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

Jonas Degelo

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

Bachelor's Thesis

Information Security Group
Department of Computer Science
ETH Zürich
February 2023

Contents

1 Introduction

1.1 Motivation

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 [7] is a tool for such runtime monitoring. It can perform 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, 8] 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. These can be physical or virtual machines and more and more applications are also containerized with technologies like Docker. Oftentimes multiple machines also perform the same kind of operation, e.g. caching servers. 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 built-in way to update the monitored policy. We aim to offer a first method for policy changes with minimal overhead in time and compute power.

We improve MonPoly in these aspects by building a wrapper that connects it with a database and also provides a new, more web friendly, interface to MonPoly.

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 [11].

1.2 Our Approach

1.3 Contributions

We extend MonPoly with a web based wrapper written in Python using Flask [10]. This wrapper provides a REST API [9] to MonPoly. Packaging MonPoly as a web app and offering a new API that offers greater flexibility in how MonPoly can be used.

Through this wrapper we connect MonPoly to a database. The addition of a database gives us more options in terms of data portability. The state of the database is consistent with that of the monitor.

The database connection 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 [6] 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.

Finally we have made a few additions to MonPoly itself that were needed for our wrapper. We added flags to MonPoly to print the schema of a given signature in SQL as well as JSON format. One of our aims was fault tolerance and for this we added some options to keep that would keep the monitor running instead of exiting when encountering certain issues. For the policy change we need to potentially reload a lot of events into the monitor. We added an option to first read events from a file and then switch to standard input. Previously the monitor would either read a file and then stop or continuously read from standard input. Another addition are capabilities to get the relative interval of a MFOTL formula and also to get the relative intervals of predicates in a formula. We have begun work on a different method of doing a policy change by changing only parts of a formula while MonPoly keeps running and can keep the state of the formula parts that are unchanged.

2 Background

2.1 Metric First-Order Temporal Logic

As mentioned in the introduction, Metric First-Order Temporal Logic (MFOTL) [5, 3, 8] is used as a policy specification language by MonPoly. Here we give a quick overview of MFOTL. MFOTL is 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"), $\lozenge_I$ ("eventually"), $\square_I$ ("always"), and $\blacksquare_I$ ("historically"). See Basin et al. 2015 [3] 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.

Basin et al. 2008 [5] define the syntax and semantics of MFOTL.

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 $\blacklozenge_I$, $\lozenge_I$, $\square$, 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.

Basin et al. 2015 [4] extends 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 [7] 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.

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

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:

```
loc_accessed(i, "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.

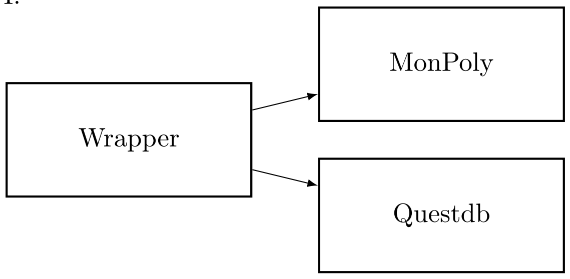
2.3 Time Series Databases

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 [11].

QuestDB uses a column-based storage model [14]. It supports the PostgreSQL wire protocol [13] 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 [12] 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.

3 Architecture

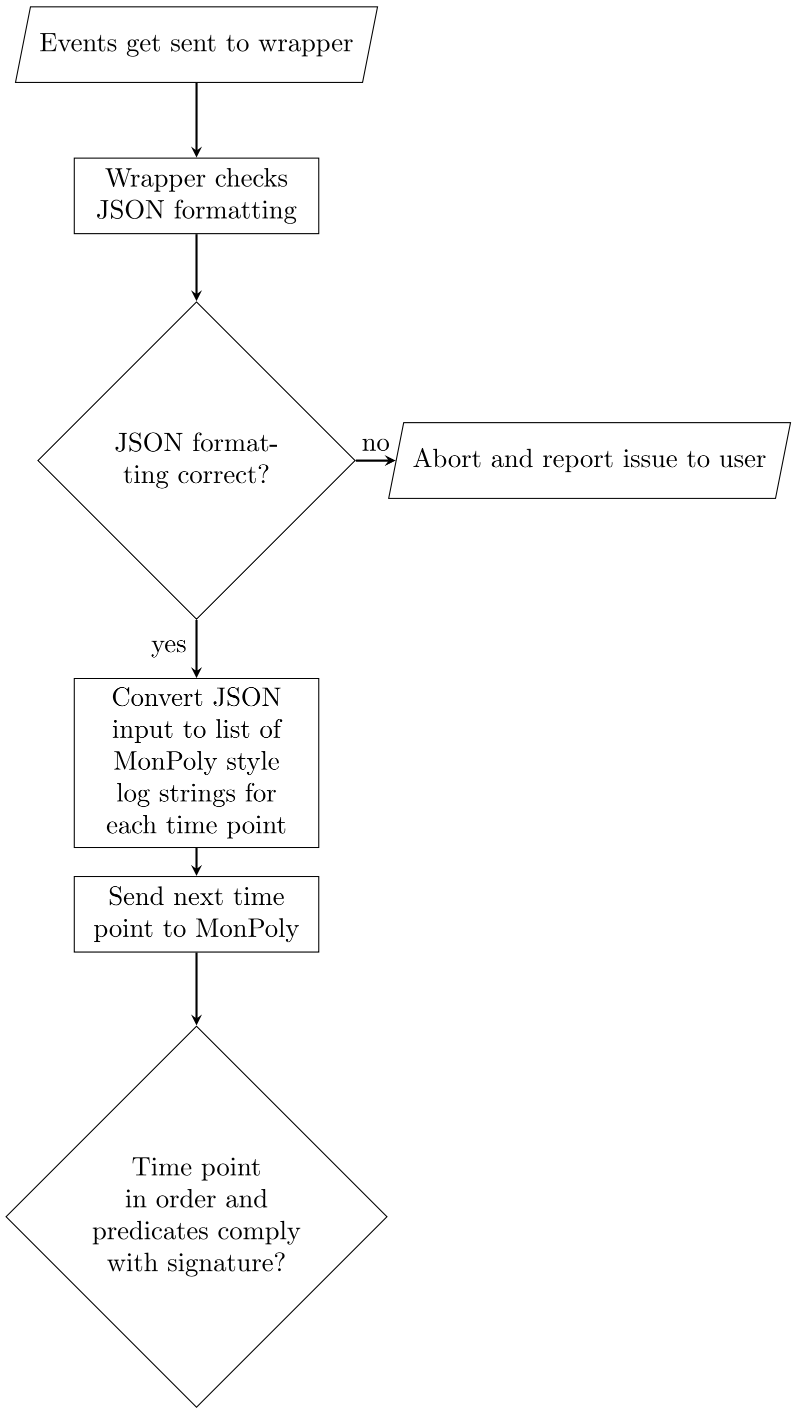
In this section we introduce the general architecture of our wrapper for MonPoly. A more in depth look at the specific technical implementation will be provided in the implementation section. In general, we have three components for this project. On one hand we have the MonPoly with a few extensions. On the other hand is QuestDB. Our wrapper acts as the glue between the two. In addition the wrapper also provides a new interface to MonPoly in the form of a REST API.



3.1 Wrapper

The wrapper can be run directly on a system with MonPoly installed. The alternative and more portable way to run it is with a docker container. To interact with the wrapper a user can use the provided REST API by sending web requests.

The wrapper runs MonPoly as a subprocess and handles all interactions with MonPoly itself. Incoming events are first parsed and checked on some major formatting errors. When the formatting is deemed acceptable the events get forwarded to MonPoly on a per time stamp basis. If MonPoly reports an issue with a certain time stamp, either it is out of order or one event at that timestamp does not comply with the given signature, this time stamp gets ignored by MonPoly, and in turn the wrapper discards it as well. If no issue is detected with a timestamp all events in at that timestamp get forwarded to the database.


Events get sent to wrapper
Wrapper checks JSON formatting
JSON formatting correct?
no
Abort and report issue to user
yes
Convert JSON input to list of MonPoly style log strings for each time point
Send next time point to MonPoly
Time point in order and predicates comply with signature?

3.2 Policy Change

This first version of a policy change functions by stopping the current monitor and starting a new one. When starting the new monitor we want to restore the state of the old one. We do this by querying old events from our database. But we do not simply query for the entirety of the database. We make two optimizations. First we use relative intervals and second we make use of constant values. For each predicate occurring in a formula we look for constant attributes in its occurrences. For every predicate and its potential occurrences with different constant values we then compute an over approximation of the relative interval. We use this information to create a SQL query that only queries the constant values combined with the interval. This way we minimize the amount of data that the new monitor has to read and process.

4 Algorithms

4.1 Relative Intervals

We extend the idea of relative intervals [6]. No event outside a MFOTL formula's relative interval can influence the truth value of that formula. This property is useful to us when restoring the state of the monitor by reloading past events, because it limits the amount of events we have to load and query from the database. Our extension further limits the number of relevant events. Events are tied to specific predicates. In our ongoing example we have the predicates `loc_accessed, perm_granted, perm_revoked`. Depending on the formula structure, different predicates might have different relative intervals which can reduce the number of relevant events. The exact details of the extension will be explained after this next part.

We combine this extension with another optimization involving constant values. Let's recall our example formula from earlier.

```
loc_accessed(i, "advertising")
IMPLIES
(
    (ONCE[0,*) perm_granted(i))
    AND
    (NOT (perm_revoked(i) SINCE[0,*) perm_granted(i)))
)
```

We scan all occurrences of any predicate and check if the attributes are constants or variable and noting down any constant values. Doing this for the example yields the following.

```
loc_accessed:
    variable, constant: "advertising"

perm_granted:
    variable

perm_revoked:
    variable
```

We call a list variable or constant attributes a mask. It is possible that the same predicate has multiple different masks, if it occurs with different constant values or if attributes at different positions are constant.

Now we have all the pieces to explore our full data structure for the extended relative intervals. We not only compute the relative interval for every predicate, but for every mask per predicate. For this we use a doubly nested map data structure. First mapping from predicate names to masks. And then mapping from the masks to their relative intervals.

In the rest of this section we explore the concrete algorithm used to compute the described data structure.

4.2

5 Implementation and Evaluation

6 Conclusion

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 Basin et al. "Scalable Offline Monitoring of Temporal Specifications". In: *Formal Methods in System Design* 49 (1 2016), pp. 75–108. ISSN: 1572-8102.

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

[8] 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.

[9] Roy Thomas Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. University of California, Irvine, 2000. ISBN: 0599871180.

[10] *Flask*. Accessed: 2023-01-30. URL: `https://flask.palletsprojects.com/`.

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

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

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

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