ should be true and $\blacklozenge\varphi$ should be false.

The semantics of next operators means that other common identities still hold but with minor alterations i.e.

$$\Box\varphi = \varphi \wedge \hat{\circ}\Box\varphi$$

$$\Diamond\varphi = \varphi \vee \circ\Diamond\varphi$$

Note that the identify for $\Box$ uses a weak-next operator as $\Box\varphi$ should be true at the end of the trace, whilst the identify for $\Diamond$ uses strong-next as $\Diamond\varphi$ should be false at the end of the trace.

We note that this is not the only way of dealing with finite traces. One could view the trace seen so far as a finite prefix of some infinite trace and talk about possible extensions. Orthogonally, one can introduce multiple verdicts differentiating the possible continuations of the observed trace. We do not consider either of these ideas here but they are compatible with our general approach.

We make the following simplifying assumption, which is sensible but not strictly required.

Definition 11 (Constant Interpretation of Non-Events). *We assume that the interpretation of function and predicate symbols is the same throughout the sequence of structures. Therefore, the only change between structures is the presence of events. This is why we separated events from predicates and fits our usual setting of events as observations with functions/predicates as fixed operations.*

Note that an alternative way of capturing events would be for event names to propositional and event arguments to be constants that are interpreted differently in each structure. We feel that the current presentation is closer to how RV systems are normally defined.

A.4 From Traces to Models

The interpretations from the previous section look a bit different from the kinds of traces we often write. This is mainly because in RV we often (not always) have an assumption that a single event occurs at each time point. In this setting a trace can be written as a finite sequence of events. We show how such a trace can be translated into an Interpretation.

Definition 12 (Event Per Time Point Assumption). *Given a trace σ captured as a finite sequence of events where an event consists of a name and optionally a sequence of argument values and a domain $\mathcal{D}$. Let $\mathcal{I} = (\mathcal{D}, \tau)$ where τ_i is a structure where the events true in τ_i are exactly the singleton event appearing at σ_i.*

Definition 13 (Event Per Time Point Axioms). *We note that it may be necessary/useful to add additional axioms to the logic when the event per time point assumption is made i.e. one can assume*

$$\bigvee_{e_i \in events} e_i$$

i.e. that at least one event should occur and

$$\bigvee_{e_i \in events} \left(e_i \rightarrow \bigwedge_{e_j \in events/e_i} \neg e_j \right)$$

Note that this is relatively informal and still requires us to define events.

This requires us to give a domain. We can also show how the domain can be extracted from the trace. This is the typical approach in much RV work.

Definition 14 (Extracting Domain from Trace). *Given a trace σ captured as a finite sequence of events where an event consists of a name and optionally a sequence of argument values. Let $\mathcal{D}_\sigma$ be the domain function for σ where, for each s in $\mathcal{S}$ the set $\mathcal{D}_\sigma(s) = \{d_i \mid e(\ldots, d_i, \ldots) \in \sigma \wedge \gamma_A(e, i) = s\}$.*

This states that if the ith argument of event e is of sort s and value d appears as the ith element of e in the trace then d is in the domain of s.

In fact this is a slight generalisation of what many RV approaches do as these approaches often fail to define notions of signature and sorts.

A.5 Including Background Theories

It is common to assume background theories such as Arithmetic. We do not formalise this notion here. But in general this can be achieved by introducing interpreted operations to the signature and extending the semantics to interpret such operations.

For example, one could add the functions $+, -, *$ and predicates $\geq, \leq$ from arithmetic. Then the set of models for formulas over this extended signature would also need to satisfy the theory of arithmetic.

A.6 Metric Operators etc

We do not have metric temporal operators in our language or any other kinds of counting quantifiers or aggregate operators. Many temporal logics for RV do have these. If you use them they must be explained. If possible also give a version of the specification without them. Even better, define the operators in terms of operators defined here!

A.7 ASCII notation

The above symbols are readable but difficult to type in a plaintext document (such as an email). We introduce an alternative ASCII syntax in the following (rather informal) grammar.

```
term := variable | constant | functionName(termList)

atom    := eventName
         | <eventName>(termList)
         | propositionName
         | predicateName(termList)

formula := atom
         | not(formula)
         | formula1 and formula2
         | formula1 or formula2
         | formula! => formula2
         | formula1 <=> formula2
         | next formula
         | last formula
         | strong-next formula
         | strong-last formula
         | weak-next formula
         | weak-last formula
         | always formula
         | always-been formula                  /* past-time always */
         | eventually formula
         | once formula                         /* past-time eventually */
         | formula1 until formula2
         | formula1 weakly until formula2
         | formula1 since formula2
         | formula1 weakly since formula2
         | formula1 release formula2
         | (forall variable:type)(formula)
         | (exists variable:type)(formula)
```

If you choose to use other syntax it must be explained.

A.8 Examples

We give two simple examples. Future versions of this document may add further examples.

HasNext. We repeat the HasNext example. This is a standard RV property often used as a Hello-World property. If you want help with writing any properties then please contact the organisers.

This can be written as follows

$$(\forall i : \mathit{iterator})\ ((\neg \texttt{next}(i)\ \mathcal{W}\ \texttt{hasNext}(i, \mathit{true})) \wedge \\ \Box(\texttt{next}(i) \rightarrow \circ(\neg \texttt{next}(i)\ \mathcal{W}\ \texttt{hasNext}(i, \mathit{true})))$$

The first part states that we cannot see a `next` until we have seen the appropriate `hasNext` and the second part says that every `next` cannot be followed by another `next` until we have seen the appropriate `hasNext`. Note the use of $\mathcal{W}$, there is no requirement to see a `hasNext`.

It can be more concisely written using past-time operators.

$$(\forall i : \mathit{iterator})\Box(\texttt{next}(i) \rightarrow \bullet(\neg \texttt{next}(i)\ \mathcal{S}\ \texttt{hasNext}(i, \mathit{true})))$$

This quite simply states that for every `next` there has been no other `next` *since* the last `hasNext`. Note that strong-last is used, meaning that there must be a previous point in the trace i.e. `next` cannot appear on the first step.

Auction Bids. Consider the property that items are listed on an auction website with a minimum amount. After listing, bids on the item must be strictly increasing and if the item is sold it should be for at least the minimum.

One way of formulating this property in first order linear temporal logic is as follows. Not how this uses the theory of arithmetic, a mixture of different quantifications, and past and future temporal operators.

$$(\forall i : \mathit{item})(\forall \mathit{min} : \mathit{int}) \\ \Box(\texttt{list}(i, \mathit{min}) \rightarrow \Box \begin{pmatrix} (\forall a, a' : \mathit{int})((a \leq a' \wedge \texttt{bid}(i, a)) \rightarrow \blacksquare \neg \texttt{bid}(a')) \\ \texttt{sold}(i) \rightarrow \blacklozenge((\exists a : \mathit{int}) a > \mathit{min} \wedge \texttt{bid}(i, a)) \end{pmatrix}$$