MONPOLY: Monitoring Usage-control Policies*

David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu

Computer Science Department, ETH Zurich, Switzerland

1 Introduction

Determining whether the usage of sensitive, digitally stored data complies with regulations and policies is a growing concern for companies, administrations, and end users alike. Classical examples of policies used for protecting and preventing the misuse of data are history-based access-control policies like the Chinese-wall policy and separation-of-duty constraints. Other policies from more specialized areas like banking involve retention, reporting, and transaction requirements. Simplified examples from this domain are that financial reports must be approved at most a week before they are published and that transactions over $10,000 must be reported within two days.

In the context of IT systems, compliance checking amounts to implementing a process that monitors, either online or offline, other processes. Such a monitor needs to temporally relate actions performed by the other processes and the data involved in these actions. Since the number of data items processed in IT systems is usually huge at each point in time and cannot be bounded over time, monitoring algorithms, in particular for propositional temporal logics, are of limited use for compliance checking.

In this paper, we present our monitoring tool MONPOLY for compliance checking. Policies are given as formulas of an expressive safety fragment of metric first-order temporal logic (MFOTL). The first-order fragment is well suited for formalizing relations on data, while the metric temporal operators can be used to specify properties depending on the times associated with past, present, and even future system events. MONPOLY processes a stream of system events with identifiers representing the data involved and reports policy violations. In the following, we describe MONPOLY and its features in more detail. We also briefly report on case studies and discuss related tools.

2 Tool Description

We describe MONPOLY's input and output and its theoretical underpinnings. Afterwards we give an overview of its implementation.

Input and Output. MONPOLY takes as command-line input a signature file, a policy file, and a log file. It outputs violations of the specified policy. We illustrate MONPOLY's input and output with an example.

* This work was funded by the Nokia Research Center, Switzerland. The authors thank the Nokia team in Lausanne for their support.

An MFOTL formalization of the policy that financial reports must be approved at most a week before they are published is

$$\Box \forall r.\, \mathit{publish}(r) \rightarrow \blacklozenge_{\leq 7\,\mathrm{days}}\, \mathit{approve}(r)\,. \tag{1}$$

We use $\Box$ for the temporal operator "always in the future" and $\blacklozenge$ for "sometimes in the past." Moreover, to express timing constraints, we attach metric constraints to these operators like $\leq 7\,\mathrm{days}$ for "within 7 days." The concrete textual input to MONPOLY for the policy (1) is

```
publish(?r) IMPLIES ONCE[0,7d] approve(?r),
```

where the arities of the predicates and the types of the arguments are specified in a signature file. The outermost temporal operator $\Box$ is implicit in the input to MONPOLY, since policies should hold at every point in time. Moreover, in our example the variable `?r` is free. This is because MONPOLY should output the reports that were published but either not approved at all or the approval was too early. That is, MONPOLY outputs for every time-point the satisfying valuations of the negated formula

```
publish(?r) AND HISTORICALLY[0,7d] NOT approve(?r).
```

A log file consists of a sequence of time-stamped system events, which are ordered by their time-stamps. Events assumed to have happened simultaneously are grouped together. For example, according to the log file

```
@1307532861 approve (52)
@1307955600 approve (63)
            publish (60)
@1308477599 approve (87)
            publish (63) (52)
```

the report with the number 52 was approved at time-point 0 with the time-stamp 1307532861 (2011-06-08, 11:34:21 in UNIX time) and it was published at time-point 2 with the time-stamp 1308477599 (i.e., on 2011-06-19) together with the report 63, which was approved on 2011-06-13.

MONPOLY processes the log file incrementally and outputs for each time-point all policy violations. For the above input, MONPOLY reports the following violations:

```
@1307955600 (time-point 1): (60)
@1308477599 (time-point 2): (52)
```

Publishing the reports 60 and 52 each violates the policy (1). Report 60 was never approved and report 52 was approved too early. MONPOLY does not produce an output for time-point 0, since there is no policy violation at this time-point.

Foundations. MONPOLY implements our monitoring algorithm [7] for time-stamped temporal structures with finite relations. To effectively monitor properties specified in MFOTL, this algorithm only handles a safety fragment of MFOTL. Namely, the formulas must be of the form $\Box \Phi$, where the temporal future operators occurring in Φ are bounded, i.e., the attached metric constraints

restrict these operators so that they range only over finitely many time-points. Roughly speaking, the monitoring algorithm iteratively processes the log file and determines for each given time-point the satisfying valuations of the formula $\neg\Phi$. Since Φ is bounded, only finitely many time-points need to be taken into account. However, the evaluation at a time-point is delayed by the monitoring algorithm until it reads the data of the relevant future time-points.

To efficiently determine at each time-point the violating elements of Φ, we evaluate the formula $\neg\Phi$ bottom-up and store intermediate results in finite relations. These are updated in each iteration and reused in later iterations. We require that $\neg\Phi$ can be rewritten to a formula so that the intermediate results are always finite relations. In particular, the use of negation and quantification is syntactically restricted. These restrictions are adapted from database query evaluation [1]. Before starting the monitoring process, MONPOLY checks whether the given formula has these properties.

Implementation. MONPOLY is written in the OCaml programming language. The code is mainly functional, making only sparse use of OCaml's imperative programming-language features and not using OCaml's object layer.

The code is structured in modules. For instance, there are modules for operations on MFOTL formulas, relations, and first-order structures. There are also modules for parsing formulas and log files. Finally, there is a module that implements the monitoring algorithm [7]. Since the algorithm manipulates relations extensively, the data structure used to represent relations has a huge impact on the monitor's efficiency. Currently, MONPOLY uses the data type for sets from OCaml's standard library, which is implemented using balanced binary trees.

Since the implementation is modular, MONPOLY can easily be modified and extended. For example, modifying MONPOLY so that it processes log files in another format is straightforward, as is using other data structures for representing and manipulating relations. The source code of MONPOLY is publicly available from the web page `http://projects.developer.nokia.com/MonPoly`.

3 Experimental Evaluation

We have evaluated MONPOLY's performance on several policies on synthetically generated data. For example, for the simple publishing policy (1), MONPOLY processes a log file with 25,000 entries in 0.4 seconds on a standard desktop computer. It uses 30 MBytes of memory. Monitoring the more complex policy where approvals must be signed by managers[1] takes MONPOLY 2.25 seconds, where 60 MBytes of memory are used. Other examples of our evaluation include MFOTL formalizations of transaction policies, the Chinese-wall policy, and

[1] The MFOTL formalization of this policy is $\Box\forall r.\,\forall a.\,\mathit{publish}(r,a) \rightarrow \blacklozenge_{\leq 7\,\mathrm{days}}\,\exists m.\,\underline{\mathit{manager}}(m,a) \wedge \mathit{approve}(r,m)$. Here, $\underline{\mathit{manager}}(m,a)$ encodes that m is a manager of the accountant a, which might change over time. It abbreviates the formula $\neg \mathit{manager}_f(m,a)\ \mathsf{S}\ \mathit{manager}_s(m,a)$, where S denotes the temporal past operator *since* and the predicates $\mathit{manager}_s$ and $\mathit{manager}_f$ represent system events that mark the start and the finish of the relation of m being a's manager.

separation-of-duty constraints, and are given in [6]. MONPOLY performs significantly better than our previous prototype implementation in Java, used in [6]. A reason for this improvement is that the fragment MONPOLY handles is slightly more restrictive but formulas in this fragment are evaluated more efficiently.

We have also used MONPOLY in a case study with industry: monitoring the usage of data within Nokia's data-collection campaign.[2] The campaign collects contextual information from cell phones of about 180 participants, including phone locations, call and SMS information, and the like. Given the data's high sensitivity, usage-control policies govern what actions may and must not be performed on the data. We formalized these policies in MFOTL, obtaining 14 formulas. We used MONPOLY to check them on different log files, each corresponding to roughly 24 hours of logged data. The largest logs contain around 85,000 time-points and one million system events. On such log files, the running times for the different policies on a standard desktop computer range from 10 seconds for simple access-control policies to 7 hours for complex policies employing nested temporal operators. The memory requirements are also modest: even for the complex policies, MONPOLY never used more than 500 MBytes of memory and these peaks occurred infrequently. Further details on the policies, the setup, MONPOLY's performance, and our findings in this case study are given in [5].

4 Related Tools

MONPOLY targets automated compliance checking in IT systems where actions are performed by distributed and heterogeneous system components. Monitoring tools for related applications are BeepBeep [10], Orchids [13], Monid [12], and LogScope [3]. BeepBeep monitors a web-client application for the conformance of its communication with the web service's interface specifications expressed in LTL-FO$^+$, a first-order extension of the linear-time temporal logic LTL. Orchids is a monitor for intrusion detection. It searches in an event stream for attack patterns, which are specified in a variant of future-time temporal logic and compiled into non-deterministic automata for fast pattern matching. Monid, similar to Orchids, is a tool for intrusion detection. It is based on the monitoring framework Eagle [2], where properties are specified by systems of parametrized equations with Boolean and temporal operators and a fixpoint semantics. LogScope can be seen as a restriction of RuleR [4]—a conditional rule-based system with an algorithm for runtime verification—tailored for log-file analysis. Properties in LogScope are given as conjunctions of data-parametrized temporal patterns and finite-state machines. These tools differ from MONPOLY in their specification languages and their underlying monitoring algorithms. For instance, LTL-FO$^+$ does not support temporal past operators but supports unbounded temporal future operators. Quantification in LTL-FO$^+$ is more restrictive than in the monitorable MFOTL fragment of MONPOLY, since quantified variables only range over data elements that appear in the system event that is currently processed. BeepBeep's monitoring algorithm for LTL-FO$^+$ is based on an extension of a tableaux construction for LTL.

[2] See `http://research.nokia.com/page/11367` for details on the campaign.

Other runtime-verification approaches, implemented in tools like Temporal Rover [9], Lola [8], J-LO [14], and MOP [11], have primarily been developed and used for monitoring the execution of programs. Programs are instrumented so that relevant actions, like procedure calls and variable assignments, either directly trigger the inlined monitors or are forwarded to external monitors. Evaluating and comparing the performance of the different underlying monitoring algorithms experimentally remains as future work.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison Wesley, 1994.
2. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. of the 5th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, vol. 2937 of *LNCS*, pp. 44–57. Springer, 2004.
3. H. Barringer, A. Groce, K. Havelund, and M. Smith. Formal analysis of log files. *J. Aero. Comput. Inform. Comm.*, 7:365–390, 2010.
4. H. Barringer, D. E. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. *J. Logic Comput.*, 20(3):675–706, 2010.
5. D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. Monitoring usage-control policies in distributed systems. In *Proc. of the 18th Int. Symp. on Temporal Representation and Reasoning*, to appear.
6. D. Basin, F. Klaedtke, and S. Müller. Monitoring security policies with metric first-order temporal logic. In *Proc. of the 15th ACM Symp. on Access Control Models and Technologies*, pp. 23–33. ACM Press, 2010.
7. D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proc. of the 28th IARCS Annu. Conf. on Foundations of Software Technology and Theoretical Computer Science*, vol. 2 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.
8. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *Proc. of the 12th Int. Symp. on Temporal Representation and Reasoning*, pp. 166–174. IEEE Computer Society, 2005.
9. D. Drusinsky. The Temporal Rover and the ATG Rover. In *Proc. of the 7th Int. SPIN Workshop*, vol. 1885 of *LNCS*, pp. 323–330. Springer, 2000.
10. S. Hallé and R. Villemaire. Browser-based enforcement of interface contracts in web applications with BeepBeep. In *Proc. of the 21st Int. Conf. on Computer Aided Verification*, vol. 5643 of *LNCS*, pp. 648–653. Springer, 2009.
11. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Int. J. Softw. Tools Technol. Trans.*, to appear.
12. P. Naldurg, K. Sen, and P. Thati. A temporal logic based framework for intrusion detection. In *Proc. of the 24th IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems*, vol. 3235 of *LNCS*, pp. 359–376. Springer, 2004.
13. J. Olivain and J. Goubault-Larrecq. The Orchids intrusion detection tool. In *Proc. of the 17th Int. Conf. on Computer Aided Verification*, vol. 3576 of *LNCS*, pp. 286–290. Springer, 2005.
14. V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *Proc. of the 5th Workshop on Runtime Verification*, vol. 144 of *ENTCS*, pp. 109–124. Elsevier Science Inc., 2006.