Kalpa Publications in Computing
Volume 3, 2017, Pages 19–28
RV-CuBES 2017. An International Workshop on
Competitions, Usability, Benchmarks, Evaluation,
and Standardisation for Runtime Verification Tools

The MonPoly Monitoring Tool

David Basin[1], Felix Klaedtke[2], and Eugen Zălinescu[3]

[1] ETH Zurich, Department of Computer Science
[2] NEC Laboratories Europe, Heidelberg
[3] Technische Universität München

Abstract

MonPoly is a monitoring tool for checking logs against formulas of metric first-order temporal logic. We provide here an overview of the tool, including its usage and history.

1 Introduction

MonPoly is a monitoring tool for checking the compliance of systems to policies, which identifies and reports all policy violations. Systems may be IT systems or other kinds of systems that produce streams of events, and policies express which system behaviors are allowed in terms of these event streams. The events can be monitored online, as they are produced by the system, or offline, by processing the events stored in log files.

MonPoly has a rich policy specification language based on metric first-order temporal logic (MFOTL). MFOTL's first-order features make it well suited for formalizing relations between event arguments. Moreover, its metric temporal operators can be used to specify qualitative and quantitative time constraints between events. Finally, its policy language includes operators for aggregating data values, which are useful in formulating many kinds of policies.

MonPoly is written in the programming language OCaml and is available at `https://sourceforge.net/projects/monpoly`. The website includes the tool's source code, and various log files and policy formalizations that were used in our case studies to evaluate the tool's performance.

In the following sections, we provide an overview of MonPoly. We describe the tool, version 1.1.7, including a brief description of its foundations and its input and output; in the appendix, we also provide additional usage guidelines. Afterwards, we recall the tool's history. This includes past case studies and we also provide pointers to some of our research results.

2 Tool Description

2.1 Foundations

We sketch here MonPoly's underlying monitoring algorithm for temporal structures with finite relations, which was first described in [8]. We refer to [9] for its algorithmic details, theoretical

underpinnings, and a description of MFOTL. We also refer to [6] for details on MFOTL's extension with aggregations operators.

The monitoring algorithm incrementally processes a temporal structure, which is an infinite sequence of timestamped first-order structures. We assume that the interpretation of each predicate symbol in each first-order structure from the input sequence is a finite relation (except for the interpretations of the rigid predicates like equality and ordering). We exploit here the fact that an event can be seen as a first-order structure, where an event is an element from the set $\Sigma \times D^*$, for some finite set Σ of event names and some infinite value domain D. Indeed, for an event $e(d_1, \ldots, d_n)$, the name e is seen as a predicate symbol with the singleton $\{(d_1, \ldots, d_n)\}$ as its interpretation. Thus, temporal structures can readily encode event streams.

We assume that policy formalizations are closed formulas of the form $\Box \forall \bar{x}.\, \varphi$ ($\Box$ stands for "always"). The universal quantification is not a restriction as the sequence $\bar{x}$ can be empty and φ can have additional topmost universal quantifiers. Our algorithm takes as input a formula ψ logically equivalent to $\neg\varphi$. It outputs for each time point (i.e. index in the input sequence of first-order structures) the satisfying valuations of the formula ψ. These represent policy violations. The universal quantification is dropped because a satisfying instantiation of the free variables $\bar{x}$ provides additional information about the violation.

To effectively monitor policies, our algorithm only handles a safety fragment of MFOTL. Namely, the temporal future operators occurring in the input formulas ψ are bounded, e.g. subformulas of the form $\Diamond_{[a,\infty)}\, \gamma$ and in particular $\Diamond \gamma$ ("eventually" γ) are disallowed, while $\Diamond_{[a,b)}\, \gamma$ is allowed, for any $a, b \in \mathbb{Q}_{\geq 0}$. Since ψ is bounded and time progresses, only finitely many future time points must accounted for when determining the satisfying valuations of ψ at a time point. This evaluation is delayed until the algorithm reads the data of the relevant future time points.

To determine at each time point the satisfying valuations of ψ, our algorithm evaluates the formula ψ bottom-up and stores intermediate results in finite relations. These are updated in each iteration and reused in later iterations. We require that ψ is such 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 satisfies these requirements.

2.2 Policy Specification

As a policy example, consider the following property from the domain of fraud detection. It requires that, at each point in time, the sum of withdrawals of each user in the last 30 days does not exceed the limit of \$10,000. We assume that the event $\mathit{withdraw}(u, a)$ captures that the user u has withdrawn the amount a, and that events are timestamped. Note that this event corresponds to the homonymous predicate symbol used in the propert specification below.

The formalization of this property in the extension of MFOTL with aggregations is as follows.

$$\Box \forall s.\, \forall u.\, [\mathsf{SUM}_a \blacklozenge_{[0,30)}\, \mathit{withdraw}(u, a) \wedge \mathit{tp}(i)](s; u) \rightarrow s < 10000 \tag{1}$$

Let us consider this formula in more detail. First, the time window of 30 days is specified by attaching the interval $[0, 30)$ to the temporal operator $\blacklozenge$ ("once"). Intuitively, the formula $\blacklozenge_I \varphi$ states that φ holds at some time point in the past within the time window represented by the interval I. In other words, the difference $\tau - \tau'$ between the timestamp τ of the current position and timestamp τ' of the past position must be within I. If the interval I includes zero, then the current time point is also considered.

Second, the occurrences of *withdraw* events at different time points, for the same user and of the same amount, are distinguished by using the built-in unary predicate tp. The predicate $tp(i)$ holds at the time point j iff $i = j$. Note that MFOTL has a set, rather than a multiset semantics. Thus the intended meaning of the policy would be incorrectly captured if this conjunct were omitted.

Third, we use the aggregation operator SUM to express the aggregation (as a sum) of the withdrawal amounts over the specified time window, grouped by the users. The operator, at the current time point, groups all withdrawals for a user u over the past 30 days and sums up their amounts a. The aggregation formula defines a binary relation where the first coordinate is the SUM's result s and the second coordinate is the user u for whom the result is calculated. In general, an aggregation formula has the form $[\omega_x \psi](y; \bar{g})$, where ω is an aggregation operator like SUM, CNT, and AVG, with the expected meanings. The semantics of aggregation formulas mimics that of aggregations and grouping operations in SQL: by viewing variables as (relation) attributes, $\bar{g}$ are the attributes on which grouping is performed, x is the attribute on which the aggregation operator ω is applied, and y is the attribute that stores the result. The remaining free variables in ψ (i.e. those different from y and not in $\bar{g}$) are bound by the aggregation formula; they do not have a corresponding attribute in the resulting relation. This is the case for the variable i in formula (1).

Finally, if a user's sum is greater than 10000, then the property is violated at the current time point. The formula therefore states that the aggregation condition must hold for each user and every time point.

2.3 Input and Output

MonPoly takes as command-line input a signature file, a formula file, and a log file. It outputs satisfying valuations of the given formula on the given log file.

The *signature file* describes the first-order signature of the formula used. MonPoly assumes sorted signatures, and it currently supports the sorts string, integer, and float. For our example policy, we use the following single-line signature file.

```
withdraw(string,int)
```

The *formula file* contains the policy as an MFOTL formula, possibly containing aggregation operators. In this example, we assume that the file contains the following formula.

```
(s <- SUM a;u ONCE[0,30] withdraw(u,a) AND tp(i))
AND
NOT s <= 10000
```

Note that this formula is the negation of the formula (1), where the outermost temporal operator $\Box$ and the universal quantifier block are dropped. The grammar for MonPoly's policy specification language is provided in the appendix.

A *log file* consists of a sequence of timestamped system events, which are ordered by their timestamps. Events that are assumed to occur simultaneously are grouped together. For example, consider the following log file.[1]

[1] For brevity, we assume here that the time unit of the timestamps is one day. Consequently, the metric constraint `[0,30]` of the temporal operator `ONCE` in the formula is interpreted as days. In practice, timestamps are often in Unix time and MonPoly's default time unit is thus one second. In particular, the metric constraints given as intervals attached to temporal operators are then in seconds. For convenience, one can use predefined abbreviations like `m` for minutes and `d` for days, e.g. one can attach the interval `[0,30d]` to a temporal operator.

```
@10 withdraw (Alice,6000)
@20 withdraw (Bob,300) (Dan,300)
@20 withdraw (Charlie,2000)
@30 withdraw (Alice,6000)
@60 withdraw (Charlie,9000)
```

This log states that Alice withdrew \$6,000 at time point 0 with the timestamp 10, Bob and Dan both withdrew \$300 at time 20, and so on. If two events have the same timestamp (possibly due to an insufficiently precise clock), but occur at different time points, then this means that the events are still ordered time-wise (possibly by additional information that determines the ordering). However, events at the same time point, like Bob and Dan's withdrawals, are not ordered.

MonPoly outputs

```
@30. (time-point 3): (12000,Alice)
```

when running on the above input:

```
monpoly -sig example.sig -formula example.mfotl -log example.log
```

where the signature, the formula, and the timestamped events from above are contained in the respective `example` files. Recall that the formula file contains the negated policy with free variables. Thus, policy violations correspond to satisfying valuations for the given formula at a time point. In fact, at the time point with timestamp 30, Alice withdrew in total \$12,000 within the last 30 days and has thus violated the policy. Note that although Charlie withdrew \$11,000 in total, he did not violate the policy since his two withdrawals—both under \$10,000—do not fall into the same 30 day time window.

Note that without the `-log` argument, MonPoly expects to receive the event sequence on standard input. This enables MonPoly to be used in an online manner, where the timestamped events are input through a Unix pipe rather than a log file.

3 Tool History

MonPoly's underlying monitoring algorithm was first presented in [8] (see also [9]) and extends Chomicki's approach [12] for dynamically checking integrity constraints in temporal databases. The monitoring algorithm also handles sequences of first-order structures that are represented by automata instead of finite relations (as assumed by MonPoly). In this more general case, intermediate results are stored as automata and the syntactic restrictions on negation and quantification are not needed. However, the performance overhead is significant.

Our first implementation of the monitoring algorithm for finite relations was programmed in Java. We used this implementation to evaluate MFOTL's suitability for expressing and checking a wide range of security policies [7]. We reimplemented the monitoring algorithm in the programming language OCaml, with several optimizations, resulting in the first version of the MonPoly tool [5]. Later, we extended MonPoly with support for function symbols (allowing, for instance, specifications involving arithmetic operators) and for operators that aggregate data values [6]. As these features often appear in policies, this extension significantly broadened the tool's scope.

We have also optimized MonPoly's treatment of subformulas of the form $\blacklozenge_I \gamma$ and $\Diamond_I \gamma$. Our optimization uses an algorithm for combining elements with an associative operator $\oplus$ in a sliding window of varying size of an infinite sequence $\bar{a}$ [10]. This algorithm is greedy and

optimal in the number of $\oplus$ applications. In MonPoly's setting, the sequence $\bar{a}$ corresponds to the sequence of relations consisting of the tuples that satisfy the subformula γ, the associative operator $\oplus$ is set union, and the sliding window is determined by the timing constraint I.

We used MonPoly in two larger case studies. In the first case study, we monitored the usage of data within Nokia's data-collection campaign [13], where contextual information from cell phones was collected, 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 collected data. The second case study was in collaboration with Google [4], where we checked policies over huge distributed log files. In comparison to the Nokia case study, the logs were 100 times larger in terms of the number of events and 50 times larger in data volume. Namely, the log files contained over 26 billion events from a two year's period, amounting to 0.4 TB of logged data in a protocol buffers format. We used the map-reduce framework to slice logs and monitor the sliced logs separately with MonPoly. Furthermore, MonPoly participated in the offline track of the first competition on runtime verification in 2014 [2], where it scored the second place.

4 Outlook

We see increasing applications for monitoring in the future. IT systems are collecting and processing ever increasing amounts of data while, at the same time, there are increasingly many regulations on how such data can be used. It is therefore important to develop monitoring approaches that scale well to "big data" scenarios. As previously mentioned, we have used our slicing framework and the map-reduce framework as a wrapper to run multiple MonPoly instances in parallel. This is suitable for scaling up monitoring in the offline setting. We are currently working on parallelizing MonPoly in the online setting.

It is also important to develop suitably expressive monitoring languages with associated algorithms that have better runtime complexity than MonPoly, where monitoring, in the worst case, requires space polynomial in the event stream prefix being monitored. Promising recent work in this regard is developing monitors whose space consumption is (almost) independent of the event rate of the stream being monitored. The recent work [3, 11] shows how this can be done for different fragments and extensions of propositional metric temporal logic. It remains to be seen whether these fragments can be extended to a first-order setting, and lead to more efficient tools than MonPoly for fragments of MFOTL.

Acknowledgments. We thank Matúš Harvan and Srdjan Marinovic for their contributions to the MonPoly tool. Furthermore, we thank Nokia Research and Google for their past support.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison Wesley, 1994.

[2] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, accepted for publication.

[3] D. Basin, B. Bhatt, and D. Traytel. Almost event-rate indepedent monitoring of metric temporal logic. In *Proc. of the 23rd Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 10206 of *LNCS*, pages 94–112. Springer, 2017.

[4] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. Scalable offline monitoring of temporal properties. *Formal Methods in System Design*, 49(1-2):75–108, 2016.

[5] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. MONPOLY: Monitoring usage-control policies. In *Proc. of the 2nd Int. Conf. on Runtime Verification (RV)*, volume 7186 of *LNCS*, pages 360–364. Springer, 2012.

[6] D. Basin, F. Klaedtke, S. Marinovic, and E. Zălinescu. Monitoring of temporal first-order properties with aggregations. *Formal Methods in System Design*, 46(3):262–285, 2015.

[7] 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 (SACMAT)*, pages 23–33. ACM Press, 2010.

[8] D. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann. Runtime monitoring of metric first-order temporal properties. In *Proc. of the 28th IARCS Annual Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2 of *Leibniz Int. Proceedings in Informatics (LIPIcs)*, pages 49–60. Schloss Dagstuhl - Leibniz Center for Informatics, 2008.

[9] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *Journal of the ACM*, 62(2), 2015.

[10] D. Basin, F. Klaedtke, and E. Zălinescu. Greedily computing associative aggregations on sliding windows. *Information Processing Letters*, 115(2):186–192, 2015.

[11] D. Basin, S. Krstić, and D. Traytel. Almost event-rate indepedent monitoring of metric dynamic logic. In *Proc. of the 17th Int. Conf. on Runtime Verification (RV)*, volume 10548 of *LNCS*, pages 85–102. Springer, 2017.

[12] J. Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Transactions on Database Systems*, 20(2):149–186, 1995.

[13] N. Kiukkonen, J. Blom, O. Dousse, D. Gatica-Perez, and J. Laurila. Towards rich mobile phone datasets: Lausanne data collection campaign. In *Proc. of the 7th Int. Conf. on Pervasive Services (ICPS)*, 2010.

A Usage Guidelines

Main Command-Line Arguments. As usual, the list of MonPoly's command-line arguments is provided by using the `-help` argument. We start by explaining MonPoly's most important arguments. Only the `-sig` and `-formula` arguments are mandatory, where the specified files contain a signature and a formula, respectively. A log file can be given to MonPoly using the `-log` argument. When the `-log` argument is missing MonPoly reads the sequence of timestamped events (or, more accurately, first-order structures) from the standard input.

When the `-negate` argument is present, MonPoly negates the given formula. In what follows, let the *input formula* be the formula contained in the formula file, or its negation in case the `-negate` argument is given. By default, MonPoly applies a series of rewrite rules to the input formula to simplify it. We call the resulting formula the *analyzed formula*. Note that the analyzed formula corresponds to the formula ψ in Section 2.1.

The rewrite rules mainly eliminate syntactic sugar[2] and double negations, and push negations inwards, in case the `-negate` argument was given. Furthermore, if the resulting formula is not monitorable, MonPoly applies additional rewrite rules that try to put the formula into a

[2] Concretely, the following equivalences are used to rewrite the formulas on the left-hand side of the $\equiv$ symbol: $\alpha \leftrightarrow \beta \equiv (\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$, $\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$, $\forall \bar{x}.\, \alpha \equiv \neg\exists \bar{x}.\, \neg\alpha$, $\Box_I\, \alpha \equiv \neg \Diamond_I \neg\alpha$, and $\blacksquare_I\, \alpha \equiv \neg \blacklozenge_I \neg\alpha$.

monitorable form. Recall that MonPoly only handles a syntactically-defined safety fragment of MFOTL. The `-no_rw` argument instructs MonPoly not to perform such rewrites. Note that with the `-no_rw` argument, the analyzed formula is syntactically equal to the input formula. Furthermore, MonPoly always reports that the analyzed formula is not monitorable when the input formula has free variables, and both arguments `-no_rw` and `-negate` are present.

Monitorable Formulas. With the `-check` argument, no monitoring is performed. Instead, MonPoly checks whether the analyzed formula is monitorable, that is, whether it satisfies the syntactic requirements needed by the monitoring algorithm. Furthermore, MonPoly outputs the analyzed formula together with its free variables. For instance, when trying to detect satisfactions of the policy from Section 2.2, by calling MonPoly as

```
monpoly -sig example.sig -formula example2.mfotl -check
```

where `example2.mfotl` contains the formula (1), again with the outermost temporal operator $\Box$ and the universal quantifier block dropped, MonPoly's output is as follows.

```
The input formula is:
  (s <- SUM a; u ONCE[0,30] (withdraw(u,a) AND tp(i)))         \
  IMPLIES s <= 10000
The analyzed formula is:
  (NOT s <- SUM a; u ONCE[0,30] (withdraw(u,a) AND tp(i)))     \
  OR s <= 10000
The sequence of free variables is: (s,u)
The analyzed formula is not monitorable because of the         \
following subformula:
  NOT s <- SUM a; u ONCE[0,30] (withdraw(u,a) AND tp(i))
Subformulas of the form NOT psi should contain no free         \
variables (except when they are part of subformulas of the form\
phi AND NOT psi, NOT psi SINCE_I phi, or NOT psi UNTIL_I phi).
```

The analyzed formula (as well as any logically equivalent formula) is not monitorable because at each time point there are infinitely many tuples (s, u) satisfying it. Note that MonPoly also outputs the subformula that causes the analyzed formula to be non-monitorable, and it provides the reason why this subformula is not monitorable. Note that with the `-negate` argument, in this example, the analyzed formula becomes the (monitorable) one from `example.mfotl`.

We remark that even when the analyzed formula is monitorable, it is generally a good idea to inspect it, using the `-check` argument, and simplify it if possible while retaining monitorability. Simplifications usually result in performance improvements during monitoring. For instance, instead of the subformula $\exists x.\, (p(x, y) \wedge q(z))$, it is better to use the logically equivalent subformula $(\exists x.\, p(x, y)) \wedge q(z)$. This is because with the latter subformula the intermediate relations are smaller than with the first one.

The following example provides further intuition about non-monitorable formulas and how to deal with them. Consider the policy that requires that any published report r must be approved by a manager m within six days:

$$\Box\, \forall r.\, \forall m.\, \mathit{publish}(r) \rightarrow \Diamond_{[0,6]}\, \mathit{approve}(m, r)$$

It may be tempting to use MonPoly directly with the formula

```
publish(r) IMPLIES EVENTUALLY[0,6] approve(m,r)
```

and the `-negate` argument. However, the formula's negation is not monitorable. The intuitive reason is that a policy violation is already identified by a non-approved report (and the time point where the report was published); no particular manager is responsible for causing the policy violation. More formally, given a report r that is it not approved in the required time window by any manager, each tuple (r, m) satisfies the analyzed formula, for any m. As we assume an infinite domain, there are infinitely many such tuples. For monitoring, we must retain the universal quantification over the managers. However, MonPoly does not succeed in rewriting the negation of the formula

```
FORALL m. publish(r) IMPLIES EVENTUALLY[0,6] approve(m,r)
```

into a monitorable formula. The rewriting must be done by hand (and MonPoly must be used without the `-negate` argument):

```
publish(r) AND NOT EVENTUALLY[0,6] EXISTS m. approve(m,r)
```

Finite Event Sequences and Future Temporal Operators. By default, MonPoly adds a last time point with a large enough timestamp (more exactly, the largest representable timestamp), at the end of the input event sequence. This ensures that temporal subformulas with a future temporal operator can be evaluated at all time points in the original event sequence. Note that this behavior introduces the assumption that "nothing happens" after the end of the input sequence; that is, that there are no time points after the last time point in the original sequence. The argument `-nonewlastts` can be used to change this default behavior.

MonPoly cannot deal with formulas that contain unbounded future temporal operators. However, when monitoring a finite sequence of events, it often suffices to replace the unbounded future operators with bounded ones, having a large enough upper bound. For instance, an upper bound bigger then the last timestamp in the input sequence is enough. Such a rewrite must however done with some care. First, in general, the resulting formula is not logically equivalent to the original one. Second, MonPoly may delay output until the last log entry is read.

Other Command-Line Arguments. By default, MonPoly filters out events that cannot influence the monitor's output. It also filters out time points with no events when they do not influence the monitor's output. The arguments `-nofilterrel` and `-nofilteremptytp` can be used to change this default behavior.

MonPoly has additional command-line arguments, with self-explanatory names, instructing it whether to ignore parse errors or out-of-order time points in the input event stream, and whether to provide debugging information or resource usage statistics.

B Grammar

B.1 Signatures

The grammar of MonPoly's signatures is as follows.

⟨*signature*⟩ ::= ⟨*predicate*⟩ ⟨*signature*⟩ | ⟨*empty*⟩

⟨*predicate*⟩ ::= ⟨*string*⟩ '`(`' ⟨*sorts*⟩ '`)`'

⟨*sort-list*⟩ ::= ⟨*sort*⟩ '`,`' ⟨*sort-list*⟩ | ⟨*sort*⟩ | ⟨*empty*⟩

⟨*sort*⟩ ::= '`string`' | '`int`' | '`float`'

B.2 Log Entries

MonPoly reads a log file incrementally, one log entry at a time. The grammar of the log entries is as follows. A log file is a sequence of log entries.

⟨*log-entry*⟩ ::= '`@`' ⟨*ts*⟩ ⟨*db*⟩
⟨*ts*⟩ ::= ⟨*integer*⟩ | ⟨*float*⟩
⟨*db*⟩ ::= ⟨*table*⟩ ⟨*db*⟩
⟨*table*⟩ ::= ⟨*string*⟩ ⟨*relation*⟩
⟨*relation*⟩ ::= ⟨*tuple*⟩ ⟨*relation*⟩ | ⟨*empty*⟩
⟨*tuple*⟩ ::= '`(`' ⟨*fields*⟩ '`)`'
⟨*fields*⟩ ::= ⟨*string*⟩ '`,`' ⟨*fields*⟩ | ⟨*string*⟩ | ⟨*empty*⟩

B.3 Policy Specification Language

The grammar of MonPoly's policy specification language is as follows.

⟨*formula*⟩ ::=
| '`(`' ⟨*formula*⟩ '`)`'
| '`FALSE`'
| '`TRUE`'
| ⟨*predicate*⟩
| ⟨*term*⟩ '`=`' ⟨*term*⟩
| ⟨*term*⟩ '`<`' ⟨*term*⟩
| ⟨*term*⟩ '`>`' ⟨*term*⟩
| ⟨*term*⟩ '`<=`' ⟨*term*⟩
| ⟨*term*⟩ '`>=`' ⟨*term*⟩
| ⟨*formula*⟩ '`EQUIV`' ⟨*formula*⟩
| ⟨*formula*⟩ '`IMPLIES`' ⟨*formula*⟩
| ⟨*formula*⟩ '`OR`' ⟨*formula*⟩
| ⟨*formula*⟩ '`AND`' ⟨*formula*⟩
| '`NOT`' ⟨*formula*⟩
| '`EXISTS`' ⟨*var-list*⟩ '`.`' ⟨*formula*⟩
| '`FORALL`' ⟨*var-list*⟩ '`.`' ⟨*formula*⟩
| ⟨*var*⟩ '`<-`' ⟨*aggreg*⟩ ⟨*var*⟩ '`;`' ⟨*var-list*⟩ ⟨*formula*⟩ // aggregation formula
| ⟨*var*⟩ '`<-`' ⟨*aggreg*⟩ ⟨*var*⟩ ⟨*formula*⟩ // variant with no group-by variables
| '`NEXT`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| '`PREV`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| '`EVENTUALLY`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| '`ONCE`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| '`ALWAYS`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| '`PAST_ALWAYS`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| ⟨*formula*⟩ '`SINCE`' ⟨*interval-opt*⟩ ⟨*formula*⟩
| ⟨*formula*⟩ '`UNTIL`' ⟨*interval-opt*⟩ ⟨*formula*⟩

⟨*predicate*⟩ ::=
| ⟨*string*⟩ '`(`' ⟨*term-list*⟩ '`)`'
| '`tp`' '`(`' ⟨*term*⟩ '`)`' // time point predicate
| '`ts`' '`(`' ⟨*term*⟩ '`)`' // timestamp predicate
| '`tpts`' '`(`' ⟨*term*⟩ '`,`' ⟨*term*⟩ '`)`' // time point and timestamp predicate

⟨*aggreg*⟩ ::=
  | 'CNT' // counting aggregation operator
  | 'MIN' // minimum aggregation operator
  | 'MAX' // maximum aggregation operator
  | 'SUM' // sum aggregation operator
  | 'AVG' // average aggregation operator
  | 'MED' // median aggregation operator

⟨*interval-opt*⟩ ::= ⟨*lbound*⟩ ',' ⟨*rbound*⟩ | ⟨*empty*⟩

⟨*lbound*⟩ ::= '(' ⟨*bound*⟩ | '[' ⟨*bound*⟩

⟨*rbound*⟩ ::= ⟨*bound*⟩ ')' | ⟨*bound*⟩ ']' | '*)'

⟨*bound*⟩ ::= ⟨*integer*⟩⟨*unit*⟩ | ⟨*integer*⟩

⟨*unit*⟩ ::= 's' | 'm' | 'h' | 'd'

⟨*term-list*⟩ ::= ⟨*term*⟩ ',' ⟨*term-list*⟩ | ⟨*term*⟩ | ⟨*empty*⟩

⟨*var-list*⟩ ::= ⟨*var*⟩ ',' ⟨*var-list*⟩ | ⟨*var*⟩ | ⟨*empty*⟩

⟨*term*⟩ ::=
  | '(' ⟨*term*⟩ ')'
  | ⟨*term*⟩ '+' ⟨*term*⟩
  | ⟨*term*⟩ '-' ⟨*term*⟩
  | ⟨*term*⟩ '*' ⟨*term*⟩
  | ⟨*term*⟩ '/' ⟨*term*⟩
  | ⟨*term*⟩ 'MOD' ⟨*term*⟩ // modulo operation
  | '-' ⟨*term*⟩
  | 'f2i' '(' ⟨*term*⟩ ')' // float to integer conversion
  | 'i2f' '(' ⟨*term*⟩ ')' // integer to float conversion
  | ⟨*cst*⟩
  | ⟨*var*⟩

⟨*cst*⟩ ::= ⟨*integer*⟩ | ⟨*rational*⟩ | '"' ⟨*string*⟩ '"'

⟨*var*⟩ ::= '_' | ⟨*string*⟩

The following table explains the correspondence between mathematical notation and MonPoly notation for the symbols denoting connectives, quantifiers, and temporal operators. The table also specifies MonPoly's convention on a symbol's precedence and associativity. The symbols are declared to associate to the left, to the right, or to be non-associative (marked by 'none'). All symbols on the same line are given the same precedence. The symbols listed in a row are given lower precedence than the symbols listed in a previous row (and higher precedence than the symbols listed in a subsequent row).

symbol	MonPoly terminal	associativity
¬	NOT	none
∧	AND	left
∨	OR	left
→	IMPLIES	right
↔	EQUIV	left
∃ ∀	EXISTS FORALL	none
● ○ ◆ ◇ ■ □	PREV NEXT ONCE EVENTUALLY PAST_ALWAYS ALWAYS	none
S U	SINCE UNTIL	right