2003-01

Execution-Based Model Checking of Interrupt-Based Systems

Drusinsky, Doron; Havelund, Klaus

Execution-Based Model Checking of Interrupt-Based Systems

Doron Drusinsky[1]
Naval Postgraduate School
Monterey, CA, USA
ddrusins@nps.navy.mil

Klaus Havelund
Kestrel Technology, NASA Ames Research Center
Moffett Field, CA, USA
havelund@email.arc.nasa.gov

Abstract

Execution-based model checking (EMC) is a verification technique based on executing a multi-threaded/multi-process program repeatedly in a systematic manner in order to explore the different interleavings of the program. This is in contrast to traditional model checking, where a model of a system is analyzed. Several execution-based model-checking tools exist at this point, such as for example Verisoft and Java PathFinder. The most common formal specification languages used by EMC tools are untimed, either just assertions, or linear-time temporal logic (LTL). An alternative verification technique is Runtime Execution Monitoring (REM), which is based on monitoring the execution of a program, checking that the execution trace conforms to a requirement specification. The Temporal Rover and DBRover are such tools. They provide a very rich specification language, being an extension of LTL with real-time constraints and time-series. We show how execution-based model checking, combined with runtime execution monitoring, can be used for the verification of a large class of safety critical systems commonly known as interrupt-based systems. The proposed approach is novel in that: (i) it supports model checking of a large class of applications not practically verifiable using conventional EMC tools, (ii) it supports verification of LTL assertions extended with real-time and time-series constraints, and (iii) it supports the verification of custom schedulers.

1 Introduction

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. Pnueli [9] suggested using LTL for reasoning about concurrent programs. Since then, several researchers have used LTL to state and measure correctness of concurrent programs, protocols, and hardware (e.g., [8]). LTL is an extension of propositional logic where, in addition to the well-known propositional logic operators, there are four future-time operators (Eventually, Always, Until, Next) and four, dual, past-time operators. Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real-time systems [2]. MTL extends LTL by supporting the specification of relative time and real-time constraints. All four LTL future time operators can be constrained by relative time (cycles) and real-time constraints specifying the duration of the temporal operator. This paper describes an additional extension to LTL and MTL suitable for the specification of time-series requirements such as stability, monotonicity, temporal average and sum values, and temporal min/max values. It is then suggested how to combine execution-based model checking with runtime execution monitoring of properties in this logic to verify event-driven interrupt-based software systems.

Execution-based model checking (EMC) is a technique for exploring the possible interleavings of a multi-

[1] CEO of and funded by Time-Rover, Inc., 11425 Charsan Ln., Cupertino, CA 95014, USA. doron@time-rover.com, www.time-rover.com. Time Rover has developed the runtime verification tools Temporal Rover, ATG Rover and DBRover.

threaded/multi-process program by executing the program repeatedly in a systematic manner, exercising a new interleaving for each new repetition. This is in contrast to traditional model checking, where a *model* of a system is exercised systematically. Verisoft [6] is an example of an EMC system, which repeatedly executes a program on its existing execution platform, using partial order reduction techniques to avoid re-executing the same interleavings. Java PathFinder (JPF) [1] is another example, which uses a special JVM as execution platform, and a memory of "visited states" to avoid re-executing interleavings. The specification languages used by such systems are usually restricted to propositional linear temporal logic at the best, not allowing the statement of real-time properties. LTL formulae are translated into Buchi-automata that are exercised together with the executing program. One common way of analyzing an application is to define an *environment generator*, which non-deterministically assigns values to variables that are used by the application, or that non-deterministically generates events to the application in case it is event-driven. Typically EMC suffers from an exponential blow-up in the number of possible interleavings as a function of the number of potentially concurrent processes being verified. This is partly due to the fact that traditionally, EMC considers the worst-case scenario, where **all** interleavings are considered. Techniques are applied (such as partial order reduction) to reduce this number, but the starting point is that all interleavings should be analyzed. In the context of embedded systems, this assumption can be relaxed, as we shall describe.

Runtime Execution Monitoring (REM) is a class of methods of tracking the temporal behavior of an underlying application. REM methods range from simple print statement logging methods to run-time tracking of complete formal requirements (e.g., written in LTL/MTL) for verification purposes. Indeed, first applications of REM were verification oriented where REM was used to track how formal specification requirements are conformed to by the actual executing system. Temporal Rover [3] and, more recently, Java PathExplorer [7], are such systems. Recent adaptations of on-line REM methods enable run time monitoring for non-verification purposes such as temporal business rule checking and temporal security rule checking [4].

Event-driven interrupt-based software systems consist of a collection of tasks, an interrupt system, and a scheduler. Tasks execute based on a schedule pattern defined by the scheduler, and are interrupted from time to time by external stimuli manifested as interrupts. Upon interrupt, given the specific composite state of the interrupt system (manifested by interrupt priority levels and interrupt masking setup), a corresponding task is enabled for execution by the scheduler. Hence, conventional timesharing based concurrency can be considered as a special-case of an interrupt-based system which consists of a single event (the system clock) and an invisible time-sharing scheduler. Conventional EMC tools do not attempt to model the system clock in the environment generator, nor do they model the scheduler as part of the underlying program. Rather, they assume the worst by exercising all possible interleavings of potentially concurrent tasks. Note that unlike desktop or workstation concurrent programs where processes are by definition concurrent, tasks may or may not be concurrent depending on the particular sequence of interrupts that triggers the system during execution; in fact, tasks might be both concurrent and sequential during the same execution. For example, two equal priority interrupt driven tasks, $task_1$ and $task_2$, triggered by external events $event_1$ and $event_2$, respectively, are concurrent if $event_1$ fires while $task_2$ is executing, but might be sequential is $event_1$ fires after $task_2$ completes its invocation and before it is invoked again due to a successive $event_2$ event.

2. Temporal Rover, Real-Time Constraints and Time Series

Temporal Rover [3] uses runtime monitoring of LTL augmented with real-time constraints (MTL) and time-series constraints. With MTL, each temporal operator is potentially qualified by a real-time constraint, such as '`Always`$_{<20}$`commandResult>0`', which means that '`commandResult>0`' must hold every cycle until 20 real-time units in the future. MTL with time-series extends MTL to cater for the specification of relative values of variables and measurements. For example, consider an automotive cruise control application with an embedded Temporal Rover stability assertion requiring "*speed to be 5% stable while cruise is set and not changed"*:

```
TRAssert{
     Always
       ({cruiseSet} ->
           {speed*0.95 < speed' &&
            speed' < speed*1.05}
                Until $speed$
            {cruiseChange || cruiseOff}
       )
}
```

In this example speed is a temporal data variable, which is associated with the Until temporal operator. This association implies that every time the Until operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in the speed variable of this instance of the Until; this value is referred to as the pivot value for this Until node instance. Future speed values used by this particular evaluation of the Until statement are referred to using the prime notation, i.e., as speed'. Hence, if the speed value was 100Kmh when *cruiseSet* is true, then the pivot value for speed is 100, while every subsequent speed is referred to as speed' and must be within 5% of the pivot speed value. Time series constants enable the specification of important categories of requirements such as stability, average, and min-max values.

The DBRover [4] is a remote version of the TemporalRover whereby assertions are monitored on a remote machine, using HTTP, sockets, or serial communication with an underlying client/target application. The DBRover includes a graphical temporal rule editor, a temporal rule simulator, and a temporal rule execution engine based on the TemporalRover code generator. In run-time, the DBRover listens for messages from client application(s) and evaluates corresponding temporal assertions. DBRover monitoring is performed on-line, namely, the DBRover operates in tandem with the target program, and re-evaluates assertions every cycle while not storing an ever-growing history trace.

3. Execution-Based Model Checking of Interrupt-Based Systems

Fig. 1 illustrates the architecture for EMC of interrupt-based systems. In this configuration, EMC is performed on the target board; the environment generator communicates with the target board using the same interface (for event interrupts and data), as does the real environment. The environment generator and the target execute in real-time, where the environment generator generates real-time stimuli for the target. The environment generator in this architecture differs from the environment generator of traditional EMC systems in that:

1. The environment generator executes in real-time and is also extended with delay commands, thereby enabling environment programs to generate their stimuli according to a real-time plan. Traditional EMC does not model real-time constructs in the environment, and does not mimic the environment in real-time.

2. The environment generator generates events instructing (via interrupts) the embedded scheduler to fire certain tasks with particular timing. This empowers the tester to generate event sequences that force varying patterns of concurrency. For example, consider the two-task example provided in the introduction. Consider that each task executes for less than one second; the tester might know that $event_2$ cannot, under certain circumstances, fire within one second after $event_1$ and that consequently $task_2$ cannot interrupt $task_1$; all the tester needs to do is to force this real-time constraint in the event generation segment of the environment generator. In contrast, traditional EMC does not model the scheduler or real-time. Hence, to achieve similar performance using traditional EMC, a tester needs analyze the system and conclude that the tasks cannot be concurrent, and then rely on complicated programming of process priority levels to achieve the same effect.
3. Since EMC is performed on the target system in real-time, using real interrupts, a real scheduler, and real interrupt system, it enables the verification of interrupt latency requirements (i.e., requirements pertaining to the delay between interrupt detection time and the time a corresponding service task fires).

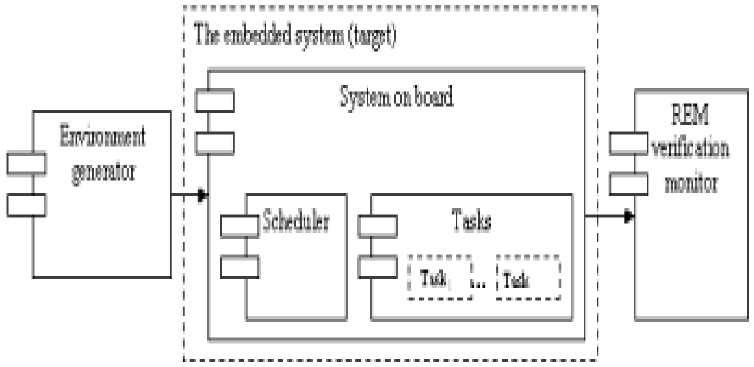

Figure 1. UML component diagram for EMC of Interrupt-based systems

Hence, Fig. 1 illustrates the application of EMC as a sophisticated test generator for embedded interrupt-based systems. Having external events represented as part of the environment program enables a focused and effective verification effort where environment programs reflect the tester's knowledge of the embedded system and the environment.

Note that conventional EMC verification time is exponential in the number of potentially concurrent tasks to be verified. Many embedded system programs, however, consist of ten or more (possibly small) tasks. Therefore, the brute force approach taken by conventional EMC tools does not scale well when applied to such embedded systems. In contrast, having the ability to check specific interleavings using event generation in the environment generator enables more efficient verification of such systems.

Unlike non-embedded software, where process scheduling is for the most part invisible to the programmer, embedded system programs often include custom schedulers. An embedded custom scheduler is responsible for scheduling event-tasks based on pending, yet un-serviced, interrupts as well as periodic background tasks. Note how the suggested architecture enables explicit modeling of the scheduler. Hence, a novel feature of the suggested architecture is its ability to verify custom schedulers; in contrast, traditional EMC must either assume the worst, i.e., that all possible interleavings of all tasks need to be verified, or must use the priority system to programmatically induce the same effect, resulting in a program that is behaviorally different than the original, thereby discrediting the verification effort.

On the verification side of Fig. 1, the architecture incorporates REM of LTL augmented with real-time and time-series constraints, as described in section 2 and [5]. Real-time measurements are provided to the REM tool from the embedded real-time system, enabling the REM tool to perform real-time constraint evaluation for temporal assertions.

4. Conclusion

Embedded real-time applications pose unique challenges to the verification community. In these systems, real-time issues, such as correctness of the scheduler, and event-response latencies, are all of primary concern. Nevertheless, EMC can be effectively used to verify such systems, provided that it is used in a specific manner and combined with REM of specifications augmented with real-time constraints.

References

1. G. Brat, K. Havelund, W. Visser - *Model Checking Programs,* Proc. 15th IEEE International Conference on Automated Software Engineering (ASE), IEEE CS Press.

2. E. Chang, A. Pnueli, Z. Manna - *Compositional Verification of Real-Time Systems*, Proc. 9'th *IEEE Symp. On Logic In Computer Science*, 1994, pp. 458-465.

3. D. Drusinsky - *The Temporal Rover and ATG Rover*. Proc. Spin 2000 Workshop, Springer Lecture Notes in Computer Science, 1885, pp. 323-329.

4. D. Drusinsky, J. Fobes - *Real-time, On-line, Low Impact, Temporal Pattern Matching*, 7th World Multi-conference on Systemics, Cybernetics and Informatics, Orlando FL, 2003; accepted for publication.

5. D. Drusinsky - Monitoring Temporal Rules Combined with Time Series, Computer Aided Verification Conference 2003; accepted for publication.

6. P. Godefroid *Model Checking for Programming Languages using VeriSoft*, Proc. of the 24th ACM Symposium on Principles of Programming Languages, pp. 174-186, 1997.

7. K. Havelund, G. Rosu - *Monitoring Java Programs with Java PathExplorer,* Proc. of the 1st International Workshop on Runtime Verification (RV'01), Elsevier Science, Electronic Notes in Theoretical Computer Science 55(2), pp. 97-114, 2001.

8. Z. Manna, A. Pnueli - *Verification of Concurrent Programs: Temporal Proof Principles*, Proc. of the Workshop on Logics of Programs, Springer LNCS, 1981 pp. 200-252.

9. A. Pnueli - *The Temporal Logic of Programs*, Proc.18th IEEE Symp. on Foundations of Computer Science, pp. 46-57, 1977.