# Implementing Runtime Monitors[⋆]

Klaus Havelund

Jet Propulsion Laboratory, California Institute of Technology, USA

Runtime Verification (RV) consists in part of checking program executions against formalized requirements. The field has within the last decade specifically focused on various notions of state machines, temporal logics such as future and past time Ltl (Linear Temporal Logic), regular expressions, grammars, and rule-based systems. Using the high-level JVM-based Scala programming language [12], which combines object-oriented and functional programming, we will illustrate how some of these systems can be implemented in an elegant manner. The logics differ in expressiveness as well as in ease of use. Some implementation schemes are more efficient than others, whereas some are easier to implement and therefore easier to adapt to changing user requirements.

## 1 Rewriting Systems versus Automata

Temporal logic can be executed for runtime verification purposes as a rewriting system, on an event-by-event manner [6]. Consider for example the until operator $p\ \mathcal{U}\ q$ ($p$ until $q$) in Linear Temporal Logic (Ltl) [11]. The formula $p\ \mathcal{U}\ q$ means: $q$ must eventually be true and until then $p$ must be true. The operator satisfies the following recursive equation:

$$p\ \mathcal{U}\ q \quad \equiv \quad q \vee (p \wedge \bigcirc(p\ \mathcal{U}\ q))$$

reading: $p\ \mathcal{U}\ q$ is true now if $q$ is true now or: $p$ is true now and in the next step $p\ \mathcal{U}\ q$ is true ($\bigcirc\psi$ means $\psi$ is true in the next step). This equation can form the basis of an implementation based on rewriting. That is, given a formula and a new incoming event, the formula can be rewritten into a new formula that has to hold in the next state. A special case of rewriting is alternating automata [13] where the target of a labelled transition from a single state can be a formula constructed from conjunctions and disjunctions of states. In general, a more efficient approach is to translate Ltl to non-deterministic or deterministic automata [7]. See also [9] for an account on teaching such concepts. However, interestingly, rule-based systems, such as Ruler [2], which are based on a form of rewriting, seem of real practical interest and will be discussed in the presentation.

## 2 Propositional versus Data Parameterized Logics

Very early monitoring systems supported monitoring of propositional specifications. For example properties such as the following (for a martian rover):

[⋆] Part of the work to be covered in this presentation was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

$$\Box(open \rightarrow \neg send\, \mathcal{U}\ close)$$

reading: it is always the case, that when a file is opened, it should eventually be closed, and it should not be sent to ground before then. However, more recent monitoring systems handle data parameterization, allowing formulas to be specific about data parameters to formulas. A data parameterized version of the above formula could for example be the following, stating that for all files $f$, if $f$ is opened then eventually it should be closed and not sent before then:

$$\forall f : File\ \bullet\ \Box(open(f) \rightarrow \neg send(f)\, \mathcal{U}\ close(f))$$

We shall present two implementations of parameterized logics, one based on rewriting and one based on automata. The rewriting solution is implemented in the TraceContract system [1], which is an internal DSL extending the Scala programming language with linear temporal logic and state machines. The automata solution is an abstract representation of the Mop system [3]. A characteristic difference between the two systems is that in TraceContract data and control are merged into the same representation, whereas in Mop data are handled separately from the temporal logic. This makes Mop more efficient and allows Mop to provide a parameterization solution for any kind of propositional logic. A logic becomes a plugin in other words. We shall outline the advantages and disadvantages of the two approaches.

## 3 External versus Internal DSLs

We can consider a logic for runtime verification as a DSL (Domain Specific Language) [5]. One distinction that we shall emphasize is that of external DSLs versus internal DSLs. An external DSL is a specialized language focusing only on the specific task it is meant to handle, in this case trace analysis. An internal DSL (or embedded DSL as it is sometimes referred to) is an extension of a general purpose programming language, in this case we consider an extension of Scala. An external DSL is usually very convenient for expressing solutions to *typical* problems, and is usually very efficiently implemented. An internal DSL may, however, deliver the expressiveness needed for special case problems that go beyond the typical case. In the extreme, an internal DSL offers the underlying programming language in case this is needed, whereas the added DSL layer can be used only for those situations where it is appropriate. An internal DSL is usually easier to implement due to the re-use of language concepts provided by the underlying programming language, and due to the avoidance of a parser. An internal DSL, on the other hand, might be less suited for automated analysis, such as for example DSL specific type checking. A separate issue is that an external DSL might be easier to learn for users not familiar with the underlying programming language, whereas an internal DSL might be preferred by programmers.

## 4 Code Instrumentation versus Design by Contract

A runtime monitor is typically developed with the purpose to monitor an executing program. Part of an RV environment is therefore traditionally an automated method to instrument code to emit events or states to the runtime monitor. We will briefly outline how Aspect-Oriented Programming (AOP) can be used to instrument software programs for monitoring, specifically with examples in using AspectJ [8] for instrumenting Java programs.

Usually monitors are written in separate files or modules, very similar to how aspects are written in separate aspect modules in AOP. Runtime verification can, however, be more tightly integrated with programming. The classical paradigm of design by contract suggests that methods/functions are developed with contracts in the form of pre/post conditions and invariants. This paradigm is supported by various programming languages, such as Eiffel [4]. One can easily imagine such contracts being extended with trace contracts stating temporal properties about interfaces, for example stating policies about the ordering of method calls. We shall illustrate how design by contracts in Scala as outlined in [10] can be extended with such trace contracts.

## References

1. H. Barringer and K. Havelund. TraceContract: A Scala DSL for trace analysis. In *17th International Symposium on Formal Methods (FM'11), Limerick, Ireland, June 20-24, 2011. Proceedings*, volume 6664 of *LNCS*, pages 57–72. Springer, 2011.
2. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: from Eagle to RuleR. *J. Log. Comput.*, 20(3):675–706, 2010.
3. F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *LNCS*, pages 246–261, 2009.
4. Eiffel. http://www.eiffel.com.
5. M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, 2010.
6. K. Havelund and G. Rosu. Monitoring programs using rewriting. In *16th ASE conference, San Diego, CA, USA*, pages 135–143, 2001.
7. G. J. Holzmann. *The Spin Model Checker – Primer and Reference Manual*. Addison-Wesley, 2004.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.
9. M. Leucker. Teaching runtime verification. In *Runtime Verification - Second Int. Conference, RV'11, San Francisco, California, USA, September 27-30, 2011. Proceedings*, volume TBD of *LNCS*. Springer, 2011.
10. M. Odersky. Contracts for Scala. In *Runtime Verification - First Int. Conference, RV'10, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.
11. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.
12. Scala. http://www.scala-lang.org.
13. M. Y. Vardi. Alternating automata and program verification. In *Computer Science Today. Proceedings*, volume 1000 of *LNCS*, pages 47–485. Springer, 1995.