Comparing Two Methods for Checking Runtime Properties

Gerard J. Holzmann

Nimble Research, Monrovia, CA 91016, USA
gholzmann@acm.org

Abstract. A number of different tools for runtime verification have been developed in the last few years, and just as many different formalisms for specifying the types of properties that runs must satisfy. Though there have been some attempts to develop them, there are as yet no general benchmark suites available with event logs of a realistic size and the corresponding properties to check. This makes it hard to compare the expressiveness and ease of use of different specification formalisms, or even the relative performance of tools. In this paper we try to address this to some extent by considering a formal logic and its supporting tool, and compare both ease of formalization and tool performance with a tool that was designed for an entirely different domain of application, namely interactive analysis of large source code archives. The static analysis tool needs just a few small adjustments to support runtime verification of event streams. The results of the comparison are surprising.

Keywords: Runtime Verification, Interactive Static Analysis, Logics, Formal Specification.

1. Introduction

For a comparison of options for runtime verification we consider a recent paper describing a new tool called *nfer* [1,2]. The goal of the *nfer* tool was to simplify the analysis of event-logs by making it easier to accurately specify properties of interest and making it possible to efficiently monitor event streams for those properties. In this paper we compare their results with runtime checks that can be performed with an extended version of the Cobra tool [3], which was originally developed to support interactive static analysis of large code archives, and consider what we can learn from any differences we find.

A driving example for the development of *nfer* was its application to the analysis of event logs from NASA/JPL's Mars Science Laboratory (MSL) mission, specifically from the Curiosity Rover. As a specific example of this analysis the authors describe a scenario where a specific type of error event, named TLM_TR_ERROR, must be ignored when it appears within the bounds of a known type of safe interval, but must be flagged as an error if it appears outside these intervals.

A formalization of the concept of an *event interval* can provide a useful abstraction for reasoning about event streams in general. It should then be possible to define interval sequences that overlap, are adjacent, or nested hierarchically, depending on application. To capture these notions, the *nfer* tool adopts a logic framework for expressing properties of intervals known as Allen Interval algebra [4]. The operators from Allen interval algebra include, for example, "*i1* before *i2*." "*i1* overlaps *i2*," "*i1* during *i2*," and "*i1* meets *i2*," where *i1* and *i2* refer to specific intervals of events.

2. Event Intervals

Allen algebra allows one to reason about event intervals and draw logical conclusions from relations between intervals. This assumes, of course, that everything of interest can indeed be formalized as an event interval. A standalone event, like the TLM_TR_ERROR event, can be considered as a special type of zero-length interval, which would then allow us to reason about its appearance inside or outside other

intervals. Using zero-length intervals, though, does introduce some peculiarities in the logic that turn out to be important.

3. Formalization in Interval Logic

The formalization of the TLM_TR_ERROR property in Allen logic, as given in [1] and [2], is as follows:

```
cmdExec :- CMD_DISPATCH before CMD_COMPLETE
      where CMD_DISPATCH.cmd = CMD_COMPLETE.cmd
      map {cmd -> CMD_DISPATCH.cmd}
okRace :- TLM_TR_ERROR during cmdExec
      where cmdExec.cmd = MOB_PRM | cmdExec.cmd = ARM_PRM
```

This formalization defines an interval of interest named *cmdExec*, which starts with a CMD_DISPATCH event and ends with CMD_COMPLETE.[1] Each event in the target log has a name and a *cmd* field, which can hold a device identifier. The *where* clause in the formalization specifies that these fields must match for the events to be considered part of the same interval.

The second part of the specification states that if event TLM_TR_ERROR (taken as a pseudo interval of zero length) occurs inside the *cmdExec* interval as defined here, then it is to be tagged *okRace* (meaning it is not an error) provided that the device identifier on the interval was either MOB_PRM or ARM_PRM.

4. Some Complicating Factors

The test lends itself naturally to an alternate formalization as a state machine, with state changes indicating whether or not we are inside an interval of interest. In that case we can check what the current state in the processing of the log is when TLM_TR_ERROR events occur. But there are some complicating factors that we must take care of to perform the check correctly.

A first complicating factor is that the target event-log captures an *interleaved* series of timestamped events that originate from *different* sources. Specifically, the TLM_TR_ERROR events come from a different source than the CMD_DISPATCH and CMD_COMPLETE events, and although all events carry a timestamp that is ordered chronologically for each source separately, events from different sources can appear *out of order* with respect to events from other sources in the log. Specifically, the timestamps on TLM_TR_ERROR events can lag those delimiting the *cmdExec* intervals.

A second complicating factor is that the target log also contains *zero-length* intervals for *pairs* of events: the log shows that CMD_DISPATCH and the matching CMD_COMPLETE events often carry the same timestamp. This can be a problem especially for the *nfer* specification because the key operators from Allen logic have a formal semantics that does not match what is needed to handle the zero-length intervals. Specifically, the semantics of the *during* operator state that the timestamp on the *first* event of the interval must precede the timestamp of the *second*. This means that it requires a < (less than) relation on timestamps, and not a ≤ relation (less than or equals).

5. A Cobra Script

The property can now be formulated in Cobra's interactive scripting language as a textual description of a small finite state machine. The check can be written in about 25 lines of text if we can assume that all events are ordered chronologically. Since this is not the case for the target log, some more information must be remembered, and the extended checker script now grows to about 30 lines, plus some helper

[1] The actual event-names in the logs are different.

functions. Both versions are of course longer than the 5-line version formalized in Allen logic, but have a fairly simple straightforward structure.

The Cobra version of the query is shown in Figure 1, handling each of the three types of relevant events in the log separately, using small helper functions, *new_interva)*, *close_interval*, and *check_interval*, to create, close, and check the relevant intervals. Timestamps follow the event name and command identifier field in the log, in a standard comma-separated values (csv) format. An example is:

```
CMD_DISPATCH, SEQ_WAIT_FOR, 517525760
CMD_COMPLETE, DAN_ABORT,    517525760
```

The Cobra tool turns names and commas into token fields that can be navigated by following standard .nxt or .prv references. For the above two lines, for instance, a sequence of 10 tokens are generated, four of which are names, two are numbers, and four are the commas used to separate the values.

```
%{
        if (#CMD_DISPATCH)
        {       id = .nxt;          # comma separator
                id = id.nxt;        # cmd id field
                if (id.txt == "ARM_PRM_SETDMP")
                {       A = new_interval(id.nxt, A);
                }
                if  (id.txt == "MOB_NAV_PRM_SET")
                {       M = new_interval(id.nxt, M);
                }
                Next;
        }
        If (#CMD_COMPLETE)
        {       id = .nxt;
                id = id.nxt;
                if (id.txt == "ARM_PRM_SETDMP")
                {       close_interval(id.nxt, A);
                }
                if (id.txt == "MOB_NAV_PRM_SET")
                {       close_interval(id.nxt, M);
                }
                Next;
        }
        If (#TLM_TR_ERROR)
        {       id = .nxt;          # ,
                id = id.nxt;        # command identifier
                ts = id.nxt;        # ,
                ts = ts.nxt;        # timestamp

                a  =  A;  while (check_interval(a,  "ARM"))  { a = a.nxt; }
                m = M; while (check_interval(m, "MOB")) { m = m.nxt; }
        }
%}
```

Figure 1 – Cobra script for the interval property

For handling a CMD_DISPATCH event, the script checks if the command identifier is of the right type and then builds a new token sequence as a linked list of intervals (named A or M) for each type of interval, preserving the timestamps.

To handle TLM_TR_ERROR events, the script checks the currently open intervals, traversing the linked list for each type, and check if the property is violated. We have to handle the fact that TLM_TR_ERROR events can appear in the log long after the intervals in which they appear were marked closed. An interval can therefore only be deleted as soon as TLM_TM_ERROR events are seen that are beyond the CMD_COMPLETE timestamp of that interval. In our version of the check, though, we did not delete any closed interval to simplify the processing.

The three helper functions are defined in an initialization segment that ends with a Stop command to shortcut the processing over all tokens, as shown in Figure 2. The initialization also initializes the two linked lists and declares a global variable named *inside* to indicate whether we are within the bounds one of the tracked intervals. Variables need not be declared in the scripting language, with the data types determined by the context in which the variables are used. A detailed description of the Cobra scripting language can be found online [5]. The source code for the tool is available on Github [6].

```
%{
	A = newtok();
	M = newtok();
	inside = 0;
	function new_interval(x, S) {
		y = newtok();
		x = x.nxt;
		y.seq = x.seq;
		y.txt = x.txt;
		y.lnr = x.lnr;
		y.nxt = S;
		return y;
	}
	function close_interval(x, S) {
		x = x.nxt;
		S.prv = x;
	}
	function check_interval(y, x) {
		if (y.seq == 0)
		{	return 0;    # end of list
		}
		b = y.prv;
		if (b.seq == 0             # interval not closed
		||  (y.txt <= ts.txt
		&&   b.txt >= ts.txt))
		{	y.mark++;
			if (y.mark == 1) {
			print x " interval " y.txt " contains TLM_TR_ERROR\n";
			}
			inside++;
			return 0;
		}
		return 1;
	}
	Stop;
%}
```

Figure 2 – Cobra definition of helper functions for processing intervals.

The traversal of the lists of open intervals, when handling a TLM_TR_ERROR event uses function *check_interval* to check if the new timestamp appears within, or is coinciding with, the stored intervals. The function returns zero either when the end of the list is seen, or when the timestamp being checked is found to be within a stored open or closed interval.

Earlier we extended the Cobra tool to allow the analysis of not just source code but also arbitrary inputs, using a new streaming input option that allows the tool to read and process live event streams from the standard input, for indefinite periods of time. This is the option we used for this application.

6. Comparison

Running the *nfer* version of the check on the target MSL event log of 50,000 events, as reported in [1] and [2], labels 4 of 45 TLM_TR_ERROR events as appearing within the designated types of intervals, and the remaining 41 occurrences to appear outside these intervals, requiring warnings to be generated.

When we repeat the verification with an independent check using the formalization in a Cobra script, we find that 6 of the 45 TLM_TR_ERROR events fall within designated intervals, and the remaining 39 are outside. Closer inspection of the intervals shows this to be the correct result. The reason for the incorrect *nfer* result is that the *before* operator from Allen logic only detects intervals where the CMD_COMPLETE events appears *later* in time than the CMD_DISPATCH event. This results in the tool pairing CMD_DISPLATCH events with CMD_COMPLETE events from later, unrelated, intervals in the event stream, that do have higher timestamps.

Performance numbers for the *nfer* checks are included in [1], and show a runtime of 251.1 seconds. Various heuristics were defined in [1] to see how they affected the accuracy of the analysis in return for a reduced runtime. From the approximate runs that give the same (but incorrect) results as the original run without the heuristics, the fastest run with the *nfer* tool took 28.5 seconds. In contrast, the Cobra result is obtained in just 0.25 seconds, or two to three orders of magnitude faster than the *nfer* run on the same event log.

7. Conclusion

This experiment illustrates how difficult it can be to correctly interpret complex statements, especially when they are expressed in a less commonly used logic. In this case, the semantics of Allen Interval logic included some surprises, but the same type of problem has caused problems with the formalization of complex properties in more mainstream logics like LTL as used in, for instance, the Spin model checker. This can lead to inaccuracies that can remain undetected for long periods of time. The flaw that our comparison revealed was unknown to the authors of *nfer* tool, until we tried to find out why the results of our verification runs differed. A simpler formalization of queries can not only be more robust, but as our comparison showed, it can be significantly more efficient to check.

Acknowledgement

We gratefully acknowledge the help of Klaus Havelund for access to the event log that was used for the *nfer* verification, and for his insights in understanding the difference in the results we obtained.

References

[1] S. Kauffman, K. Havelund, and R. Joshi, **"***nfer - A Notation and System for Inferring Event Stream Abstractions,"* In Ylies Falone and Cesar Sanchez editors, 16th Int.l Conf. on Runtime Verification. 23-30 Sep. 2016 - Madrid. Springer-Verlag LNCS, Vol. 10012, pp 235-250.

[2] S. Kauffman, K. Havelund, R. Joshi, and S. Fischmeister*, "Inferring Event Stream Abstractions,"* Formal Methods in System Design, Springer-Verlag, 2018. (Journal version of [1].)

[3] G.J. Holzmann, "*Cobra: a light-weight tool for static and dynamic program analysis,"* Innovations in Systems and Software Engineering, Vol. 13, No. 1, pp. 35-49 (2017)

[4] https://en.wikipedia.org/wiki/Allen's_interval_algebra

[5] http://spinroot.com/cobra/

[6] https://github.com/nimble-code/Cobra/