A Real-Time, Flexible Logging and Monitoring Infrastructure for MonPoly

Jonas Degelo

Advisor:
François Hublet
Professor:
Prof. Dr. David Basin

Bachelor's Thesis

Information Security Group
Department of Computer Science
ETH Zürich
February 2023

1 Introduction

Our digital world consists of many hardware and software systems. These systems are continuously performing a lot of actions. For a variety of reasons one might want to monitor those actions and make sure that they do not violate some predefined specification. One way to achieve this is to log relevant actions and analyze these logs. Such monitoring is part of the field of Runtime Verification (RV) [1]. The analysis of the logs can either be done *online* while the system is running or it can be done *offline* after the system has terminated.

Consider a social media site. It is bound by an increasing number of privacy laws and regulations. Let a hypothetical piece of regulation be that a user's location information may not be used to tailor advertisements to that user unless the user gave specific permission. Then the site could log every time instance when a user's location data is accessed and the purpose of the access. In words the predefined specification the site wants to check then could be: "If a user's location data is accessed and the purpose of the access is for tailoring advertisements, the user must have previously given permission for there location data to be used for advertising purposes".

MonPoly [6] is a tool for such runtime monitoring. It can do both online and offline monitoring. For online monitoring it accepts new events via standard input. For offline monitoring it can read a timestamped log file that was generated during the runtime of a system. It uses Metric First-Order Temporal Logic (MFOTL) [5, 3, 7] as a formal specification language, which we will introduce in the background section.

MonPoly in its current state has some limitations that we want to improve. For one, online monitoring can not easily be done on a different machine from which the system is running on. This does not fit well with the way many modern systems operate. Modern systems are often very distributed. It is common that different functions of a system run on different machines. Be that physical or virtual ones. Often time multiple machines also perform the same kind of operation. Take caching servers as an example. MonPoly in its current state does not fit well into this world of interconnected microservices.

Another issue that MonPoly faces currently is data portability. MonPoly can store its execution state to disk before stopping. It can also restore that state, but only on the same system as the way it stores the state is tied to the physical memory configuration of the system. We would like the ability to have MonPoly run on one system, then shut it down and restart it on a different system where it resumes with the same state that the monitor had before shutting down on the first system.

Further there is no way to update the monitored policy. We aim to offer a first method for policy changes with minimal overhead in time and compute power.

1.1 Our Contributions

We extend MonPoly with a web based backend written in Python using Flask [**empty citation**]. This backend provides a REST API [**empty citation**] to MonPoly. Packaging MonPoly as a web app and offering a new API offers greater flexibility in how MonPoly can be used.

Through this backend we also connect MonPoly to a database. MonPoly works with timestamped and tabular data. We make use of this fact for the choice of database. The temporal nature of the data leads us to time series databases [**empty citation**], which, as the name implies, are optimized for temporal data. By far the most common type of databases are relational databases. This is a great coincidence, because relational databases make extensive use of tables for storing data. We looked at a few different relational time series databases. In the end we opted for QuestDB [9].

The addition of a database gives us more options in terms of data portability. The state of MonPoly is now not only stored in the system memory of the system currently running MonPoly, but also in the database.

This allows us to stop MonPoly on one system and resume the monitoring on a different system, by querying the database. We make use of relative intervals [**empty citation**] to get a good over approximation of the data needed to continue monitoring a specific formula.

We use the database to offer a first version of a policy change by stopping the monitor and starting a new one with the events within the relative interval of the new policy already loaded.

2 Background

2.1 Metric First-Order Temporal Logic

As mentioned in the introduction, Metric First-Order Temporal Logic (MFOTL) [5, 3, 7] is used as a policy specification language by MonPoly. Here we give a quick overview of MFOTL. MFOTL well suited to express a variety of policies one might want to monitor. It combines First Order Logic (FOL) with metric temporal operators. FOL provides us with common logic operators like $\wedge$ ("and"), $\vee$ ("or"), and $\neg$ ("not") as well as quantifiers $\forall$ ("for all") and $\exists$ ("exists"). The metric temporal operators in MFOTL are $\mathcal{U}_I$ ("until"), $\mathcal{S}_I$ ("since"), $\bullet_I$ ("previous"), and $\bigcirc_I$ ("next"). These operators can be used to construct further syntactic sugar operators such as $\blacklozenge_I$ ("once"), $\Diamond_I$ ("eventually"), $\Box_I$ ("always"), and $\blacksquare_I$ ("historically"). See Hublet et al. 20222 [8] for the concrete derivations of these additional operators. The metric aspect of these operators is the interval I they are bound by. This interval denotes a time frame in which the formula needs to be satisfied.

Metric First-Order *Dynamic* Logic (MFODL) [2] is an even more expressive specification language than MFOTL. MFODL introduces the notion of regular expressions. For an exact definition of these regular expressions and the two new operators they introduce see figure 4 of Basin et al. 2020 [2]. Similarly

to how $\blacklozenge_I$, $\Diamond_I$, $\Box$, and $\blacksquare$ can be derived from the four core operators $\mathcal{U}_I$, $\mathcal{S}_I$, $\bullet_I$, and $\bigcirc_I$, these core operators could theoretically be replaced by the two new regular expression operators. The exact conversion can also be seen in Basin et al. 2020 [2]. In practice, it is often useful to keep the basic temporal operators as we can apply specialized optimizations to them that cannot be done with regular expressions.

In Basin et al. 2015 [4] they extended MFOTL with aggregations. Aggregation operations like SUM are commonly seen in database contexts. When considering an example like a monthly spending limit for a credit card it becomes clear how aggregations can be useful in policy monitoring.

2.2 MonPoly

MonPoly [6] is a policy monitoring tool written in OCaml that supports MFOTL with aggregations and in its newest iterations it also has support for MFODL. MonPoly can monitor a fragment of MFOTL/MFODL where all future operators must be bounded. One major exception to that rule is an (implicit) always operator around the desired policy.

Let's return to the social media example from the introduction and look at how we would go about monitoring that policy with MonPoly. We recall our description in words: "If a user's location data is accessed and the purpose of the access is for tailoring advertisements, the user must have previously given permission for there location data to be used for advertising purposes" In MonPoly a policy is tied to a signature. A signature can be compared with a database schema and describes the arity and types of possible events. So let's consider a possible signature for our example:

```
loc_accessed(user_id: int, purpose: string)
perm_granted(user_id: int)
perm_revoked(user_id: int)
```

This is a basic signature with 3 predicates. The first one means that a users location data has been used for a specified purpose. The last two events get triggered when a user either grants or revokes permission for their location data to be used for advertising purposes. Let's now define the policy in a formal manner.

$$\Box((\exists x \cdot \texttt{loc\_accessed(i, x)} \wedge (x = \text{"advertising"})) \implies (\blacklozenge_{[0,\infty)}\texttt{perm\_granted(i)} \wedge \neg(\texttt{perm\_revoked(i)}\mathcal{S}_{[0,\infty)}\texttt{perm\_granted(i)})))$$

For MonPoly we first get rid of the surrounding $\Box$, because MonPoly implicitly adds an always-operator around any policy. The remaining formula in MonPoly syntax is the following:

```
(EXISTS x. (loc_accessed(i, x) AND x = "advertising"))
IMPLIES
(
```

```
    (ONCE[0,*) perm_granted(i))
    AND
    (NOT (perm_revoked(i) SINCE[0,*) perm_granted(i)))
)
```

While MonPoly cannot actually monitor this formula directly, it can monitor the negation of this formula. For this on can use the `-negate` flag when running MonPoly.

2.3 Time Series Databases

Time series databases are a class of databases optimized for timestamped data. For example, they optimize for data retrieval within a certain time range. With the advance of internet of things devices with built-in sensors time series databases are experiencing explosive growth. And as we have established they happen to fit well with our monitoring goals. There are many different options of time series databases available. We were looking for something with good performance, good support for tables of data, and good usability. We have opted for QuestDB [9].

It has a column-based storage model [12]. QuestDB supports the PostgreSQL wire protocol [11] for querying and inserting data. It further provides a REST API and has a web console for both inserting and querying data. For best performance it supports the InfluxDB Line Protocol [10] with client libraries for most popular modern programming languages. QuestDB itself is written in Java, open source, and licensed under the Apache 2.0 license.

References

[1] Ezio Bartocci et al. *Lectures on Runtime Verification*. Ed. by Ezio Bartocci and Yliès Falcone. Vol. 10457. Springer International Publishing, 2018. ISBN: 978-3-319-75631-8. DOI: 10.1007/978-3-319-75632-5. URL: http://link.springer.com/10.1007/978-3-319-75632-5.

[2] David Basin et al. "A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic". In: *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part I 10*. Springer. 2020, pp. 432–453.

[3] David Basin et al. "Monitoring Metric First-Order Temporal Properties". In: *Journal of the ACM (JACM)* 62 (2 2015), pp. 1–45. ISSN: 0004-5411.

[4] David Basin et al. "Monitoring of Temporal First-Order Properties with Aggregations". In: *Formal methods in system design* 46 (2015), pp. 262–285.

[5] David Basin et al. "Runtime Monitoring of Metric First-Order Temporal Properties". In: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[6] David A Basin, Felix Klaedtke, and Eugen Zalinescu. "The MonPoly Monitoring Tool." In: *RV-CuBES* 3 (2017), pp. 19–28.

[7] Jan Chomicki. "Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding". In: *ACM Transactions on Database Systems (TODS)* 20 (2 1995), pp. 149–186. ISSN: 0362-5915.

[8] François Hublet, David Basin, and Srđan Krstić. "Real-Time Policy Enforcement with Metric First-Order Temporal Logic". In: Springer, 2022, pp. 211–232.

[9] *QuestDB*. Accessed: 2023-01-24. URL: `https://questdb.io/`.

[10] *QuestDB - InfluxDB Line Protocol*. Accessed: 2023-01-24. URL: `https://questdb.io/docs/reference/api/ilp/overview/`.

[11] *QuestDB - Postgres Wire Protocol*. Accessed: 2023-01-24. URL: `https://questdb.io/docs/develop/insert-data/#postgresql-wire-protocol`.

[12] *QuestDB - Storage Model*. Accessed: 2023-01-24. URL: `https://questdb.io/docs/concept/storage-model/`.