Dissertation ETH No. 28348

Efficient, Expressive, and Verified Temporal Query Evaluation

A dissertation submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZÜRICH

presented by
MARTIN RASZYK
born on April 16, 1995
Master of Science ETH in Computer Science

accepted on the recommendation of
Prof. Dr. David Basin
Prof. Dr. Bernd Finkbeiner
Prof. Dr. Juraj Hromkovič
Prof. Dr. Dmtriy Traytel

2022

Abstract

Runtime monitoring (or runtime verification) is an approach to checking compliance of a system's execution with a specification (e.g., a temporal query). The system's execution is logged into a *trace*—a sequence of time-points, each consisting of a time-stamp and observed events. A *monitor* is an algorithm that produces *verdicts* on the satisfaction of a temporal query on a trace. This thesis develops new monitoring algorithms for expressive temporal query languages that are more time- and space-efficient than the state-of-the-art and produce detailed verdicts.

An *online* monitor reads the trace forwards, one time-point after another. An *offline* monitor processes the time-points in an arbitrary order. We propose a novel paradigm—*multi-head* monitoring—that fills a middle-ground between online and offline monitoring. A multi-head monitor uses multiple reading heads that read the trace forwards. We develop multi-head monitors for metric temporal logic (MTL) and metric dynamic logic (MDL). Our monitors improve upon the state-of-the-art by optimizing the time and space complexity while producing a sequence of Boolean verdicts denoting the temporal query's satisfaction at every time-point. We have implemented our monitors, empirically confirmed that their performance improves upon existing approaches, and formally verified the correctness of our monitors using the Isabelle/HOL proof assistant.

MTL and MDL events cannot have parameters and queries cannot contain variables. Metric first-order temporal logic (MFOTL) generalizes MTL with parametric events and first-order variables ranging over an arbitrary domain. Hence, a monitor evaluates an MFOTL query at every time-point to a relation representing the valuations of the free variables satisfying the query. An online monitor for MFOTL had been developed in previous work and later formally verified using Isabelle/HOL. The formally verified monitor is called VeriMon. In this thesis, we optimize the time complexity of evaluating VeriMon's Since and Until temporal operators.

Query evaluation for a proper subset of MFOTL queries can be efficiently implemented using relational algebra operations. Alternatively, an arbitrary MFOTL query can be evaluated using structures that represent arbitrary relations satisfying the subqueries of the query, e.g., automatic structures or binary decision diagrams. However, these alternative representations have a negative impact on the performance of the resulting monitor. Hence, in this thesis we use relational algebra operations to evaluate an arbitrary MFOTL query. We first investigate the case of relational calculus (RC), i.e., MFOTL without temporal operators. We develop a novel approach to RC query evaluation by translating an arbitrary RC query into a pair of relational algebra normal form (RANF) queries that can be evaluated using relational algbra operations on finite tables: one characterizes the original query's relative safety (i.e., whether it evaluates to a finite relation) and the other one is equivalent to the original query if the original query is relatively safe. We implement our translation and empirically confirm that the performance of evaluating the queries produced by our translation improves upon existing approaches to RC query evaluation. Finally, we generalize our translation of RC queries to MFOTL queries. This way, we obtain a monitor for an arbitrary MFOTL query that decides for every time-point if it evaluates to a finite relation and computes the relation if it is finite.

Zusammenfassung

Runtime Monitoring (oder Runtime Verifikation) ist ein Verfahren, um zu überprüfen, ob eine Systemausführung eine Spezifikation (z.B. eine temporale Abfrage) erfüllt. Die Systemausführung wird durch ein *Trace*—eine Folge von Zeitpunkten—dargestellt, die jeweils aus einem Zeitstempel und beobachteten Events bestehen. Ein *Monitor* ist ein Algorithmus, der *Verdikte* über die Erfüllung einer temporalen Abfrage bezüglich einem Trace berechnet. Diese Arbeit entwirft neue Monitore für ausdrucksstarke Sprachen von temporalen Abfragen, die zeit- und platzeffizienter als bestehende Ansätze sind und detailliertere Verdikte berechnen.

Ein *Online-Monitor* liest das Trace vorwärts, d.h. einen Zeitpunkt nach dem anderen. Ein *Offline-Monitor* bearbeitet die Zeitpunkte in einer beliebigen Reihenfolge. Wir entwerfen ein neues Paradigma—*Mehrkopf-Monitoring*—das zwischen Online- und Offline-Monitoring liegt. Ein Mehrkopf-Monitor benutzt mehrere Leseköpfe, die das Trace lesen und sich dabei vorwärts bewegen. Wir entwickeln Mehrkopf-Monitore für die metrische temporale Logik (MTL) und metrische dynamische Logik (MDL). Unsere Monitore verbessern den Stand der Technik, indem sie die Zeit- und Platzkomplexität optimieren und eine Folge von Booleschen Verdikten erzeugen, die die Erfüllung der temporalen Abfrage zu jedem Zeitpunkt angeben. Wir haben unsere Monitore implementiert, ihre verbesserte Effizienz im Vergleich zu bestehenden Ansätzen empirisch nachgewiesen und ihre Korrektheit mithilfe des Beweisassistenten Isabelle/HOL verifiziert.

Die Events in MTL und MDL dürfen keine Parameter haben und Abfragen dürfen keine Variablen enthalten. Die metrische temporale Logik erster Stufe (MFOTL) erweitert MTL um parametrisierte Events, Variablen mit einem beliebigen Bereich und Quantoren. Des Weiteren wertet ein Monitor eine MFOTL-Abfrage zu jedem Zeitpunkt in eine Relation aus, die die erfüllenden Belegungen der freien Variablen der Abfrage enthält. Ein Online-Monitor für MFOTL wurde in vorheriger Arbeit entwickelt und später mithilfe von Isabelle/HOL formal verifiziert. Der formal verifizierte Monitor heisst VeriMon. In dieser Arbeit optimieren wir die Laufzeitkomplexität der Auswertung von VeriMon's Since und Until temporalen Operatoren.

Die Auswertung von Abfragen kann für eine echte Teilmenge von MFOTL-Abfragen effizient durch Operationen der relationalen Algebra (RA) umgesetzt werden. Alternativ kann eine beliebige MFOTL-Abfrage mithilfe von Datenstrukturen ausgewertet werden, die eine beliebige erfüllende Relation für jede Teilabfrage darstellen können, z.B. automatische Strukturen oder binäre Entscheidungsdiagramme. Diese alternativen Darstellungen haben allerdings einen negativen Einfluss auf die Effizienz vom jeweiligen Monitor. In dieser Arbeit verwenden wir deshalb Operationen der relationalen Algebra, um eine beliebige MFOTL-Abfrage auszuwerten. Wir untersuchen zunächst den Fall vom Relationenkalkül (RC, aus dem Englischen *relational calculus*), d.h. MFOTL ohne temporale Operatoren. Wir entwerfen ein neues Verfahren zur Auswertung von beliebigen RC-Abfragen durch ihre Übersetzung in zwei RA Normalform (RANF) Abfragen, die durch relationale Operationen auf endlichen Tabellen ausgewertet werden können: die eine entscheidet die relative Sicherheit (aus dem Englischen *relative safety*) der ursprünglichen Abfrage (d.h. ob sie zu einer endlichen Relation auszuwerten ist) und die andere ist äquivalent zu der ursprünglichen Abfrage, wenn diese relativ sicher ist. Wir implementieren unser Übersetzungsverfahren und weisen empirisch nach, dass unser Verfahren Abfragen liefert, deren Auswertung effizienter ausführbar ist, als die Auswertung der ursprünglichen Abfragen mit bestehenden Ansätzen zur Auswertung von RC-Abfragen. Schliesslich erweitern wir unser

Übersetzungsverfahren von RC-Abfragen auf MFOTL-Abfragen. Somit erhalten wir einen Monitor für eine beliebige MFOTL-Abfrage, der für jeden Zeitpunkt entscheidet, ob die erfüllende Relation für diesen Zeitpunkt endlich ist und die Relation berechnet, wenn sie endlich ist.

Acknowledgments

I would like to thank David Basin for giving me the opportunity to carry out my thesis in his research group and proposing the topic of runtime verification, for his valuable feedback, and last but not least for his flexibility throughout my thesis. Thank you! I would also like to thank Dmitriy Traytel for his guidance, helpful feedback, and great support throughout my thesis. Спасибо! I would like to thank my colleagues Srđan Krstić and Joshua Schneider for our thriving collaboration. Хвала! I would also like to thank Joshua for being a great office mate. Merci! I appreciate the feedback on my talks from my fellows in the Information Security group. Thank you! I am grateful for Bernd Finkbeiner's willingness to serve as my co-examiner. Danke!

I am also grateful for the opportunity to work with Juraj Hromkovič. His great research workshops were my first opportunity to get in touch with research and connect to his group for our subsequent fruitful collaboration. Ďakujem! I am happy to have worked with Hans-Joachim Böckenhauer, Elisabet Burjons, Fabian Frei, and David Wehner. Merci i gràcies!

I would like to thank David Lanzenberger for our long discussions on various topics and his willingness to always help me. Danke!

I am deeply indebted to my parents for their neat care and huge support. Moc děkuji!

Contents

Chapter 1

Introduction

Complex (software) systems are ubiquitous nowadays and plenty of bugs regularly found in the vast majority of such systems show that it is very difficult to make sure that these systems behave according to their specification, i.e., to check their *correctness*. A traditional approach to checking the correctness of systems is to perform extensive *testing* before the system is deployed and then to *monitor* the system's execution while the system is running. However, testing and monitoring can never provide absolute guarantees on the system's correctness because these methods are inherently incomplete, i.e., they do not check every possible system behaviour. Hence, critical systems have also been subject to *formal verification* of their correctness, e.g., by model-checking the system's behaviour or by proving the system's correctness using a proof assistant. Although formal verification provides the highest level of trustworthiness, it is expensive in terms of development cost. Hence, testing and monitoring are indispensable for systems where the development cost of formal verifiction is not affordable.

1.1 Runtime Verification

Runtime monitoring (or runtime verification [5]) is an approach to checking compliance of a system's execution with a specification. Monitoring can be implemented as an interaction between a human (having a specification in mind) and sensors measuring some parameters of the system. If the specification can be formalized as a precise mathematical statement (a formal specification) and if the system's execution can be abstracted as a sequence of events, then the monitoring task can be automated by executing a monitoring algorithm observing the sequence of events produced by the system and computing verdicts on the compliance of the system's execution with the specification.

A diagram of the runtime monitoring architecture design is depicted in Figure 1.1. At the top we have a running system that logs its execution into a *trace*—a sequence of time-points, each consisting of a time-stamp and a set of observed events. On the left we have a specification (a temporal query) formalizing the system's intended behaviour. Both the specification and

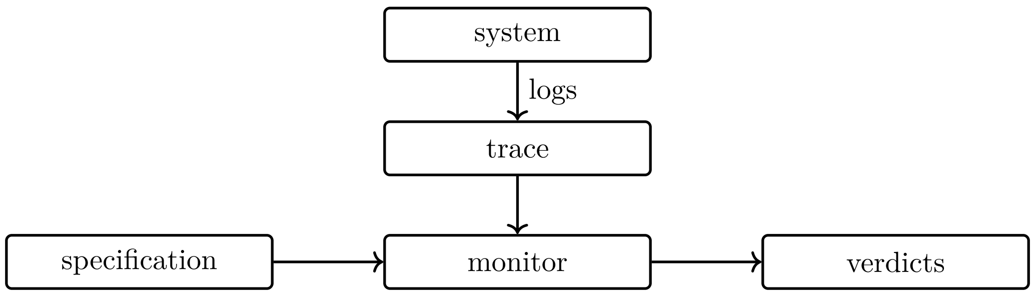


Figure 1.1. Runtime monitoring architecture.

Input trace		Output verdicts	Input trace		Output verdicts
@0	auth	✓	@0	auth	✓
@2	err	✓	@2	err	✓
@2		✓	@2	auth	✓
@6	err	✓	@6	err	✓
@10	err	✓	@10	err	✓
@12	auth	✗	@12	auth	✓
@63	auth	✓	@63	auth	✓

Figure 1.2. Input traces and output verdicts for Example 1.1.

the trace are inputs of a monitor that produces verdicts attesting to the system's execution's compliance with the specification.

Because a monitor is running in addition to the actual system (Figure 1.1), runtime monitoring incurs a runtime cost. This thesis develops new monitoring algorithms for rich temporal query languages that produce detailed verdicts and that are more time- and space-efficient than the state-of-the-art (thus decreasing the runtime cost of monitoring).

Example 1.1. Consider an example of a monitoring specification for a system managing user authentication. Many such systems follow a specification like: "A user should not be able to authenticate after entering a wrong password three times within the last hour without authenticating in between." We suppose that the user authentication system logs its execution into a trace of discrete time-points. Every time-point (one line) is characterized by a time-stamp (prefixed by @) and a set of events that happened at that time-point. For a fixed user, we write auth for the event "User authenticated" and err for the event "User entered a wrong password". The output of a monitor for the specification is a sequence of Boolean verdicts denoting whether the specification is satisfied (✓) or violated (✗) at each time-point in the trace.

A pair of example traces are depicted in Figure 1.2. They consist of seven time-points each, with time-stamps in seconds. A singleton set of events were observed at each time-point, except for the third time-point of the trace on the left at which no event was observed (such a time-point can still be logged, e.g., if the time-points are produced periodically or only events irrelevant for the policy at hand had happened). There is a violation of the specification at the penultimate time-point of the trace on the left at which an authentication auth was observed after three wrong attempts to enter the passowrd err within the last 10 seconds, i.e., within the last minute. There is no violation of the specification at the last time-point of the trace on the left because a sufficient amount of time (61 seconds) has ellapsed since the first out of the three wrong attempts to enter the password. There is no violation of the specification at any time-point of the trace on the right in Figure 1.2 because the three wrong attempts to enter the password are interrupted by a successful authentication.

The output verdicts could be produced by a human (e.g., the thesis author who produced the verdicts in Figure 1.2) or by an ad-hoc program for this particular specification. A more productive and less error-prone solution is offered by formalizing the specification in a suitable specification language (Chapter 2) for a runtime monitor (Chapters 3, 4). A verified monitor also gives the author of the thesis solid guarantees that he computed the verdicts in Figure 1.2 correctly. ◇

In the following, we discuss the three main aspects of runtime monitoring considered in this thesis due to their impact on efficiency, expressiveness, and trustworthiness:

- mode of operation (e.g., online, offline, multi-head monitoring),
- specification language features (e.g., regular expressions, first-order queries),
- correctness (e.g., formal verification using proof assistants).

Mode of Operation Monitors can be classified by their mode of operation. An *online* monitor reads the trace forwards, one time-point after another. In particular, the entire trace does not have to be available when the monitor starts its computation. This means that an online monitor can run parallel with the system. An *offline* monitor processes the time-points in an arbitrary order, e.g., it could read the trace backwards. In particular, an offline monitor needs the entire trace from the very beginning. This means that an offline monitor can only run after the system finished its execution. The space requirements for online monitoring might grow linearly with the number of incoming events because the monitor might have to buffer the incoming events.

To distinguish monitors that only require little working memory if they can read events several times, we propose a novel paradigm—*multi-head* monitoring—that fills a middle-ground between online and offline monitoring. A multi-head monitor uses multiple reading heads that read the trace forwards and their number only depends on the specification, i.e., their number does not depend on the observed events. Formally, one can view a multi-head monitor as a multi-tape Turing machine that can read the trace on the input tape with multiple reading heads that only move left-to-right on the input tape. The standard definition of space complexity for multi-tape Turing machines carries over to multi-head monitors: The space complexity of a multi-head monitor is the maximum number of cells on the working tapes (ignoring the read-only input tape) ever used during the computation (i.e., the maximum amount of working memory ever used by the monitor, ignoring the length of the read-only trace). A multi-head monitor might read the trace at several positions, as usual in offline monitoring. However, as its reading heads read the trace forwards, parts of the trace that were processed by all reading heads can be discarded and new time-points can be appended at the end of the trace (before the reading heads actually reach the end of the trace), as usual in online monitoring. This means that a multi-head monitor can run parallel with the system. Finally, an online monitor can be seen as a special case of a multi-head monitor that uses a single reading head.

Specification Language Features Time constraints in specifications can be expressed using the discrete (natural numbers) or dense (real numbers) time domains. To combine several time constraints (potentially over mutliple time domains), we generalize the notion of time to an abstract time domain. Example instantiations of our abstract time domain include the discrete (natural numbers) and dense (real numbers) time domains and direct and lexicographic products of time domains. A product of time domains can be used to enforce several time constraints.

We conjecture that the specification from Example 1.1 can only be implemented by standard temporal logic operators if the time domain is discrete. Still, the resulting formal specification is huge. Regular expressions are a powerful tool to succinctly express patterns, such as that from Example 1.1. Regular expressions can also help to comprehend a specification. Hence, expressiveness, succinctness, and comprehensibility are the main benefits of regular expressions.

Example 1.1 implements a security specification for a single user. A typical system for user authentication manages a collection of multiple users. In this setting, the specification from Example 1.1 can be monitored by running a sequence of monitors, one for every user. Alternatively, one can directly formulate the specification over multiple users by making the user a parameter of the specification. Formally, one formulates the specification as a first-order query and the parameter becomes a first-order variable. The alternative approach is particularly beneficial if the specification contains several first-order variables with complex dependencies between them. However, first-order monitoring is a much harder computational task than propositional monitoring. Instead of plain Boolean values, a first-order monitor must represent and manipulate valuations of the variables (also called *tuples*) over an infinite domain.

A major challenge in first-order monitoring is to find a suitable representation and efficient algorithms manipulating sets of tuples during temporal query evaluation. A *table* is a finite set of tuples. Standard database management systems (e.g., PostgreSQL) use finite tables to represent and efficiently manipulate finite sets of tuples using relational algebra operations. Hence, they restrict the class of supported queries to those for which relational algebra operations on finite tables are sufficient. These restrictions can be generalized to temporal queries. Then a first-order monitor can also use relational algebra operations on finite tables to evaluate temporal queries. Still, designing an efficient algorithm dealing with time constraints in a temporal query is challenging. For example, a monitoring algorithm checking if an event was observed at least one day ago and at most a week ago must efficiently maintain a sliding window over the trace.

Another challenging aspect is that a formalization closely following the natural language specification could yield a query that is not supported by first-order monitors that use finite tables to represent sets of tuples.

Example 1.2. Consider a shop in which brands (unary finite relation B of brands) sell products (binary finite relation P relating brands and products) and products are reviewed by users with a score (ternary finite relation S relating products, users, and scores). The relations B, P, and S might be tables in a relational database. Alternatively, they might be obtained by evaluating a temporal query over a sequence of events. We consider a brand *suspicious* if there is a user and a score such that all the brand's products were reviewed by that user with that score. A first-order query computing all suspicious brands is

$$Q^{susp} := \mathsf{B}(b) \land \exists u, s.\ \forall p.\ \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s).$$

This formalization is close to the natural language definition of suspicious brands, but it is not directly supported by first-order monitors that use finite tables to represent sets of tuples. ◇

Instead of relying on finite tables to represent finite sets of tuples, arbitrary sets of tuples can be represented and manipulated using automatic structures or binary decision diagrams. However, these alternative representations negatively impact the resulting monitor's performance.

Yet another option, ubiquitous in computer science, is to reduce our computational problem to another one. In this case, we could translate a query that is not supported by an existing first-order monitor into an equivalent query that is supported. This way, we could benefit from the optimizations implemented in existing first-order monitors.

Example 1.3. Finding suspicious brands (Example 1.2) using relational algebra (RA) or SQL is a challenge, which only the best students from an undergraduate database course could accomplish. We give away an RA answer next (where $-$ is the set difference operator and $\triangleright$ is the anti-join):

$$\pi_{brand}((\pi_{user,score}(\mathsf{S}) \times \mathsf{B}) - \pi_{brand,user,score}((\pi_{user,score}(\mathsf{S}) \times \mathsf{P}) \triangleright \mathsf{S})) \cup (\mathsf{B} - \pi_{brand}(\mathsf{P})).$$

The expressions $\pi_{user,score}(\mathsf{S})$ are called *generators.* They ensure that the left operands of the anti-join and set difference operators include or have the same columns (i.e., are union-compatible) as the corresponding right operands. ◇

Correctness The verdicts computed by a monitor are only trustworthy if the monitoring algorithm and its implementation are correct. To increase their trustworthiness, we formalize our monitors using the Isabelle/HOL proof assistant. To this end, we formulate precise definitions of the monitoring algorithms, formulate their correctness as a mathematical statement, and write a formal proof of the mathematical statement using the proof assistant. Proof assistants, e.g., Coq and Isabelle/HOL, are tools that help humans write formal proofs of mathematical statements and mechanically check these proofs. Isabelle/HOL also features a code generator producing OCaml code from the precise definitions of the monitoring algorithms. A proof assistant only accepts sound proofs that can be decomposed into small steps passing through a small and well-understood kernel. Hence, the trusted code base is reduced to that kernel and the OCaml compiler used to compile the extracted code. Numerous bugs have been found in unverified software (including runtime verification software) through its formal verification using proof assistants. This confirms that formal verification of monitoring algorithms is indispensable to guarantee that our monitors can detect all violations and do not report any false positives. Formally verified software can also be used as a testing oracle for software development, e.g., by automatically generating many test cases and correctly solving them using formally verified code.

1.2 Related Work

We present related work divided into several areas.

1.2.1 Propositional Monitoring

Linear temporal logic (LTL) was introduced by Pnueli [58]. A *formula* is a specification formulated in LTL or its extensions which we introduce later. An online monitor for past-only linear temporal logic based on dynamic programming was developed by Havelund and Roşu [43]. Roşu and Havelund [69, 70] also developed an offline monitor for future-only LTL based on dynamic programming that traverses the trace backwards. Finkbeiner and Sipma [32] developed an offline monitor for past and future LTL based on alternating automata. All these LTL monitors produce a single Boolean verdict denoting the satisfaction of an LTL formula at the first time-point of the trace.

Metric temporal logic (MTL), introduced by Koymans [47], extends LTL with metric (quantitative) time constraints. For instance, an LTL formula can express a simplified version of the speficiation from Example 1.1: "A user should not be able to authenticate after entering a wrong password." An MTL formula can further restrict the time: "A user should not be able to authenticate after entering a wrong password within the last hour." The first online monitor for metric temporal logic has been developed by Thati and Roşu [72]. It generalizes the LTL monitor by Havelund and Roşu [43] based on dynamic programming. The MTL monitor also produces a single Boolean verdict denoting the satisfaction of an LTL formula at the first time-point of the trace. Its time complexity to process one time-point and its space complexity do not depend on the number of time-points processed so far. Monitors with this highly desirable property are called *trace-length independent*. However, the worst-case time and space complexity is exponential in the number of subformulas of the MTL formula and the magnitude of time constraints.

The trace-length independent monitors [32, 43, 69, 70, 72] produce a single Boolean verdict for the first time-point of the trace. Basin et al. [7] generalize the online monitor by Thati and

Roşu [72] to an online monitor AERIAL that produces a verdict for every time-point. That is, rather than outputting whether a trace violates a specification, AERIAL outputs every position where a violation occurs. This output provides more insight into why and when the property was violated, but the associated computational problem is harder. Basin et al. [7] observe that an online monitor for MTL with both past and future temporal operators producing a verdict for every time-point cannot be trace-length independent. For example, Kane et al. [45] developed an online MTL monitor `EgMon` that produces a verdict for every time-point, but `EgMon` is not trace-length independent. Basin et al. [7] call a monitor *event-rate independent* if its space complexity does not depend on the number of events in a fixed time unit. The notion can be naturally generalized to time complexity. To develop an event-rate independent online monitor, they design a monitor that might also compute verdicts relating the satisfaction of the query at several time-points. This means that deriving a Boolean verdict for a time-point might amount to resolving chains of equivalences between several time-points.

AERIAL is actually only an *almost* event-rate independent monitor because it must still refer to time-points and a time-point can only be represented in logarithmic space. We abstract over this detail by assuming that time-points and time-stamps from the trace take constant space. The time and space complexity of AERIAL is worst-case exponential in the number of subqueries and in the magnitude of time constraints.

The suboptimal performance of AERIAL was also observed empirically and improved upon by Ulus [75] who designed an online monitor REELAY for past-only MTL. However, REELAY's time and space complexity is still linear in the magnitude of time constraints. Moreover, time-stamps are (implicitly) equal to time-points for REELAY (in particular, time-stamps are not even part of REELAY's input). Moosbrugger et al. [54] have developed an online monitor R2U2 for MLTL [48]. However, R2U2 does not support specifications mixing past and future temporal operators and time-stamps are (implicitly) equal to time-points (similarly to REELAY). Under these assumptions, MLTL is as expressive as the standard MTL [48]. However, the translation of an MTL formula might yield an equivalent MLTL formula of exponential size. The papers presenting REELAY [75] and R2U2 [54] lack detailed complexity analysis.

Linear dynamic logic (LDL), introduced by De Giacomo and Vardi [36], extends LTL with regular expressions. For instance, an LDL specification can express a simplified version of the speficiation from Example 1.1: "A user should not be able to authenticate after entering a wrong password three times without successfully authenticating in between." In a subsequent work, De Giacomo and Vardi [37] show how to monitor an LDL specification using finite automata that accept a finite trace iff it satisfies the LDL specification (i.e., a single Boolean verdict for the entire trace is produced). Metric dynamic logic, introduced by Basin et al. [12], extends LDL with metric time constraints. An metric dynamic logic specification can express the original specification from Example 1.1 including the time constraint of one hour. Basin et al. [12] also extend their MTL monitor AERIAL [7] to metric dynamic logic. In a subsequent work, Basin et al. [6] adjust the dynamic modalities in metric dynamic logic (abbreviated here as $\text{MDL}^{\mathsf{Aerial}}$) and present their joint online monitor AERIAL for MTL and $\text{MDL}^{\mathsf{Aerial}}$.

1.2.2 Relational Calculus

Relational calculus (RC) is a database query language based on first-order logic. It is well-known that satisfiability is undecidable for first-order logic (proved independently by Church [15] and Turing [74]), i.e., given a first-order query (and an *infinite* domain), it is undecidable whether there exists a structure satisfying the query. This impossibility result was also proved for first-order logic over *finite* domains by Trakhtenbrot [73]. Nevertheless, given a fixed structure and a fixed (finite or infinite) domain, it is decidable if a first-order query is satisfied over the fixed structure and the fixed domain [2].

Next we recall the fundamental notion of *capturability.* Kifer [46] calls a query class capturable if there is an algorithm that, given a query in the class and a structure, enumerates the query's evaluation result, i.e., all tuples satisfying the query. Avron and Hirshfeld [3] observe that Kifer's notion is restricted because it requires every query in a capturable class to be domain independent (a query is *domain-independent* if its evaluation result does not depend on the underlying domain). Hence, they propose an alternative definition that we also use in this thesis: A query class is capturable if there is an algorithm that, given a query in the class, a (finite or infinite) domain, and a structure, determines whether the query's evaluation result for the given structure and domain is finite (such a query is *relatively safe* with respect to the structure and domain) and enumerates the result in this case.

To measure the time and space complexity of query evaluation, Vardi [77] proposed two measures of query complexity: Data complexity [77] is the complexity of recognizing if a tuple satisfies a fixed query over a structure (database), as a function of the database size. Expression complexity [77] is the complexity of recognizing if a tuple satisfies a query over a fixed structure, as a function of the query size. Because queries are typically small and fixed while databases are large, data complexity provides a reasonable measure of real-world query complexity.

We now present approaches to evaluating RC queries grouped into three categories.

Structure reduction. The classical approach to handling arbitrary RC queries is to evaluate them under a finite structure [49]. The core question here is whether the evaluation produces the same result as defined by the natural semantics, which typically considers infinite domains. Codd's theorem [21] affirmatively answers this question for domain-independent queries, restricting the structure to the *active domain*, i.e., elements occurring in the query or database. Ailamazyan et al. [2] show that RC is a capturable query class by extending the active domain with a few additional elements, whose number depends only on the query, and evaluating the query over this finite domain. *Natural–active collapse* results generalize Ailamazyan et al.'s [2] result to extensions of RC with order relations by combining the structure reduction with a translation-based approach [14]. Hull and Su [44] study several semantics of RC that guarantee the finiteness of the query's evaluation result. In particular, the "output-restricted unlimited interpretation" only restricts the query's evaluation result to tuples that only contain elements in the active domain, but the quantified variables still range over the (finite or infinite) underlying domain.

Query translation. Another strategy is to translate a given query into one that can be evaluated efficiently, e.g., using a sequence of relational algebra (RA) operations. Van Gelder and Topor pioneered this approach [33, 34] for RC. A core component of their translation is the choice of generators, which replace the active domain restrictions from structure reduction approaches and thereby improve the time complexity. Extensions to scalar and complex function symbols have also been studied [27, 50]. All these approaches focus on syntactic classes of RC, for which domain-independence is given, e.g., the *evaluable* queries [34, Definition 5.2].

Evaluation with infinite relations. Constraint databases [68] obviate the need for using finite tables when evaluating queries and support extensions of RC with order relations. Yet the efficiency of the quantifier elimination procedures employed by constraint databases cannot compare with the simple evaluation of a projection operation in RA. Similarly, automatic structures [16] and binary decision diagrams [10, 17, 41, 42, 53] can represent the results of arbitrary RC queries finitely, but struggle with large quantities of data.

1.2.3 First-Order Monitoring

Parametric trace slicing [18] is a technique to monitor a first-order property by running a collection of propositional monitors, one for each partition of the parameters' valuations. A significant drawback of parametric trace slicing is the lack of quantifier alternation. Quantified event automata [4] extend parametric trace slicing with quantifier alternation and quantified variables ranging over elements observed in the trace. However, automata-based specifications are often large and might be difficult to comprehend.

Basin et al. [10] proposed a logic-based first-order specification language—metric first-order temporal logic (MFOTL). This declarative specification language can be seen as an extension of relational calculus with metric temporal operators and features quantifier alternation with quantified variables ranging over an arbitrary domain. Hence, MFOTL query evaluation inherits the difficulties of RC query evaluation. In particular, it has to deal with potentially infinite sets of tuples satisfying the MFOTL query or its subqueries.

Basin et al. [10] deal with infinite sets of satisfying tuples by developing two (online) monitors: MonPoly-reg uses automatic structures to represent an arbitrary (finite or infinite) set of tuples and MonPoly-fin (MonPoly for short) uses relational algebra operations on finite tables to evaluate MFOTL queries from a syntactically restricted fragment of MFOTL. The empirical evaluation clearly shows that MonPoly outperforms MonPoly-reg. On the other hand, even with heuristics to rewrite arbitrary MFOTL queries to equivalent ones that can be monitored by MonPoly, many queries fail to be monitorable by MonPoly. The monitors MonPoly-reg and MonPoly produce a verdict for every time-point in the trace (unless no verdict can be computed due to dependence on time-points beyond the finite input trace).

Schneider et al. [71] have formally verified the core of MonPoly (omitting optimizations) using the Isabelle/HOL proof assistant. The formally verified monitoring algorithm is called VeriMon. The formally verified algorithm has been optimized and extended to support regular expressions and aggregations (MonPoly also supports aggregations, but not regular expressions) in a subsequent work [8].

Havelund et al. [41, 42] deal with infinite sets of satisfying tuples by using binary decision diagrams (BDDs) to represent an arbitrary (finite or infinite) set of tuples. They develop a monitor that supports past-only metric first-order temporal logic, i.e., their monitor does not support future temporal operators. Moreover, queries may not contain free variables and the quantified variables sometimes only range over elements observed in the trace (e.g., if an equality occurs in the query). Last but not least, every time-point carries exactly one event, i.e., there cannot be concurrent events at a single time-point.

Havelund [40] has also extended first-order monitoring with recursive rules in which the recursive predicates refer to the previous time-point. For instance, such recursive rules can be used to express transitive closure over a sequence of events observed at various time-points. A recent work by Zingg et al. [79] extends the formally verified algorithm VeriMon by generalizing the recursive rules of Havelund [40]. The recursive predicates in VeriMon can refer to any time-point that is strictly in the past (not just to the previous time-point).

1.2.4 Stream runtime verification

Stream runtime verification (SRV) generalizes monitoring a single event stream to recursive programs using multiple stream expressions. The stream-based specification language Lola [23] extends propositional temporal logic specifications evaluating to Boolean verdicts with collecting statistical information about the input streams. Given a sequence of input streams, a Lola specification defines a sequence of output streams. The stream-based specification language rtLola [29, 30] extends Lola with asynchronous streams, sliding window aggregations, and parameterization. rtLola's parameterization can be seen as a generalization of parametric trace slicing [18] with the dynamic creation and termination of streams and the aggregation of statistics over the instances of a stream template [29]. Hence, rtLola supports first-order properties with implicit universal quantification on all variables (parameters). TeSSLa [22] and Striver [38] are further examples of stream-based specification languages.

If a Lola specification does not have cyclic dependencies (i.e., it is *well-formed*), then the output streams can be computed by a monitoring algorithm. Some Lola specifications can be efficiently monitored in constant space, but this fragment is rather restricted: specifications may only refer to a bounded number of future events and the bound must be fixed in advance. Efficiently monitorable rtLola specifications are restricted even further: they must not contain any references into the future. In contrast, monitors for temporal logic specifications can efficiently evaluate specifications referring to an arbitrary number of future time-points. Note that intervals of temporal logic operators only restrict time-stamps and there might be arbitrarily many time-points with the same time-stamp.

To guarantee correctness of monitoring rtLola specifications, a verified translation of rtLola specifications to Rust code was developed [31]. Manual translation of a desired property into an rtLola specification might lead to faulty specifications. To discover faulty specifications, a framework to formally prove guarantees on rtLola output streams was developed [25].

Most monitoring algorithms proceed by directly evaluating a given specification. Similarly to optimizing compilers transforming the source code's intermediate representation in a pipeline of optimization steps, transformations of rtLola's intermediate representation optimizing the monitoring algorithm's performance were developed [13].

1.3 Contributions

In the following, we give an overview of our contributions.

Semantics We generalize the notion of time in specification languages and redefine metric dynamic logic, introduced in previous work and abbreviated here as $\mathrm{MDL}^{\mathsf{Aerial}}$, to make regular expressions easier to write and comprehend.

Multi-Head Monitoring We develop multi-head monitors for two propositional specification languages: metric temporal logic (MTL) and metric dynamic logic (MDL). Our monitors improve upon the state-of-the-art by producing a sequence of Boolean verdicts denoting the query's satisfaction at every time-point in the order they appear in the trace and by optimizing the time and space complexity. We have implemented our monitors, empirically confirmed that their performance improves upon existing approaches, and formally verified the correctness of our monitors using the Isabelle/HOL proof assistant.

Monitoring Any MFOTL Query using Relational Algebra A proper subset of first-order queries can be efficiently evaluated using relational algebra operations on finite tables. We show that an arbitrary (temporal) first-order query can be evaluated using relational algebra operations on finite tables.

We first investigate relational calculus (RC). We formally verify the approach by Ailamazyan et al. [2] to evaluate an arbitrary RC query over a structure using relational algebra operations. This approach uses the constants from the RC query and the structure and a few additional elements to obtain a finite domain for evaluating the query. It shows that evaluating an arbitrary RC query is feasible, but the actual performance is rather poor.

Hence, we propose a novel approach to RC query evaluation by translating an arbitrary RC query into a pair of relational algebra normal form (RANF) queries that can be evaluated using relational algebra operations on finite tables: one characterizes the original query's relative safety (i.e., whether it evaluates to a finite relation) and the other is equivalent to the original query if the original query is relatively safe. We implement our translation and empirically confirm that the performance of evaluating the queries produced by our translation improves upon existing approaches to RC query evaluation.

Finally, we generalize our translation of RC queries to MFOTL queries. This way, we obtain a monitor for an arbitrary MFOTL query that decides for every time-point if the corresponding relation is finite and computes the relation if it is finite.

Optimizing Temporal Operators in First-Order Monitoring We optimize the time complexity of evaluating VeriMon's Since and Until temporal operators.

Publications

This thesis is built upon the following peer-reviewed conference publications.

[66] Martin Raszyk, David Basin, Dmitriy Traytel. From Nondeterministic to Multi-Head Deterministic Finite-State Transducers. In 46th International Colloquium on Automata, Languages and Programming (ICALP 2019), LIPIcs 132, pp. 127:1–127:14, Schloss Dagstuhl – Leibniz-Zentrum für Informatik 2019.

[64] Martin Raszyk, David Basin, Srđan Krstić, Dmitriy Traytel. Multi-Head Monitoring of Metric Temporal Logic. In 17th International Symposium on Automated Technology for Verification and Analysis (ATVA 2019), LNCS 11781, pp. 151–170, Springer 2019.

[8] David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, Dmitriy Traytel. A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic. In 10th International Joint Conference on Automated Reasoning (IJCAR 2020), LNCS 12166, pp. 432–453, Springer 2020.

[67] Martin Raszyk, David Basin, Dmitriy Traytel. Multi-head Monitoring of Metric Dynamic Logic. In 18th International Symposium on Automated Technology for Verification and Analysis (ATVA 2020), LNCS 12302, pp. 233–250, Springer 2020.

[65] Martin Raszyk, David Basin, Srđan Krstić, Dmitriy Traytel. Practical Relational Calculus Query Evaluation. In 25th International Conference on Database Theory (ICDT 2022), LIPICcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik 2022.

Parts of the thesis have been formally verified using the Isabelle/HOL proof assistant and the proofs have been published in the Archive of Formal Proofs (AFP). In the following, we list the AFP entries and places in the thesis to which they refer:

Section 3.2	[63]	Martin Raszyk: Multi-Head Monitoring of Metric Dynamic Logic.
Section 4.2.7	[61]	Martin Raszyk: First-Order Query Evaluation.
Section 4.4	[24]	Thibault Dardinier, Lukas Heimes, Martin Raszyk, Joshua Schneider, Dmitriy Traytel: Formalization of an Optimized Monitoring Algorithm for Metric First-Order Dynamic Logic with Aggregations.

Parts of the thesis have also been implemented as executable tools used for our empirical evaluations. In the following, we list these tools and and places in the thesis to which they refer:

Section 3.2	[59]	Martin Raszyk: HYDRA.
Section 4.2	[60]	Martin Raszyk: RC2SQL.
Section 4.3	[62]	Martin Raszyk: MFOTL2RANF.
Section 4.4	[9]	Thibault Dardinier, Lukas Heimes, Nicolas Kaletsch, Srđan Krstić, Emanuele Marsicano, Martin Raszyk, Joshua Schneider, Dmitriy Traytel, Sheila Zinggs: VERIMON.

1.4 Thesis outline

Finally, we outline the structure of the thesis.

Chapter 2 In this chapter, we introduce specification languages to formally describe the intended system behaviour: metric temporal logic (MTL), metric dynamic logic (MDL), and metric first-order temporal logic (MFOTL).

Chapter 3 In this chapter, we present our multi-head monitors for metric temporal logic [64] and metric dynamic logic [66, 67].

Chapter 4 In this chapter, we present our results on efficiently evaluating arbitrary (temporal) first-order queries using relational algebra operations on finite tables [65]. In particular, we present our optimizations of temporal operators in VERIMON [8].

Chapter 2

Specification Languages

In this chapter, we introduce specification languages that are used to formally describe the *intended* system behaviour. A specification can be an input of a monitor in addition to a trace that describes the *observed* system behaviour. Specification languages can be classified according to various dimensions that have been investigated in a taxonomy by Falcone et al. [28]. In the following, we summarize these dimensions and position our work with respect to them. We refer to the taxonomy by Falcone et al. [28] for example specifications illustrating the classification.

Implicit and explicit specifications An *implicit* specification is a built-in specification describing common properties of a well-understood intended system behaviour, e.g., (memory) safety, concurrency, and (system) security. An *explicit* specification is a custom specification provided by a user of a monitor. We focus on explicit specifications that give users more flexibility to formalize the intended system behaviour. In general, the price for this flexibility is the quality of the monitor's output and the monitor's time and space complexity. In our work, we develop efficient monitors that provide detailed output.

Operational and declarative specifications An *operational* specification describes *how* the monitor should check the behaviour of the target system, e.g., using a finite-state automaton. A *declarative* specification describes *what* the monitor should check in the behaviour of the target system, e.g., using a temporal logic formula. We focus on declarative specifications.

Time A specification can also express constraints over time. Such constraints refer either to *logical* time or *physical* time. Logical time only constraints the relative ordering of events. An example specification language relying purely on logical time is linear temporal logic (LTL) [58]. The relative ordering of events can be either *total* (e.g., in a single-threaded program) or *partial* (e.g., in a distributed system). In our work, we consider the standard semantics of temporal logics based on Kripke structures: the observed system behaviour is modeled by an infinite sequence of sets of events. A set of events in the sequence is called a *time-point* and the events in the set are considered to happen concurrently, i.e., the events at a single time-point are not ordered. This semantics of events can also be viewed as a partial ordering of events that are grouped into totally ordered groups of events. Within one such group (time-point), events are not relatively ordered, i.e., they are considered to happen concurrently.

A specification can also refer to physical time of events. In this case, the target system must associate a *time-stamp* with every event. Metric temporal logic (MTL) [47] extends LTL with constraints on physical time. The taxonomy [28] distinguishes between discrete (e.g., natural numbers) and dense (e.g., real numbers) physical time domains. We generalize these time domains to an abstract time domain with associated algebraic properties.

Modality The *modality* of a specification restricts constraints over logical time to past, current, and future events. The simplest form of assertions supported by most programming languages can only express properties of events that happen concurrently at the *current* time-point. A temporal logic adds constraints over past or future events (or both). The various fragments (e.g., past-only, future-only, and arbitrary) result in trade-offs in terms of expressiveness (what properties can be expressed), conciseness (how succinctly a property can be expressed), and performance (how efficient a property can be checked by a monitor). Our monitors support all three modalities. To mitigate a negative impact on the performance, we propose a novel monitoring paradigm—*multi-head* monitoring.

Data The target system may produce *propositional* events from a finite set of events (characterized only by their name) or *parameterized* events (characterized by a name and a list of data values, called parameters). A collection of parameterized events can be viewed as a database. Consequently, a specification that refers to parameterized events can be viewed as a (temporal) database query. Parameterized events substantially extend the class of properties that can be checked, but they also make monitoring a substantially harder computational problem. In our work, we consider both specifications that refer to propositional events as well as specifications that refer to parameterized events.

Output A specification can be checked at a single position (time-point) in the trace or at every time-point in the trace. The latter option provides more details about the system's compliance with the specification by distinguishing the time-points at which the specification is satisfied from time-points at which the specification is not satisfied. On the other hand, the former option can promote the performance of the monitor. In our work, we opt for the latter option and develop monitors that compute Boolean verdicts (for propositional logics) and tuples of data values (for first-order logics) for every time-point in the trace.

2.1 Time Domain

We generalize the discrete (e.g., natural numbers) and dense (e.g., real numbers) time domains used in previous work to an abstract time domain $\mathbb{T}$. Formally, $\mathbb{T}$ must form an additive commutative monoid $(\mathbb{T}, +, 0)$, a partial order $(\mathbb{T}, \leq)$, and a join-semilattice $(\mathbb{T}, \sqcup)$. The partial order must be consistent with $\sqcup$ and addition, i.e., $a \leq a \sqcup b$, $b \leq a \sqcup b$, $(a \leq c \text{ and } b \leq c) \implies a \sqcup b \leq c$, $b \leq c \implies a + b \leq a + c$, for all $a, b, c \in \mathbb{T}$. Moreover, we denote by $\mathbb{T}_{\text{fin}}$ a subset of time-stamps, $0 \in \mathbb{T}_{\text{fin}}$, $\mathbb{T}_{\text{fin}} \subseteq \mathbb{T}$, that are considered *finite*. Specifically, the subset $\mathbb{T}_{\text{fin}}$ must be closed under addition. From the partial order $(\mathbb{T}, \leq)$, we derive the strict partial order $(\mathbb{T}, <)$ as follows: $\tau < \tau'$ if and only if $\tau \leq \tau'$ and not $\tau = \tau'$. Then we assume that $0 < c \Longrightarrow a < a + c$, for all $a \in \mathbb{T}_{\text{fin}}$ and $c \in \mathbb{T}$. Finally, we assume the existence of an order-preserving embedding ι of natural numbers into finite time-stamps that are progressing by any finite amount of time, i.e., for every $i \in \mathbb{N}$ we have $\iota(i) \in \mathbb{T}_{\text{fin}}$ and for every $\Delta \in \mathbb{T}_{\text{fin}}$ there exists some $j \in \mathbb{N}$ such that $\iota(j) \leq \iota(i) + \Delta$ does not hold. For example, these assumptions are satisfied by both the discrete natural numbers $\mathbb{T} = \mathbb{N} \cup \{\infty\}$ extended with infinity and the dense real numbers $\mathbb{T} = \mathbb{R} \cup \{-\infty, +\infty\}$ extended with infinity.

We call a time domain $\mathbb{T}$ *total* if it is totally ordered and no positive infinite time-stamp is less than a positive finite time-stamp: $a \leq b$ holds for all $a \in \mathbb{T}_{\text{fin}}$ and $b \in \mathbb{T} - \mathbb{T}_{\text{fin}}$ such that $0 \leq a$ and $0 \leq b$. We call a time domain $\mathbb{T}$ *strict* if, for all $a, b, c \in \mathbb{T}$, $b < c \implies a + b < a + c$

holds. For example, the natural numbers $\mathbb{T} = \mathbb{N}$ and the real numbers $\mathbb{T} = \mathbb{R}$ are total and strict time domains.

We can also combine two time domains $\mathbb{T}_1$ and $\mathbb{T}_2$ into a new time domain $\mathbb{T}_1 \times \mathbb{T}_2$ that is ordered by the product order, i.e., $(\tau_1, \tau_2) \leq_\times (\tau_1', \tau_2')$ if and only if $\tau_1 \leq_1 \tau_1'$ and $\tau_2 \leq_2 \tau_2'$. If $\mathbb{T}_1$ and $\mathbb{T}_2$ are total and strict time domains, then the new time domain $\mathbb{T}_1 \times \mathbb{T}_2$ is a total and strict time domain if it is ordered by the lexicographic order, i.e., $(\tau_1, \tau_2) \leq_\times (\tau_1', \tau_2')$ if and only if $\tau_1 <_1 \tau_1'$ or ($\tau_1 = \tau_1'$ and $\tau_2 \leq_2 \tau_2'$).

Intervals We define the following four types of *intervals* over $\mathbb{T}$, where $a \in \mathbb{T}_{\text{fin}}$, $b \in \mathbb{T}$, and $0 \leq a \leq b$:

Type	Notation	Condition
closed	$[a, b]$	
open	$]a, b[$	$b \neq 0$
left-open	$]a, b]$	
right-open	$[a, b[$	$b \neq 0$

We denote the set of all (closed, open, left- and right-open) intervals as $\mathbb{I}$. Given an interval $I \in \mathbb{I}$, we denote by $\mathsf{right}(I)$ its upper bound. Formally, we define $\mathsf{right}([a, b]) = \mathsf{right}(]a, b[) = \mathsf{right}(]a, b]) = \mathsf{right}([a, b[) = b$. The lower bound $\mathsf{left}(I)$ could be defined analogously, but we do not need this notion explicitly in the thesis. Given a pair of time-stamps $\tau, \tau' \in \mathbb{T}$, $\tau \leq \tau'$, and an interval $I \in \mathbb{I}$, we say that the time-stamps τ and τ' satisfy the interval's lower ($\mathsf{memL}(\tau, \tau', I)$) and upper ($\mathsf{memR}(\tau, \tau', I)$) bound condition, respectively, if

$$\begin{array}{llll}
\mathsf{memL}(\tau, \tau', [a, b]) \text{ iff } \tau + a \leq \tau', & & \mathsf{memR}(\tau, \tau', [a, b]) \text{ iff } \tau' \leq \tau + b, \\
\mathsf{memL}(\tau, \tau',]a, b[) \text{ iff } \tau + a < \tau', & & \mathsf{memR}(\tau, \tau',]a, b[) \text{ iff } \tau' < \tau + b, \\
\mathsf{memL}(\tau, \tau',]a, b]) \text{ iff } \tau + a < \tau', & & \mathsf{memR}(\tau, \tau',]a, b]) \text{ iff } \tau' \leq \tau + b, \\
\mathsf{memL}(\tau, \tau', [a, b[) \text{ iff } \tau + a \leq \tau', & & \mathsf{memR}(\tau, \tau', [a, b[) \text{ iff } \tau' < \tau + b.
\end{array}$$

Given a pair of time-stamps $\tau, \tau' \in \mathbb{T}$ and an interval $I \in \mathbb{I}$, we say that the time-stamps τ and τ' satisfy the interval condition ($\mathsf{mem}(\tau, \tau', I)$) if they satisfy the interval's lower and upper bound conditions:

$$\mathsf{mem}(\tau, \tau', I) \text{ iff } \mathsf{memL}(\tau, \tau', I) \text{ and } \mathsf{memR}(\tau, \tau', I).$$

We call an interval I *full* if $\mathsf{mem}(\tau, \tau', I)$ holds for all $\tau, \tau' \in \mathbb{T}_{\text{fin}}$ such that $0 \leq \tau \leq \tau'$. Given a total time domain $\mathbb{T}$, intervals I such that $\mathsf{memL}(0, 0, I)$ and $\mathsf{right}(I) \notin \mathbb{T}_{\text{fin}}$ are full intervals. Note that a full interval might not be unique for a time domain with multiple time-stamps $b, c \in \mathbb{T} - \mathbb{T}_{\text{fin}}$, $0 \leq b$, and $0 \leq c$.

For instance, for the time domain $\mathbb{T} = \mathbb{N}$ of natural numbers, we have $\mathsf{memL}(\tau, \tau', [a, b]) \iff a \leq \tau' - \tau$, $\mathsf{memR}(\tau, \tau', [a, b]) \iff \tau' - \tau \leq b$, and $\mathsf{mem}(\tau, \tau', [a, b]) \iff \tau' - \tau \in \{n \mid a \leq n \leq b\}$. For $\mathbb{T} = \mathbb{N}$, no full interval exists. For $\mathbb{T} = \mathbb{N} \cup \{\infty\}$, the interval $[0, \infty]$ is full.

Given an interval $I \in \mathbb{I}$, we define $\mathsf{dropL}(I)$ to be an interval such that $\mathsf{memL}(\tau, \tau', \mathsf{dropL}(I))$ holds, for all $\tau \leq \tau'$, and $\mathsf{memR}(\tau, \tau', \mathsf{dropL}(I)) \iff \mathsf{memR}(\tau, \tau', I)$ holds, for all τ, τ'. Formally, we define $\mathsf{dropL}([a, b]) = \mathsf{dropL}(]a, b]) = [0, b]$ and $\mathsf{dropL}([a, b[) = \mathsf{dropL}(]a, b[) = [0, b[$.

Given an interval $I \in \mathbb{I}$ over a total time domain $\mathbb{T}$ whose lower bound is not 0 (inclusive), i.e., not $\mathsf{memL}(0, 0, I)$, we define $\mathsf{flipL}(I)$ to be an interval such that $\mathsf{memL}(\tau, \tau', \mathsf{flipL}(I))$ holds, for all $\tau \leq \tau'$, and $\mathsf{memR}(\tau, \tau', \mathsf{flipL}(I)) \iff$ not $\mathsf{memL}(\tau, \tau', I)$ holds, for all τ, τ'. Formally, we define $\mathsf{flipL}([a, b]) = \mathsf{flipL}(]a, b]) = [0, b[$ and $\mathsf{flipL}([a, b[) = \mathsf{flipL}(]a, b[) = [0, b]$.

$(d1)$ $d(t,t') = 0 \iff t = t'$;
$(d2)$ $d(t,t') = d(t',t)$;
$(d3)$ if $t < t' < t''$ then $d(t,t'') = d(t,t') + d(t',t'')$ and $d(t'',t) = d(t'',t') + d(t',t)$;
$(\Delta 1)$ $\delta + \delta' = \delta' + \delta$;
$(\Delta 2)$ $(\delta + \delta') + \delta'' = \delta + (\delta' + \delta'')$;
$(\Delta 3)$ $\delta + 0 = \delta = 0 + \delta$;
$(\Delta 4)$ $(\delta + \delta' = \delta + \delta'' \implies \delta' = \delta'')$ and $(\delta + \delta'' = \delta' + \delta'' \implies \delta = \delta')$;
$(\Delta 5)$ $\delta + \delta' = 0 \implies \delta = 0$ and $\delta' = 0$;
$(\Delta 6)$ $\exists \delta''. (\delta = \delta' + \delta''$ or $\delta' = \delta + \delta'')$.

Figure 2.1. Conditions on a metric point structure.

Related Work: Metric Point Structure To abstractly model time, Koymans [47] introduced the notion of a *metric point structure*.

Definition 2.1 ([47, Definition 4.1]). *A* metric point structure *is a two-sorted structure* $(T, \Delta, <, d, +, 0)$ *with signature* $< \subseteq T \times T$, $d : T \times T \to \Delta$, $+ : \Delta \times \Delta \to \Delta$, $0 \in \Delta$ *such that*

(i) $<$ *is total,*

(ii) d *is surjective and satisfies (d1)–(d3),*

(iii) $(\Delta, +, 0)$ *satisfies* $(\Delta 1)$*–*$(\Delta 6)$*.*

Δ and d are called the *metric domain* and the *temporal distance function*, respectively. Koymans [47, Section 4] also derives a strict total order $\ll$ on Δ from the temporal distance function. The conditions (d1)–(d3) and $(\Delta 1)$–$(\Delta 6)$ are defined in [47, Section 4]. We summarize them in Figure 2.1. Because Koymans assumes the strict precedence relation $<$ on time-stamps to be “transitive, irreflexive and comparable”, the precedence relation is actually a partial order.

We now provide a conversion of a metric point structure to our abstract time domain $\mathbb{T}$. To this end, we define $\mathbb{T} := \Delta \cup \{\infty\}$, where ∞ is a special value representing an infinite time-stamp, we further define $\mathbb{T}_{\text{fin}} := \Delta$, we derive $\sqcup$ from the strict total order $\ll$ on Δ, and assume the existence of an order-preserving embedding ι of natural numbers into Δ such that the values $\iota(i)$ are progressing by an arbitrary value $x \in \Delta$, i.e., for every $i \in \mathbb{N}$ we have $\iota(i) \in \Delta$ and for every $x \in \Delta$ there exists some $j \in \mathbb{N}$ such that $\iota(j) \leq \iota(i) + x$ does not hold. The existence of an order-preserving embedding ι is necessary to guarantee that there exists an infinite trace such that the satisfaction of future temporal operators with conditions $d(t,t') \ll x$ only depends on finitely many time-points of the trace. We map Koymans' time-stamps $t \in T$ in a trace to $d(t_0, t) \in \Delta \subseteq \mathbb{T}$, where t_0 is the initial time-stamp of the trace. Then the conditions $d(t,t') \ll x$, $d(t,t') = x$, and $d(t,t') \gg x$ can be expressed by our interval conditions as shown in the following lemmas.

Lemma 2.2 ([63, metric_point_structure_lt_mem]). *Let* t_0 *be the initial time-stamp of a trace and* t, t' *be arbitrary two time-stamps in the trace such that* $t_0 \leq t \leq t'$*. Let* $x \in \Delta$ *be such that* $0 \ll x$*. Then* $d(t,t') \ll x \iff \mathsf{mem}(d(t_0,t), d(t_0,t'), [0, x[)$*.*

Lemma 2.3 ([63, metric_point_structure_eq_mem]). *Let* t_0 *be the initial time-stamp of a trace and* t, t' *be arbitrary two time-stamps in the trace such that* $t_0 \leq t \leq t'$*. Let* $x \in \Delta$*. Then* $d(t,t') = x \iff \mathsf{mem}(d(t_0,t), d(t_0,t'), [x, x])$*.*

$(\rho, i) \models p$	iff $p \in \Gamma_i$
$(\rho, i) \models \neg\phi$	iff not $(\rho, i) \models \phi$
$(\rho, i) \models \phi \vee \psi$	iff $(\rho, i) \models \phi$ or $(\rho, i) \models \psi$
$(\rho, i) \models \bullet_I\ \phi$	iff $i > 0$, $\mathsf{mem}(\tau_{i-1}, \tau_i, I)$, and $(\rho, i-1) \models \phi$
$(\rho, i) \models \bigcirc_I\ \phi$	iff $\mathsf{mem}(\tau_i, \tau_{i+1}, I)$ and $(\rho, i+1) \models \phi$
$(\rho, i) \models \phi\ \mathsf{S}_I\ \psi$	iff j exists with $j \leq i$, $\mathsf{mem}(\tau_j, \tau_i, I)$, $(\rho, j) \models \psi$, and $(\rho, k) \models \phi$, for all $j < k \leq i$
$(\rho, i) \models \phi\ \mathsf{U}_I\ \psi$	iff j exists with $j \geq i$, $\mathsf{mem}(\tau_i, \tau_j, I)$, $(\rho, j) \models \psi$, and $(\rho, k) \models \phi$, for all $i \leq k < j$

Figure 2.2. Semantics of MTL.

Lemma 2.4 ([63, metric_point_structure_gt_mem]). *Let t_0 be the initial time-stamp of a trace and t, t' be arbitrary two time-stamps in the trace such that $t_0 \leq t \leq t'$. Let $x \in \Delta$. Then $x \ll d(t, t') \iff \mathsf{mem}(d(t_0, t), d(t_0, t'),]x, \infty])$.*

On the other hand, because the strict order $\ll$ on Δ must be total by condition (Δ6), we cannot convert the product time domain on pairs of natural numbers $\mathbb{T} := \mathbb{N} \times \mathbb{N}$ ordered by the product order, i.e., $(\tau_1, \tau_2) \leq_\times (\tau_1', \tau_2')$ if and only if $\tau_1 \leq_1 \tau_1'$ and $\tau_2 \leq_2 \tau_2'$, which is not total, to a metric point structure with the natural choice $\Delta \subseteq T = \mathbb{T}$. This means that a nontrivial Δ and a nontrivial temporal distance function d would be needed in such a case. We could not come up with any such Δ and d. We thus conjecture that our abstract notion of time is more general than Koymans' metric point structures.

2.2 Propositional Temporal Logics

In this section, we formally define the syntax and semantics of MTL and MDL—an extension of MTL with regular expressions [6]. We first formalize the observed system behaviour and then a specification expressing the intended system behaviour. We assume that the observed system behaviour is a finite prefix $\rho_{<\ell} = (\rho_i)_{i<\ell}$ (called a *trace*) of an infinite stream $\rho = \langle(\Gamma_i,\ \tau_i)\rangle_{i\in\mathbb{N}}$ over a finite set of atomic propositions Σ. The stream is an infinite sequence of time-points $i \in \mathbb{N}$, each consisting of a set of atomic propositions $\Gamma_i \subseteq \Sigma$ and a time-stamp $\tau_i \in \mathbb{T}_{\text{fin}}$. The time-stamps must be monotone ($\tau_i \leq \tau_{i+1}$, for all $i \in \mathbb{N}$) and progressing by an arbitrary finite amount of time (for every $i \in \mathbb{N}$ and $\Delta \in \mathbb{T}_{\text{fin}}$ there exists some $j \in \mathbb{N}$ such that $\tau_j \leq \tau_i + \Delta$ does not hold). Consecutive time-points may carry the same time-stamp and there might be time-stamps that no time-point carries.

The intended system behaviour can be expressed by an MTL formula. The syntax of MTL formulas ϕ over a finite set of atomic propositions Σ is defined recursively:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \bullet_I\ \phi \mid \bigcirc_I\ \phi \mid \phi\ \mathsf{S}_I\ \phi \mid \phi\ \mathsf{U}_I\ \phi,$$

where $p \in \Sigma$ and $I \in \mathbb{I}$. This minimal syntax includes Boolean operators $\neg$ (*not*) and $\vee$ (*or*) and the temporal operators $\bullet_I$ (*previous*), S_I (*since*), $\bigcirc_I$ (*next*), and U_I (*until*). We employ the usual syntactic sugar for additional Boolean constants and operators *true* $= p \vee \neg p$, *false* $= \neg$*true*, $\phi \wedge \psi = \neg(\neg\phi \vee \neg\psi)$, and additional temporal operators $\blacklozenge_I\ \phi =$ *true* $\mathsf{S}_I\ \phi$ (*once*), $\blacksquare_I\ \phi = \neg\blacklozenge_I\ \neg\phi$ (*historically*), $\lozenge_I\ \phi =$ *true* $\mathsf{U}_I\ \phi$ (*eventually*), and $\square_I\ \phi = \neg\lozenge_I\ \neg\phi$ (*always*).

We define the standard point-based semantics of MTL in Figure 2.2. We refer to Basin et al. [11] for a comprehensive comparison of alternative semantics. We remark that it might not be possible to evaluate an MTL formula at a time-point of a trace without knowing its continuation,

$$
\begin{array}{llll}
(\rho, i) \models^{\mathsf{Aerial}} & \langle r|_I & \text{iff } j \text{ exists with } j \leq i,\ \tau_i - \tau_j \in I \text{ and } (j, i) \in \mathcal{R}^{\mathsf{Aerial}}_\rho(r) \\
(\rho, i) \models^{\mathsf{Aerial}} & |r\rangle_I & \text{iff } j \text{ exists with } j \geq i,\ \tau_j - \tau_i \in I \text{ and } (i, j) \in \mathcal{R}^{\mathsf{Aerial}}_\rho(r)
\end{array}
$$

$$
\begin{array}{llll}
(\rho, i) \models & \langle r|_I & \text{iff } j \text{ exists with } j \leq i,\ \mathsf{mem}(\tau_j, \tau_i, I) \text{ and } (j, i+1) \in \mathcal{R}_\rho(r) \\
(\rho, i) \models & |r\rangle_I & \text{iff } j \text{ exists with } j \geq i,\ \mathsf{mem}(\tau_i, \tau_j, I) \text{ and } (i, j+1) \in \mathcal{R}_\rho(r)
\end{array}
$$

$$
\begin{array}{llll}
\mathcal{R}^{\mathsf{Aerial}}_\rho(\phi?) & = \{(i,i) \mid (\rho, i) \models^{\mathsf{Aerial}} \phi\} & \mathcal{R}_\rho(\phi?) & = \{(i,i) \mid (\rho, i) \models \phi\} \\
\mathcal{R}^{\mathsf{Aerial}}_\rho(\star) & = \{(i,i+1) \mid i \in \mathbb{N}\} & \mathcal{R}_\rho(\phi) & = \{(i,i+1) \mid (\rho, i) \models \phi\} \\
\mathcal{R}^{\mathsf{Aerial}}_\rho(r+s) & = \mathcal{R}^{\mathsf{Aerial}}_\rho(r) \cup \mathcal{R}^{\mathsf{Aerial}}_\rho(s) & \mathcal{R}_\rho(r+s) & = \mathcal{R}_\rho(r) \cup \mathcal{R}_\rho(s) \\
\mathcal{R}^{\mathsf{Aerial}}_\rho(r \cdot s) & = \mathcal{R}^{\mathsf{Aerial}}_\rho(r) \cdot \mathcal{R}^{\mathsf{Aerial}}_\rho(s) & \mathcal{R}_\rho(r \cdot s) & = \mathcal{R}_\rho(r) \cdot \mathcal{R}_\rho(s) \\
\mathcal{R}^{\mathsf{Aerial}}_\rho(r^*) & = \mathcal{R}^{\mathsf{Aerial}}_\rho(r)^* & \mathcal{R}_\rho(r^*) & = \mathcal{R}_\rho(r)^*
\end{array}
$$

Figure 2.3. Semantics of MDL$^{\mathsf{Aerial}}$ and MDL.

i.e., the infinite stream. For instance, the formula $\bigcirc_I\, p$ cannot be evaluated at the last time-point $\ell - 1$ of a trace $\rho_{<\ell}$ without the next time-point ℓ of the infinite stream, i.e., Γ_ℓ and τ_ℓ.

Basin et al. [6] introduced metric dynamic logic (refered to as MDL$^{\mathsf{Aerial}}$ in this work) as an extension of MTL with regular expressions. The time domain of MDL$^{\mathsf{Aerial}}$ is fixed to the natural numbers $\mathbb{T} = \mathbb{N}$. The syntax of MDL$^{\mathsf{Aerial}}$ formulas ϕ over a finite set of atomic propositions Σ and MDL$^{\mathsf{Aerial}}$ regular expressions r is defined by mutual recursion:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \langle r|_I \mid |r\rangle_I \qquad \text{and} \qquad r ::= \phi? \mid \star \mid r + r \mid r \cdot r \mid r^*,$$

where $p \in \Sigma$ and $I \in \mathbb{I}$. This minimal syntax of MDL$^{\mathsf{Aerial}}$ formulas includes Boolean operators and regular expression match operators. The future match operator $|r\rangle_I$ evaluated at a time-point i expresses that there exists some future time-point j, $i \leq j$, whose time-stamp τ_j satisfies $\tau_j - \tau_i \in I$ and the regular expression r matches at the time-points between i and j. The past match operator $\langle r|_I$ expresses the dual property about past time-points. MDL$^{\mathsf{Aerial}}$ regular expressions consist of MDL$^{\mathsf{Aerial}}$ formulas ϕ to characterize the matched time-points in the form of *lookaheads* $\phi?$. The MDL$^{\mathsf{Aerial}}$ regular expression $\star$ always matches a time-point and advances regular expression matching to the next time-point. The operators $+$, $\cdot$, and * are the standard alternation, concatenation, and (Kleene) star operators.

The point-based semantics of an MDL$^{\mathsf{Aerial}}$ formula ϕ on a stream ρ, $(\rho, i) \models^{\mathsf{Aerial}} \phi$, and MDL$^{\mathsf{Aerial}}$ regular expressions is defined by mutual recursion in Figure 2.3. We omit the cases of atomic propositions and Boolean operators that are identical to MTL. The semantics of an MDL$^{\mathsf{Aerial}}$ regular expression r on a stream ρ is a relation $\mathcal{R}^{\mathsf{Aerial}}_\rho(r) \subseteq \mathbb{N} \times \mathbb{N}$, where $(i, j) \in \mathcal{R}^{\mathsf{Aerial}}_\rho(r)$ iff r matches the time-points between i (inclusive) and j (inclusive). We call $(i, j) \in \mathcal{R}^{\mathsf{Aerial}}_\rho(r)$ a *match*. A match of the form (i, i) is called *empty*.

It is possible to express MTL temporal operators by regular expressions. For instance, one can express $\phi\ \mathsf{S}_I\ \psi$ by the regular expression $\psi \cdot \phi^*$ matched backwards (and ignoring the interval I for now). However, $\psi \cdot \phi^*$ is not an MDL$^{\mathsf{Aerial}}$ regular expression because the formulas ψ and ϕ can only appear as lookaheads $\psi?$ and $\phi?$ in MDL$^{\mathsf{Aerial}}$ regular expressions. Moreover, we need to insert the MDL$^{\mathsf{Aerial}}$ regular expression $\star$ into the gaps between the lookaheads and then obtain the MDL$^{\mathsf{Aerial}}$ regular expression $\psi? \cdot (\star \cdot \phi?)^*$. Finally, taking the interval I into account, the MTL formula $\phi\ \mathsf{S}_I\ \psi$ can be equivalently expressed by the MDL$^{\mathsf{Aerial}}$ formula $\langle \psi? \cdot (\star \cdot \phi?)^*|_I$. We introduce our definition of metric dynamic logic (refered to as MDL in this work) derived from MDL$^{\mathsf{Aerial}}$ with the goal to avoid inserting $\star$ into the gaps between the individual lookaheads. This makes it easier for a user to write MDL regular expressions (compared to $\text{MDL}^{\mathsf{Aerial}}$). Moreover, we conjecture that MDL regular expressions are asymptotically more succinct than equivalent $\text{MDL}^{\mathsf{Aerial}}$ regular expressions. In more detail, for the following class of well-formed MDL regular expressions r_i, $i \in \mathbb{N}$:

$$r_i = \begin{cases} \phi_0^* & \text{if } i = 0, \\ (\phi_i \cdot r_{i-1})^* & \text{if } i > 0, \end{cases}$$

we could not derive smaller equivalent $\text{MDL}^{\mathsf{Aerial}}$ regular expressions than $\tilde{r_i} := \mathsf{embed}'(\mathsf{rderive}(r_i))$ (Figure 2.7) that are of quadratic size:

$$\tilde{r_i} = \begin{cases} (\phi_0? \cdot \star)^* \cdot \phi_0? & \text{if } i = 0, \\ (\phi_i? \cdot \star \cdot \hat{r}_{i-1})^* \cdot (\phi_i? + \phi_i? \cdot \star \cdot \tilde{r_{i-1}}) & \text{if } i > 0, \end{cases}$$

where

$$\hat{r_i} = \begin{cases} (\phi_0? \cdot \star)^* & \text{if } i = 0, \\ (\phi_i? \cdot \star \cdot \hat{r}_{i-1})^* & \text{if } i > 0. \end{cases}$$

The MDL regular expressions r_0, r_1, r_2 have the following simple form: $(\phi_0)^*, (\phi_1 \cdot (\phi_0)^*)^*, (\phi_2 \cdot (\phi_1 \cdot (\phi_0)^*)^*)^*$ while the equivalent $\text{MDL}^{\mathsf{Aerial}}$ regular expressions $\tilde{r_0}, \tilde{r_1}, \tilde{r_2}$ have the following form: $(\phi_0? \cdot \star)^* \cdot \phi_0?, (\phi_1? \cdot \star \cdot (\phi_0? \cdot \star)^*)^* \cdot (\phi_1? + \phi_1? \cdot \star \cdot (\phi_0? \cdot \star)^* \cdot \phi_0?), (\phi_2? \cdot \star \cdot (\phi_1? \cdot \star \cdot (\phi_0? \cdot \star)^*)^*)^* \cdot (\phi_2? + \phi_2? \cdot \star \cdot (\phi_1? \cdot \star \cdot (\phi_0? \cdot \star)^*)^* \cdot (\phi_1? + \phi_1? \cdot \star \cdot (\phi_0? \cdot \star)^* \cdot \phi_0?))$.

We formally define the syntax of MDL formulas ϕ by extending the syntax of MTL formulas with past and future regular expression match operators: $\langle r|_I$ and $|r\rangle_I$ and define the syntax of MDL regular expressions r by mutual recursion:

$$r ::= \phi? \mid \phi \mid r + r \mid r \cdot r \mid r^*.$$

We call an MDL formula ϕ a *direct subformula* of an MDL regular expression r if ϕ occurs in r and ϕ is not a proper subformula of any other formula that occurs in r. We denote the set of all direct subformulas of an MDL regular expression r as $\mathsf{SF}(r)$. Because MTL operators can be evaluated more efficiently than the equivalent MDL regular expressions (using a general algorithm for MDL regular expressions), we include MTL operators in MDL. We also generalize $\mathbb{T} = \mathbb{N}$ to an abstract domain $\mathbb{T}$ (Section 2.1).

We define the point-based semantics of MDL formulas ϕ, $(\rho, i) \models \phi$, and MDL regular expressions by mutual recursion in Figure 2.3. We omit the cases of atomic propositions and Boolean and temporal operators that are identical to MTL. The semantics of an MDL regular expression r on a stream ρ is a relation $\mathcal{R}_\rho(r) \subseteq \mathbb{N} \times \mathbb{N}$, where $(i, j) \in \mathcal{R}_\rho(r)$ iff r matches the time-points between i (inclusive) and j (exclusive). We exclude the time-point j to conveniently distinguish matches of MDL regular expressions like $\phi?$ that can be combined with subsequent matches from the same time-point and matches of MDL regular expressions like ϕ that can be combined with subsequent matches from the next time-point. For instance, the MDL regular expression $\phi \cdot \psi$ matches a pair of consecutive time-points if the formulas ϕ and ψ hold at these time-points while $\phi? \cdot \psi$ matches a single time-point if the formulas ϕ and ψ hold at that time-point.

Unlike in $\text{MDL}^{\mathsf{Aerial}}$, an empty match $(i, i) \in \mathcal{R}_\rho(r^*)$ of the MDL regular expression r^* does not make the MDL formula $\langle r|_I$ or $|r\rangle_I$ satisfied at the time-point i if $0 \in I$ because the match

$$\begin{array}{ll}
\text{not } \mathsf{wfr}(\phi?), \mathsf{wfr}(\phi) & \varepsilon(\phi?), \text{not } \varepsilon(\phi) \\
\mathsf{wfr}(r+s) \iff \mathsf{wfr}(r) \text{ and } \mathsf{wfr}(s) & \varepsilon(r+s) \iff \varepsilon(r) \text{ or } \varepsilon(s) \\
\mathsf{wfr}(r \cdot s) \iff \mathsf{wfr}(s) \text{ and } (\text{not } \varepsilon(s) \text{ or } \mathsf{wfr}(r)) & \varepsilon(r \cdot s) \iff \varepsilon(r) \text{ and } \varepsilon(s) \\
\mathsf{wfr}(r^*) \iff \mathsf{wfr}(r) & \varepsilon(r^*)
\end{array}$$

Figure 2.4. Well-formed and nullable regular expressions r.

must always include the time-point i (in particular, the match must be nonempty). This is without loss of generality because such an empty match $(i, i) \in \mathcal{R}_\rho(r^*)$ would make the formula $\langle r|_I$ or $|r\rangle_I$ trivially satisfied (if $0 \in I$) or would not affect the satisfaction at all (if $0 \notin I$).

We use the same semantics $\mathcal{R}_\rho(r)$ of regular expressions for both past and future regular expression match operators. Our semantics $\mathcal{R}_\rho(\phi) = \{(i, i+1) \mid (\rho, i) \models \phi\}$ resembles the semantics of linear dynamic logic (LDL) [36] which has only future modalities. The semantics of past-time linear dynamic logic (PLDL) [35] which has only past modalities defines $\mathcal{R}_\rho^{\text{PLDL}}(\phi) = \{(i, i+1) \mid (\rho, i+1) \models^{\text{PLDL}} \phi\}$, where $\models^{\text{PLDL}}$ is the semantics of PLDL formulas. This effectively means that a match $(j, i) \in \mathcal{R}_\rho^{\text{PLDL}}(r)$ matches the time-points between j (exclusive) and i (inclusive) which is convenient for the semantics of the match operator $\langle\!\langle r \rangle\!\rangle \phi$ in PLDL:

$$(\rho, i) \models^{\text{PLDL}} \langle\!\langle r \rangle\!\rangle \phi \text{ iff } j \text{ exists with } j \leq i,\ (j, i) \in \mathcal{R}_\rho^{\text{PLDL}}(r), \text{ and } (\rho, j) \models^{\text{PLDL}} \phi.$$

In MDL, we do not separate the formula ϕ from the regular expression r, i.e., we express the PLDL formula $\langle\!\langle r \rangle\!\rangle \phi$ by the MDL formula $\langle \phi \cdot r|_I$ with a regular expression $\phi \cdot r$. This allows us to define the semantics of MDL regular expressions independently of their modality, i.e., whether an MDL regular expression r is used in a past match formula $\langle r|_I$ or a future match formula $|r\rangle_I$.

We call an MDL regular expression *well-formed* if all lookaheads refer to time-points matched by the regular expression. Hence, a match (i, j) of a well-formed MDL regular expression r only depends on the semantics of the MDL formulas ϕ contained in r at the time-points included in the match, i.e., between time-points i (inclusive) and j (exclusive). For instance, the regular expressions $\phi? \cdot \psi$ is well-formed, but the regular expression $\psi \cdot \phi?$ is not. Indeed, the regular expression $\psi \cdot \phi?$ has a match $(i, i+1)$ on ρ if and only if $(\rho, i) \models \psi$ and $(\rho, i+1) \models \phi$, i.e., the match depends on the semantics of ϕ at the excluded time-point $i+1$. We write $\mathsf{wfr}(r)$ if the regular expression r is well-formed. We call an MDL formula *well-formed* if all its regular expressions are well-formed. We write $\mathsf{wf}(\phi)$ if the formula ϕ is well-formed. To formally define well-formed regular expressions, we say that an MDL regular expression r is *nullable* if it has empty matches, e.g., $\phi?$ or r^*. We write $\varepsilon(r)$ if the regular expression r is nullable. We formally define well-formed and nullable regular expressions in Figure 2.4.

Given formulas ϕ and ψ, we could express the MTL operators $\bullet_I\ \phi$ as $\langle \phi \cdot \mathit{true}|_I$, $\bigcirc_I\ \phi$ as $|\mathit{true} \cdot \phi\rangle_I$, $\phi\ \mathsf{S}_I\ \psi$ as $\langle \psi \cdot \phi^*|_I$, and $\phi\ \mathsf{U}_I\ \psi$ as $|\phi^* \cdot \psi\rangle_I$. The provided well-formed MDL regular expressions are equivalent to the MTL operators.

Example 2.5. Many systems for user authentication follow a specification like: "A user should not be able to authenticate after entering a wrong password three times within the last hour without authenticating in between." For a fixed user, we write auth for the event "User authenticated" and err for the event "User entered a wrong password". This means we consider the set $\Sigma = \{\mathsf{auth}, \mathsf{err}\}$ of atomic propositions in this example. Then the MDL formula $\neg\langle \mathsf{err} \cdot (\neg\mathsf{auth})^* \cdot \mathsf{err} \cdot (\neg\mathsf{auth})^* \cdot \mathsf{err} \cdot (\neg\mathsf{auth})^* \cdot \mathsf{auth}|_{[0,3600]}$ captures this specification: it is violated at time-points at which the user authenticated after entering a wrong password three times in the last 3 600 seconds without an intermediate authentication.

We can express this property in MTL by nesting three temporal operators, namely one *strict* since operator for each err in the above MDL regular expression. The semantics of the strict since operator $\phi \mathrel{\dot{\mathsf{S}}_I} \psi$ differs from the since operator $\phi \mathrel{\mathsf{S}_I} \psi$ by requiring that the formula ψ must hold at a time-point that is strictly before the time-point at which the (strict) since operator is evaluated:

$$(\rho, i) \models \phi \mathrel{\dot{\mathsf{S}}_I} \psi \text{ iff } j \text{ exists with } j < i,\ \mathsf{mem}(\tau_j, \tau_i, I),\ (\rho, j) \models \psi, \text{ and } (\rho, k) \models \phi, \text{ for all } j < k \leq i$$

For $\mathbb{T} = \mathbb{N}$, the *strict* since operator $\phi \mathrel{\dot{\mathsf{S}}_{[a,b]}} \psi$ can be expressed by the following equivalent MTL formulas:

$$\phi \mathrel{\dot{\mathsf{S}}_{[a,b]}} \psi \equiv \begin{cases} (\phi \wedge \bullet_{[0,0]} (\phi \mathrel{\mathsf{S}_{[0,0]}} \psi)) & \text{if } a = 0,\, b = 0, \\ (\phi \wedge \bullet_{[0,0]} (\phi \mathrel{\mathsf{S}_{[0,0]}} \psi)) \vee (\phi \mathrel{\mathsf{S}_{[1,b]}} \psi) & \text{if } a = 0,\, b \neq 0, \\ \phi \mathrel{\mathsf{S}_{[a,b]}} \psi & \text{if } a \neq 0. \end{cases}$$

Yet, it is unclear which intervals to use with the three strict since operators beyond the fact that their upper bounds should sum up to 3 600. For $\mathbb{T} = \mathbb{N}$, a rather impractical solution exploits the fact that there are only finitely many ways to split the upper bound 3 600 into a sum of three upper bounds:

$$\neg \bigvee_{\substack{x_1, x_2, x_3 \in \mathbb{N} \\ x_1 + x_2 + x_3 = 3600}} \Big(\mathsf{auth} \wedge \Big((\bullet\ \neg\mathsf{auth}) \mathrel{\dot{\mathsf{S}}_{[0,x_1]}} \Big(\mathsf{err} \wedge \Big((\bullet\ \neg\mathsf{auth}) \mathrel{\dot{\mathsf{S}}_{[0,x_2]}} \Big(\mathsf{err} \wedge \Big((\bullet\ \neg\mathsf{auth}) \mathrel{\dot{\mathsf{S}}_{[0,x_3]}} \mathsf{err}\Big)\Big)\Big)\Big)\Big)\Big)$$

and constructs the disjunction of all possible splits which yields $\binom{3603}{3} \approx 8 \cdot 10^9$ disjuncts in this case. For $\mathbb{T} = \mathbb{R}$, the previous solution no longer works and we conjecture that no equivalent MTL formula exists. MDL remediates these difficulties regardless of the time domain. $\diamond$

Given a trace $\rho_{<\ell}$ (a finite prefix of an infinite stream $\rho = \langle(\Gamma_i,\, \tau_i)\rangle_{i \in \mathbb{N}}$) and a well-formed MDL formula ϕ, it might not be possible to check for every time-point $j < \ell$ if $(\rho, j) \models \phi$, e.g., if ϕ contains future temporal operators $\bigcirc_I$ or U_I. Hence, we define the *progress* $\mathsf{prog}(\phi, \overline{\tau})$ of a well-formed MDL formula ϕ on a monotone sequence of time-stamps $\overline{\tau}$ to be the number of time-points j, $j < \mathsf{prog}(\phi, \overline{\tau})$, for which it is possible to check if $(\rho, j) \models \phi$ when given, for all $k < |\overline{\tau}|$, the time-points $\tau_k = \overline{\tau}_k$ and sets of events Γ_k. We formally define the function $\mathsf{prog}(\phi, \overline{\tau})$ in Figure 2.20, following the definition of progress by Schneider et al. [71], and capture its core property in the following lemma.

Lemma 2.6. *Let ϕ be an MDL formula and $\overline{\tau}$ be a monotone sequence of time-stamps. Then $(\rho, j) \models \phi \iff (\rho', j) \models \phi$ holds for all time-points $j < \mathsf{prog}(\phi, \overline{\tau})$ and for every two infinite streams $\rho = \langle(\Gamma_i,\, \tau_i)\rangle_{i \in \mathbb{N}}$ and $\rho' = \langle(\Gamma'_i,\, \tau'_i)\rangle_{i \in \mathbb{N}}$ such that, for all $k < |\overline{\tau}|$, $\tau_k = \tau'_k = \overline{\tau}_k$ and $\Gamma_k = \Gamma'_k$.*

Note that Figure 2.5 does not necessarily provide the maximum possible value of $\mathsf{prog}(\phi, \overline{\tau})$ for which Lemma 2.20 holds. For instance, if a formula ϕ is a tautology, then $(\rho, j) \models \phi$ holds for all ρ and j. Hence, $\mathsf{prog}(\phi, \overline{\tau})$ could be arbitrarily high. Still, Figure 2.5 provides a lower bound on the number of time-points j for which $(\rho, j) \models \phi$ can be definitely checked.

We assume that the intervals I of U_I and $|r\rangle_I$ temporal operators are bounded, i.e., $\mathsf{right}(I) \in \mathbb{T}_{\text{fin}}$, because a monitor might not be able to produce a verdict for a formula with unbounded future temporal operators on a (finite) trace. We call a formula *bounded-future* if the intervals

```
input:  A well-formed MDL formula $\phi$ and a monotone sequence $\overline{\tau}$ of time-stamps.
output: The number of time-points $j$ for which $(\rho, j) \models \phi$ can be checked given, for all
        $k < |\overline{\tau}|$, the time-points $\tau_k = \overline{\tau}_k$ and sets of events $\Gamma_k$.
function $\mathsf{prog}(\phi, \overline{\tau}) =$
    switch $\phi$ do
        case $\neg\phi_0$ do return $\mathsf{prog}(\phi_0, \overline{\tau})$;
        case $\phi_1 \vee \phi_2$ do return $\min\{\mathsf{prog}(\phi_1, \overline{\tau}), \mathsf{prog}(\phi_2, \overline{\tau})\}$;
        case $\bullet_I\ \phi_0$ do return $\min\{|\overline{\tau}|, \mathsf{prog}(\phi_0, \overline{\tau}) + 1\}$;
        case $\bigcirc_I\ \phi_0$ do
            $k := \mathsf{prog}(\phi_0, \overline{\tau})$;
            if $k = 0$ then return $0$;
            else return $k - 1$;
        case $\phi_1\ \mathsf{S}_I\ \phi_2$ do return $\min\{\mathsf{prog}(\phi_1, \overline{\tau}), \mathsf{prog}(\phi_2, \overline{\tau})\}$;
        case $\phi_1\ \mathsf{U}_I\ \phi_2$ do
            if $|\overline{\tau}| = 0$ then return $0$;
            else
                $k := \min\{|\overline{\tau}| - 1, \mathsf{prog}(\phi_1, \overline{\tau}), \mathsf{prog}(\phi_2, \overline{\tau})\}$;
                return $\min\{j \mid 0 \leq j \leq k \wedge \mathsf{memR}(\overline{\tau}_j, \overline{\tau}_k, I)\}$;
        case $\langle r|_I$ do return $\min\{\mathsf{prog}(\psi, \overline{\tau}) \mid \psi \in \mathsf{SF}(r)\}$;
        case $|r\rangle_I$ do
            if $|\overline{\tau}| = 0$ then return $0$;
            else
                $k := \min\{|\overline{\tau}| - 1, \min\{\mathsf{prog}(\psi, \overline{\tau}) \mid \psi \in \mathsf{SF}(r)\}\}$;
                return $\min\{j \mid 0 \leq j \leq k \wedge \mathsf{memR}(\overline{\tau}_j, \overline{\tau}_k, I)\}$;
        otherwise do return $|\overline{\tau}|$;
```

Figure 2.5. The function $\mathsf{prog}(\phi, \overline{\tau})$.

of its U_I and $|r\rangle_I$ temporal operators temporal operators are bounded. Still, the intervals of the future temporal operators $\bigcirc_I$ can be unbounded because these operators never refer to any time-point beyond the next time-point. For instance, the formula $\Diamond_{[0,\infty]}\ p$, where $p \in \Sigma$ is an atomic proposition, is not bounded-future. This formula is satisfied at a time-point $i \in \mathbb{N}$ if the atomic proposition p is satisfied at a later time-point $j \in \mathbb{N}$, $i \leq j$. Unless the monitor encounters such a time-point j with $p \in \Gamma_j$, no Boolean verdict can ever be computed. We point out that the semantics of such a formula is still well-defined with respect to an *infinite* stream. However, a monitor can only process a *finite* prefix (trace) of such a stream.

We conclude this section by providing transformations between MDL$^{\mathsf{Aerial}}$ and MDL. These transformations constructively show that MDL$^{\mathsf{Aerial}}$ formulas and well-formed MDL formulas are equally expressive. Moreover, the transformations provide upper bounds on the size trade-offs between MDL$^{\mathsf{Aerial}}$ and MDL. We formally define the transformation $\mathsf{mdl2mdl}(\phi)$ of MDL$^{\mathsf{Aerial}}$ formulas ϕ into well-formed MDL formulas by mutual recursion in Figure 2.6. We remark that the size of the well-formed MDL formula $\mathsf{mdl2mdl}(\phi)$ is asymptotically bounded by the size of the MDL$^{\mathsf{Aerial}}$ formula ϕ. The correctness of the function $\mathsf{mdl2mdl}(\phi)$ is expressed by the following lemma.

Lemma 2.7 ([63, mdlstar2mdl]). *Let ϕ be an* MDL$^{\mathsf{Aerial}}$ *formula. Then* $\mathsf{mdl2mdl}(\phi)$ *is a*

$$\begin{aligned}
\mathsf{mdl2mdl}(p) &= p & \mathsf{embed}(\phi?) &= \mathsf{mdl2mdl}(\phi)?\\
\mathsf{mdl2mdl}(\neg\phi) &= \neg\mathsf{mdl2mdl}(\phi) & \mathsf{embed}(\star) &= \mathit{true}\\
\mathsf{mdl2mdl}(\phi \vee \psi) &= \mathsf{mdl2mdl}(\phi) \vee \mathsf{mdl2mdl}(\psi) & \mathsf{embed}(r+s) &= \mathsf{embed}(r) + \mathsf{embed}(s)\\
\mathsf{mdl2mdl}(\langle r|_I) &= \langle \mathsf{embed}(r)\cdot \mathit{true}|_I & \mathsf{embed}(r\cdot s) &= \mathsf{embed}(r)\cdot\mathsf{embed}(s)\\
\mathsf{mdl2mdl}(|r\rangle_I) &= |\mathsf{embed}(r)\cdot \mathit{true}\rangle_I & \mathsf{embed}(r^*) &= \mathsf{embed}(r)^*
\end{aligned}$$

Figure 2.6. Transformation of MDL$^{\mathsf{Aerial}}$ formulas into well-formed MDL formulas.

$$\begin{aligned}
\mathsf{mdl2mdl}'(p) &= p & \mathsf{embed}'(\phi?) &= \mathsf{mdl2mdl}'(\phi)?\\
\mathsf{mdl2mdl}'(\neg\phi) &= \neg\mathsf{mdl2mdl}'(\phi) & \mathsf{embed}'(\phi) &= \mathsf{mdl2mdl}'(\phi)?\cdot\star\\
\mathsf{mdl2mdl}'(\phi \vee \psi) &= \mathsf{mdl2mdl}'(\phi) \vee \mathsf{mdl2mdl}'(\psi) & \mathsf{embed}'(r+s) &= \mathsf{embed}'(r) + \mathsf{embed}'(s)\\
\mathsf{mdl2mdl}'(\langle r|_I) &= \langle \mathsf{embed}'(\mathsf{rderive}(r))|_I & \mathsf{embed}'(r\cdot s) &= \mathsf{embed}'(r)\cdot\mathsf{embed}'(s)\\
\mathsf{mdl2mdl}'(|r\rangle_I) &= |\mathsf{embed}'(\mathsf{rderive}(r))\rangle_I & \mathsf{embed}'(r^*) &= \mathsf{embed}'(r)^*
\end{aligned}$$

$$\begin{aligned}
\mathsf{rderive}(\phi?) &= \mathit{false}?\\
\mathsf{rderive}(\phi) &= \phi?\\
\mathsf{rderive}(r+s) &= \mathsf{rderive}(r) + \mathsf{rderive}(s)\\
\mathsf{rderive}(r\cdot s) &= \begin{cases}\mathsf{rderive}(r) + (r\cdot\mathsf{rderive}(s)) & \text{if } \varepsilon(s)\\ r\cdot\mathsf{rderive}(s) & \text{otherwise}\end{cases}\\
\mathsf{rderive}(r^*) &= r^*\cdot\mathsf{rderive}(r)
\end{aligned}$$

Figure 2.7. Transformation of well-formed MDL formulas into MDL$^{\mathsf{Aerial}}$ formulas.

well-formed MDL formula that is equivalent to ϕ. Formally, $\mathsf{wf}(\mathsf{mdl2mdl}(\phi))$ *holds and*

$$(\rho, i) \models \mathsf{mdl2mdl}(\phi) \iff (\rho, i) \models^{\mathsf{Aerial}} \phi.$$

Finally, we formally define the transformation $\mathsf{mdl2mdl}'(\phi)$ of well-formed MDL formulas ϕ into MDL$^{\mathsf{Aerial}}$ formulas by mutual recursion in Figure 2.7. We transform a well-formed MDL regular expression r into an MDL$^{\mathsf{Aerial}}$ regular expression $\mathsf{embed}'(\mathsf{rderive}(r))$ by computing the right derivative $\mathsf{rderive}(r)$ of r, analogously to computing the right derivate of an MDL$^{\mathsf{Aerial}}$ regular expression [6, Section 7.1]. We point out that computing the right derivative of a regular expression may result in an MDL$^{\mathsf{Aerial}}$ regular expression of quadratic size. Consequently, the size of the MDL$^{\mathsf{Aerial}}$ formula $\mathsf{mdl2mdl}'(\phi)$ is only asymptotically bounded by a square of the size of the well-formed MDL formula ϕ. The correctness of the function $\mathsf{mdl2mdl}'(\phi)$ is expressed by the following lemma.

Lemma 2.8 ([63, mdl2mdlstar]). *Let ϕ be a well-formed MDL formula. Then* $\mathsf{mdl2mdl}'(\phi)$ *is an* MDL$^{\mathsf{Aerial}}$ *formula that is equivalent to ϕ. Formally,*

$$(\rho, i) \models^{\mathsf{Aerial}} \mathsf{mdl2mdl}'(\phi) \iff (\rho, i) \models \phi.$$

2.3 First-Order Logics

Parameterized events (characterized by a name and a list of data values) substantially extend the class of properties that can be checked by a monitor. For instance, the specification from Example 2.5 can be generalized from a fixed user to multiple users by parameterizing the events

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\mathcal{S},\alpha) \not\models \mathit{false};\ (\mathcal{S},\alpha) \models \mathit{true};} \\
(\mathcal{S},\alpha) \models (x \approx t) & \text{iff} & \alpha(x) = \alpha(t); \\
(\mathcal{S},\alpha) \models r(t_1,\ldots,t_{\iota(r)}) & \text{iff} & (\alpha(t_1),\ldots,\alpha(t_{\iota(r)})) \in r^{\mathcal{S}}; \\
(\mathcal{S},\alpha) \models (\neg Q) & \text{iff} & \text{not } (\mathcal{S},\alpha) \models Q; \\
(\mathcal{S},\alpha) \models (Q_1 \lor Q_2) & \text{iff} & (\mathcal{S},\alpha) \models Q_1 \text{ or } (\mathcal{S},\alpha) \models Q_2; \\
(\mathcal{S},\alpha) \models (Q_1 \land Q_2) & \text{iff} & (\mathcal{S},\alpha) \models Q_1 \text{ and } (\mathcal{S},\alpha) \models Q_2; \\
(\mathcal{S},\alpha,i) \models (\exists x.\, Q) & \text{iff} & (\mathcal{S},\alpha[x \mapsto d],i) \models Q, \text{for some } d \in \mathcal{D}.
\end{array}
$$

Figure 2.8. Semantics of RC.

auth and err by a user u: $\mathsf{auth}(u)$ and $\mathsf{err}(u)$. Then a monitor would compute a set of users violating the specification for every time-point.

In this section, we introduce pure first-order logic, also known as relational calculus (RC), and metric first-order temporal logic (MFOTL). An RC query is interpreted over a fixed structure consisting of relations interpreting the predicate symbols in the query. Metric first-order temporal logic can express properties about a sequence of structures that change over time.

2.3.1 Relational Calculus

We introduce the syntax and semantics of RC and define relevant classes of RC queries that can be evaluated using relational algebra (RA) operations on finite tables. A signature σ is a triple $(\mathcal{C},\mathcal{R},\iota)$, where $\mathcal{C}$ and $\mathcal{R}$ are disjoint finite sets of constant and predicate symbols, and the function $\iota : \mathcal{R} \to \mathbb{N}$ maps each predicate symbol $r \in \mathcal{R}$ to its arity $\iota(r)$. Let $\sigma = (\mathcal{C},\mathcal{R},\iota)$ be a signature and $\mathcal{V}$ a countably infinite set of variables disjoint from $\mathcal{C} \cup \mathcal{R}$. The syntax of RC queries Q is defined recursively:

$$Q ::= \mathit{false} \mid \mathit{true} \mid x \approx t \mid r(t_1,\ldots,t_{\iota(r)}) \mid \neg Q \mid Q \lor Q \mid Q \land Q \mid \exists x.\, Q.$$

Here, $r \in \mathcal{R}$ is a predicate symbol, $t, t_1, \ldots, t_{\iota(r)} \in \mathcal{V} \cup \mathcal{C}$ are terms, and $x \in \mathcal{V}$ is a variable. We write $\exists \vec{v}.\, Q$ as a shorthand for $\exists v_1.\ldots\exists v_k.\, Q$ and $\forall \vec{v}.\, Q$ for $\neg\exists\vec{v}.\,\neg Q$, where $\vec{v}$ is a variable sequence $v_1,\ldots,v_k$. If $k = 0$, then both $\exists\vec{v}.\, Q$ and $\forall\vec{v}.\, Q$ denote the query Q. Quantifiers have lower precedence than conjunctions and disjunctions, e.g., $\exists x.\, Q_1 \land Q_2$ means $\exists x.\, (Q_1 \land Q_2)$. We use $\approx$ to denote the equality of terms in RC to distinguish it from $=$, which denotes syntactic object identity. We also write $Q_1 \longrightarrow Q_2$ for $\neg Q_1 \lor Q_2$. However, defining $Q_1 \lor Q_2$ as a shorthand for $\neg(\neg Q_1 \land \neg Q_2)$ would complicate later definitions, e.g., the definition of safe-range queries. We denote by $\mathsf{fv}(Q)$ the set of *free variables* in Q and by $\vec{\mathsf{fv}}(Q)$ the sequence of free variables in Q based on some fixed ordering of variables.

We define the subquery partial order $\sqsubseteq$ on queries recursively on the structure of RC queries, e.g., Q_2 is a subquery of the query $Q_1 \land \neg\exists y.\, Q_2$. One can also view $\sqsubseteq$ as the (reflexive and transitive) subterm relation on the datatype of RC queries. We denote by $\mathsf{sub}(Q)$ the set of subqueries of an RC query Q.

A structure $\mathcal{S}$ over a signature $(\mathcal{C},\mathcal{R},\iota)$ consists of a non-empty domain $\mathcal{D}$ and interpretations $\mathsf{c}^{\mathcal{S}} \in \mathcal{D}$ and $r^{\mathcal{S}} \subseteq \mathcal{D}^{\iota(r)}$, for each $\mathsf{c} \in \mathcal{C}$ and $r \in \mathcal{R}$. We assume that all relations $r^{\mathcal{S}}$ interpreting the predicate symbols $r \in \mathcal{R}$ in a structure $\mathcal{S}$ are *finite*. Note that this assumption does *not* yield a finite structure (in the sense of finite model theory [49]) since the domain $\mathcal{D}$ can still be infinite.

A (*variable*) *assignment* is a mapping $\alpha : \mathcal{V} \to \mathcal{D}$. We additionally extend α to constant symbols $\mathsf{c} \in \mathcal{C}$ by defining $\alpha(\mathsf{c}) = \mathsf{c}^{\mathcal{S}}$. We write $\alpha[x \mapsto d]$ for the assignment that maps x to $d \in \mathcal{D}$ and is otherwise identical to α. We lift this notation to sequences $\vec{x}$ and $\vec{d}$ of pairwise distinct

$$\begin{array}{llllllllllll}
x \approx x & \equiv & \mathit{true}, & \neg \mathit{false} & \equiv & \mathit{true}, & \neg \mathit{true} & \equiv & \mathit{false}, & & & \\
Q \wedge \mathit{false} & \equiv & \mathit{false}, & \mathit{false} \wedge Q & \equiv & \mathit{false}, & Q \wedge \mathit{true} & \equiv & Q, & \mathit{true} \wedge Q & \equiv & Q, \\
Q \vee \mathit{false} & \equiv & Q, & \mathit{false} \vee Q & \equiv & Q, & Q \vee \mathit{true} & \equiv & \mathit{true}, & \mathit{true} \vee Q & \equiv & \mathit{true}, \\
\exists x.\, \mathit{false} & \equiv & \mathit{false}, & \exists x.\, \mathit{true} & \equiv & \mathit{true}. & & & & & &
\end{array}$$

Figure 2.9. Constant propagation rules.

variables and arbitrary domain elements of the same length. The semantics of RC queries Q for a structure $\mathcal{S}$ and an assignment α, i.e., $(\mathcal{S}, \alpha) \models Q$, is defined in Figure 2.8.

Queries Q_1 and Q_2 over the same signature are *equivalent*, written $Q_1 \equiv Q_2$, if $(\mathcal{S}, \alpha) \models Q_1 \iff (\mathcal{S}, \alpha) \models Q_2$, for every $\mathcal{S}$ and α. Queries Q_1 and Q_2 over the same signature are *inf-equivalent*, written $Q_1 \overset{\infty}{\equiv} Q_2$, if $(\mathcal{S}, \alpha) \models Q_1 \iff (\mathcal{S}, \alpha) \models Q_2$, for every structure $\mathcal{S}$ with an *infinite* domain $\mathcal{D}$ and every α. Clearly, equivalent queries are also inf-equivalent. However, the queries $\exists x.\, \neg x \approx \mathsf{c}$ and *true* are only inf-equivalent. Indeed, the query $\exists x.\, \neg x \approx \mathsf{c}$ does not hold if the domain is $\mathcal{D} = \{\mathsf{c}\}$.

We write $\alpha \models Q$ for $(\mathcal{S}, \alpha) \models Q$ if the structure $\mathcal{S}$ is fixed in the given context. For a fixed $\mathcal{S}$, only the assignments to Q's free variables influence $\alpha \models Q$, i.e., $\alpha \models Q$ is equivalent to $\alpha' \models Q$, for every variable assignment α' that agrees with α on $\mathsf{fv}(Q)$. A query Q with no free variables, i.e., $\mathsf{fv}(Q) = \emptyset$, is called *closed*. For closed queries Q, we write $\models Q$ and say that Q holds, since closed queries either hold for all variable assignments or for none of them. We call a finite sequence $\vec{d}$ of domain elements $d_1, \dots d_k \in \mathcal{D}$ a *tuple*. Given an RC query Q and a structure $\mathcal{S}$, we denote the set of satisfying tuples for Q by

$$[\![Q]\!]^{\mathcal{S}} = \{\vec{d} \in \mathcal{D}^{|\vec{\mathsf{fv}}(Q)|} \mid (\mathcal{S}, \alpha[\vec{\mathsf{fv}}(Q) \mapsto \vec{d}]) \models Q, \text{ for some assignment } \alpha\}.$$

We omit $\mathcal{S}$ from $[\![Q]\!]^{\mathcal{S}}$ if $\mathcal{S}$ is fixed in the given context. We call values from $[\![Q]\!]^{\mathcal{S}}$ assigned to $x \in \mathsf{fv}(Q)$ as Q's column x.

The *active domain* $\mathsf{adom}^{\mathcal{S}}(Q)$ of an RC query Q and a structure $\mathcal{S}$ is a subset of the domain $\mathcal{D}$ containing the interpretations $\mathsf{c}^{\mathcal{S}}$ of all constant symbols that occur in Q and the values in the relations $r^{\mathcal{S}}$ interpreting all predicate symbols that occur in Q. We omit $\mathcal{S}$ from $\mathsf{adom}^{\mathcal{S}}(Q)$ if $\mathcal{S}$ is fixed in the given context. Since $\mathcal{C}$ and $\mathcal{R}$ are finite and all $r^{\mathcal{S}}$ are finite relations of a finite arity $\iota(r)$, the active domain $\mathsf{adom}^{\mathcal{S}}(Q)$ is a also finite set.

A query Q is *domain-independent* if $[\![Q]\!]^{\mathcal{S}_1} = [\![Q]\!]^{\mathcal{S}_2}$ holds for every two structures $\mathcal{S}_1$ and $\mathcal{S}_2$ that agree on the interpretations of constants ($\mathsf{c}^{\mathcal{S}_1} = \mathsf{c}^{\mathcal{S}_2}$) and predicate symbols ($r^{\mathcal{S}_1} = r^{\mathcal{S}_2}$), while their domains $\mathcal{D}_1$ and $\mathcal{D}_2$ may differ. Agreement on the interpretations implies $\mathsf{adom}^{\mathcal{S}_1}(Q) = \mathsf{adom}^{\mathcal{S}_2}(Q) \subseteq \mathcal{D}_1 \cap \mathcal{D}_2$. It is undecidable whether an RC query is domain-independent [57, 76].

Safe-range queries Before defining safe-range queries, we introduce some helper notions and notation. We introduce constant propagation rules in Figure 2.9. We denote by $\mathsf{cp}(Q)$ the query obtained from a query Q by exhaustively applying the rules in Figure 2.9. Note that $\mathsf{cp}(Q)$ is either of the form *false* or *true* or contains neither *false* nor *true* as a subquery. The following definitions introduce substitution of a variable by another variable and removing all free occurrences of a free variable.

$$\begin{array}{ll}
\mathsf{gen}(x, \mathit{false}, \emptyset); & \\
\mathsf{gen}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{gen}(x, \neg\neg Q, \mathcal{G}) & \text{if } \mathsf{gen}(x, Q, \mathcal{G}); \\
\mathsf{gen}(x, \neg(Q_1 \vee Q_2), \mathcal{G}) & \text{if } \mathsf{gen}(x, (\neg Q_1) \wedge (\neg Q_2), \mathcal{G}); \\
\mathsf{gen}(x, \neg(Q_1 \wedge Q_2), \mathcal{G}) & \text{if } \mathsf{gen}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{G}); \\
\mathsf{gen}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{gen}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{gen}(x, Q_2, \mathcal{G}_2); \\
\mathsf{gen}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{gen}(x, Q_1, \mathcal{G}) \text{ or } \mathsf{gen}(x, Q_2, \mathcal{G}); \\
\mathsf{gen}(x, Q \wedge x \approx y, \mathcal{G}[y \mapsto x]) & \text{if } \mathsf{gen}(y, Q, \mathcal{G}); \\
\mathsf{gen}(x, Q \wedge y \approx x, \mathcal{G}[y \mapsto x]) & \text{if } \mathsf{gen}(y, Q, \mathcal{G}); \\
\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{gen}(x, Q_y, \mathcal{G}).
\end{array}$$

Figure 2.10. The *generated* relation for RC.

Definition 2.9. *The substitution of the form $Q[x \mapsto y]$ is the query $\mathsf{cp}(Q')$ where Q' is obtained from a query Q by replacing all occurrences of the free variable x by the variable y, potentially also renaming bound variables to avoid capture.*

Definition 2.10. *The substitution of the form $Q[x/\mathit{false}]$ is the query $\mathsf{cp}(Q')$ where Q' is obtained from a query Q by replacing with false every atomic predicate or equality containing the free variable x, except for $(x \approx x) \equiv \mathit{true}$.*

Queries of the form $r(t_1, \dots, t_{\iota(r)})$ and $x \approx \mathsf{c}$ are called *atomic predicates.* We define the predicate $\mathsf{ap}(\cdot)$ characterizing atomic predicates, i.e., $\mathsf{ap}(Q)$ is true iff Q is an atomic predicate. Queries of the form $\exists \vec{v}.\, r(t_1, \dots, t_{\iota(r)})$ and $\exists \vec{v}.\, x \approx \mathsf{c}$ are called *quantified predicates.* We denote by $\tilde{\exists} x.\, Q$ the query obtained by existentially quantifying a variable x from an RC query Q if x is free in Q, i.e., $\tilde{\exists} x.\, Q := \exists x.\, Q$ if $x \in \mathsf{fv}(Q)$ and $\tilde{\exists} x.\, Q := Q$ otherwise. We use $\tilde{\exists} x.\, Q$ (instead of $\exists x.\, Q$) when constructing an RC query to avoid introducing bound variables that do not occur free in Q.

The class of *safe-range* queries [1] is a decidable subset of domain-independent RC queries. Its definition is based on the notion of range-restricted variables of an RC query. A variable is called *range-restricted* if "its possible values all lie within the active domain of the query" [1, Section 5.4]. Intuitively, atomic predicates restrict the possible values of a variable that occurs in them as a term. An equality $x \approx y$ between variables can also extend the set of range-restricted variables in a conjunction $Q \wedge x \approx y$: If x or y is a range-restricted variable in Q, then both x and y are range-restricted variables in $Q \wedge x \approx y$. We formalize range-restricted variables using the *generated* relation $\mathsf{gen}(x, Q, \mathcal{G})$, defined in Figure 2.10. Specifically, $\mathsf{gen}(x, Q, \mathcal{G})$ holds if x is a range-restricted variable in Q and every satisfying assignment for Q satisfies some quantified predicate, referred to as *generator*, from $\mathcal{G}$.

Note that, unlike in a similar definition by Van Gelder and Topor [34, Figure 5] which defines the rule $\mathsf{gen}(x, \exists y.\, Q_y, \mathcal{G})$ if $x \neq y$ and $\mathsf{gen}(x, Q_y, \mathcal{G})$, we modify the rule's conclusion to existentially quantify the bound variable y from all queries in $\mathcal{G}$ where y occurs: $\mathsf{gen}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G})$. Hence, $\mathsf{gen}(x, Q, \mathcal{G})$ implies $\mathsf{fv}(\mathcal{G}) \subseteq \mathsf{fv}(Q)$. We now formalize these relationships.

Lemma 2.11. *Let Q be an RC query, $x \in \mathsf{fv}(Q)$, and $\mathcal{G}$ be a set of quantified predicates such that $\mathsf{gen}(x, Q, \mathcal{G})$. Then (i) $x \in \mathsf{fv}(Q_{qp})$ and $\mathsf{fv}(Q_{qp}) \subseteq \mathsf{fv}(Q)$ hold for every $Q_{qp} \in \mathcal{G}$, (ii) for every $\mathcal{S}$ and α such that $(\mathcal{S}, \alpha) \models Q$, there exists $Q_{qp} \in \mathcal{G}$ such that $(\mathcal{S}, \alpha) \models Q_{qp}$, and (iii) $Q[x/\mathit{false}] = \mathit{false}$.*

$$
\begin{array}{ll}
\mathsf{gen_{vgt}}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{gen_{vgt}}(x, \neg\neg Q, \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, Q, \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, \neg(Q_1 \vee Q_2), \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, (\neg Q_1) \wedge (\neg Q_2), \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, \neg(Q_1 \wedge Q_2), \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, \neg\exists y.\, Q_y, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{gen_{vgt}}(x, \neg Q_y, \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{gen_{vgt}}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{gen_{vgt}}(x, Q_2, \mathcal{G}_2); \\
\mathsf{gen_{vgt}}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, Q_1, \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, Q_2, \mathcal{G}); \\
\mathsf{gen_{vgt}}(x, \exists y.\, Q_y, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{gen_{vgt}}(x, Q_y, \mathcal{G}); \\
\\
\mathsf{con_{vgt}}(x, Q, \emptyset) & \text{if } x \notin \mathsf{fv}(Q); \\
\mathsf{con_{vgt}}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{con_{vgt}}(x, \neg\neg Q, \mathcal{G}) & \text{if } \mathsf{con_{vgt}}(x, Q, \mathcal{G}); \\
\mathsf{con_{vgt}}(x, \neg(Q_1 \vee Q_2), \mathcal{G}) & \text{if } \mathsf{con_{vgt}}(x, (\neg Q_1) \text{ and } (\neg Q_2), \mathcal{G}); \\
\mathsf{con_{vgt}}(x, \neg(Q_1 \wedge Q_2), \mathcal{G}) & \text{if } \mathsf{con_{vgt}}(x, (\neg Q_1) \vee (\neg Q_2), \mathcal{G}); \\
\mathsf{con_{vgt}}(x, \neg\exists y.\, Q_y, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{con_{vgt}}(x, \neg Q_y, \mathcal{G}); \\
\mathsf{con_{vgt}}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{con_{vgt}}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{con_{vgt}}(x, Q_2, \mathcal{G}_2); \\
\mathsf{con_{vgt}}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, Q_1, \mathcal{G}); \\
\mathsf{con_{vgt}}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{gen_{vgt}}(x, Q_2, \mathcal{G}); \\
\mathsf{con_{vgt}}(x, Q_1 \wedge Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{con_{vgt}}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{con_{vgt}}(x, Q_2, \mathcal{G}_2); \\
\mathsf{con_{vgt}}(x, \exists y.\, Q_y, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{con_{vgt}}(x, Q_y, \mathcal{G}).
\end{array}
$$

Figure 2.11. The relations $\mathsf{gen_{vgt}}(x, Q, \mathcal{G})$ and $\mathsf{con_{vgt}}(x, Q, \mathcal{G})$ [34, Figure 5].

Definition 2.12. *We define* $\mathsf{gen}(x, Q)$ *to hold iff there exists a set* $\mathcal{G}$ *such that* $\mathsf{gen}(x, Q, \mathcal{G})$*. Let* $\mathsf{nongens}(Q) := \{x \in \mathsf{fv}(Q) \mid \mathsf{gen}(x, Q)$ *does not hold*$\}$ *be the set of free variables in an RC query* Q *that are not range-restricted. A query* Q *has* range-restricted free variables *if every free variable of* Q *is range-restricted, i.e.,* $\mathsf{nongens}(Q) = \emptyset$*. A query* Q *has* range-restricted bound variables *if the bound variable* y *in every subquery* $\exists y.\, Q_y$ *of* Q *is range-restricted, i.e.,* $\mathsf{gen}(y, Q_y)$ *holds. A query is* safe-range *if it has range-restricted free and range-restricted bound variables.*

Evaluable and Allowed Queries The classes of *evaluable* queries [34, Definition 5.2] and *allowed* queries [34, Definition 5.3] are decidable subsets of domain-independent RC queries. Among queries with no repeated predicate symbols, the evaluable queries characterize exactly the domain-independent queries [34, Theorem 10.5]. Every evaluable query can be translated to an equivalent allowed query [34, Theorem 8.6] and every allowed query can be translated to an equivalent RANF query [34, Theorem 9.6].

Definition 2.13. *A query* Q *is called* evaluable *if*

- *every variable* $x \in \mathsf{fv}(Q)$ *satisfies* $\mathsf{gen_{vgt}}(x, Q)$ *and*
- *the bound variable* y *in every subquery* $\exists y.\, Q_y$ *of* Q *satisfies* $\mathsf{con_{vgt}}(y, Q_y)$*.*

A query Q *is called* allowed *if*

- *every variable* $x \in \mathsf{fv}(Q)$ *satisfies* $\mathsf{gen_{vgt}}(x, Q)$ *and*

```
function measure($Q$) =
    switch $Q$ do
        case $\neg Q'_1$ do return $2 \cdot \mathsf{measure}(Q')$;
        case $Q'_1 \vee Q'_2$ do return $2 \cdot (\mathsf{measure}(Q'_1) + \mathsf{measure}(Q'_2) + 1)$ ;
        case $Q'_1 \wedge Q'_2$ do return $\mathsf{measure}(Q'_1) + \mathsf{measure}(Q'_2) + 1$;
        case $\exists x.\, Q_x$ do return $2 \cdot \mathsf{measure}(Q_x)$;
        otherwise do return $1$;
```

Figure 2.12. The measure $\mathsf{measure}(Q)$ on RC queries.

- *the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies* $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$*,*

where the relation $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ *is defined to hold iff there exists a set $\mathcal{G}$ such that* $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ *and the relation* $\mathsf{con}_{\mathsf{vgt}}(x, Q)$ *is defined to hold iff there exists a set $\mathcal{G}$ such that* $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$*, respectively. The relations* $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ *and* $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ *are defined in Figure 2.11.*

We relate our definition of $\mathsf{gen}(x, Q, \mathcal{G})$ from Figure 2.10 and the definition of $\mathsf{gen}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ by Van Gelder Topor [34] from Figure 2.11 in the following lemmas. We prove them by induction on queries using the measure $\mathsf{measure}(Q)$, defined in Figure 2.12, that decreases for proper subqueries, after pushing negation, and after distributing existential quantification over disjunction.

Lemma 2.14. *Let x and y be free variables in a query Q such that* $\mathsf{gen}_{\mathsf{vgt}}(x, \neg Q)$ *and* $\mathsf{gen}_{\mathsf{vgt}}(y, Q)$ *hold. Then we get a contradiction.*

Proof. The lemma is proved by induction on the query Q using the measure $\mathsf{measure}(Q)$ on queries, which decreases in every case of the definition in Figure 2.11. □

Lemma 2.15. *Let Q be a query such that* $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ *holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Suppose that* $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ *holds for a free variable* $x \in \mathsf{fv}(Q)$*. Then* $\mathsf{gen}(x, Q)$ *holds.*

Proof. The lemma is proved by induction on the query Q using the measure $\mathsf{measure}(Q)$ on queries, which decreases in every case of the definition in Figure 2.11.

Lemma 2.14 and the assumption that $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q imply that $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ cannot be derived using the rule $\mathsf{gen}_{\mathsf{vgt}}(x, \neg\exists y.\, Q_y)$, i.e., Q cannot be of the form $\neg\exists y.\, Q_y$. Every other case in the definition of $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ has a corresponding case in the definition of $\mathsf{gen}(x, Q)$. □

Lemma 2.16. *Let Q be an* allowed *query, i.e.,* $\mathsf{gen}_{\mathsf{vgt}}(x, Q)$ *holds for every free variable* $x \in \mathsf{fv}(Q)$ *and* $\mathsf{gen}_{\mathsf{vgt}}(y, Q_y)$ *holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q. Then Q is a safe-range query, i.e.,* $\mathsf{gen}(x, Q)$ *holds for every free variable* $x \in \mathsf{fv}(Q)$ *and* $\mathsf{gen}(y, Q_y)$ *holds for the bound variable y in every subquery $\exists y.\, Q_y$ of Q.*

Proof. The lemma is proved by applying Lemma 2.15 to every free variable of Q and to the bound variable y in every subquery of Q of the form $\exists y.\, Q_y$. □

Lemma 2.16 shows that every allowed query is safe-range. But there exist safe-range queries that are not allowed, e.g., $\mathsf{B}(x) \wedge x \approx y$.

$$\begin{tikzcd}
 & \boxed{\text{Safe-range RC}} \arrow[dl, "\mathsf{srnf}(Q)"'] \arrow[dr, "\mathsf{enf}(Q)"] \arrow[dd, dashed, bend right=80, "\mathsf{sr2ranf}(Q)"'] & \\
\boxed{\text{SRNF}} \arrow[dr, "\mathsf{sr2ranf}(Q_{srnf}, \emptyset)"'] & & \boxed{\text{ENF}} \arrow[dl, "{[27]}"] \\
 & \boxed{\text{RANF}} &
\end{tikzcd}$$

Figure 2.13. Overview of query normal forms.

Relational Calculus Query Normal Forms In the following paragraphs, we introduce the following RC query normal forms: safe-range normal form (SRNF), existential normal form (ENF), and relational algebra normal form. The query normal forms are used to translate a safe-range query to an equivalent relational algebra expression: a safe-range query is first translated to an equivalent query in SRNF or ENF and then to an equivalent query in RANF that can be directly mapped to a relational algebra expression. Note that a query normal form (SRNF, ENF, and RANF) concerns the query's structure rather than functional dependencies between attributes in relations (e.g., as in 1NF, 2NF, 3NF).

The translation of safe-range queries in SRNF or ENF to equivalent RANF queries proceeds by subquery rewriting using the following rules [1, Algorithm 5.4.7], [27, Lemma 7.8]:

$$\begin{array}{lcll}
Q \land (Q_1 \lor Q_2) & \equiv & (Q \land Q_1) \lor (Q \land Q_2), & (R1) \\
Q \land (\exists x.\, Q_x) & \equiv & (\exists x.\, Q \land Q_x), & (R2) \\
Q \land \neg Q' & \equiv & Q \land \neg(Q \land Q'). & (R3)
\end{array}$$

Figure 2.13 shows an overview of the RC fragments and query normal forms (nodes) and the functions to translate between them (edges). The dashed edge shows the translation of a safe-range query to RANF we opt for in this thesis. It is the composition of the two translations from safe-range RC to SRNF and from SRNF to RANF, respectively. For the sake of completeness, we also present ENF and an example (Example 2.18) showing that using SRNF can be beneficial over ENF in terms of the intermediate relation sizes when evaluating the resulting RANF query.

Safe-Range Normal Form A query Q is in safe-range normal form (SRNF) if the query Q' in every subquery $\neg Q'$ of Q is an atomic predicate, equality, or an existentially quantified query [1]. Figure 2.14 defines the function $\mathsf{srnf}(Q)$ that yields a SRNF query equivalent to a query Q. The function $\mathsf{srnf}(Q)$ proceeds by pushing negation [1, Section 5.4], distributing existential quantifiers over disjunction [34, Rule (T9)], and dropping bound variables that never occur [34, Definition 9.2]. We include the last two rules to optimize the intermediate relation sizes when evaluating the equivalent RANF query after translating the SRNF query to RANF. The termination of the function $\mathsf{srnf}(Q)$ follows using the measure $\mathsf{measure}(Q)$, defined in Figure 2.12.

A query might be safe-range, but not in safe-range normal form: e.g., the query $\mathsf{P}_2(x, y) \land \neg(\mathsf{P}_1(x) \land \mathsf{P}_1(y))$. Moreover, a query might be in safe-range normal form, but not safe-range: e.g., the query $\neg\mathsf{P}_1(x)$. Still, given a safe-range query Q, the function $\mathsf{srnf}(Q)$ yields an equivalent safe-range query in safe-range normal form. We prove this fact in the following lemma that is also used as a precondition for translating safe-range SRNF queries to RANF queries.

Lemma 2.17. *Let Q be a safe-range query. Then $\mathsf{srnf}(Q)$ is a safe-range query in SRNF and $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x and subquery $\neg Q'$ of $\mathsf{srnf}(Q)$.*

```
**input:** An RC query $Q$.
**output:** A SRNF query $Q_{srnf}$ such that $Q \equiv Q_{srnf}$, $\mathsf{fv}(Q) = \mathsf{fv}(Q_{srnf})$.
function $\mathsf{srnf}(Q) =$
  switch $Q$ do
    case $\neg Q'$ do
      switch $Q'$ do
        case $\neg Q''$ do return $\mathsf{srnf}(Q'')$;
        case $Q_1 \vee Q_2$ do return $\mathsf{srnf}((\neg Q_1) \wedge (\neg Q_2))$;
        case $Q_1 \wedge Q_2$ do return $\mathsf{srnf}((\neg Q_1) \vee (\neg Q_2))$;
        case $\exists \vec{v}.\, Q_{\vec{v}}$ do
          if $\vec{v} \cap \mathsf{fv}(Q_{\vec{v}}) = \emptyset$ then return $\mathsf{srnf}(\neg Q_{\vec{v}})$;
          else
            switch $\mathsf{srnf}(Q_{\vec{v}})$ do
              case $Q_1 \vee Q_2$ do return $\mathsf{srnf}((\neg\exists \vec{v}.\, Q_1) \wedge (\neg\exists \vec{v}.\, Q_2))$;
              otherwise do return $\neg\exists \vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, \mathsf{srnf}(Q_{\vec{v}})$;
        otherwise do return $\neg\mathsf{srnf}(Q')$;
    case $Q_1 \vee Q_2$ do return $\mathsf{srnf}(Q_1) \vee \mathsf{srnf}(Q_2)$;
    case $Q_1 \wedge Q_2$ do return $\mathsf{srnf}(Q_1) \wedge \mathsf{srnf}(Q_2)$;
    case $\exists \vec{v}.\, Q_{\vec{v}}$ do
      switch $\mathsf{srnf}(Q_{\vec{v}})$ do
        case $Q_1 \vee Q_2$ do return $\mathsf{srnf}((\exists \vec{v}.\, Q_1) \vee (\exists \vec{v}.\, Q_2))$;
        otherwise do return $\exists \vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, \mathsf{srnf}(Q_{\vec{v}})$;
    otherwise do return $Q$;
```

Figure 2.14. Translation to SRNF.

Proof. The lemma is proved by induction on the query Q using the measure $\mathsf{measure}(Q)$ on queries, which decreases in every recursive call of $\mathsf{srnf}(Q)$. Using Figure 2.10, $\mathsf{gen}(x, \neg Q')$ can only hold if $\neg Q'$ has the form $\neg\neg Q''$, $\neg(Q_1 \vee Q_2)$, or $\neg(Q_1 \wedge Q_2)$. The SRNF query $\mathsf{srnf}(Q)$ cannot have a subquery $\neg Q'$ that has any such form. □

Existential Normal Form Existential normal form (ENF) was introduced by Van Gelder and Topor [34] to translate an allowed query (Definition 2.13) into an equivalent RANF query. Given a safe-range query in ENF, the rules $(R1)$–$(R3)$ can be applied to obtain an equivalent RANF query [27, Lemma 7.8]. We remark that the rules $(R1)$–$(R3)$ are not sufficient to yield an equivalent RANF query for the original definition of ENF [34]. This issue has been identified and fixed by Escobar-Molano et al. [27]. Unlike SRNF, a query in ENF can have a subquery of the form $\neg(Q_1 \wedge Q_2)$, but no subquery of the form $\neg Q_1 \vee Q_2$ or $Q_1 \vee \neg Q_2$. A function $\mathsf{enf}(Q)$ that yields an ENF query equivalent to a query Q can be defined in terms of subquery rewriting using the rules in [27, Figure 2]. Although applying the rules $(R1)$–$(R3)$ to $\mathsf{enf}(Q)$ instead of $\mathsf{srnf}(Q)$ may result in a RANF query with fewer subqueries, the intermediate relation sizes when evaluating the resulting RANF query can be arbitrarily larger. We illustrate this in the following example that is also included in our artifact [60]. We thus opt for using SRNF instead of ENF for translating safe-range queries into RANF.

Example 2.18. The safe-range query $Q_{enf} := \mathsf{P}_2(x, y) \wedge \neg(\mathsf{P}_1(x) \wedge \mathsf{P}_1(y))$ is in ENF and RANF,

$\mathsf{ranf}(\mathit{false})$; $\mathsf{ranf}(\mathit{true})$;
$\mathsf{ranf}(Q)$ if $\mathsf{ap}(Q)$;
$\mathsf{ranf}(\neg Q)$ if $\mathsf{ranf}(Q)$ and $\mathsf{fv}(Q) = \emptyset$;
$\mathsf{ranf}(Q_1 \vee Q_2)$ if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$ and $\mathsf{fv}(Q_1) = \mathsf{fv}(Q_2)$;
$\mathsf{ranf}(Q_1 \wedge Q_2)$ if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$;
$\mathsf{ranf}(Q_1 \wedge \neg Q_2)$ if $\mathsf{ranf}(Q_1)$ and $\mathsf{ranf}(Q_2)$ and $\mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1)$;
$\mathsf{ranf}(Q \wedge (x \approx y))$ if $\mathsf{ranf}(Q)$ and $\{x, y\} \cap \mathsf{fv}(Q) \neq \emptyset$;
$\mathsf{ranf}(Q \wedge \neg(x \approx y))$ if $\mathsf{ranf}(Q)$ and $\{x, y\} \subseteq \mathsf{fv}(Q)$;
$\mathsf{ranf}(\exists x.\, Q_x)$ if $\mathsf{ranf}(Q_x)$ and $x \in \mathsf{fv}(Q_x)$.

Figure 2.15. Characterization of RC queries in RANF.

but not SRNF. Applying the rule $(R1)$ to $\mathsf{srnf}(Q_{\mathit{enf}})$ yields the RANF query $Q_{\mathit{srnf}} := (\mathsf{P}_2(x,y) \wedge \neg\mathsf{P}_1(x)) \vee (\mathsf{P}_2(x,y) \wedge \neg\mathsf{P}_1(y))$ that is equivalent to Q_{enf}. The sizes of the intermediate relations when evaluating these RANF queries over a structure $\mathcal{S}$ are $2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{\mathit{enf}}]\!]|$ and $2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y)]\!]| + |[\![\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y) \wedge \neg\mathsf{P}_1(x)]\!]| + 2 \cdot |[\![\mathsf{P}_2(x,y) \wedge \neg\mathsf{P}_1(y)]\!]| + 2 \cdot |[\![Q_{\mathit{srnf}}]\!]|$, respectively. Note that the intermediate relation sizes for Q_{enf} can be arbitrarily larger if $\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)$ evaluates to a large intermediate result, i.e., $|[\![\mathsf{P}_1(x) \wedge \mathsf{P}_1(y)]\!]| \gg |[\![\mathsf{P}_2(x,y)]\!]|$. In contrast, the intermediate relation sizes for Q_{srnf} can only be larger by a constant factor. ◇

Relational Algebra Normal Form Relational algebra normal form (RANF) [1, Section 5.4] is a class of safe-range queries whose operators can be directly mapped to relational algebra (RA) operators (e.g., natural joins and projections) and thus evaluated using relational algebra operations. Figure 2.15 defines the predicate $\mathsf{ranf}(\cdot)$ characterizing RC queries in RANF.

The translation [1, Algorithm 5.4.7] of safe-range queries in SRNF to equivalent RANF queries proceeds by nondeterministically applying the rules (R1)–(R3) to subqueries that satisfy additional conditions imposed to prevent unnecessary rule applications and to guarantee that the rewriting terminates. The choices of subqueries and rules applied to them affects the intermediate relation sizes when evaluating the resulting RANF query. Because enumerating all possible RANF queries obtained by subquery rewriting is infeasible due to a combinatorial blow-up, we define a recursive function with a restricted search space to translate a safe-range query in SRNF to an equivalent RANF query. Our recursive function $\mathsf{sr2ranf}(Q, \mathcal{Q})$ propagates subsets of queries $\mathcal{Q}$ throughout the recursion and these subsets can be locally optimized, e.g., by approximating the intermediate relation sizes when evaluating the resulting RANF queries.

The function $\mathsf{sr2ranf}(Q, \mathcal{Q}) = (\hat{Q}, \overline{\mathcal{Q}})$, defined in Figure 2.16, where $\mathsf{sr2ranf}$ stands for *safe-range to relational algebra normal form*, takes a safe-range query $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ in SRNF and returns a RANF query $\hat{Q}$ such that $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$. To restrict variables in Q, the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$ conjoins a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ to Q. Given a safe-range query Q, we first convert it into SRNF and set $\mathcal{Q} = \emptyset$. Then we define $\mathsf{sr2ranf}(Q) := \hat{Q}$, where $(\hat{Q}, \_) := \mathsf{sr2ranf}(\mathsf{srnf}(Q), \emptyset)$, to be a RANF query $\hat{Q}$ equivalent to Q. The termination of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ follows from the lexicographic measure $(2 \cdot \mathsf{measure}(Q) + \mathsf{eqneg}(Q) + 2 \cdot \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{measure}(\overline{Q}) + 2 \cdot |\mathcal{Q}|, \mathsf{measure}(Q) + \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{measure}(\overline{Q}))$, where $\mathsf{measure}(Q)$ is defined in Figure 2.12, $\mathsf{eqneg}(Q) := 1$ if Q is an equality between two variables or the negation of a query, and $\mathsf{eqneg}(Q) := 0$ otherwise.

Next we describe the definition of $\mathsf{sr2ranf}(Q, \mathcal{Q})$, inspired by the rules (R1)–(R3) and [1, Algorithm 5.4.7]. Note that no constant propagation (Figure 2.9 in Section 2.3.1) is needed

```
input:  A safe-range RC query $Q \land \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ such that $\mathsf{gen}(x, \neg Q')$ does not hold for any
        variable $x$ and subquery $\neg Q'$.
output: A RANF query $\hat{Q}$ and a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ such that
        $Q \land \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \land \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$; for all $\mathcal{S}$ and $\alpha$, $(\mathcal{S}, \alpha) \models \hat{Q} \Longrightarrow (\mathcal{S}, \alpha) \models \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$
        holds; $\hat{Q} = \mathsf{cp}(\hat{Q})$; and $\mathsf{fv}(Q) \subseteq \mathsf{fv}(\hat{Q}) \subseteq \mathsf{fv}(Q) \cup \mathsf{fv}(\mathcal{Q})$, unless $\hat{Q} = \mathit{false}$.
function $\mathsf{sr2ranf}(Q, \mathcal{Q}) =$
    if $\mathsf{ranf}(Q)$ then return $(\mathsf{cp}(Q), \emptyset)$;
    switch $Q$ do
        case $x \approx y$ do return $\mathsf{sr2ranf}(x \approx y \land \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}, \emptyset)$;
        case $\neg Q'$ do
            $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid (\neg Q') \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range$\}$;
            if $\overline{\mathcal{Q}} = \emptyset$ then
                $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \emptyset)$;
                return $(\mathsf{cp}(\neg \hat{Q}'), \emptyset)$;
            else return $\mathsf{sr2ranf}((\neg Q') \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
        case $Q_1 \lor Q_2$ do
            $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid \bigvee_{Q' \in \mathsf{flat}^{\lor}(Q)} (Q' \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q})$ is safe-range$\}$;
            foreach $Q' \in \mathsf{flat}^{\lor}(Q)$ do $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q' \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
            return $(\mathsf{cp}(\bigvee_{Q' \in \mathsf{flat}^{\lor}(Q)} \hat{Q}'), \overline{\mathcal{Q}})$;
        case $Q_1 \land Q_2$ do
            $\mathcal{Q}^- := \{Q' \in \mathsf{flat}^{\land}(Q) \cup \mathcal{Q} \mid \mathsf{neg}(Q')\}$; $\mathcal{Q}^+ := (\mathsf{flat}^{\land}(Q) \cup \mathcal{Q}) - \mathcal{Q}^-$;
            $\mathcal{Q}^{\approx} := \{Q' \in \mathcal{Q}^+ \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^+ := \mathcal{Q}^+ - \mathcal{Q}^{\approx}$;
            $\mathcal{Q}^{\not\approx} := \{\neg Q' \in \mathcal{Q}^- \mid \mathsf{eq}(Q')\}$; $\mathcal{Q}^- := \mathcal{Q}^- - \mathcal{Q}^{\not\approx}$;
            foreach $Q' \in \mathcal{Q}^+$ do $(\hat{Q}', \mathcal{Q}_{Q'}) := \mathsf{sr2ranf}(Q', (\mathcal{Q}^+ \cup \mathcal{Q}^{\approx}) - \{Q'\})$ ;
            foreach $\neg Q' \in \mathcal{Q}^-$ do $(\hat{Q}', \_) := \mathsf{sr2ranf}(Q', \mathcal{Q}^+ \cup \mathcal{Q}^{\approx})$ ;
            $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q}^+ \mid \mathcal{Q}^+ \subseteq \bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cup \{Q'\})\}$;
            return $(\mathsf{cp}(\mathsf{sort}^{\land}(\bigcup_{Q' \in \overline{\mathcal{Q}}} \{\hat{Q}'\} \cup \mathcal{Q}^{\approx} \cup \bigcup_{\neg Q' \in \mathcal{Q}^-} \{\neg \hat{Q}'\} \cup \mathcal{Q}^{\not\approx})),$
              $\bigcup_{Q' \in \overline{\mathcal{Q}}} (\mathcal{Q}_{Q'} \cap \mathcal{Q}))$;
        case $\exists \vec{v}.\, Q_{\vec{v}}$ do
            if $\mathsf{fv}(\mathcal{Q}) \cap \vec{v} \neq \emptyset$ then $\vec{w} \leftarrow \{\vec{w} \mid |\vec{w}| = |\vec{v}|$ and $((\mathsf{fv}(Q_{\vec{v}}) - \vec{v}) \cup \mathsf{fv}(\mathcal{Q})) \cap \vec{w} = \emptyset\}$;
            else $\vec{w} := \vec{v}$;
            $Q_{\vec{w}} := Q_{\vec{v}}[\vec{v} \mapsto \vec{w}]$;
            $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid Q_{\vec{w}} \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range$\}$;
            $(\hat{Q}_{\vec{w}}, \_) := \mathsf{sr2ranf}(Q_{\vec{w}} \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}, \emptyset)$;
            return $(\mathsf{cp}(\exists \vec{w}.\, \hat{Q}_{\vec{w}}), \overline{\mathcal{Q}})$;
        otherwise do return $(\mathsf{cp}(Q), \emptyset)$;
```

Figure 2.16. Translation of safe-range SRNF to RANF.

in [1, Algorithm 5.4.7], because the constants *false* and *true* are not in the textbook's query syntax [1, Section 5.3]. Because $\mathsf{gen}(x, \mathit{false})$ holds and $x \notin \mathsf{fv}(\mathit{false})$, we need to perform constant propagation to guarantee that every disjunct in a disjunction has the same set of free variables (e.g., the query $\mathit{false} \lor \mathsf{B}(x)$ must be translated to $\mathsf{B}(x)$ to be in RANF). We flatten the disjunction and conjunction using $\mathsf{flat}^{\vee}(\cdot)$ and $\mathsf{flat}^{\wedge}(\cdot)$, respectively. Formally, the function $\mathsf{flat}^{\oplus}(Q)$, where $\oplus \in \{\vee, \wedge\}$, computes the following set of queries: $\mathsf{flat}^{\oplus}(Q) := \mathsf{flat}^{\oplus}(Q_1) \cup \mathsf{flat}^{\oplus}(Q_2)$ if $Q = Q_1 \oplus Q_2$ and $\mathsf{flat}^{\oplus}(Q) := \{Q\}$ otherwise. In the case of a conjunction $Q^{\wedge}$, we first split the queries from $\mathsf{flat}^{\wedge}(Q^{\wedge})$ and $\mathcal{Q}$ into queries $\mathcal{Q}^{+}$ that do not have the form of a negation and queries $\mathcal{Q}^{-}$ that do. Then we take out equalities between two variables and negations of equalities between two variables from the sets $\mathcal{Q}^{+}$ and $\mathcal{Q}^{-}$, respectively. To partition $\mathsf{flat}^{\wedge}(Q^{\wedge}) \cup \mathcal{Q}$ this way, we define the predicates $\mathsf{neg}(Q)$ and $\mathsf{eq}(Q)$ characterizing equalities between two variables and negations, respectively, i.e., $\mathsf{neg}(Q)$ is true iff Q has the form $\neg Q'$ and $\mathsf{eq}(Q)$ is true iff Q has the form $x \approx y$. Finally, the function $\mathsf{sort}^{\wedge}(\mathcal{Q})$ converts a set of queries into a conjunction in RANF (in particular, a left-associative conjunction). Note that the function $\mathsf{sort}^{\wedge}(\mathcal{Q})$ must place the queries $x \approx y$ so that either x or y is free in some preceding conjunct, e.g., $\mathsf{B}(x) \wedge x \approx y \wedge y \approx z$ is in RANF, but $\mathsf{B}(x) \wedge y \approx z \wedge x \approx y$ is not. In the case of an existentially quantified query $\exists \vec{v}.\, Q_{\vec{v}}$, we rename the variables $\vec{v}$ to avoid clash of the free variables in the set of queries $\mathcal{Q}$ with the bound variables $\vec{v}$.

Query Cost To assess the time complexity of evaluating a RANF query Q, we define the *cost* of Q over a structure $\mathcal{S}$, denoted $\mathsf{cost}^{\mathcal{S}}(Q)$, to be the sum of intermediate relation sizes over all RANF subqueries of Q. Formally, $\mathsf{cost}^{\mathcal{S}}(Q) := \sum_{Q' \sqsubseteq Q,\ \mathsf{ranf}(Q')} \left| [\![Q']\!]^{\mathcal{S}} \right| \cdot |\mathsf{fv}(Q')|$. This corresponds to evaluating Q following its RANF structure using the RA operations for projection, column duplication, selection, set union, binary join, and anti-join. The complexity of these operations is linear in the combined input and output size (ignoring logarithmic factors due to set operations). The output size (the number of tuples times the number of variables) is counted in $\left| [\![Q']\!]^{\mathcal{S}} \right| \cdot |\mathsf{fv}(Q')|$ and the input size is counted as the output size for the input subqueries. Repeated subqueries are only considered once, which does not affect the asymptotics of query cost. In practice, the evaluation results for common subqueries can be reused.

2.3.2 Metric First-Order Temporal Logic

We introduce the syntax and semantics of MFOTL and define relevant classes of MFOTL queries. We obtain MFOTL by combining MTL and RC, i.e., first-order logic (FOL). Let $(\mathcal{C}, \mathcal{R}, \iota)$ be a signature and $\mathcal{V}$ a countably infinite set of variables disjoint from $\mathcal{C} \cup \mathcal{R}$. The syntax of MFOTL queries Q is defined recursively:

$$\begin{array}{rllllllll} Q & ::= & \mathit{false} & \mid \mathit{true} & \mid x \approx t & \mid r(t_1, \ldots, t_{\iota(r)}) & \mid \neg Q & \mid Q \vee Q & \mid Q \wedge Q \quad \mid \exists x.\, Q \mid \\ & & \bullet_I\, Q & \mid \bigcirc_I\, Q & \mid Q \,\mathsf{S}_I\, Q & \mid Q \,\mathsf{U}_I\, Q. & & & \end{array}$$

Here, $r \in \mathcal{R}$ is a predicate symbol, $t, t_1, \ldots, t_{\iota(r)} \in \mathcal{V} \cup \mathcal{C}$ are terms, $x \in \mathcal{V}$ is a variable, and $I \in \mathbb{I}$ is an interval. We employ the usual syntactic sugar: $\blacklozenge_I\, Q = \mathit{true}\, \mathsf{S}_I\, Q$ (*once*), $\blacksquare_I\, Q = \neg \blacklozenge_I \neg Q$ (*historically*), $\lozenge_I\, Q = \mathit{true}\, \mathsf{U}_I\, Q$ (*eventually*), and $\square_I\, Q = \neg \lozenge_I \neg Q$ (*always*).

A *temporal structure* $\bar{\mathcal{S}}$ over a signature $(\mathcal{C}, \mathcal{R}, \iota)$ consists of a non-empty domain $\mathcal{D}$, interpretations $\mathsf{c}^{\bar{\mathcal{S}}} \in \mathcal{D}$, for each $\mathsf{c} \in \mathcal{C}$, and an infinite stream $\rho^{\bar{\mathcal{S}}} = \langle (\Gamma_i,\, \tau_i) \rangle_{i \in \mathbb{N}}$. We assume that the observed system behaviour is a finite prefix $\rho^{\bar{\mathcal{S}}}_{<\ell} = (\rho^{\bar{\mathcal{S}}}_i)_{i<\ell}$ (called a *trace*) of the infinite stream $\rho^{\bar{\mathcal{S}}} = \langle (\Gamma_i,\, \tau_i) \rangle_{i \in \mathbb{N}}$ in a temporal structure $\bar{\mathcal{S}}$. The stream is an infinite sequence of time-points $i \in \mathbb{N}$, each consisting of a database $\Gamma_i \subseteq \mathcal{R} \times \mathcal{D}^*$ and a time-stamp $\tau_i \in \mathbb{T}_{\mathrm{fin}}$. The database Γ_i contains named tuples of domain values $(r, \vec{d})$, where $r \in \mathcal{R}$ and $|\vec{d}| = \iota(r)$, which provide interpretations of the predicate symbols at the time-point $i \in \mathbb{N}$. We make the same assumptions on the time-stamps τ_i (e.g., monotonicity) as for MTL and MDL. Basin et al. [10]

$$
\begin{array}{lll}
\multicolumn{3}{l}{(\bar{\mathcal{S}}, \alpha, i) \not\models \mathit{false};\ (\bar{\mathcal{S}}, \alpha, i) \models \mathit{true};} \\
(\bar{\mathcal{S}}, \alpha, i) \models (x \approx t) & \text{iff} & \alpha(x) = \alpha(t); \\
(\bar{\mathcal{S}}, \alpha, i) \models r(t_1, \dots, t_{\iota(r)}) & \text{iff} & (r, (\alpha(t_1), \dots, \alpha(t_{\iota(r)}))) \in \Gamma_i; \\
(\bar{\mathcal{S}}, \alpha, i) \models (\neg Q) & \text{iff} & \text{not } (\bar{\mathcal{S}}, \alpha, i) \models Q; \\
(\bar{\mathcal{S}}, \alpha, i) \models (Q_1 \vee Q_2) & \text{iff} & (\bar{\mathcal{S}}, \alpha, i) \models Q_1 \text{ or } (\bar{\mathcal{S}}, \alpha, i) \models Q_2; \\
(\bar{\mathcal{S}}, \alpha, i) \models (Q_1 \wedge Q_2) & \text{iff} & (\bar{\mathcal{S}}, \alpha, i) \models Q_1 \text{ and } (\bar{\mathcal{S}}, \alpha, i) \models Q_2; \\
(\bar{\mathcal{S}}, \alpha, i) \models (\exists x.\, Q) & \text{iff} & (\bar{\mathcal{S}}, \alpha[x \mapsto d], i) \models Q, \text{for some } d \in \mathcal{D}; \\
(\bar{\mathcal{S}}, \alpha, i) \models \bullet_I\, Q & \text{iff} & i > 0 \text{ and } \mathsf{mem}(\tau_{i-1}, \tau_i, I) \text{ and } (\bar{\mathcal{S}}, \alpha, i-1) \models Q; \\
(\bar{\mathcal{S}}, \alpha, i) \models \bigcirc_I\, Q & \text{iff} & \mathsf{mem}(\tau_i, \tau_{i+1}, I) \text{ and } (\bar{\mathcal{S}}, \alpha, i+1) \models Q; \\
(\bar{\mathcal{S}}, \alpha, i) \models Q_1 \mathbin{\mathsf{S}_I} Q_2 & \text{iff} & j \text{ exists with } j \le i \text{ and } \mathsf{mem}(\tau_j, \tau_i, I) \text{ and } (\bar{\mathcal{S}}, \alpha, j) \models Q_2 \text{ and} \\
& & (\bar{\mathcal{S}}, \alpha, k) \models Q_1, \text{for all } j < k \le i; \\
(\bar{\mathcal{S}}, \alpha, i) \models Q_1 \mathbin{\mathsf{U}_I} Q_2 & \text{iff} & j \text{ exists with } j \ge i \text{ and } \mathsf{mem}(\tau_i, \tau_j, I) \text{ and } (\bar{\mathcal{S}}, \alpha, j) \models Q_2 \text{ and} \\
& & (\bar{\mathcal{S}}, \alpha, k) \models Q_1, \text{for all } i \le k < j.
\end{array}
$$

Figure 2.17. Semantics of MFOTL.

$$
\begin{array}{ll}
\mathsf{gen}(x, \bullet_I\, Q, \{\bullet_I\, Q_{tqp} \mid Q_{tqp} \in \mathcal{G}\}) & \text{if } \mathsf{gen}(x, Q, \mathcal{G}); \\
\mathsf{gen}(x, \bigcirc_I\, Q, \{\bigcirc_I\, Q_{tqp} \mid Q_{tqp} \in \mathcal{G}\}) & \text{if } \mathsf{gen}(x, Q, \mathcal{G}); \\
\mathsf{gen}(x, Q_1 \mathbin{\mathsf{S}_I} Q_2, \{\blacklozenge_I\, Q_{tqp} \mid Q_{tqp} \in \mathcal{G}\}) & \text{if } \mathsf{gen}(x, Q_2, \mathcal{G}); \\
\mathsf{gen}(x, Q_1 \mathbin{\mathsf{U}_I} Q_2, \{\lozenge_I\, Q_{tqp} \mid Q_{tqp} \in \mathcal{G}\}) & \text{if } \mathsf{gen}(x, Q_2, \mathcal{G}).
\end{array}
$$

Figure 2.18. The *temporal* generated relation for MFOTL. The cases in the generated relation for RC are omitted here.

define a temporal structure as a sequence of (first-order) structures with the same domains and interpretations of constant symbols. Our definition of a temporal structure avoids these assumptions.

The semantics of MFOTL queries Q for a temporal structure $\bar{\mathcal{S}}$ and a variable assignment α at a time-point $i \in \mathbb{N}$, i.e., $(\bar{\mathcal{S}}, \alpha, i) \models Q$, is defined in Figure 2.17. Queries Q_1 and Q_2 over the same signature are *equivalent*, written $Q_1 \equiv Q_2$, if $(\bar{\mathcal{S}}, \alpha, i) \models Q_1 \iff (\bar{\mathcal{S}}, \alpha, i) \models Q_2$, for every $\bar{\mathcal{S}}$, α, and i. Given an MFOTL query Q and a temporal structure $\bar{\mathcal{S}}$, we denote the set of satisfying tuples for Q at a time-point i by

$$[\![Q]\!]_i^{\bar{\mathcal{S}}} = \{\vec{d} \in \mathcal{D}^{|\mathsf{fv}(Q)|} \mid (\bar{\mathcal{S}}, \alpha[\mathsf{fv}(Q) \mapsto \vec{d}], i) \models Q, \text{ for some assignment } \alpha\}.$$

An MFOTL query Q is *domain-independent* if $[\![Q]\!]_i^{\bar{\mathcal{S}}_1} = [\![Q]\!]_i^{\bar{\mathcal{S}}_2}$ holds for all time-points $i \in \mathbb{N}$ and for every two temporal structures $\bar{\mathcal{S}}_1$ and $\bar{\mathcal{S}}_2$ that agree on the interpretations of constants ($\mathsf{c}^{\bar{\mathcal{S}}_1} = \mathsf{c}^{\bar{\mathcal{S}}_2}$) and predicate symbols ($\rho^{\bar{\mathcal{S}}_1} = \rho^{\bar{\mathcal{S}}_2}$), while their domains $\mathcal{D}_1$ and $\mathcal{D}_2$ may differ. It is undecidable whether an MFOTL query is domain-independent because it is undecidable whether an RC query is domain-independent [57, 76] and MFOTL extends RC. We do not extend the notion of the active domain from RC to MFOTL.

Safe-range MFOTL queries We introduce the following additional constant propagation rules for MFOTL:

$$\bullet_I\, \mathit{false} \equiv \mathit{false}, \quad \bigcirc_I\, \mathit{false} \equiv \mathit{false}, \quad Q \mathbin{\mathsf{S}_I} \mathit{false} \equiv \mathit{false}, \quad Q \mathbin{\mathsf{U}_I} \mathit{false} \equiv \mathit{false}.$$

We do not introduce any additional constant propagation rules for *true* and the left-hand sides of S_I and U_I because such rules cannot yield constants (*false* or *true*) in general.

$$\begin{array}{ll}
\mathsf{ranf}(\CIRCLE_I\ Q) & \text{if } \mathsf{ranf}(Q);\\
\mathsf{ranf}(\bigcirc_I\ Q) & \text{if } \mathsf{ranf}(Q);\\
\mathsf{ranf}(Q_1\ \mathsf{S}_I\ Q_2) & \text{if } \mathsf{ranf}(Q_1) \text{ and } \mathsf{ranf}(Q_2) \text{ and } \mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1);\\
\mathsf{ranf}(\neg Q_1\ \mathsf{S}_I\ Q_2) & \text{if } \mathsf{ranf}(Q_1) \text{ and } \mathsf{ranf}(Q_2) \text{ and } \mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1);\\
\mathsf{ranf}(Q_1\ \mathsf{U}_I\ Q_2) & \text{if } \mathsf{ranf}(Q_1) \text{ and } \mathsf{ranf}(Q_2) \text{ and } \mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1);\\
\mathsf{ranf}(\neg Q_1\ \mathsf{U}_I\ Q_2) & \text{if } \mathsf{ranf}(Q_1) \text{ and } \mathsf{ranf}(Q_2) \text{ and } \mathsf{fv}(Q_2) \subseteq \mathsf{fv}(Q_1).
\end{array}$$

Figure 2.19. Characterization of MFOTL queries in RANF. The cases in the characterization of RC queries in RANF are omitted here.

A *temporal* quantified predicate is an atomic predicate to which a sequence of unary temporal operators $\CIRCLE_I$, $\bigcirc_I$, $\blacklozenge_I$, $\lozenge_I$ and existential quantifications $\exists x.$ is applied in some order. Formally, an atomic predicate is a temporal quantified predicate and if Q_{tqp} is a temporal quantified predicate, then $\CIRCLE_I\ Q_{tqp}$, $\bigcirc_I\ Q_{tqp}$, $\blacklozenge_I\ Q_{tqp}$, $\lozenge_I\ Q_{tqp}$, and $\exists x.\, Q_{tqp}$ are temporal quantified predicates, for every $I \in \mathbb{I}$ and $x \in \mathcal{V}$. We extend the notion of range-restricted variables to MFOTL and extend the generated relation $\mathsf{gen}(x, Q, \mathcal{G})$, defined for RC in Figure 2.10, to *temporal* generated relation for MFOTL in Figure 2.18. The set $\mathcal{G}$ in $\mathsf{gen}(x, Q, \mathcal{G})$ for MFOTL is a set of temporal quantified predicates. We also generalize Lemma 2.11 to MFOTL. Definition 2.12 of safe-range RC queries immediately extends to MFOTL.

Lemma 2.19. *Let Q be an MFOTL query, $x \in \mathsf{fv}(Q)$, and $\mathcal{G}$ be a set of temporal quantified predicates such that $\mathsf{gen}(x, Q, \mathcal{G})$. Then (i) $x \in \mathsf{fv}(Q_{tqp})$ and $\mathsf{fv}(Q_{tqp}) \subseteq \mathsf{fv}(Q)$ hold for every $Q_{tqp} \in \mathcal{G}$, (ii) for every $\bar{\mathcal{S}}$, α, and i such that $(\bar{\mathcal{S}}, \alpha, i) \models Q$, there exists $Q_{tqp} \in \mathcal{G}$ such that $(\bar{\mathcal{S}}, \alpha, i) \models Q_{tqp}$, and (iii) $Q[x/\mathit{false}] = \mathit{false}$.*

Relational Algebra Normal Form Relational algebra normal form (RANF) for MFOTL queries is a class of safe-range MFOTL queries that can be directly evaluated using relational algebra operations. Figure 2.19 extends the predicate $\mathsf{ranf}(\cdot)$ characterizing RC queries in RANF to MFOTL queries. The cases of temporal operators in Figure 2.19 match the corresponding cases in the definition of the monitorable fragment of MFOTL by Schneider et al. [71]. We omit the cases $Q^{\approx}\ \mathsf{S}_I\ Q$ and $Q^{\approx}\ \mathsf{U}_I\ Q$, $Q^{\approx} \in \{x \approx y, \neg(x \approx y)\}$, because these MFOTL queries can be equivalently expressed as

$$Q^{\approx}\ \mathsf{S}_I\ Q \iff \begin{cases} Q \vee ((\blacklozenge_I\ Q) \wedge Q^{\approx}) & \text{if } 0 \in I,\\ (\blacklozenge_I\ Q) \wedge Q^{\approx} & \text{otherwise;}\end{cases}$$
$$Q^{\approx}\ \mathsf{U}_I\ Q \iff \begin{cases} Q \vee ((\lozenge_I\ Q) \wedge Q^{\approx}) & \text{if } 0 \in I,\\ (\lozenge_I\ Q) \wedge Q^{\approx} & \text{otherwise;}\end{cases}$$

and the RANF conditions would be derived from the equivalent formulations.

Progress Given a trace $\rho^{\bar{\mathcal{S}}}_{<\ell}$ (a finite prefix of a temporal structure $\bar{\mathcal{S}}$), a domain $\mathcal{D}$, interpretations $\mathsf{c}^{\bar{\mathcal{S}}}$ of constant symbols $\mathsf{c} \in \mathcal{C}$, and an MFOTL query Q, it might not be possible to compute $[\![Q]\!]^{\bar{\mathcal{S}}}_j$ for all time-points $j < \ell$, e.g., if Q contains future temporal operators $\bigcirc_I$ or U_I. Hence, we define the *progress* $\mathsf{prog}(Q, \overline{\tau})$ of an MFOTL query Q on a monotone sequence of time-stamps $\overline{\tau}$ to be the number of time-points j, $j < \mathsf{prog}(Q, \overline{\tau})$, for which it is possible to compute $[\![Q]\!]^{\bar{\mathcal{S}}}_j$ when given, for all $k < |\overline{\tau}|$, the time-points $\tau_k = \overline{\tau}_k$ and databases Γ_k of $\bar{\mathcal{S}}$, the domain $\mathcal{D}$, and the

```
input:  An MFOTL query $Q$ and a monotone sequence $\overline{\tau}$ of time-stamps.
output: The number of time-points $j$ for which $[\![Q]\!]^{\bar{\mathcal{S}}}_j$ can be computed given, for all
        $k < |\overline{\tau}|$, the time-points $\tau_k = \overline{\tau}_k$ and databases $\Gamma_k$ of $\bar{\mathcal{S}}$, the domain $\mathcal{D}$, and the
        interpretations $\mathsf{c}^{\bar{\mathcal{S}}}$ of constant symbols $\mathsf{c} \in \mathcal{C}$.
function prog($Q, \overline{\tau}$) =
    switch $Q$ do
        case $\exists x. Q_x$ do return prog($Q_x, \overline{\tau}$);
        ...
```

Figure 2.20. The function $\mathsf{prog}(Q, \overline{\tau})$.

interpretations $\mathsf{c}^{\bar{\mathcal{S}}}$ of constant symbols $\mathsf{c} \in \mathcal{C}$. We extend the function $\mathsf{prog}(\phi, \overline{\tau})$ on well-formed MDL formulas, defined in Figure 2.5, to MFOTL queries in Figure 2.20 and capture its core property in the following lemma.

Lemma 2.20. *Let Q be an MFOTL query and $\overline{\tau}$ be a monotone sequence of time-stamps. Then $[\![Q]\!]^{\bar{\mathcal{S}}_1}_j = [\![Q]\!]^{\bar{\mathcal{S}}_2}_j$ holds for all time-points $j < \mathsf{prog}(Q, \overline{\tau})$ and for every two temporal structures $\bar{\mathcal{S}}_1$ and $\bar{\mathcal{S}}_2$ that agree on their domains ($\mathcal{D}_1 = \mathcal{D}_2$), on the interpretations of constants ($\mathsf{c}^{\bar{\mathcal{S}}_1} = \mathsf{c}^{\bar{\mathcal{S}}_2}$), and, for all $k < |\overline{\tau}|$, on the entries $\rho^{\bar{\mathcal{S}}_1}_k = \rho^{\bar{\mathcal{S}}_2}_k$ of the infinite streams $\rho^{\bar{\mathcal{S}}_1}$ and $\rho^{\bar{\mathcal{S}}_2}$, where $\tau_k = \overline{\tau}_k$.*

We observe that the maximum possible value of $\mathsf{prog}(Q, \overline{\tau})$ for which Lemma 2.20 holds is not computable. To this end, we show that it is undecidable if such a maximum possible value of $\mathsf{prog}(Q, \overline{\tau})$ is zero: Given an arbitrary RC query Q, we first determine a variable x that does not occur free in Q and a unary predicate symbol r that does not occur in Q. We then consider the query $(\bigcirc_{[0,\infty]}\, r(x)) \land Q$. If Q is not satisfiable, then $[\![(\bigcirc_{[0,\infty]}\, r(x)) \land Q]\!]^{\bar{\mathcal{S}}}_0 = \emptyset$ holds for every temporal structure $\bar{\mathcal{S}}$ and thus the maximum possible value of $\mathsf{prog}((\bigcirc_{[0,\infty]}\, r(x)) \land Q, \overline{\tau})$, where $|\overline{\tau}| = 1$, equals one. If Q is satisfiable, then $[\![(\bigcirc_{[0,\infty]}\, r(x)) \land Q]\!]^{\bar{\mathcal{S}}}_0$ contains the tuples from $[\![Q]\!]^{\bar{\mathcal{S}}}_0$ extended by the values d for x satisfying $r(x)$ at the next time-point 1, i.e., such that $(r, d) \in \Gamma_1$. Hence, the maximum possible value of $\mathsf{prog}((\bigcirc_{[0,\infty]}\, r(x)) \land Q, \overline{\tau})$, where $|\overline{\tau}| = 1$, equals zero. Because it is undecidable if an arbitrary RC query Q is satisfiable, it is also undecidable if the maximum possible value of $\mathsf{prog}(Q, \overline{\tau})$ is zero.

Query Cost To assess the time complexity of evaluating an MFOTL query Q in RANF, we define the *temporal query cost* of Q over a trace $\rho^{\bar{\mathcal{S}}}_{<\ell}$ (a finite prefix of a temporal structure $\bar{\mathcal{S}}$), denoted as $\mathsf{cost}^{\bar{\mathcal{S}}}_{<\ell}(Q)$, to be the sum of intermediate result sizes over all RANF subqueries Q' of Q and all time-points $j < \ell$ at which Q' can be evaluated given the trace $\rho^{\bar{\mathcal{S}}}_{<\ell}$. We approximate these time-points j using the notion of progress: we consider the time-points $j < \mathsf{prog}(Q', (\tau_k)_{k<\ell})$, where $(\tau_k)_{k<\ell}$ is the monotone sequence of time-stamps from the trace $\rho^{\bar{\mathcal{S}}}_{<\ell}$. Formally,

$$\mathsf{cost}^{\bar{\mathcal{S}}}_{<\ell}(Q) := \sum_{\substack{Q' \sqsubseteq Q,\ \mathsf{ranf}(Q') \\ j < \mathsf{prog}(Q', (\tau_k)_{k<\ell})}} \left| [\![Q']\!]^{\bar{\mathcal{S}}}_j \right| \cdot |\mathsf{fv}(Q')|.$$

This corresponds to evaluating Q by a monitoring algorithm following the query's RANF structure using the RA operations (e.g., natural joins and projections) for nontemporal operators and an algorithm for evaluating temporal operators whose time complexity is linear in the combined input and output size of the tables satisfying the temporal operator and its arguments (ignoring logarithmic factors due to set operations). Such an algorithm clearly exists for the temporal operators $\CIRCLE_I$ and $\Circle_I$ whose evaluation merely requires to shift the tables in time and we show a suitable evaluation algorithm for the temporal operators S_I and U_I in Section 4.4.

Chapter 3

Multi-Head Monitoring

3.1 Introduction

In this chapter, we develop multi-head monitoring algorithms for metric temporal logic (MTL) and metric dynamic logic (MDL). We first describe how a multi-head monitor can be derived from the online monitor by Basin et al. [7]. We turn their monitor into a nondeterministic finite transducer and then simulate its nondeterministic computation deterministically with mutliple reading heads. This preliminary multi-head monitor inspired the development of an optimized multi-head monitor that we formally describe in the subsequent sections. We have implemented the optimized multi-head monitor for MTL and MDL, empirically confirmed that its performance improves upon existing approaches, and formally verified its correctness using the Isabelle/HOL proof assistant.

A *finite transducer* is a finite-state machine that sequentially reads some input, updates its state, and produces some output at every step. Let us suppose that the time-stamps (elements $\tau \in \mathbb{T}$) in the trace are represented such that the interval condition for every pair of time-points can be checked in constant space (e.g., if the time-stamps are natural numbers $\mathbb{T} = \mathbb{N}$, they can be represented by the time-stamp differences between consecutive time-points). Then the online monitor by Basin et al. [7] can be interpreted as a deterministic finite transducer for a fixed formula that works as follows: Given a fixed formula, it maintains a buffer that stores a *Boolean expression* for every time-point for which the monitor has not produced any verdict yet. The Boolean expression is a propositional formula that represents the dependency of the verdict on future time-points. At every step, the monitor updates the Boolean expressions in the buffer and computes a Boolean expression for the current time-point. If the Boolean expression for the current time-point is equivalent to a Boolean value, then the monitor outputs this Boolean value as the Boolean verdict for the current time-point. Otherwise, it buffers the Boolean expression for the current time-point. Finally, the monitor produces equivalence verdicts for buffered time-points that have a matching time-point with an equivalent Boolean expression in the buffer and Boolean verdicts for buffered time-points whose Boolean expressions are equivalent to a Boolean value. The monitor also removes the entries for the time-points from the buffer for which a (equivalence or Boolean) verdict was produced. An equivalence verdict denotes the fact that the Boolean verdicts (to be determined later) for the time-points related by the verdict are equal. This way, no pair of Boolean expressions are equivalent. For a fixed formula, there are only finitely many possible Boolean expressions that are pairwise not equivalent. Hence, the buffer has constant size (independent of the observed events) and the monitor is indeed a *finite* transducer. We remark that a finite transducer cannot store a sequence number that refers to a buffered time-point and thus we assume that the monitor refers to these time-points relatively to the set of buffered time-points for which no verdict has been produced yet, e.g., as the *first* buffered time-point, the *second* buffered time-point etc.

The deterministic finite transducer produces equivalence verdicts (in addition to Boolean verdicts) and it does not necessarily produce the verdicts in the order in which the corresponding time-points appear in the trace. Hence, we turn the deterministic finite transducer into a nondeterministic one that produces a sequence of Boolean verdicts for every time-point in the order in which the time-points appear in the trace. The nondeterministic transducer nondeterministically guesses and outputs a Boolean verdict for the current time-point even if the Boolean expression for the current time-point is not equivalent to a Boolean value. The guesses are always stored together with the Boolean expressions in the buffer and the output is rejected if the monitor discovers an inconsistency (a Boolean expression becomes equivalent to a Boolean value that does not match the guess or two Boolean expressions in the buffer with distinct guesses become equivalent). Unfortunately, a nondeterministic finite transducer cannot be directly implemented in a real-world system. Hence, we simulate a nondeterministic finite transducer by a deterministic one that uses multiple reading heads to read the input. The core of the simulation is the *all-suffix regular matching problem*, which is the problem of deciding for each suffix of an input word whether it is accepted by a finite automaton. We show that a multi-head deterministic finite transducer can solve this problem by maintaining a bounded collection of automaton runs that yield unique states at the current position in the input word. In the context of monitoring, we adapt this approach to our *window* data structure (Section 3.2.2) for matching MDL regular expressions. In the context of finite transducers, we provide a detailed description of the simulation in [66].

We replace the window data structure by custom data structures tailored to the MTL operators to achieve a better time and space complexity for MTL. To analyze the time and space complexity, we assume that time-points (natural numbers whose magnitude is at most the trace length ℓ) and time-stamps (elements $\tau \in \mathbb{T}$) can be manipulated in constant time and stored in constant space. Then the time and space complexity of our monitors is linear in the number of subformulas for MTL and exponential for MDL. In practice, however, formulas are small, while traces are huge. It usually poses no problem for monitors to be exponential in the number of subformulas, whereas a linear dependence on the event-rate or on the magnitude of time constraints is prohibitive. Our monitors are event-rate independent and their time and space complexity does not depend on the magnitude of time constraints, i.e., their time and space complexity is *interval-oblivious*.

3.2 Multi-Head Monitoring Algorithm

A multi-head monitor uses multiple reading heads that read a trace forwards. In the following, let us fix a trace $\rho_{<\ell} = \langle(\Gamma_i,\, \tau_i)\rangle_{i<\ell}$. We model a reading head h over the trace $\rho_{<\ell}$ as an object from a set $\mathbb{H}$. The content of the trace $\rho_{<\ell}$ can be accessed via a function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H}\times\mathbb{T}\times\mathcal{P}(\Sigma))\cup\{\bot\}$ applied iteratively to the reading head $\mathsf{init_h}$, $\mathsf{init_h} \in \mathbb{H}$, positioned at the initial time-point of the trace $\rho_{<\ell}$. Given a reading head h, the function $\mathsf{adv_h}$ yields a special value $\bot$, i.e., $\mathsf{adv_h}(h) = \bot$, if the reading head h has reached the time-point ℓ, i.e., the end of the (finite) trace, and no more time-point can be provided. Otherwise, if the reading head h has reached the time-point $i < \ell$, then $\mathsf{adv_h}(h) = (h', \tau_i, \Gamma_i)$, where h' is the reading head advanced to the next time-point. We also use a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to read the time-stamp of the same time-point several times in a row without advancing the reading head. We use this interface ($\mathsf{init_h}$, $\mathsf{adv_h}$, and $\mathsf{read_h}$) in our monitoring algorithm to enforce that the monitoring algorithm is not reading time-points arbitrarily, but using a (constant) number of reading heads that read the time-points forwards.

We model a multi-head monitor's state for a formula ϕ over a trace $\rho_{<\ell}$ as an object from a set $\mathbb{M}$. Given a formula ϕ, the function $\mathsf{init_m}(\phi)$ yields the initial state of our multi-head monitor for the formula ϕ over the trace $\rho_{<\ell}$. In particular, the function $\mathsf{init_m}(\phi)$ uses $\mathsf{init_h}$ to get reading heads positioned at the initial time-point of the trace $\rho_{<\ell}$. The verdicts of our monitor can be obtained via the function $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$ applied iteratively to the initial state of the monitor $\mathsf{init_m}(\phi)$, $\mathsf{init_m}(\phi) \in \mathbb{M}$, where $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ are Boolean values. Given a state of the monitor m, the function $\mathsf{adv_m}$ yields a special value $\bot$, i.e., $\mathsf{adv_m}(m) = \bot$, if the monitor cannot compute any verdict, e.g., because it would require further time-points beyond the end of the (finite) trace. Otherwise, if the monitor has reached the state m from the initial state for the formula ϕ after producing i verdicts, then $\mathsf{adv_m}(m) = (m', \tau_i, \beta_i)$, where m' is the next state of the monitor and β_i is equivalent to $(\rho, i) \models \phi$. We remark that our multi-head monitor may output $\bot$ even if the trace implies a unique Boolean verdict for the formula, e.g., if the formula is a tautology. Still, our monitor's completeness theorem (Section 3.2.4) states that it outputs a verdict for a time-point if the reading heads could read *sufficiently many* time-points afterwards.

In the following, we formally define the set of our multi-head monitor's states $\mathbb{M}$, we define the function $\mathsf{init_m}$ computing the initial state of the monitor for a formula ϕ, and the function $\mathsf{adv_m}$ computing Boolean verdicts.

We present our multi-head monitor for MTL in Section 3.2.1, our *window* data structure for matching MDL regular expressions in Section 3.2.2, our multi-head monitor for MDL in Section 3.2.3, and our monitors' correctness and complexity analyses in Sections 3.2.4 and 3.2.5, respectively.

3.2.1 MTL Monitor

The set of our multi-head MTL monitor's states is defined as the smallest set satisfying the following:

$$\begin{aligned}\mathbb{M} = \{\bot\} \cup\ & \\ & (\Sigma \times \mathbb{H}) \cup \\ & (\{\neg\} \times \mathbb{M}) \cup \\ & (\{\vee\} \times \mathbb{M} \times \mathbb{M}) \cup \\ & (\{\bullet_I\} \times \mathbb{M} \times \mathbb{H} \times ((\mathbb{T} \times \mathbb{B}) \cup \{\bot\})) \cup \\ & (\{\bigcirc_I\} \times \mathbb{M} \times \mathbb{H} \times (\mathbb{T} \cup \{\bot\})) \cup \\ & (\{\mathsf{S}_I\} \times \mathbb{M} \times \mathbb{M} \times \mathbb{H} \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \cup \{\bot\}) \times (\mathbb{T} \cup \{\bot\})) \cup \\ & (\{\mathsf{U}_I\} \times \mathbb{H} \times \mathbb{M} \times \mathbb{M} \times \mathbb{H} \times \mathbb{N} \times ((\mathbb{T} \times \mathbb{B} \times \mathbb{B}) \cup \{\bot\})).\end{aligned}$$

The first component of every monitor's state (except $\bot$) denotes the main operator of the monitored formula or the atomic proposition $p \in \Sigma$ to be monitored. The meaning of the remaining components is explained in the following paragraphs. We remark that $\bot$ is also a state of the monitor used if a verdict can still be computed, but no proper next state of the monitor can be computed (e.g., because the reading head reached the end of the trace).

We formally define the initialization function $\mathsf{init_m}$ in Figure 3.1 and the function $\mathsf{adv_m}$ advancing the monitor's state and computing verdicts in Figure 3.2. The functions computing verdicts for the MTL temporal operators $\bullet_I\ \phi_0$, $\bigcirc_I\ \phi_0$, $\phi_1\ \mathsf{S}_I\ \phi_2$, and $\phi_1\ \mathsf{U}_I\ \phi_2$ are defined separately in Figure 3.3, 3.4, 3.5, and 3.6. They recursively evaluate $\mathsf{adv_m}$ for the submonitors.

Atomic Propositions and Boolean Operators For an atomic proposition $p \in \Sigma$, a simple one-head monitor with the state (p, h) reads the trace with a head h and computes the corresponding Boolean verdicts by checking if $p \in \Gamma_i$ (Figure 3.2).

```
context: A reading head $\mathsf{init_h}$ positioned at the initial time-point of a trace.
input:   A bounded-future MTL formula $\phi$.
output:  The initial state $\mathsf{init_m}(\phi) \in \mathbb{M}$ of our multi-head monitor.
function $\mathsf{init_m}(\phi) =$
    switch $\phi$ do
        case $p$ do return $(p, \mathsf{init_h})$;
        case $\neg\phi_0$ do return $(\neg, \mathsf{init_m}(\phi_0))$;
        case $\phi_1 \vee \phi_2$ do return $(\vee, \mathsf{init_m}(\phi_1), \mathsf{init_m}(\phi_2))$;
        case $\bullet_I\ \phi_0$ do return $(\bullet_I, \mathsf{init_m}(\phi_0), \mathsf{init_h}, \bot)$;
        case $\bigcirc_I\ \phi_0$ do return $(\bigcirc_I, \mathsf{init_m}(\phi_0), \mathsf{init_h}, \bot)$;
        case $\phi_1\ \mathsf{S}_I\ \phi_2$ do return $(\mathsf{S}_I, \mathsf{init_m}(\phi_1), \mathsf{init_m}(\phi_2), \mathsf{init_h}, 0, 0, \bot, \bot)$;
        case $\phi_1\ \mathsf{U}_I\ \phi_2$ do return $(\mathsf{U}_I, \mathsf{init_h}, \mathsf{init_m}(\phi_1), \mathsf{init_m}(\phi_2), \mathsf{init_h}, 0, \bot)$;
```

Figure 3.1. The initial state of our multi-head monitor $\mathsf{init_m}(\phi)$ for an MTL formula ϕ.

A monitor for a Boolean operator recursively consists of monitors for the subformulas of the Boolean operator. The state $(\vee, m_1, m_2)$ of the monitor for a formula $\phi_1 \vee \phi_2$ consists of two *submonitors* for the subformulas ϕ_1 and ϕ_2. The submonitors for ϕ_1 and ϕ_2 are synchronous, i.e., the submonitors are evaluated in rounds producing verdicts for the same time-point. Analogously, the state $(\neg, m_0)$ of the monitor for a formula $\neg\phi_0$ consists of a single submonitor for the subformula ϕ_0. The verdicts for a Boolean operator are computed by applying the Boolean operator to the verdicts computed recursively by the submonitors (Figure 3.2).

Previous **and** ***Next*** **MTL Temporal Operators** A monitor for a unary MTL temporal operator $\bullet_I\ \phi_0$ (Figure 3.3) or $\bigcirc_I\ \phi_0$ (Figure 3.4) uses a submonitor for the subformula ϕ_0 to shift the sequence of Boolean values from the submonitor's verdicts by a single time-point either in the past or in the future while replacing the Boolean values by ff at time-points for which the interval condition is not met. The state of the monitor consists of a submonitor m_0 for the subformula ϕ_0 (used to compute the Boolean verdicts for ϕ_0), a time-stamp (used to check the interval condition from the semantics of the temporal operator), and a Boolean value from the previous verdict (the Boolean value from the previous verdict is only needed for the $\bullet_I$ operator). Before reading the first verdict, the time-stamp and the Boolean value from the previous verdict are initialized to $\bot$. To be able to compute a verdict for $\bullet_I\ \phi_0$ or $\bigcirc_I\ \phi_0$ if the submonitor for ϕ_0 could not produce a verdict, but the interval condition is not met, the monitor for $\bullet_I\ \phi_0$ and $\bigcirc_I\ \phi_0$ uses an additional reading head h to check the interval condition independently of the submonitor for ϕ_0. Formally, the state for the $\bullet_I$ operator is $(\bullet_I, m_0, h, z)$, where z is the time-stamp and Boolean value from the previous time-point (or $\bot$ if there is no such time-point yet), and $(\bullet_I, m_0, h, \tilde{\tau}_0)$, where $\tilde{\tau}_0$ is the time-stamp from the previous time-point (or $\bot$ if there is no such time-point yet).

Since **MTL Temporal Operator** A monitor for the binary MTL temporal operator $\phi_1\ \mathsf{S}_I\ \phi_2$ (Figure 3.5) uses two submonitors (for the subformulas ϕ_1 and ϕ_2) that are asynchronous, i.e., the submonitors are not evaluated in rounds producing verdicts for the same time-point (as it was the case for the binary Boolean operators). Formally, its state is $(\mathsf{S}_I, m_1, m_2, h, c_1, c_2, \tilde{c}_2^+, \tilde{\tau}_2^+)$. The submonitor m_1 for ϕ_1 produces a verdict for the time-point i at which the monitor for $\phi_1\ \mathsf{S}_I\ \phi_2$ produces a verdict. This way, the monitor for $\phi_1\ \mathsf{S}_I\ \phi_2$ can maintain the number c_1 of time-points

context: A function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$ to read a time-point using a reading head and advance the reading head, a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to only read the time-stamp of a time-point using a reading head (without advancing the reading head).
input: A state $m \in \mathbb{M}$ of our multi-head monitor.
output: A tuple (m', τ, β) or $\bot$ if no verdict can be computed, where $m' \in \mathbb{M}$ is the next state of our multi-head monitor and $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$ is the Boolean verdict for the current time-point.

```
function $\mathsf{adv_m}(m)$ =
  switch $m$ do
    case $(p, h)$ do
      switch $\mathsf{adv_h}(h)$ do
        case $(h', \tau, \Gamma)$ do return $((p, h'), \tau, p \in \Gamma)$;
        case $\bot$ do return $\bot$;
    case $(\neg, m_0)$ do
      switch $\mathsf{adv_m}(m_0)$ do
        case $(m'_0, \tau, \beta)$ do return $((\neg, m'_0), \tau, \text{not } \beta)$;
        case $\bot$ do return $\bot$;
    case $(\vee, m_1, m_2)$ do
      switch $\mathsf{adv_m}(m_1)$ do
        case $(m'_1, \tau_1, \beta_1)$ do
          switch $\mathsf{adv_m}(m_2)$ do
            case $(m'_2, \_, \beta_2)$ do return $((\vee, m'_1, m'_2), \tau_1, \beta_1 \text{ or } \beta_2)$;
            case $\bot$ do return $\bot$;
        case $\bot$ do return $\bot$;
    case $(\bullet_I, \_, \_, \_)$ do return $\mathsf{doPrev}(\mathsf{adv_m}, m)$;
    case $(\bigcirc_I, \_, \_)$ do return $\mathsf{doNext}(\mathsf{adv_m}, m)$;
    case $(\mathsf{S}_I, \_, \_, \_, \_, \_, \_, \_)$ do return $\mathsf{doSince}(\mathsf{adv_m}, m)$;
    case $(\mathsf{U}_I, \_, \_, \_, \_, \_, \_)$ do return $\mathsf{doUntil}(\mathsf{adv_m}, m)$;
    case $\bot$ do return $\bot$;
```

Figure 3.2. The function $\mathsf{adv_m}$ computing verdicts of our MTL multi-head monitor.

context: A function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$ to read a time-point using a reading head and advance the reading head, a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to only read the time-stamp of a time-point using a reading head (without advancing the reading head).
input: A state $m \in \mathbb{M}$ of our multi-head monitor for $\bullet_I\ \phi$ and the function $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$ computing verdicts for submonitors of m.
output: A tuple (m', τ, β) or $\bot$ if no verdict can be computed, where $m' \in \mathbb{M}$ is the next state of our multi-head monitor and $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$ is the Boolean verdict for the current time-point.

```
function doPrev($\mathsf{adv_m}$, $(\bullet_I, m_0, h, z)$) =
    switch $\mathsf{adv_h}(h)$ do
        case $(h', \tau, \_)$ do
            switch $z$ do
                case $(\tau_0, \beta_0)$ do $\beta := \beta_0$ and $\mathsf{mem}(\tau_0, \tau, I)$;
                case $\bot$ do $\beta := \mathsf{ff}$;
            switch $\mathsf{adv_m}(m_0)$ do
                case $(m_0', \_, \beta')$ do return $((\bullet_I, m_0', h', (\tau, \beta')), \tau, \beta)$;
                case $\bot$ do return $(\bot, \tau, \beta)$;
        case $\bot$ do return $\bot$;
```

Figure 3.3. The function $\mathsf{doPrev}(\mathsf{adv_m}, m)$ computing verdicts for $\bullet_I\ \phi$.

context: A function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$ to read a time-point using a reading head and advance the reading head, a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to only read the time-stamp of a time-point using a reading head (without advancing the reading head).
input: A state $m \in \mathbb{M}$ of our multi-head monitor for $\bigcirc_I\ \phi$ and the function $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$ computing verdicts for submonitors of m.
output: A tuple (m', τ, β) or $\bot$ if no verdict can be computed, where $m' \in \mathbb{M}$ is the next state of our multi-head monitor and $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$ is the Boolean verdict for the current time-point.

```
function doNext($\mathsf{adv_m}$, $(\bigcirc_I, m_0, h, \tilde{\tau}_0)$) =
    switch $\mathsf{adv_h}(h)$ do
        case $(h', \tau, \_)$ do
            if $\tilde{\tau}_0 = \bot$ then
                switch $\mathsf{adv_m}(m_0)$ do
                    case $(m_0', \_, \_)$ do return doNext($\mathsf{adv_m}$, $(\bigcirc_I, m_0', h', \tau)$);
                    case $\bot$ do return $\bot$;
            switch $\mathsf{adv_m}(m_0)$ do
                case $(m_0', \_, \beta)$ do return $((\bigcirc_I, m_0', h', \tau)), \tilde{\tau}_0, \beta$ and $\mathsf{mem}(\tilde{\tau}_0, \tau, I))$;
                case $\bot$ do
                    if $\mathsf{mem}(\tilde{\tau}_0, \tau, I)$ then return $\bot$;
                    else return $(\bot, \tilde{\tau}_0, \mathsf{ff})$;
        case $\bot$ do return $\bot$;
```

Figure 3.4. The function $\mathsf{doNext}(\mathsf{adv_m}, m)$ computing verdicts for $\bigcirc_I\ \phi$.

context: A function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$ to read a time-point using a reading head and advance the reading head, a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to only read the time-stamp of a time-point using a reading head (without advancing the reading head).
input: A state $m \in \mathbb{M}$ of our multi-head monitor for $\phi_1 \ \mathsf{S}_I \ \phi_2$ and the function $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$ computing verdicts for submonitors of m.
output: A tuple (m', τ, β) or $\bot$ if no verdict can be computed, where $m' \in \mathbb{M}$ is the next state of our multi-head monitor and $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$ is the Boolean verdict for the current time-point.

```
function doSince($\mathsf{adv_m}$, ($\mathsf{S}_I, m_1, m_2, h, c_1, c_2, \tilde{c}_2^+, \tilde{\tau}_2^+$)) =
    switch $\mathsf{adv_m}(m_1)$ do
        case $(m_1', \tau, \beta_1)$ do
            if $\beta_1$ then $c_1 := c_1 + 1$;
            else $c_1 := 0$;
            $c_2 := c_2 + 1$;
            if $\tilde{c}_2^+ \neq \bot$ then $\tilde{c}_2^+ := \tilde{c}_2^+ + 1$;
            while $c_2 > 0$ and $\mathsf{memL}(\mathsf{read_h}(h), \tau, I)$ do
                switch $\mathsf{adv_h}(h)$ do
                    case $(h', \_, \_)$ do $h := h'$;
                switch $\mathsf{adv_m}(m_2)$ do
                    case $(m_2', \tau_2, \beta_2)$ do
                        $m_2 := m_2'$;
                        if $\beta_2$ then $\tilde{c}_2^+ := c_2$; $\tilde{\tau}_2^+ := \tau_2$;
                    case $\bot$ do return $\bot$;
                $c_2 := c_2 - 1$;
            $\beta := \tilde{c}_2^+ \neq \bot$ and $\tilde{c}_2^+ - 1 \leq c_1$ and $\mathsf{memR}(\tilde{\tau}_2^+, \tau, I)$;
            return $((\mathsf{S}_I, m_1', m_2, h, c_1, c_2, \tilde{c}_2^+, \tilde{\tau}_2^+), \tau, \beta)$;
        case $\bot$ do return $\bot$;
```

Figure 3.5. The function $\mathsf{doSince}(\mathsf{adv_m}, m)$ computing verdicts for $\phi_1 \ \mathsf{S}_I \ \phi_2$.

context: A function $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$ to read a time-point using a reading head and advance the reading head, a function $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$ to only read the time-stamp of a time-point using a reading head (without advancing the reading head).
input: A state $m \in \mathbb{M}$ of our multi-head monitor for $\phi_1 \mathsf{U}_I \phi_2$ and the function $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$ computing verdicts for submonitors of m.
output: A tuple (m', τ, β) or $\bot$ if no verdict can be computed, where $m' \in \mathbb{M}$ is the next state of our multi-head monitor and $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$ is the Boolean verdict for the current time-point.

```
function loopCondUntil($I, \tau, h_2, c, z$) =
    if $c \neq 0$ then
        switch $z$ do
            case $(\tau', \beta_1, \beta_2)$ do
                if ($\beta_2$ and memL($\tau, \tau', I$)) or not $\beta_1$ then return ff;
    switch read_h($h_2$) do
        case $(\tau', \_)$ do return memR($\tau, \tau', I$);
        case $\bot$ do return ff;
function doUntil(adv_m, $(\mathsf{U}_I, h_1, m_1, m_2, h_2, c, z)$) =
    switch adv_h($h_1$) do
        case $(h_1', \tau, \_)$ do
            while loopCondUntil($I, \tau, h_2, c, z$) do
                switch adv_m($m_1$) do
                    case $(m_1', \_, \beta)$ do $m_1 := m_1'$; $\beta_1 := \beta$;
                    case $\bot$ do return $\bot$;
                switch adv_m($m_2$) do
                    case $(m_2', \_, \beta)$ do $m_2 := m_2'$; $\beta_2 := \beta$;
                    case $\bot$ do return $\bot$;
                $(h_2, \tau', \_) :=$ adv_h($h_2$);
                $c := c + 1$;
                $z := (\tau', \beta_1, \beta_2)$;
            if $c = 0$ then return $\bot$;
            else
                switch $z$ do
                    case $(\tau', \beta_1, \beta_2)$ do
                        if $\beta_2$ and memL($\tau, \tau', I$) then return
                         $((\mathsf{U}_I, h_1', m_1, m_2, h_2, c-1, z), \tau, \mathsf{tt})$;
                        else if not $\beta_1$ then return $((\mathsf{U}_I, h_1', m_1, m_2, h_2, c-1, z), \tau, \mathsf{ff})$;
                        else if read_h($h_2$) $= \bot$ then return $\bot$;
                        else return $((\mathsf{U}_I, h_1', m_1, m_2, h_2, c-1, z), \tau, \mathsf{ff})$;
        case $\bot$ do return $\bot$;
```

Figure 3.6. The function $\mathsf{doUntil}(\mathsf{adv_m}, m)$ computing verdicts for $\phi_1 \mathsf{U}_I \phi_2$.

before (and including i) at which ϕ_1 is satisfied. The submonitor m_2 for ϕ_2 is evaluated up until the most recent time-point that satisfies the interval condition. To check the interval condition independently of the submonitor for ϕ_2, the monitor for $\phi_1 \mathsf{S}_I \phi_2$ uses an additional reading head h. Furthermore, the monitor for $\phi_1 \mathsf{S}_I \phi_2$ maintains in constant space the most recent time-point j satisfying ϕ_2 and also satisfying the interval condition. This time-point j is represented by its time-stamp $\tilde{\tau}_2^+$ and the difference $\tilde{c}_2^+ := i - j$ in the state, where $\tilde{\tau}_2^+ = \bot$ and $\tilde{c}_2^+ = \bot$ if no such time-point j exists. To be able to update $\tilde{c}_2^+$, the monitor for $\phi_1 \mathsf{S}_I \phi_2$ always maintains the difference c_2 between the number of verdicts produced by the submonitors for ϕ_1 and ϕ_2, respectively. If the time-point i satisfies the interval condition (e.g., $0 \in I$), then the submonitor for ϕ_2 is evaluated until it produces a verdict for the time-point i (in particular, the submonitor for ϕ_2 never overtakes the submonitor for ϕ_1). To compute a Boolean verdict for $\phi_1 \mathsf{S}_I \phi_2$ at the time-point i, the monitor checks that $\tilde{c}_2^+ \neq \bot$ (otherwise no j from the semantics of $\phi_1 \mathsf{S}_I \phi_2$ exists), it also checks that ϕ_1 holds *since* ϕ_2 (the condition $\tilde{c}_2^+ - 1 \leq c_1$, where $\tilde{c}_2^+ = i + 1 - j$ holds on Line 17 in Figure 3.5), and it checks the interval condition ($\mathsf{memR}(\tilde{\tau}_2^+, \tau, I)$).

***Until* MTL Temporal Operator** A monitor $(\mathsf{U}_I, h_1, m_1, m_2, h_2, c, z)$ for the binary MTL temporal operator $\phi_1 \mathsf{U}_I \phi_2$ (Figure 3.6) uses two submonitors m_1 and m_2 for the subformulas ϕ_1 and ϕ_2 and one additional reading head h_1 that is used to obtain the time-stamp of the time-point i at which a Boolean verdict for $\phi_1 \mathsf{U}_I \phi_2$ is computed. The submonitors for ϕ_1 and ϕ_2 are synchronous, i.e., the submonitors are evaluated in rounds producing verdicts for the same time-point, and ahead of the reading head h_1. They are evaluated as long as the formula ϕ_1 is satisfied until a satisfaction of ϕ_2 satisfying the interval bound is found (then $\phi_1 \mathsf{U}_I \phi_2$ is satisfied) or a time-point beyond the interval bound is encountered (then $\phi_1 \mathsf{U}_I \phi_2$ is not satisfied). The formula $\phi_1 \mathsf{U}_I \phi_2$ is not satisfied at the time-point i if ϕ_1 does not hold at a time-point $j \geq i$ and no satisfaction of ϕ_2 satisfying the interval bounds has been found so far. To be able to compute a verdict for $\phi_1 \mathsf{U}_I \phi_2$ if the submonitor for ϕ_1 or ϕ_2 could not produce a verdict, but the interval condition is not met, the monitor for $\phi_1 \mathsf{U}_I \phi_2$ uses an additional reading head h_2 to check the interval condition independently of the submonitors for ϕ_1 and ϕ_2. Because the function $\mathsf{adv_m}$ returns the next state of the monitor and applying $\mathsf{adv_m}$ to the next state of the monitor would yield verdicts for the next time-point, the monitor for $\phi_1 \mathsf{U}_I \phi_2$ caches the most recent verdicts z of the submonitors for ϕ_1 and ϕ_2. To know when the time-point i at which a Boolean verdict for $\phi_1 \mathsf{U}_I \phi_2$ is computed reaches the time-point k at which the verdicts z were computed, the monitor for $\phi_1 \mathsf{U}_I \phi_2$ also stores the number of time-points c between i and k. Note that z can only be $\bot$ if $c = 0$.

Example 3.1. Figure 3.7 shows our multi-head monitor's state and the positions of its reading heads while monitoring the formula $\phi := (a \mathsf{S}_{[2,4]} b) \wedge (a \mathsf{U}_{[0,4]} b)$ on the trace with time-stamps $0, 3, 4, 6, 8, 8, 14$ and sets of events $\{a, b\}, \{a\}, \{a\}, \{a\}, \{a, b\}, \{a\}, \{b\}$ after the monitor just computed the Boolean verdict tt for the third time-point with time-stamp 4. The reading heads are depicted as arrows. The submonitor m_1 of $\mathsf{S}_{[2,4]}$ as well as the reading head h_1 of $\mathsf{U}_{[0,4]}$ are positioned at the next time-point for which the overall monitor is going to produce a verdict (the fourth time-point with time-stamp 6). The reading head h and the submonitor m_2 of $\mathsf{S}_{[2,4]}$ are positioned at the first time-point that does not satisfy the interval's lower bound with respect to the third time-point at which the $\mathsf{S}_{[2,4]}$ is evaluated (second time-point with time-stamp 3 because $0 + 2 \leq 4$ holds, but $3 + 2 \leq 4$ does not hold anymore). The remaining entries of the state for $\mathsf{S}_{[2,4]}$ have the following values:

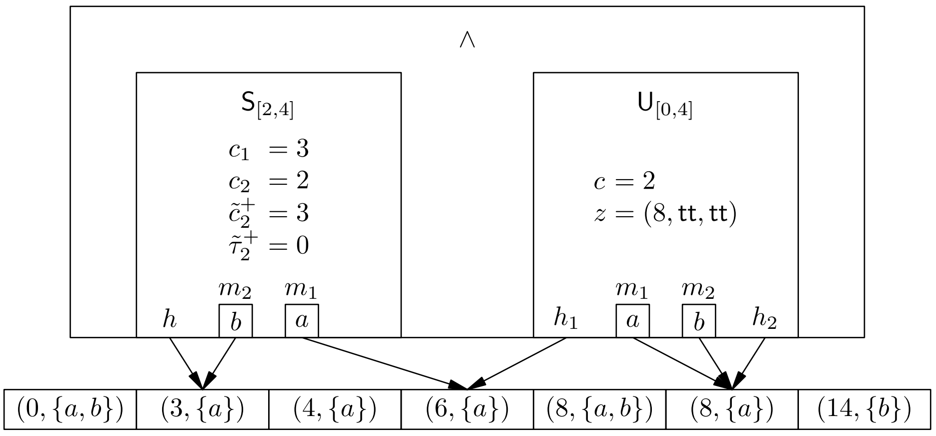


Figure 3.7. An example of our multi-head monitor's state and the positions of its reading heads for the formula $\phi \coloneqq (a \mathsf{S}_{[2,4]} b) \wedge (a \mathsf{U}_{[0,4]} b)$.

- $c_1 = 3$ because a is satisfied at the last three time-points (up to and including the third time-point);
- $c_2 = 2$ because there are 2 time-points between the positions of the submonitors m_1 and m_2;
- $\tilde{c}_2^+ = 3$ because there are 3 time-points between the most recent satisfaction of b within the interval and the position of the submonitor m_1;
- $\tilde{\tau}_2^+ = 0$ because the time-stamp of the most recent satisfaction of b within the interval (the initial time-point of the trace) is 0.

Then the condition for the satisfaction of $a \mathsf{S}_{[2,4]} b$ at the third time-point with time-stamp 4 is $\tilde{c}_2^+ \neq \bot$ and $\tilde{c}_2^+ - 1 \leq c_1$ and $\mathsf{memR}(\tilde{\tau}_2^+, \tau, [2,4])$, where $\tau = 4$. This condition is satisfied and thus the multi-head monitor for $a \mathsf{S}_{[2,4]} b$ yields the Boolean verdict tt.

The submonitors m_1 and m_2 as well as the reading head h_2 of $\mathsf{U}_{[2,4]}$ are at the next time-point after a satisfaction of b within the interval at the fifth time-point with time-stamp 8. The Boolean verdicts for a and b (both tt) at this time-point as well as the time-stamp 8 are cached in $z = (8, \mathsf{tt}, \mathsf{tt})$. Because b is satisfied and the interval condition $\mathsf{mem}(4, 8, [2,4])$ is satisfied, the multi-head monitor for $a \mathsf{U}_{[0,4]} b$ yields the Boolean verdict tt. The value $c = 2$ represents the number of time-points between h_1 and h_2 (m_1 and m_2 are positioned at the same time-point as h_2).

Finally, the multi-head monitor for the formula $\phi \coloneqq (a \mathsf{S}_{[2,4]} b) \wedge (a \mathsf{U}_{[0,4]} b)$ combines the two Boolean verdicts tt for the $\wedge$ operator into the overall Boolean verdict tt. $\diamond$

3.2.2 Matching Regular Expressions

In this section, we focus on a fixed MDL regular expression r independently of whether r is used in a past or future match formula and independently of the interval I of the match formula. We first convert r into an automaton over the alphabet $\mathbb{B}^k$, where k is the number of r's direct subformulas ψ_j, $1 \leq j \leq k$, according to an arbitrary formula ordering, e.g., left-to-right with respect to the first occurrence of the formula in the regular expression. For each time-point, the automaton's input symbol is constructed from k Boolean verdicts for r's direct subformulas at

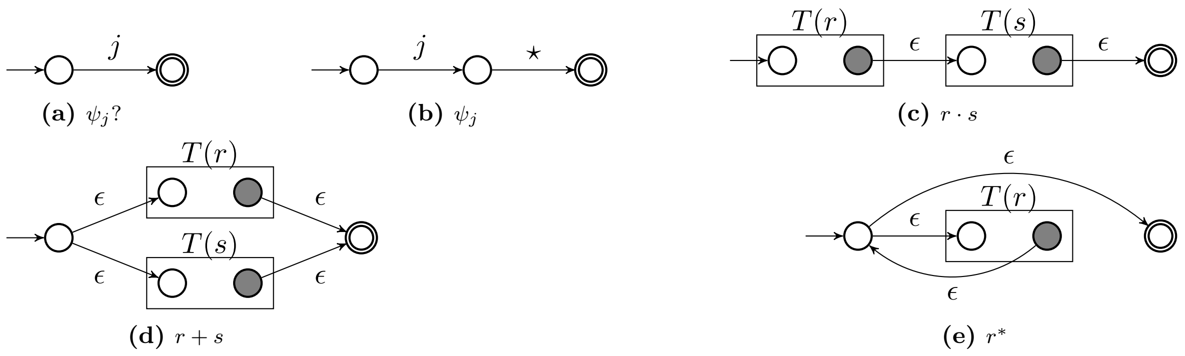


Figure 3.8. Recursive conversion T of an MDL regular expression into the transition relation of a nondeterministic automaton. The states that are eliminated by our implementation are marked gray.

this time-point. To compute an input symbol $b = (b_1, \ldots, b_k)$, a multi-head submonitor is run for each formula ψ_j, $1 \leq j \leq k$, to determine $b_j \in \mathbb{B}$, i.e., k synchronous multi-head monitors are run to compute $b = (b_1, \ldots, b_k)$.

A central component of our multi-head monitor for a match formula $\langle r|_I$ or $|r\rangle_I$ is a *window* data structure for the MDL regular expression r that maintains a summary of the automaton runs on a finite subword of the automaton's input stream. The subword starts at a position i (called *window's start*) and ends at j (called *window's end*), where i and j can be arbitrarily far apart, but the window's size does not depend on the difference $j - i$. The window data structure uses a reading head over the trace and a sequence of submonitors for the direct subformulas of r. They are used to update the time-stamps stored in the window data structure and compute the input symbols for the automaton. We include the reading heads over the trace to obtain the time-stamps at the window's start and end although the verdicts of the submonitors yields exactly these time-stamps: the regular expression r might have no direct subformulas (e.g., the regular expression *true* · *true* matching a pair of time-points) in which case there are no submonitors and the monitoring algorithm evaluating the match operator needs these reading heads to check the interval condition independently of the submonitors that might not be able to compute a Boolean verdict. Hence, it makes no sense to distinguish whether r has a direct subformula or not because the reading heads over the trace would be needed by the monitoring algorithm evaluating the match operator anyway.

Converting an MDL regular expression to a nondeterministic automaton We first convert an MDL regular expression r into a nondeterministic automaton with ϵ-transitions over the alphabet $\mathbb{B}^k$. There are three types of transitions in the nondeterministic automaton:

- ϵ-transitions that implement lookahead regular expressions ψ? and thus passing such an ϵ-transition depends on the input symbol, we denote by them by the index j of the direct subformula $\psi = \psi_j$ in the input symbol of the automaton;

- $\star$ transitions that consume an arbitrary input symbol;

- ϵ-transitions that implement nondeterministic choice that can be passed on an arbitrary input symbol.

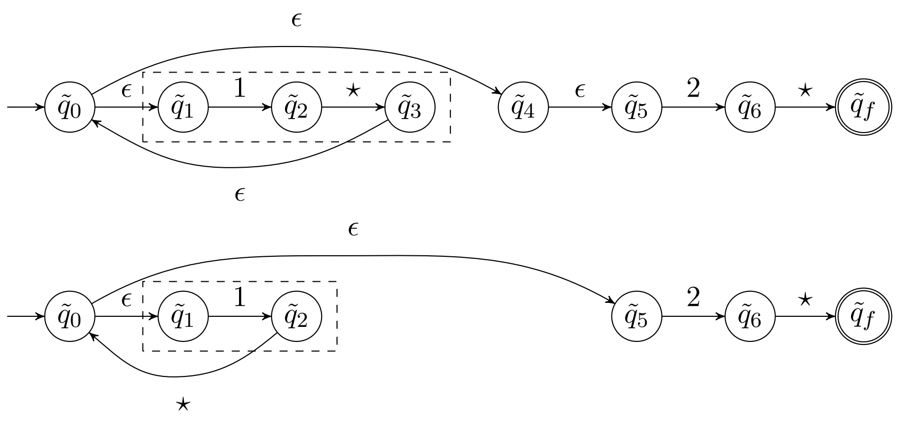


Figure 3.9. The ϵ-NFAs for $p^* \cdot q$, where $p = \psi_1$ and $q = \psi_2$, with the dashed rectangles showing the ϵ-NFA for $p = \psi_1$. The ϵ-NFA at the top is obtained by directly applying the construction in Figure 3.8. The ϵ-NFA at the bottom is obtained by eliminitaing the states marked gray in Figure 3.8.

To construct the transition relation of the nondeterministic automaton for an MDL regular expression, we use Thompson's standard construction mildly adapted to MDL regular expressions and the three types of edges described before. A recursive function T on MDL regular expressions that computes the transition relation of a regular expression together with an initial and accepting state is defined in Figure 3.8. We remark that states whose outgoing transitions are only ϵ-transitions implementing nondeterministic choice can be eliminated by redirecting all incoming transitions to all the target states of the outgoing transitions. The states that are eliminated by our implementation are marked gray in Figure 3.8.

Converting a nondeterministic automaton for MDL to a deterministic automaton Because our window data structure requires a deterministic automaton, we determinize the ϵ-NFA $\mathcal{A}_{\mathrm{N}}$ for an MDL regular expression using the subset construction. We label $\mathcal{A}_{\mathrm{N}}$'s nondeterministic states by $\tilde{q}$ and sets of nondeterministic states by $\tilde{Q}$. A difficulty arises from the ϵ-transitions that implement lookahead regular expressions $\psi?$ (labeled by the index j of the direct subformula $\psi = \psi_j$ in the input symbol of the automaton). We introduce two types of closures for a set of states $\tilde{Q}$ to handle them:

- the $\epsilon?$-closure that depends on the input symbol b is obtained by following ϵ-transitions that implement lookahead regular expressions $\psi?$ whose formulas are satisfied in b and by following ϵ-transitions that implement nondeterministic choice;

- the ϵ-closure that is obtained by only following ϵ-transitions that implement nondeterministic choice, indepedently of the input symbol.

The transition function $\delta(\tilde{Q}, b)$ first computes the $\epsilon?$-closure $\tilde{Q}_b^{\epsilon?}$ of $\tilde{Q}$ with respect to the input symbol b and then computes the set $\tilde{Q}_b^{\epsilon?\star}$ of states reachable from a state in $\tilde{Q}_b^{\epsilon?}$ by following a *single* $\star$-transition, which consumes the input symbol b. In particular, the set $\tilde{Q}_b^{\epsilon?\star}$ is not necessarily $\epsilon?$-closed with respect to the next input symbol or ϵ-closed. When checking if a set of states $\tilde{Q}$ is accepting, we first compute the ϵ-closure $\tilde{Q}^{\epsilon}$ of $\tilde{Q}$ and then check if an accepting state is in $\tilde{Q}^{\epsilon}$.

To summarize, we convert an MDL regular expression r into a DFA $\mathcal{A}_{\mathrm{D}} = (Q, \mathbb{B}^k, \delta, q_0, F)$, where

- Q is the set of $\mathcal{A}_{\mathrm{D}}$'s states consisting of all subsets of the set of $\mathcal{A}_{\mathrm{N}}$'s states;
- $\delta : Q \times \mathbb{B}^k \to Q$ is the transition function;
- q_0 is $\mathcal{A}_{\mathrm{D}}$'s initial state, i.e., the singleton set consisting of $\mathcal{A}_{\mathrm{N}}$'s initial state;
- F is the set of $\mathcal{A}_{\mathrm{D}}$'s accepting states.

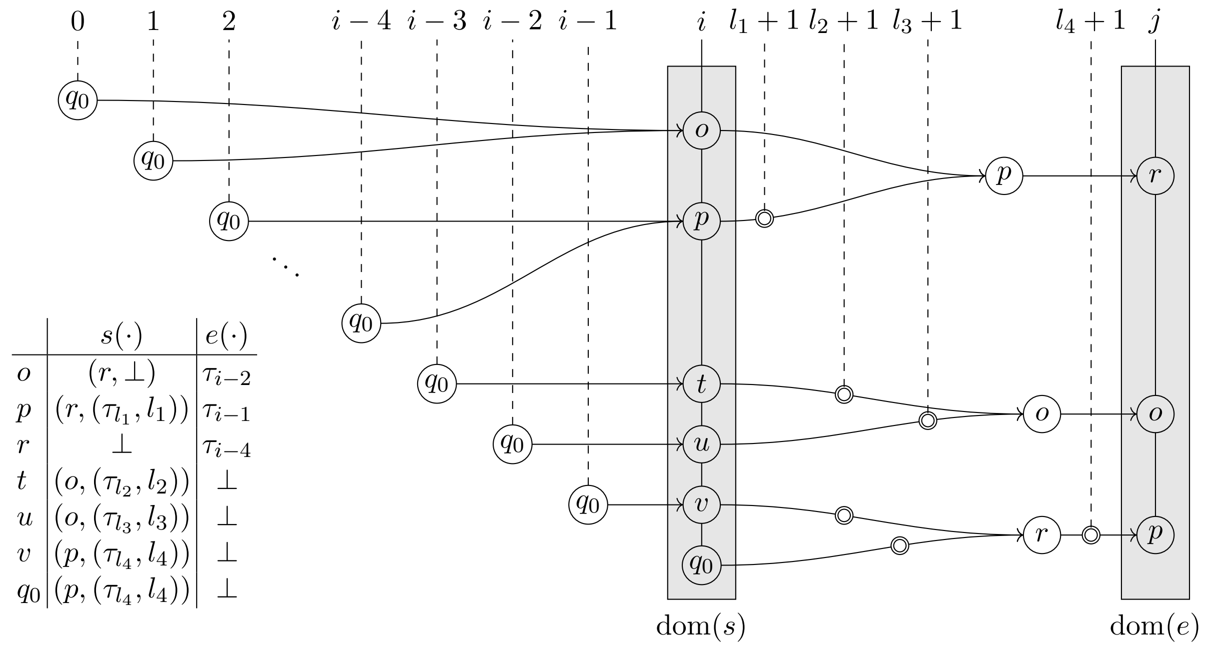
Example 3.2. Figure 3.9 shows the ϵ-NFA computed for the regular expression $p^* \cdot q$. $\diamond$

Given a pair of time-points (i, j), $i \leq j$, we say that the DFA $\mathcal{A}_{\mathrm{D}}$ *reaches a state q' from a state q on (i, j)*, denoted $q \rightsquigarrow_{(i,j)} q'$, iff the state q' is reached by running $\mathcal{A}_{\mathrm{D}}$ from the state q at time-point i until time-point j. In particular, we have $q \rightsquigarrow_{(i,i)} q$, for all q and i. Furthermore, we say that $\mathcal{A}_{\mathrm{D}}$ *accepts from a state q on (i, j)*, denoted $q \rightsquigarrow_{(i,j)} \checkmark$, iff the state q' reached by $\mathcal{A}_{\mathrm{D}}$ from q on (i, j) is accepting, i.e., $q' \in F$. We point out that the input symbol for the time-point j is not needed to decide if $q \rightsquigarrow_{(i,j)} q'$ holds and also to decide if $q \rightsquigarrow_{(i,j)} \checkmark$ holds.

Window Data Structure The window data structure for an MDL regular expression r consists of a pair of time-points (i, j), with $i \leq j$, two functions $s : Q \to Q \times ((\mathbb{T} \times \mathbb{N}) \cup \{\bot\}) \cup \{\bot\}$ and $e : Q \to \mathbb{T} \cup \{\bot\}$, two reading heads h_1 and h_2 reading the trace, and two sequences of submonitors $\bar{m}_1$ and $\bar{m}_2$ for the direct subformulas of r. The function s represents the runs of $\mathcal{A}_{\mathrm{D}}$ from a given state at the window's start by the state reached at the window's end and the last *accepting* time-point (along with the corresponding time-stamp) within the window, i.e., the last time-point after which the run was in an accepting state (if such a time-point exists). The function e yields the time-stamp of the latest time-point before the window's start from which a given state at the window's end can be reached from the initial state. The reading head h_1 and the submonitors $\bar{m}_1$ are positioned at the window's start (time-point i) while the reading head h_2 and the submonitors $\bar{m}_2$ are positioned at the window's end (time-point j). The function $\mathsf{adv}_{\mathsf{ms}}(\bar{m}) : \mathbb{M}^* \to (\mathbb{M}^* \times \mathbb{B}^*) \cup \{\bot\}$ evaluates $\mathsf{adv}_{\mathsf{m}}(m)$ for every m in $\bar{m}$, and returns the next states of the submonitors and the Boolean values assembled from their verdicts. If $\mathsf{adv}_{\mathsf{m}}(m) = \bot$, for some m in $\bar{m}$, then $\mathsf{adv}_{\mathsf{ms}}(\bar{m}) = \bot$.

Figure 3.10 visualizes the window data structure and the underlying automaton runs. The window is comprised of the time-points i and j and the functions s and e represented by the table on the left. The reading heads h_1 and h_2 and the submonitors $\bar{m}_1$ and $\bar{m}_2$ are also part of the window, but they are left out in Figure 3.10 for the sake of simplicity. Figure 3.10 shows $\mathcal{A}_{\mathrm{D}}$'s runs justifying the table's content. The individual runs are depicted by arrows from the initial state q_0. We use standard notation for accepting states, including the smaller circles, which denote states whose name is irrelevant. We also use the following notation: $\mathrm{dom}(f)$ of a partial function $f : X \to Y \cup \{\bot\}$ denotes f's domain, i.e., $\mathrm{dom}(f) = \{x \in X \mid f(x) \neq \bot\}$.

The domain of s are all the states reached by running $\mathcal{A}_{\mathrm{D}}$ from the initial state q_0 at a time-point $l \leq i$ before the window's start i until i (including the initial state itself obtained by running from i to i). The value of $s(q) = (q', \mathit{tstp})$ for a state $q \in \mathrm{dom}(s)$ is obtained by running $\mathcal{A}_{\mathrm{D}}$ further from the state q at the window's start i until the window's end j to a state q'. For example, the state r at the window's end j is reached by running $\mathcal{A}_{\mathrm{D}}$ from the states o and p at the window's start i in Figure 3.10. Moreover, tstp represents the maximum accepting time-point



	$s(\cdot)$	$e(\cdot)$
o	$(r, \bot)$	τ_{i-2}
p	$(r, (\tau_{l_1}, l_1))$	τ_{i-1}
r	$\bot$	τ_{i-4}
t	$(o, (\tau_{l_2}, l_2))$	$\bot$
u	$(o, (\tau_{l_3}, l_3))$	$\bot$
v	$(p, (\tau_{l_4}, l_4))$	$\bot$
q_0	$(p, (\tau_{l_4}, l_4))$	$\bot$

Figure 3.10. The window data structure with start i and end j

l after i and strictly before j, i.e., the maximum l, $i \leq l < j$ such that $q \rightsquigarrow_{(i,l+1)} \checkmark$. Hence, we have $s(o) = (r, \bot)$ in Figure 3.10 because there is no such time-point l. In contrast, we have $s(p) = (r, (\tau_{l_1}, l_1))$ because the run from p to r contains an accepting state after time-point l_1 (which is the only accepting state in this run and thus also the maximum one). Similarly, we have $s(q_0) = (p, (\tau_{l_4}, l_4))$ because the time-point l_4 is the maximum of the two accepting time-points in the run from the initial state q_0 at time-point i to the state p at time-point j.

The domain of e are all the states reached by running $\mathcal{A}_{\mathrm{D}}$ from the initial state q_0 at a time-point l strictly before the window's start i until the window's end j. The value of $e(q) = \tau$ for a state $q \in \mathrm{dom}(e)$ is the time-stamp of the maximum time-point l from which q was reached from the initial state q_0. For example, $e(p) = \tau_{i-1}$ in Figure 3.10 because p is reached by running from q_0 at time-point $i-1$ until j. Note that p is also reached by running from i, but i is not strictly before the window's start and thus i is not considered.

Formally, a window satisfies the invariant $\mathsf{window}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ if

- h_1, $\bar{m}_1$ are at time-point i; h_2, $\bar{m}_2$ are at time-point j;
- the domain of s, i.e., $\mathrm{dom}(s)$, are all states q such that $q_0 \rightsquigarrow_{(l,i)} q$, for some $l \leq i$;
- the domain of e, i.e., $\mathrm{dom}(e)$, are all states q such that $q_0 \rightsquigarrow_{(l,j)} q$, for some $l < i$;
- for any $q \in \mathrm{dom}(s)$: $s(q) = (q', \mathit{tstp})$, where $q \rightsquigarrow_{(i,j)} q'$ and $\mathit{tstp} = (\tau_l, l)$ for the maximum time-point l with $i \leq l < j$ and $q \rightsquigarrow_{(i,l+1)} \checkmark$, or $\mathit{tstp} = \bot$ if no such l exists;
- for any $q \in \mathrm{dom}(e)$: $e(q) = \tau$, where $\tau = \tau_l$ is the time-stamp of the maximum time-point $l < i$ such that $q_0 \rightsquigarrow_{(l,j)} q$.

Example 3.3. We now exemplify the window data structure by tracing its evolution through a sequence of window updates, which we manually selected. In the actual monitor, the update

τ_i	10	20	30	40
π_i	$\{p, q\}$	$\{q\}$	$\{p\}$	$\{\}$

q	$s(\cdot)$	$e(\cdot)$
$\{\tilde{q}_0\}$	$(\{\tilde{q}_0\}, \bot)$	$\bot$
$\tilde{Q}_1$	$\bot$	$\bot$
$\{\tilde{q}_f\}$	$\bot$	$\bot$
(i, j)	$(0, 0)$	

$\xrightarrow{\mathsf{adv_e}}$

q	$s(\cdot)$	$e(\cdot)$
$\{\tilde{q}_0\}$	$(\tilde{Q}_1, (10, 0))$	$\bot$
$\tilde{Q}_1$	$\bot$	$\bot$
$\{\tilde{q}_f\}$	$\bot$	$\bot$
(i, j)	$(0, 1)$	

$\xrightarrow{\mathsf{adv_e}}$

q	$s(\cdot)$	$e(\cdot)$
$\{\tilde{q}_0\}$	$(\{\tilde{q}_f\}, (20, 1))$	$\bot$
$\tilde{Q}_1$	$\bot$	$\bot$
$\{\tilde{q}_f\}$	$\bot$	$\bot$
(i, j)	$(0, 2)$	

$\xrightarrow{\mathsf{adv_s}}$

q	$s(\cdot)$	$e(\cdot)$
$\{\tilde{q}_0\}$	$(\{\tilde{q}_f\}, (20, 1))$	$\bot$
$\tilde{Q}_1$	$(\{\tilde{q}_f\}, (20, 1))$	$\bot$
$\{\tilde{q}_f\}$	$\bot$	10
(i, j)	$(1, 2)$	

Figure 3.11. The trace and windows for Example 3.3

sequence is derived from the time-stamps in the event stream and the match operator's intervals. A single update of the window data structure consists of advancing the window's start or end by one. We formally define the algorithms $\mathsf{adv_s}$ and $\mathsf{adv_e}$ that implement the window's start and end updates and their invariants later in this section.

Consider again the MDL regular expression $r = p^* \cdot q$ from Example 3.2 and the corresponding ϵ-NFA in Figure 3.9. In this example, we use the ϵ-NFA at the bottom in Figure 3.9. Recall that a deterministic state is a subset of the nondeterministic states. For instance, the initial deterministic state is $q_0 = \{\tilde{q}_0\}$. We analyze the trace and the sequence of window updates given in Figure 3.11. The window's end is advanced twice and then the window's start is advanced once. Figure 3.11 depicts the window's state after initialization ($i = 0$ and $j = 0$) and after each update. The reading heads h_1 and h_2 and the submonitors $\bar{m}_1$ and $\bar{m}_2$ are left out in Figure 3.11.

At the beginning ($i = 0$ and $j = 0$), the domain of s contains only $\{\tilde{q}_0\}$ because no other state could be reached from a previous time-point so far. We have $s(\{\tilde{q}_0\}) = (\{\tilde{q}_0\}, \bot)$ because of $\{\tilde{q}_0\} \rightsquigarrow_{(0,0)} \{\tilde{q}_0\}$ and because there is no time-point l such that $i = 0 \leq l < 0 = j$. The domain of e stays empty until the window's start advances because there is no time-point strictly before the initial time-point.

We have $s(\{\tilde{q}_0\}) = (\tilde{Q}_0, \bot)$, where $\tilde{Q}_0 \coloneqq \{\tilde{q}_0\}$. The ϵ?-closure of $\tilde{Q}_0$ at time-point 0 is $\tilde{Q}_0^{\epsilon?} = \{\tilde{q}_0, \tilde{q}_1, \tilde{q}_2, \tilde{q}_5, \tilde{q}_6\}$. In particular, it contains both $\tilde{q}_2$ and $\tilde{q}_6$ because p as well as q are satisfied at time-point 0 and thus the corresponding ϵ-transitions from $\tilde{q}_1$ to $\tilde{q}_2$ and from $\tilde{q}_5$ to $\tilde{q}_6$ can be taken. To update s, we perform a transition from $\tilde{Q}_0$ at time-point 0 by following $\star$-transitions from $\tilde{Q}_0^{\epsilon?}$. This way, we arrive at the next state $\{\tilde{q}_0, \tilde{q}_f\}$. Because $\{\tilde{q}_0, \tilde{q}_f\}^{\epsilon} = \{\tilde{q}_0, \tilde{q}_1, \tilde{q}_5, \tilde{q}_f\}$ contains the accepting state $\tilde{q}_f$, the state $\{\tilde{q}_0, \tilde{q}_f\}$ is accepting. Hence, we add time-point 0 (along with the corresponding time-stamp 10) to $s(\{\tilde{q}_0\})$.

We now have $s(\{\tilde{q}_0\}) = (\tilde{Q}_1, \bot)$, where $\tilde{Q}_1 \coloneqq \{\tilde{q}_0, \tilde{q}_f\}$. The ϵ?-closure of $\tilde{Q}_1$ at time-point 1 is $\tilde{Q}_1^{\epsilon?} = \{\tilde{q}_0, \tilde{q}_1, \tilde{q}_5, \tilde{q}_6, \tilde{q}_f\}$. In particular, it does not contain $\tilde{q}_2$ because p is not satisfied at time-point 1 and thus the corresponding ϵ-transition from $\tilde{q}_1$ to $\tilde{q}_2$ cannot be taken. On the other hand, $\tilde{Q}_1^{\epsilon?}$ contains $\tilde{q}_6$ because q is satisfied at time-point 1 and thus the corresponding ϵ-transition from $\tilde{q}_5$ to $\tilde{q}_6$ can be taken. To update the function s, we perform a transition from $\tilde{Q}_1$ at time-point 1 and arrive at the state $\{\tilde{q}_f\}$ because the only $\star$-transition from a state in $Q_1^{\epsilon?}$ is that from $\tilde{q}_6$ to $\tilde{q}_f$. Because $\{\tilde{q}_f\}^{\epsilon} = \{\tilde{q}_f\}$, the state $\{\tilde{q}_f\}$ is accepting and thus we update the time-stamp to 20 and time-point to 1 in $s(\{\tilde{q}_0\})$.

We now advance the window's start, i.e., update the window to $(i, j) = (1, 2)$. To this end, we set $e(\{\tilde{q}_f\}) = 10$ because from $s(\{\tilde{q}_0\}) = (\{\tilde{q}_f\}, (20, 1))$ we derive that the state $\{\tilde{q}_f\}$ is reached at the window's end 2 starting from the initial deterministic state $\{\tilde{q}_0\}$ at time-point 0. Next, we perform a transition (at time-point 0) from the only state $\{\tilde{q}_0\}$ in $\mathrm{dom}(s)$, which yields the state

$\tilde{Q}_1$. Since the maximum accepting time-point 1 is within the new window $(1,2)$, we keep it and arrive at $s(\tilde{Q}_1) = (\{\tilde{q}_f\}, (20,1))$. To compute $s(\{\tilde{q}_0\})$ for the initial deterministic state $\{\tilde{q}_0\}$, we perform two runs starting at time-point 1, one from $\{\tilde{q}_0\}$ and one from $\tilde{Q}_1$, until the two states in the runs collapse or the window's end is reached. In this example, we carry out a single step and the two states collapse into $\{\tilde{q}_f\}$ at time-point 2 (and the window's end is reached as well). Because time-point 1 in $s(\tilde{Q}_1)$ is strictly before the collapse at time-point 2, we cannot take it for $s(\{\tilde{q}_0\})$. However, since $\{\tilde{q}_0\}$ is accepting at time-point 1, we have $s(\{\tilde{q}_0\}) = (\{\}, (20,1))$. $\diamond$

Initialization and Update of the Window Data Structure The algorithms $\mathsf{init_w}$, $\mathsf{adv_s}$, and $\mathsf{adv_e}$ initializing and updating the window data structure are defined in Figure 3.12, 3.13, and 3.14. The function $\mathsf{adv_e}$ may return $\perp$ if the reading head h_2 over the trace or the submonitors $\bar{m}_2$ cannot compute their verdicts. The remaining functions are only evaluated if all reading heads and submonitors can compute all their necessary verdicts.

The window is initialized to time-points $(0,0)$ using $\mathsf{init_w}$ (Figure 3.12), which also establishes the invariant.

Lemma 3.4. *Let $\bar{m}$ be a sequence of submonitors at the initial time-point. Then the invariant $\mathsf{window}(\mathsf{init_w}(\mathsf{init_h}, \bar{m}))$ holds for the initial window.*

The window $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ can be updated to time-points $(i, j+1)$ using the function $\mathsf{adv_e}$ (Figure 3.14). The algorithm first reads the time-stamp τ_j and the input symbol b^j at the window's end (lines 2–9). Then $\mathsf{adv_e}$ updates the function e (lines 10–17). The updated domain of e is obtained by performing a transition at the window's end from all states in the original domain (line 13) and whenever two states q and q' collapse into a single state q_{new} after performing the transition, the function e associates q_{new} with the supremum of $e_{old}(q)$ and $e_{old}(q')$, using $e(q_{new})$ as an accumulator. This is because among the time-points $l < i$ and $l' < i$ such that $q_0 \rightsquigarrow_{(l,j+1)} q_{new}$, $\tau_l = e_{old}(q)$, and $q_0 \rightsquigarrow_{(l',j+1)} q_{new}$, $\tau_{l'} = e_{old}(q')$, we have to take $e(q_{new}) = \sup\{\tau_l, \tau_{l'}\}$. Next $\mathsf{adv_e}$ updates the function s (lines 18–24). Its domain does not change because the window's start i remains the same. However, for any state $q \in \mathrm{dom}(s)$ with $s(q) = (q', \mathit{tstp})$, a transition is performed on the state q' at the window's end (extending $q \rightsquigarrow_{(i,j)} q'$ to $q \rightsquigarrow_{(i,j+1)} q'_{new}$) and tstp is updated to (τ_j, j) if time-point j is accepting, i.e., $q \rightsquigarrow_{(i,j+1)} \checkmark$. Overall, $\mathsf{adv_e}$ preserves the window invariant.

Lemma 3.5. *Assume that the invariant $\mathsf{window}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ holds. Then the invariant holds after advancing the window's end, i.e., $\mathsf{window}(\mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2))$.*

To advance the window's start, we must advance the domain of s and then compute $s(q_0)$ at the new window's start. We first generalize the part of the window invariant characterizing s to take into account that $s(q_0)$ might not be computed yet. To this end, we define the invariant $\mathsf{svalid}(i, i', j, s)$ for s, which states that s is valid for the window (i', j), but the domain of s only contains states reached by running from a time-point before (and including) i. In particular, $\mathsf{window}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ implies $\mathsf{svalid}(i, i, j, s)$, but not vice-versa. Formally, $\mathsf{svalid}(i, i', j, s)$ holds if:

- $\mathrm{dom}(s)$ consists of all states q such that $q_0 \rightsquigarrow_{(l,i')} q$, for some $l \le i$;
- for any $q \in \mathrm{dom}(s)$: $s(q) = (q', \mathit{tstp})$, where $q \rightsquigarrow_{(i',j)} q'$ and $\mathit{tstp} = (\tau_l, l)$ for the maximum time-point l with $i' \le l < j$ and $q \rightsquigarrow_{(i',l+1)} \checkmark$, or $\mathit{tstp} = \perp$ if no such l exists.

```
function $\mathsf{init_w}(\mathsf{init_h}, \bar{m}) =$
    $s := (\lambda q.\, \bot)$;
    $s(q_0) := (q_0, \bot)$;
    $e := (\lambda q.\, \bot)$;
    return $(0, 0, s, e, \mathsf{init_h}, \bar{m}, \mathsf{init_h}, \bar{m})$;
```

Figure 3.12. Initialize state

```
function
  $\mathsf{adv_s}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2) =$
    $(h'_1, \tau_i, \_) := \mathsf{adv_h}(h_1)$;
    $(\bar{m}'_1, b^i) := \mathsf{adv_{ms}}(\bar{m}_1)$;
    $(q', tstp) = s(q_0)$;
    $e(q') = \tau_i$;
    $s := \mathsf{adv_d}(s, i, \tau_i, b^i)$;
    $q_{cur} := q_0$;
    $tstp_{cur} := \bot$;
    $s_{cur} := s$;
    $i_{cur} := i + 1$;
    $h_{cur} := h'_1$;
    $\bar{m}_{cur} := \bar{m}'_1$;
    while $i_{cur} < j$ and $q_{cur} \notin \mathrm{dom}(s_{cur})$
     do
        $(h'_{cur}, \tau_{i_{cur}}, \_) := \mathsf{adv_h}(h_{cur})$;
        $(\bar{m}'_{cur}, b^{i_{cur}}) := \mathsf{adv_{ms}}(\bar{m}_{cur})$;
        $q_{cur} := \delta(q_{cur}, b^{i_{cur}})$;
        if $q_{cur} \in F$ then
            $tstp_{cur} := (\tau_{i_{cur}}, i_{cur})$;
        $s_{cur} := \mathsf{adv_d}(s_{cur}, i_{cur}, \tau_{i_{cur}}, b^{i_{cur}})$;
        $i_{cur} := i_{cur} + 1$;
        $h_{cur} := h'_{cur}$;
        $\bar{m}_{cur} := \bar{m}'_{cur}$;
    if $q_{cur} \in \mathrm{dom}(s_{cur})$ then
        $(q', tstp) = s_{cur}(q_{cur})$;
        if $tstp \neq \bot$ then
            $s(q_0) := (q', tstp)$;
        else
            $s(q_0) := (q', tstp_{cur})$;
    else
        $s(q_0) := (q_{cur}, tstp_{cur})$;
    return $(i + 1, j, s, e, h'_1, \bar{m}'_1, h_2, \bar{m}_2)$;
```

Figure 3.13. Advance start

```
function
  $\mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2) =$
    switch $\mathsf{adv_h}(h_2)$ do
        case $(h', \tau, \_)$ do
            $h'_2 := h'$; $\tau_j := \tau$;
        case $\bot$ do return $\bot$;
    switch $\mathsf{adv_{ms}}(\bar{m}_2)$ do
        case $(\bar{m}', b)$ do
            $\bar{m}'_2 := \bar{m}'$; $b^j := b$;
        case $\bot$ do return $\bot$;
    $e_{old} := e$;
    $e := (\lambda q.\, \bot)$;
    for $q \in \mathrm{dom}(e_{old})$ do
        $q_{new} := \delta(q, b^j)$;
        if $q_{new} \in \mathrm{dom}(e)$ then
            $e(q_{new}) := e(q_{new}) \sqcup e_{old}(q)$;
        else
            $e(q_{new}) := e_{old}(q)$;
    for $q \in \mathrm{dom}(s)$ do
        $(q', tstp) = s(q)$;
        $q'_{new} := \delta(q', b^j)$;
        if $q'_{new} \in F$ then
            $s(q) := (q'_{new}, (\tau_j, j))$;
        else
            $s(q) := (q'_{new}, tstp)$;
    return $(i, j + 1, s, e, h_1, \bar{m}_1, h'_2, \bar{m}'_2)$;
```

Figure 3.14. Advance end

```
function $\mathsf{adv_d}(s, i, \tau_i, b^i) =$
    $s_{old} := s$;
    $s := (\lambda q.\, \bot)$;
    for $q \in \mathrm{dom}(s_{old})$ do
        $(q', tstp) = s_{old}(q)$;
        $q_{new} := \delta(q, b^i)$;
        if $tstp = (\tau_i, i)$ then
            $s(q_{new}) := (q', \bot)$;
        else
            $s(q_{new}) := (q', tstp)$;
    return $s$;
```

Figure 3.15. Advance dom(s)

The auxiliary function $\mathsf{adv_d}$ (Figure 3.15) updates s by advancing time-point i' in the invariant $\mathsf{svalid}(i, i', j, s)$. To do so, it performs a transition from every state $q \in \mathrm{dom}(s_{old})$ at time-point i' to the new state $q_{new} := \delta(q, b^{i'})$, where $b^{i'}$ is the input symbol at time-point i', and resets the latest accepting time-point to $\bot$ if it refers to time-point i', i.e., it is no longer valid at $i' + 1$. The function $\mathsf{adv_d}$ is used when advancing the domain of s from i to $i + 1$ and when computing $s(q_0)$. The correctness of the function $\mathsf{adv_d}$ is captured by the following lemma.

Lemma 3.6. *Assume that the invariant* $\mathsf{svalid}(i, i', j, s)$ *holds and* $i' < j$. *Furthermore, let* $(\tau_{i'}, b^{i'})$ *be the time-stamp and input symbol for the time-point* i'. *Then the invariant holds for the updated function* s, *i.e.,*

$$\mathsf{svalid}(i, i' + 1, j, \mathsf{adv_d}(s, i', \tau_{i'}, b^{i'})).$$

The window $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ with $i < j$ can be updated to the time-points $(i + 1, j)$ using the function $\mathsf{adv_s}$ (Figure 3.13). This algorithm first reads the time-stamp τ_i and the input symbol b^i at the window's end (lines 2–3). Then $\mathsf{adv_s}$ updates e (lines 4–5) to account for the run $q_0 \rightsquigarrow_{(i,j)} q'$, where the state q' is obtained from the function s (line 4), which always contains the initial state q_0 in its domain.

Next $\mathsf{adv_s}$ updates s (lines 6–30). First, the domain of s is advanced by $\mathsf{adv_d}$ (line 6). This way, the invariant on s becomes $\mathsf{svalid}(i, i + 1, j, s)$. However, $\mathsf{svalid}(i + 1, i + 1, j, s)$ is required to establish $\mathsf{window}(i + 1, j, s, e, h'_1, \bar{m}'_1, h_2, \bar{m}_2)$. Thus, it remains to compute the value of $s(q_0)$ and update s accordingly. To this end, $\mathsf{adv_s}$ performs runs from q_0 as well as from all states in $\mathrm{dom}(s)$ until the current state q_{cur} in the run from q_0 collapses with the current state of the run from a state $q \in \mathrm{dom}(s)$ or the window's end is reached (lines 7–22). The run from q_0 is simulated by updating the current state q_{cur} (initialized to q_0 on line 7). The runs from all states in $\mathrm{dom}(s)$ are simulated by updating a copy s_{cur} of the function s to $\mathsf{adv_d}(s_{cur}, i_{cur}, \tau_{i_{cur}}, b^{i_{cur}})$ at the current time-point i_{cur} of the simulation. This way, s_{cur} satisfies $\mathsf{svalid}(i, i_{cur}, j, s_{cur})$. In particular, the function s_{cur} contains the state reached at the window's end j and the latest accepting time-point on (i_{cur}, j) for all states in its domain. To account for accepting time-points on $(i + 1, i_{cur})$, the algorithm also tracks the maximum accepting time-point l (represented by the pair $tstp_{cur} = (\tau_l, l) \in \mathbb{T} \times \mathbb{N}$) such that $i + 1 \leq l < i_{cur}$ and $q_0 \rightsquigarrow_{(i+1,l+1)} \checkmark$.

After the loop on lines 13–22 terminates, $\mathsf{adv_s}$ proceeds branching based on whether the current state q_{cur} collapsed with the current state of the run from a state $q \in \mathrm{dom}(s)$.

(1) If yes, then we have $q_0 \rightsquigarrow_{(i+1,i_{cur})} q_{cur}$ and also $q \rightsquigarrow_{(i+1,i_{cur})} q_{cur}$. Because the states are deterministic, the two runs from q_0 and q continue the same after i_{cur}. Hence, the run from q_{cur} at i_{cur} reaches the state q' from $s_{cur}(q_{cur}) = (q', tstp)$. If $tstp \neq \bot$, then $tstp$ represents the latest accepting time-point following q_{cur} at i_{cur} which is also the latest accepting time-point time-point pair following q_0 at $i + 1$. On the other hand, if $tstp = \bot$, then there is no accepting time-point following q_{cur} at i_{cur}. Hence, the latest accepting time-point following q_0 at $i + 1$ is $tstp_{cur}$.

(2) If the current state q_{cur} did not collapse with the current state of the run from any state $q \in \mathrm{dom}(s)$, then the window's end must have been reached (due to the loop condition on line 13). Then we have $i_{cur} = j$ and thus $s(q_0) = (q_{cur}, tstp_{cur})$.

Overall, $\mathsf{adv_s}$ preserves the window invariant.

Lemma 3.7. *Assume that the invariant* $\mathsf{window}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ *holds and* $i < j$. *Then the invariant holds after advancing the window's start, i.e.,* $\mathsf{window}(\mathsf{adv_s}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2))$.

```
input:  A formula $\langle r|_I$, a window $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$, a function $\mathsf{adv_h}$ to read a
        time-point and advance the reading head, $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$, a
        function $\mathsf{read_h}$ to only read the time-stamp of a time-point using a reading head,
        $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$, the function $\mathsf{adv_m}$ computing verdicts for submonitors in
        $\bar{m}_1$ and $\bar{m}_2$, $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$.
output: A tuple $(m', \tau, \beta)$, where $m'$ is the next state of our multi-head monitor,
        $m' \in \mathbb{M}$, and $(\tau, \beta)$ is the Boolean verdict for the next time-point,
        $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$, or $\bot$ if no verdict can be computed.
function $\mathsf{doMatch_P}((\langle r|_I, (i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)))$ =
    switch $\mathsf{read_h}(h_2)$ do
        case $(\tau', \_)$ do $\tau := \tau'$;
        case $\bot$ do return $\bot$;
    switch $\mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ do
        case $\bot$ do return $\bot$;
        otherwise do $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2) := \mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$;
    while $i < j$ and $\mathsf{memL}(\mathsf{read_h}(h_1), \tau, I)$ do
        $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2) := \mathsf{adv_s}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$;
    $\beta := (\exists q \in \mathrm{dom}(e) \cap F.\ \mathsf{memR}(e(q), \tau, I))$;
    return $((\langle r|_I, (i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)), \tau, \beta)$;
```

Figure 3.16. The function $\mathsf{eval_P}$ computing verdicts for $\langle r|_I$.

3.2.3 MDL Monitor

Past Match MDL Temporal Operator A monitor $(\langle r|_I, w)$ for a past temporal match formula $\langle r|_I$ consists of a window data structure w for the MDL regular expression r. To compute a Boolean verdict at time-point j for $\langle r|_I$, we check if there exists a match of the MDL regular expression r from a past time-point l, $l \leq j$, until time-point j such that $\mathsf{mem}(\tau_l, \tau_j, I)$, i.e., $\mathsf{memL}(\tau_l, \tau_j, I)$ and $\mathsf{memR}(\tau_l, \tau_j, I)$.

Our multi-head monitor maintains a window w on (i, j) such that the invariant $\mathsf{window}(w)$ holds and $\mathsf{memL}(\tau_l, \tau_j, I)$, for all $l < i$, i.e., all time-points l strictly before the window's start i satisfy the lower bound from the interval condition. To compute a Boolean verdict at a time-point j with $\tau := \tau_j$, the monitor advances the window's end j to $j' := j + 1$ (so that matches until and including j are considered) and then repeatedly advances the window's start so that the time-points l that are strictly before the window's start ($l < i$) are exactly those that satisfy the lower bound from the interval condition, i.e., the monitor advances the window's start i to the maximum i, $i \leq j'$, such that $\mathsf{memL}(\tau_l, \tau, I)$, for all $l < i$.

Then we seek to find a past match from a time-point l strictly before the window's start i, $l < i$, i.e., satisfying the lower bound from the interval condition, that also satisfies the upper bound from the interval condition, i.e., $\mathsf{memR}(\tau_l, \tau, I)$. Using $\mathsf{window}(i, j', s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$, this amounts to checking whether there exists some accepting $q \in \mathrm{dom}(e)$ such that $\mathsf{memR}(e(q), \tau, I)$. The maximality of i implies that no candidate time-point for the beginning of a past match is missed. Formally, we define the algorithm for past match operator $\langle r|_I$ in Figure 3.16.

Future Match MDL Temporal Operator A monitor $(|r\rangle_I, w)$ for a future match formula $|r\rangle_I$ consists of a window data structure w for the MDL regular expression r. To compute a

```
input:  A formula $|r\rangle_I$, a window $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$, a function $\mathsf{adv_h}$ to read a
        time-point and advance the reading head, $\mathsf{adv_h} : \mathbb{H} \to (\mathbb{H} \times \mathbb{T} \times \mathcal{P}(\Sigma)) \cup \{\bot\}$, a
        function $\mathsf{read_h}$ to only read the time-stamp of a time-point using a reading head,
        $\mathsf{read_h} : \mathbb{H} \to \mathbb{T} \cup \{\bot\}$, the function $\mathsf{adv_m}$ computing verdicts for submonitors in
        $\bar{m}_1$ and $\bar{m}_2$, $\mathsf{adv_m} : \mathbb{M} \to (\mathbb{M} \times \mathbb{T} \times \mathbb{B}) \cup \{\bot\}$.
output: A tuple $(m', \tau, \beta)$, where $m'$ is the next state of our multi-head monitor,
        $m' \in \mathbb{M}$, and $(\tau, \beta)$ is the Boolean verdict for the next time-point,
        $(\tau, \beta) \in \mathbb{T} \times \mathbb{B}$, or $\bot$ if no verdict can be computed.
function $\mathsf{loopCondMatch_F}(I, \tau, h_2)$ =
    switch $\mathsf{read_h}(h_2)$ do
        case $(\tau', \_)$ do return $\mathsf{memR}(\tau, \tau', I)$;
        case $\bot$ do return $\mathsf{ff}$;
function $\mathsf{doMatch_F}((|r\rangle_I, (i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)))$ =
    switch $\mathsf{read_h}(h_1)$ do
        case $(\tau', \_)$ do $\tau := \tau'$;
        case $\bot$ do return $\bot$;
    while $\mathsf{loopCondMatch_F}(I, \tau, h_2)$ do
        switch $\mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ do
            case $\bot$ do return $\bot$;
            otherwise do $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2) := \mathsf{adv_e}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$;
    if $\mathsf{read_h}(h_2) = \bot$ then return $\bot$;
    $(q', \mathit{tstp}) = s(q_0)$;
    switch $\mathit{tstp}$ do
        case $(\tau_{i'}, i')$ do return $\beta := \mathsf{memL}(\tau, \tau_{i'}, I)$;
        case $\bot$ do $\beta := \mathsf{ff}$;
    return $((|r\rangle_I, \mathsf{adv_s}(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)), \tau, \beta)$;
```

Figure 3.17. The function $\mathsf{eval_F}$ computing verdicts for $|r\rangle_I$.

Boolean verdict at time-point i for $|r\rangle_I$, we check if there exists a match of the MDL regular expression r from time-point i until a future time-point l, $l \geq i$, such that $\mathsf{mem}(\tau_i, \tau_l, I)$, i.e., $\mathsf{memL}(\tau_i, \tau_l, I)$ and $\mathsf{memR}(\tau_i, \tau_l, I)$.

Our multi-head monitor maintains a window w on (i, j) such that the invariant $\mathsf{window}(w)$ holds and $\mathsf{memR}(\tau_i, \tau_l, I)$, for all $i \leq l < j$, i.e., all time-points strictly before the window's end j satisfy the upper bound from the interval condition. To compute a Boolean verdict at time-point i, the monitor repeatedly advances the window's end so that the time-points l that are strictly before the window's end ($i \leq l < j$) are exactly those that satisfy the upper bound from the interval condition, i.e., the monitor advances the window's end j to the maximum j such that $\mathsf{memR}(\tau_i, \tau_l, I)$, for all $i \leq l < j$.

Then the invariant $\mathsf{window}(i, j', s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ implies that $q_0 \in \mathrm{dom}(s)$ and that the latest accepting time-point within the window is stored in $s(q_0) = (q', \mathit{tstp})$. Note that the latest acceping time-point within the window satisfies the upper bound from the interval condition (according to the condition for advacing window's end). It remains to check $\mathit{tstp} \neq \bot$ (i.e., if an accepting time-point within the window exists) and if yes, whether the time-stamp $\tau_{i'}$ of the latest accepting time-point i' within the window satisfies the lower bound from the interval condition, i.e., $\mathsf{memL}(\tau_i, \tau_{i'}, I)$.

```
function $\mathsf{init_m}(\phi)$ =
    switch $\phi$ do
        case $\langle r|_I$ do
            $\{\phi_1, \dots, \phi_k\} \coloneqq \mathsf{SF}(r)$;
            $\bar{m} \coloneqq (\mathsf{init_m}(\phi_i))_{i=1}^k$;
            return $(\langle r|_I, \mathsf{init_w}(\mathsf{init_h}, \bar{m}))$;
        case $|r\rangle_I$ do
            $\{\phi_1, \dots, \phi_k\} \coloneqq \mathsf{SF}(r)$;
            $\bar{m} \coloneqq (\mathsf{init_m}(\phi_i))_{i=1}^k$;
            return $(|r\rangle_I, \mathsf{init_w}(\mathsf{init_h}, \bar{m}))$;
        ...
function $\mathsf{adv_m}(m)$ =
    switch $m$ do
        case $(\langle r|_I, w)$ do return $\mathsf{doMatch_P}((\langle r|_I, w))$;
        case $(|r\rangle_I, w)$ do return $\mathsf{doMatch_F}((|r\rangle_I, w))$;
        ...
```

Figure 3.18. Extension of $\mathsf{init_m}(\phi)$ and $\mathsf{adv_m}(\rho_{<\ell})$ to MDL.

Formally, we define the algorithm for future match operator $|r\rangle_I$ in Figure 3.17.

Integration into Multi-Head Monitor We integrate the window data structure and the functions $\mathsf{doMatch_P}$ and $\mathsf{doMatch_F}$, defined in Figure 3.16 and Figure 3.17, respectively, into our multi-head monitor by extending the set of our multi-head monitor's states, defined at the beginning of Section 3.2.1, with the cases of temporal match operators:

$$\begin{aligned}\mathbb{M} = {} & (\{\langle r|_I\} \times \{w \mid w \text{ is a window data structure}\}) \cup \\ & (\{|r\rangle_I\} \times \{w \mid w \text{ is a window data structure}\}) \cup \dots,\end{aligned}$$

and by extending the functions $\mathsf{init_m}$ and $\mathsf{adv_m}$ with the cases of temporal match operators as shown in Figure 3.18. In the function $\mathsf{init_m}(\phi)$, $\bar{m} = (\mathsf{init_m}(\phi_i))_{i=1}^k$ is a sequence of initial states of the submonitors for the direct subformulas $\mathsf{SF}(r) = \{\phi_1, \dots, \phi_k\}$, $k \in \mathbb{N}$, of the MDL regular expression r.

3.2.4 Correctness

The soundness and completeness of our multi-head monitor follows by induction on well-formed MDL formulas using an invariant $\mathsf{wf_m} : \{\phi \mid \phi \text{ is a well-formed MDL formula}\} \times \mathbb{N} \times \mathbb{M} \to \mathbb{B}$ characterizing states of the monitor. We have $\mathsf{wf_m}(\phi, i, m) = \mathsf{tt}$ if m is a valid state of our monitor for the formula ϕ at time-point i. The definition of the invariant $\mathsf{wf_m}$ formalizes the description of our monitor presented so far. In Section 2.2, we defined MDL as an extension of MTL with MDL regular expressions r and the corresponding regular expression match formulas $\langle r|_I$ and $|r\rangle_I$. Hence, the correctness of our multi-head monitor on MTL formulas follows from its correctness on MDL formulas. The initial state of the monitor $\mathsf{init_m}(\phi)$ satisfies the invariant, $\mathsf{wf_m}(\phi, 0, \mathsf{init_m}(\phi))$, for every formula ϕ, and $\mathsf{adv_m}$ preserves the invariant $\mathsf{wf_m}$ while advancing the time-point and computing Boolean verdicts.

$$\begin{aligned}
&\mathsf{time}(p) && && = \ell \cdot \mathsf{c_t}\\
&\mathsf{time}(\neg\phi) && && = \mathsf{time}(\phi) + \ell \cdot \mathsf{c_t}\\
&\mathsf{time}(\phi \vee \psi) && && = \mathsf{time}(\phi) + \mathsf{time}(\psi) + \ell \cdot \mathsf{c_t}\\
&\mathsf{time}(\bullet_I\ \phi) && = \mathsf{time}(\bigcirc_I\ \phi) && = \mathsf{time}(\phi) + \ell \cdot \mathsf{c_t}\\
&\mathsf{time}(\phi_1\ \mathsf{S}_I\ \phi_2) && = \mathsf{time}(\phi_1\ \mathsf{U}_I\ \phi_2) && = \mathsf{time}(\phi) + \mathsf{time}(\psi) + \ell \cdot \mathsf{c_t}\\
&\mathsf{time}(\langle r|_I) && = \mathsf{time}(|r\rangle_I) && = (3 + 8^{|r|}) \cdot 64^{|r|} \cdot \ell \cdot \mathsf{c_t} + (2 + 8^{|r|}) \cdot \textstyle\sum_{\phi \in \mathsf{SF}(r)} \mathsf{time}(\phi)
\end{aligned}$$

Figure 3.19. The recursive function $\mathsf{time}(\phi)$.

$$\begin{aligned}
&\mathsf{space}(p) && && = \mathsf{c_s}\\
&\mathsf{space}(\neg\phi) && && = \mathsf{space}(\phi) + \mathsf{c_s}\\
&\mathsf{space}(\phi \vee \psi) && && = \mathsf{space}(\phi) + \mathsf{space}(\psi) + \mathsf{c_s}\\
&\mathsf{space}(\bullet_I\ \phi) && = \mathsf{space}(\bigcirc_I\ \phi) && = \mathsf{space}(\phi) + \mathsf{c_s}\\
&\mathsf{space}(\phi_1\ \mathsf{S}_I\ \phi_2) && = \mathsf{space}(\phi_1\ \mathsf{U}_I\ \phi_2) && = \mathsf{space}(\phi) + \mathsf{space}(\psi) + \mathsf{c_s}\\
&\mathsf{space}(\langle r|_I) && = \mathsf{space}(|r\rangle_I) && = (1 + 64^{|r|}) \cdot \mathsf{c_s} + 3 \cdot \textstyle\sum_{\phi \in \mathsf{SF}(r)} \mathsf{space}(\phi)
\end{aligned}$$

Figure 3.20. The recursive function $\mathsf{space}(\phi)$.

Soundness The soundness theorem states that all verdicts produced by the our multi-head monitor are correct according to the semantics of MDL (that also includes the semantics of MTL).

Theorem 3.8 ([63, vydra_sound]). *Let ϕ be a well-formed bounded-future MDL formula, $n \in \mathbb{N}$, and m our multi-head monitor's state, $m \in \mathbb{M}$, obtained by applying $\mathsf{adv_m}$ to $\mathsf{init_m}(\phi)$ n-times. Let $\mathsf{adv_m}(m) = (m', (\tau, \beta))$. Then, (i) $\tau = \tau_n$ and (ii) β if and only if $(\rho, n) \models \phi$.*

The soundness theorem has been formally proved using the Isabelle/HOL proof assistant [63].

Completeness Because a trivial monitoring algorithm that never produces a verdict (e.g., $\mathsf{adv_m}(m) = \bot$, for every $m \in \mathbb{M}$) is also sound, we have to guarantee that our monitoring algorithm actually produces some verdicts. We capture this in a completeness theorem. The completeness theorem states that the monitor outputs a verdict for a time-point if the reading heads over the trace could read sufficiently many time-points afterwards. Quantitatively, the monitor is guaranteed to compute a verdict for time-point n, $n \in \mathbb{N}$, if the reading heads over the trace could read a (monotone) sequence of time-stamps $\overline{\tau}$ and $n < \mathsf{prog}(\phi, \overline{\tau})$.

Theorem 3.9 ([63, vydra_complete]). *Let ϕ be a well-formed bounded-future MDL formula and let $n \in \mathbb{N}$ be a time-point. Suppose that applying $\mathsf{adv_h}$ to $\mathsf{init_h}$ iteratively yields a sequence of time-stamps $\overline{\tau}$. If $n < \mathsf{prog}(\phi, \overline{\tau})$, then $\mathsf{adv_m}$ can be successfully applied to $\mathsf{init_m}(\phi)$ n-times and yields a monitor's state m, $m \in \mathbb{M}$, such that $\mathsf{adv_m}(m) = (m', (\tau, \beta))$, for some $m' \in \mathbb{M}$, $\tau \in \mathbb{T}$, and $\beta \in \mathbb{B}$.*

The completeness theorem has been formally proved using the Isabelle/HOL proof assistant [63].

3.2.5 Complexity Analysis

To analyze the time and space complexity of our multi-head monitor, we assume that time-points (natural numbers whose magnitude is at most the trace length ℓ) and time-stamps (elements

$$\begin{array}{lll}
||p|| & & = 1 \\
||\neg\phi|| & & = ||\phi|| + 1 \\
||\phi \vee \psi|| & & = \max\{||\phi||, ||\psi||\} + 1 \\
||\bullet_I\ \phi|| & = ||\bigcirc_I\ \phi|| & = ||\phi|| + 1 \\
||\phi_1\ \mathsf{S}_I\ \phi_2|| & = ||\phi_1\ \mathsf{U}_I\ \phi_2|| & = \max\{||\phi||, ||\psi||\} + 1 \\
||\langle r|_I|| & = |||r\rangle_I|| & = \max_{\phi\in\mathsf{SF}(r)}\{||\phi||\} + 10\cdot|r| + 1
\end{array}$$

Figure 3.21. The recursive function $||\phi||$.

$$\begin{array}{lll}
|p| & & = 1 \\
|\neg\phi| & & = |\phi| + 1 \\
|\phi \vee \psi| & & = |\phi| + |\psi| + 1 \\
|\bullet_I\ \phi| & = |\bigcirc_I\ \phi| & = |\phi| + 1 \\
|\phi_1\ \mathsf{S}_I\ \phi_2| & = |\phi_1\ \mathsf{U}_I\ \phi_2| & = |\phi| + |\psi| + 1 \\
|\langle r|_I| & = ||r\rangle_I| & = (\sum_{\phi\in\mathsf{SF}(r)} |\phi|) + |r| + 1
\end{array}$$

Figure 3.22. The recursive function $|\phi|$.

$\tau \in \mathbb{T}$) can be manipulated in constant time and stored in constant space. Moreover, we assume that the set of atomic propositions Σ is fixed and thus its size $|\Sigma|$ is constant.

Let a trace $\rho_{<\ell}$ of a trace length ℓ be fixed. Because the time complexity of computing individual verdicts using our multi-head monitor may vary, we analyze the total time complexity of computing all verdicts on the fixed trace $\rho_{<\ell}$ of trace length ℓ, i.e., evaluating $\mathsf{adv_m}$ starting from $\mathsf{init_m}(\phi)$ until we get $\bot$, and report the *amortized* time complexity per time-point (obtained by dividing the total time complexity on the trace $\rho_{<\ell}$ by ℓ). We define the recursive functions $\mathsf{time}(\phi)$ and $\mathsf{space}(\phi)$ on MDL formulas ϕ (Figure 3.19 and Figure 3.20) that bound the total time and space complexity, respectively, of computing all verdicts for ϕ on the trace $\rho_{<\ell}$. Then we derive an upper bound on the functions $\mathsf{time}(\phi)$ and $\mathsf{space}(\phi)$ to get a more intuitive upper bound on the complexity of our monitor in terms of the measure $||\phi||$ on MDL formulas ϕ (Figure 3.21). Finally, we define the measure $|\phi|$ (Figure 3.22) that simply counts the number of subformulas and subexpressions of MDL regular expressions in an MDL formula ϕ. We use the measure $|\phi|$ to derive an altenrative upper bound on the complexity of our monitor. The relationships between time, space, and the measures $||\cdot||$ and $|\cdot|$ are expressed by the following lemma, where $\mathsf{c_t}$ and $\mathsf{c_s}$ are sufficiently large constants independent of the trace $\rho_{<\ell}$, its length ℓ, and the formula ϕ.

Lemma 3.10. *The following upper bounds hold for an MTL formula ϕ:*

$$\begin{array}{ll}
\mathsf{time}(\phi) & \le ||\phi|| \cdot \ell \cdot \mathsf{c_t} \\
\mathsf{space}(\phi) & \le ||\phi|| \cdot \mathsf{c_s}.
\end{array}$$

The following upper bounds hold for an MDL formula ϕ:

$$\begin{array}{ll}
\mathsf{time}(\phi) & \le 3^{||\phi||} \cdot \ell \cdot \mathsf{c_t} \\
\mathsf{space}(\phi) & \le 3^{||\phi||} \cdot \mathsf{c_s} \\
||\phi|| & \le 10 \cdot |\phi|.
\end{array}$$

MTL Monitor's Complexity We observe that the state of our multi-head monitor $m \in \mathbb{M}$ for an MTL formula ϕ consists of $|\phi|$ submonitors for the subformulas of ϕ and its space complexity is at most $|\phi| \cdot \mathsf{c_s}$, where $\mathsf{c_s}$ is a sufficiently large constant. We can prove this observation for the initial state of our multi-head monitor $\mathsf{init_m}(\phi)$ by induction over ϕ and for the state $\mathsf{adv_m}(m)$ after computing a verdict by induction over m.

The time complexity of computing $\mathsf{init_m}(\phi)$ is at most $|\phi| \cdot \mathsf{c_t}$, where $\mathsf{c_t}$ is a sufficiently large constant, because $\mathsf{init_m}(\phi)$ is a simple recursive function on ϕ. On the trace $\rho_{<\ell}$ of trace length ℓ, the function $\mathsf{adv_m}$ can be evaluated at most ℓ-times before it yields $\bot$. This can be proved by induction over ℓ. To bound the time complexity of the loops in the functions $\mathsf{doSince}(\mathsf{adv_m}, m)$ and $\mathsf{doUntil}(\mathsf{adv_m}, m)$, we observe that $\mathsf{adv_m}(m_2)$ is evaluated within each loop iteration. Hence, the total number of loop iterations is bounded by ℓ. It follows that the total time complexity of evaluating $\mathsf{adv_m}$ starting from $\mathsf{init_m}(\phi)$ until we get $\bot$ on the trace $\rho_{<\ell}$ is bounded by $\mathsf{time}(\phi)$. Given an MTL formula ϕ, $\mathsf{time}(\phi)$ can be bounded by $|\phi| \cdot \ell \cdot \mathsf{c_t}$.

Finally, we summarize the amortized time and space complexity of our multi-head MTL monitor in the following theorem.

Theorem 3.11. *The amortized time complexity of computing the verdicts for an MTL formula ϕ is at most $|\phi| \cdot \mathsf{c_t}$. The space complexity of representing the multi-head monitor's state and computing the verdicts for an MTL formula ϕ is at most $|\phi| \cdot \mathsf{c_s}$.*

MDL Monitor's Complexity Because an MDL monitor extends an MTL monitor with the window data structure for the temporal match operators, we focus on analyzing the time and space complexity of the window data structure.

Given an MDL regular expression r, we define $|r|$ to be the number of all subexpressions of r (including r, but not regular expressions occurring in direct subformulas of r). By induction on MDL regular expressions r, we observe that the number of states in the ϵ-NFA $\mathcal{A}_\mathrm{N}$ computed by the function T (Figure 3.8) for r is at most $3 \cdot |r|$ (every subexpression adds at most 3 states to the ϵ-NFA $\mathcal{A}_\mathrm{N}$). Because the set of DFA's states Q consists of all subsets of the set of NFA's states, we derive $|Q| \le 2^{3 \cdot |r|} = 8^{|r|}$. We also observe that a window data structure consists of a constant number of time-points and reading heads over the trace, taking up at most $\mathsf{c_s}$ space, functions s, e associating each $\mathcal{A}_\mathrm{D}$'s state with at most one $\mathcal{A}_\mathrm{D}$'s state and a constant number of time-points and time-stamps, taking up at most $|Q|^2 \cdot \mathsf{c_s}$ space, and 2 submonitors for each direct subformula of r, taking up at most $2 \cdot \sum_{\phi \in \mathsf{SF}(r)} \mathsf{space}(\phi)$ space. When advacing the window's start, one more collection of submonitors for the direct subformulas of r is created (Line 12 in Figure 3.13). Hence, using $|Q| \le 8^{|r|}$, the space complexity of the window data structure and the functions advacing the window's start and end is at most $(1 + 64^{|r|}) \cdot \mathsf{c_s} + 3 \cdot \sum_{\phi \in \mathsf{SF}(r)} \mathsf{space}(\phi)$.

Next we analyze the time complexity of initializing and updating the window data structure for an MDL regular expression r. The window data structure can be initialized in time $|Q| \cdot \mathsf{c_t}$. To advance the window's end, the function $\mathsf{adv_e}$ has to update, for every $\mathcal{A}_\mathrm{D}$'s state, a couple of $\mathcal{A}_\mathrm{D}$'s states, time-points, and time-stamps, and evaluate the submonitors $\mathsf{adv_{ms}}(\bar{m}_1)$, with one submonitor for every $\phi \in \mathsf{SF}(r)$. Hence, the total time complexity of advancing the window's end on the trace $\rho_{<\ell}$ is at most $|Q|^2 \cdot \ell \cdot \mathsf{c_t} + \sum_{\phi \in \mathsf{SF}(r)} \mathsf{time}(\phi)$.

We now analyze the time complexity of advancing the window's start. Let c_{while} denote the total number of iterations of the loop in Lines 13–22 in Figure 3.13. Because an iteration of the loop for a specific value of i_{cur} is only performed if $q_{cur} \notin \mathrm{dom}(s_{cur})$ and we have $q_{cur} \in \mathrm{dom}(s_{cur})$ next time a loop iteration for i_{cur} is performed (in a subsequent evaluation of $\mathsf{adv_s}$), a loop iteration for a specific value of i_{cur} is performed at most $|Q|$-times in total. Because $i_{cur} < j$ and $j \le \ell$, the total number of loop iterations is at most $c_{\mathsf{while}} \le \ell \cdot |Q|$.

Hence, the total time complexity of advancing the window's start on the trace $\rho_{<\ell}$ is at most $(1+|Q|)\cdot|Q|^2\cdot\ell\cdot\mathsf{c_t}+(1+|Q|)\cdot\sum_{\phi\in\mathsf{SF}(r)}\mathsf{time}(\phi)$.

We observe that evaluating $\mathsf{doMatch_P}$ or $\mathsf{doMatch_F}$ until we get $\perp$ consists of advancing the window's start and end according to the time-stamps in the trace and computing the verdicts based on the window data structure. Overall, the total time complexity of evaluating $\mathsf{doMatch_P}$ or $\mathsf{doMatch_F}$ until we get $\perp$ is at most

$$|Q|^2\cdot\ell\cdot\mathsf{c_t}+\sum_{\phi\in\mathsf{SF}(r)}\mathsf{time}(\phi)+(1+|Q|)\cdot|Q|^2\cdot\ell\cdot\mathsf{c_t}+(1+|Q|)\cdot\sum_{\phi\in\mathsf{SF}(r)}\mathsf{time}(\phi)+|Q|\cdot\ell\cdot\mathsf{c_t}.$$

Using $|Q|\leq 8^{|r|}$, this is at most

$$(3+8^{|r|})\cdot 64^{|r|}\cdot\ell\cdot\mathsf{c_t}+(2+8^{|r|})\cdot\sum_{\phi\in\mathsf{SF}(r)}\mathsf{time}(\phi).$$

Given an MDL formula ϕ, the time complexity of computing all verdicts on the trace $\rho_{<\ell}$ is at most $\mathsf{time}(\phi)$ which can be further bounded by $3^{||\phi||}\cdot\ell\cdot\mathsf{c_t}$. The space complexity of computing all verdicts on the trace $\rho_{<\ell}$ is at most $\mathsf{space}(\phi)$ which can be further bounded by $3^{||\phi||}\cdot\mathsf{c_s}$. Finally, we summarize the amortized time and space complexity of our multi-head MDL monitor in the following theorem.

Theorem 3.12. *The amortized time complexity of computing the verdicts for an MDL formula ϕ is at most $3^{||\phi||}\cdot\mathsf{c_t}$ which is further at most $3^{10\cdot|\phi|}\cdot\mathsf{c_t}$. The space complexity of representing the multi-head monitor's state and computing the verdicts for the formula ϕ is at most $3^{||\phi||}\cdot\mathsf{c_s}$ which is further at most $3^{10\cdot|\phi|}\cdot\mathsf{c_s}$.*

We remark that the upper bounds in Theorem 3.12 are worst-case upper bounds. Our empirical evaluation (Section 3.3) confirms that our multi-head monitor can handle 100 000 time-points in less than one second on average.

3.3 Implementation and Evaluation

We have implemented our multi-head monitor in a tool called Hydra [59], consisting of a few thousand lines of C++ code. Our implementation mirrors the structure of the multi-head monitor presented here and consists of C++ classes for monitoring atomic propositions, Boolean operators, MTL temporal operators, and MDL temporal match operators.

In addition, we have exported verified OCaml code from our Isabelle/HOL formalization and augmented this verified code with unverified OCaml and C code for parsing the formula and trace file and outputting verdicts. We call the resulting tool Vydra [59]. We have used Vydra to successfully test the correctness of Hydra on thousands of pseudo-random formulas and traces.

We empirically evaluate the time and space complexity of Hydra and Vydra by answering the following four research questions:

RQ1: *How do Hydra and Vydra scale with respect to the magnitude of time constaints?*

RQ2: *How do Hydra and Vydra scale with respect to the number of subformulas and subexpressions in a formula?*

RQ3: *How do Hydra and Vydra perform on inputs that trigger worst-case space complexity for online monitors?*

RQ4: *How do Hydra and Vydra perform compared to the state-of-the-art monitoring tools?*

To answer these research questions, we conduct average-case and worst-case experiments measuring the time and space usage of Hydra, Vydra, and state-of-the-art tools supporting MTL temporal operators and having an available implementation: Aerial [6], MonPoly [10], VeriMon [8], and Reelay [75]. Aerial offers several modes of operation and several representations of Boolean expressions. We use the Global mode of Aerial because this mode yields the best space complexity guarantees on formulas with large interval bounds and traces with high event rate and the alternative Local mode is "only marginally better" than the Global mode in the empirical evaluation [6]. We use the "direct representation" (expr) of Boolean expressions in Aerial because expr is the fastest representation in the empirical evaluation [6]. We do not include r2u2 [54] in our empirical evaluation because its implementation computes incorrect verdicts for some formulas, e.g., for the formula $\blacklozenge_{[2,2]}\, a_0$ on the trace $(\{a_0\}, \emptyset, \emptyset)$, where we omit the time-stamps because r2u2 requires time-stamps to be equal to time-points. RQ4 is commonly addressed by both average-case and worst-case experiments. Our empirical evaluation can be reproduced using a publicly available artifact [60].

Experimental Setup We run our experiments on an Intel Core i5-4200U CPU computer with 8 GB RAM. We measure the tools' total execution time and maximal writeable memory usage with a custom tool that performs two repetitions of each run. The tool measures the total execution time in one repetition and reads the `/proc` directory in a loop to determine the maximal writeable memory usage in another repetition. Having thoroughly tested the tools' outputs separately, we discard any output during the experiments to minimize the impact of IO on performance. Each unfilled data point in our plots shows the median for the tool invocations with the same input parameters. Each filled data point shows the median over a collection of the tool's data points with the same x-coordinate. We include trend lines over the filled data points in all plots. The space usage is not plotted if an execution times out because the tool could have used more space if it did not time out. Note that the y-axis is always plotted in the logarithmic scale. Consequently, an exponential growth of a quantity looks linear and a polynomial growth looks logarithmic in the plots.

Optimizations Hydra implements some optimizations that were omitted from our presentation. Because the function e in the window data structure $(i, j, s, e, h_1, \bar{m}_1, h_2, \bar{m}_2)$ is not needed to evaluate the future match operator $|r\rangle_I$, it is omitted in our implementation of the monitor's state for $|r\rangle_I$. Moreover, the monitor's state buffers the time-stamps τ_k and input symbols b^k for $\mathcal{O}(|r|)$ time-points k following the window's start i. Hence, the submonitors $\bar{m}_{cur}$ for the direct subformulas of r do not have to be evaluated on Line 15 in Figure 3.13 as long as at most $\mathcal{O}(|r|)$ iterations of the loop on Lines 13–22 have been performed. For instance, this is always the case with regular expressions not containing the Kleene star, for which the current state q_{cur} in Figure 3.13 is equal to the empty state $q_{cur} = \{\}$ after at most $\mathcal{O}(|r|)$ loop iterations and the empty state $q = \{\}$ is always part of the domain of the function s in our implementation. Similarly, the submonitors $\bar{m}_1$ do not have to be evaluated as long as the number of time-points in the window $j - i$ is at most $\mathcal{O}(|r|)$. Note that our optimizations of time complexity do not impact the space complexity upper bounds in Theorem 3.12 asymptotically because there are always at most $3^{|\phi|}$ monitor state instances (since the window data structure uses up to 3 submonitors for each direct subformula of the regular expression) and the additional space usage overhead is thus at most $\mathcal{O}(3^{|\phi|} \cdot |\phi|)$.

Traces for Average-case Experiments The average-case traces with $\mathbb{T} = \mathbb{N}$ (the time-stamps are natural numbers) are produced by a pseudorandom trace generator, for a predefined length ℓ, event rate er, i.e., the number of time-points with the same time-stamp, and a maximum time-stamp difference Δ between consecutive time-points. The time-stamp differences between segments of er time-points with the same time-stamp are distributed uniformly in $\{1, \ldots, \Delta\}$. The set of atomic propositions Γ_i at a time-point i is generated as follows: (i) independently with probability $1 - \frac{1}{er}$, an atomic proposition $p_0, \ldots, p_3$ is included in Γ_i; (ii) independently with probability $\frac{1}{2}$, an atomic proposition $p_4, \ldots, p_{15}$ is included in Γ_i.

Formulas for Average-case Experiments The average-case formulas ϕ are produced by mutually recursive pseudorandom formula and regular expression generators, for a predefined number of subformulas and subexpressions $|\phi|$ and a maximum interval bound $b_{\max}$. A well-formed MDL formula ϕ with $k := |\phi|$, $k \geq 1$, is generated as follows: (i) if $k = 1$, then $\phi = p$, for an atomic proposition $p \in \{p_0, \ldots, p_{15}\}$ that is chosen uniformly at random; (ii) if $k = 2$, a top-level operator $\mathsf{op} \in \{\neg, \bullet_I, \bigcirc_I\}$ is chosen uniformly at random; (iii) if $k \geq 3$, a top-level operator $\mathsf{op} \in \{\neg, \wedge, \vee, \bullet_I, \bigcirc_I, \mathsf{S}_I, \mathsf{U}_I, \langle\cdot|_I, |\cdot\rangle_I\}$ is chosen uniformly at random. If the top-level operator op is a temporal operator with an interval I, then I is generated as follows: (i) with probability $\frac{1}{4}$, $I = [0, 0]$; (ii) with probability $\frac{1}{4}$, $I = [0, b]$, where $b \in \{0, \ldots, b_{\max}\} \cup \{\infty\}$ is chosen uniformly at random; (iii) with probability $\frac{1}{2}$, $I = [a, b]$, where $a \in \{0, \ldots, b_{\max}\}$ and $b \in \{l, \ldots, b_{\max}\} \cup \{\infty\}$ are chosen uniformly at random. We only allow $b = \infty$ for past temporal operators $\bullet_I, \mathsf{S}_I, \langle\cdot|_I$. Finally, if the top-level operator $\mathsf{op} \in \{\neg, \bullet_I, \bigcirc_I\}$ is a unary operator, a pseudorandom subformula ψ with $|\psi| = k - 1$ is generated recursively; if $\mathsf{op} \in \{\wedge, \vee, \mathsf{S}_I, \mathsf{U}_I\}$ is a binary operator, two pseudorandom subformulas ψ_1, ψ_2 with $|\psi_1| = k_1$ and $|\psi_2| = k - 1 - k_1$ are generated recursively, where $k_1 \in \{1, \ldots, k - 2\}$ is chosen uniformly at random; and if $\mathsf{op} \in \{\langle\cdot|_I, |\cdot\rangle_I\}$ is a temporal match operator, an MDL regular expression r with $(\sum_{\psi \in \mathsf{SF}(r)} |\psi|) + |r| = k - 1$ is generated recursively. A regular expression r with $k := (\sum_{\psi \in \mathsf{SF}(r)} |\psi|) + |r|$, $k \geq 2$, is generated as follows: (i) if $k = 2$, then $r = \psi$, for a formula ψ; (ii) if $k \in \{3, 4\}$, then we choose uniformly between $r = \psi$, for a formula ψ, and $r = s^*$, for an MDL regular expression s; (iii) if $k \geq 5$, then we choose uniformly between $r = \psi$, for a formula ψ, $r = s_1 \cdot s_2$, $r = s_1 + s_2$, and $r = s^*$, for MDL regular expressions s_1, s_2, s. Finally, if $r = \psi$, for a formula ψ, then ψ is generated recursively with $|\psi| = k - 1$; if $r = s_1 \cdot s_2$ or $r = s_1 + s_2$, for MDL regular expressions s_1, s_2, then s_1, s_2 are generated recursively with $(\sum_{\psi \in \mathsf{SF}(s_1)} |\psi|) + |r| = k_1$ and $(\sum_{\psi \in \mathsf{SF}(s_2)} |\psi|) + |r| = k - 1 - k_1$, where $k_1 \in \{2, \ldots, k - 3\}$ is chosen uniformly at random; if $r = s^*$, for an MDL regular expression s, then s is generated recursively with $(\sum_{\psi \in \mathsf{SF}(s)} |\psi|) + |r| = k - 1$.

Pseudorandom MTL formulas are obtained by ruling out the temporal match operators $\mathsf{op} \in \{\langle\cdot|_I, |\cdot\rangle_I\}$ when choosing a top-level operator. Past-only MTL formulas are obtained by additionally ruling out the future MTL operators $\mathsf{op} \in \{\bigcirc_I, \mathsf{U}_I\}$ when choosing a top-level operator. Equivalent $\text{MDL}^{\mathsf{Aerial}}$ formulas are obtained by applying the function $\mathsf{mdl2mdl}'(\phi)$ to well-formed MDL formulas ϕ produced by our pseudorandom formula generator. In our empirical evaluation, we have exported verified OCaml code for the function $\mathsf{mdl2mdl}'(\phi)$ from our Isabelle/HOL formalization and augmented this verified code with unverified OCaml code for parsing the MDL formula and outputting the $\text{MDL}^{\mathsf{Aerial}}$ formula.

Configuration	pMTL	MTL	MDL
Formulas	past-only MTL	MTL	MDL
Formula size $\|\phi\|$	25	25	25
Max. interval bounds $b_{\max}$	1 000	1 000	100
Trace length ℓ	100 000	2 000	2 000
Event rate er	1	2	2
Max. time-stamp difference Δ	1	100	100

Figure 3.23. Summary of configurations for the experiment “Large Intervals”.

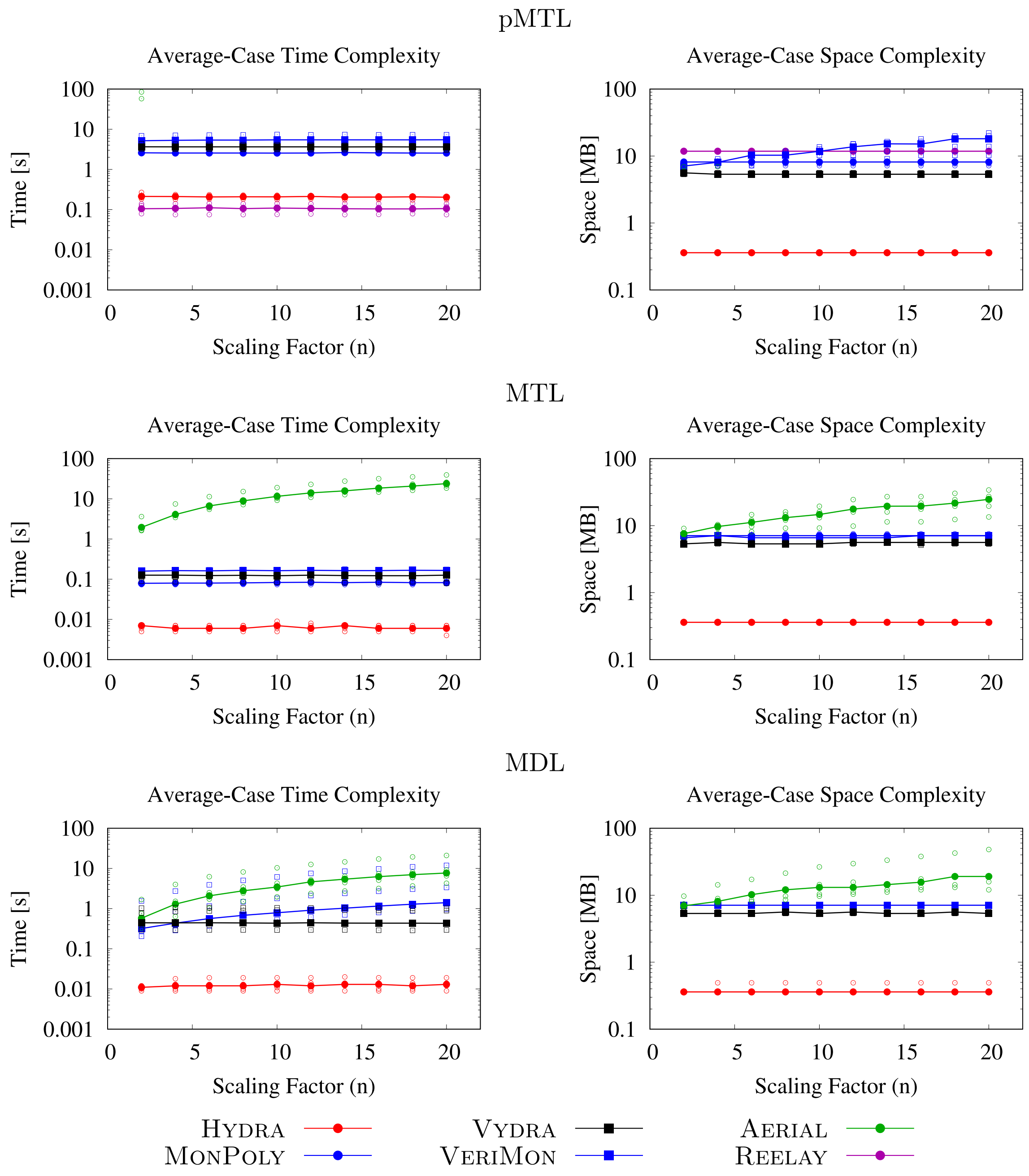


Figure 3.24. Evaluation results for the average-case experiments “Large Intervals”.

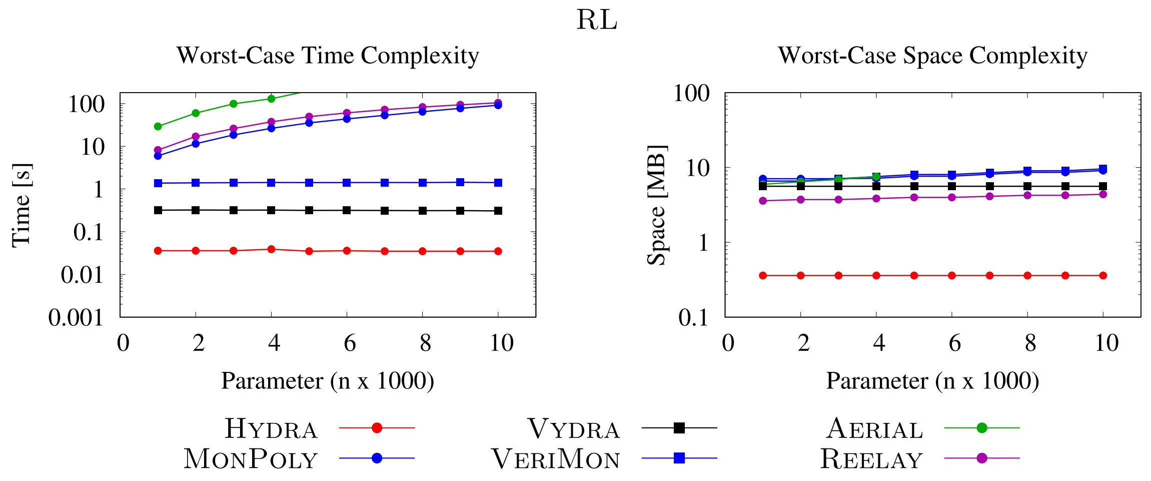


Figure 3.25. Evaluation results for the experiment RL.

3.3.1 RQ1: Large Intervals

In this section, we answer RQ1 by validating that the time and space complexity of HYDRA and VYDRA do not depend on the magnitude of time constraints, i.e., that their complexity is interval-oblivious. To this end, we generate a trace of a fixed length and a formula ϕ of a fixed size. Then we scale the interval bounds of ϕ's temporal operators by a scaling factor (2–20). In this experiment, we use three configurations whose individual parameters are summarized in Figure 3.23.

We also perform an experiment (abbreviated RL) using formulas and traces described by Ulus [75]. The MTL formulas $\mathsf{DELAY}(n)$ are of the form: $\mathsf{DELAY}(n) = p \, \mathsf{S}_{[n,n]} \, q$, where p and q are atomic propositions. A trace, parameterized by $n \in \mathbb{N}$, for the experiment RL is constructed with p being always true and q being true at every other time-point. The trace length of RL traces is $\ell = 100\,000$ and their time-stamps are equal to time-points.

We now state our expectations on the tools' behaviour. AERIAL is based on dynamic programming over *interval-shifted* formulas [6], i.e., formulas whose intervals are shifted down by offsets up to the interval bounds. Hence, AERIAL is not interval-oblivious in either time or space complexity. REELAY stores collections of intervals when evaluating temporal operators and implements an optimization merging overlapping intervals. Nevertheless, this optimization fails in the worst-case. Hence, we REELAY is not interval-oblivious in either time or space complexity. MONPOLY is not interval-oblivious in either time or space complexity. Finally, VERIMON is only interval-oblivious in time complexity for MTL, but not for MDL and also not in space complexity.

Figure 3.24 and Figure 3.25 contain the evaluation results for the average-case experiments and the experiment RL, respectively. The evaluation results confirm our expectations: HYDRA and VYDRA are interval-oblivious in both time and space complexity while the other tools are not interval-oblivious. The space complexity of MONPOLY and REELAY is plausibly interval-oblivious according to the actual space usage of these tools in our empirical evaluation. However, we confirmed that their time complexity is not interval-oblivious. Because all tools are deterministic algorithms that always halt, their space complexity cannot be constant if the time complexity is growing (otherwise the same state would be encountered twice during the computation and the tool could not halt). Hence, the space complexity of MONPOLY and REELAY cannot be interval-oblivious. Overall, we observe that HYDRA outperforms all other tools benchmarked in our experiments. Finally, note that the y-axis is logarithmic and thus a constant offset towards the y-axis corresponds to changing the order of magnitude.

3.3.2 RQ2: Large Formulas

In this section, we answer RQ2 by benchmarking Hydra and Vydra on formulas of increasing size. To this end, we generate a trace of a fixed length and formulas ϕ of increasing size $|\phi| \in \{6, 17, \ldots, 50\}$. In this experiment, we use three configurations whose individual parameters are summarized in Figure 3.26.

Figure 3.27 summarizes the evaluation results. The experiments show that Hydra can handle large formulas efficiently. In more detail, it can monitor a formula ϕ with $|\phi| = 50$ on 100 000 time-points in less than one second on average. Overall, we observe that Hydra outperforms all other tools benchmarked in our experiments except Reelay. However, Reelay only supports past-only MTL formulas and traces in which time-stamps are equal to time-points.

3.3.3 Special Cases

To answer RQ3 and RQ4, we conduct two more experiments on worst-case formulas for online monitoring and worst-case regular expressions for $\mathrm{MDL}^{\mathsf{Aerial}}$.

Worst-case Formulas for Online Monitoring We consider a family of MTL formulas $\langle \Phi_n \rangle_{n \in \mathbb{N}}$ that exhibit worst-case space complexity of online monitoring even when restricted to produce a *single* Boolean verdict for the first time-point. The formula Φ_n is defined over the set of atomic propositions $\Sigma_n = \{p_1, \ldots, p_n, e\}$:

$$\Phi_n = \bigcirc_{[1,1]} \left(\neg e \ \mathsf{U}_{[0,0]} \ \left(\neg e \wedge \textstyle\bigwedge_{i=1}^{n} (p_i \Rightarrow \Box_{[0,0]} \ (e \Rightarrow p_i)) \wedge \textstyle\bigwedge_{i=1}^{n} (\neg p_i \Rightarrow \Box_{[0,0]} \ (e \Rightarrow \neg p_i))\right)\right).$$

The family of traces, for a fixed $n \in \mathbb{N}$, on which the space complexity of online monitoring for Φ_n becomes at least 2^n bits looks as follows: the first time-point has a time-stamp τ_0 and an empty set of atomic propositions $\Gamma_0 = \emptyset$. Then for each subset $X \in \mathcal{X} \subseteq 2^{\Sigma_n - \{e\}}$ of atomic propositions without e, we add a time-point i with the atomic propositions $\Gamma_i = X$ and the time-stamp $\tau_0 + 1$. Next, for some $X \subseteq \Sigma_n - \{e\}$, we add a time-point i with the atomic propositions $\Gamma_i = X \cup \{e\}$ and the time-stamp $\tau_0 + 1$. Finally, we add a time-point with an empty set of atomic propositions and the time-stamp $\tau_0 + 3$, so that the trace uniquely determines the Boolean verdict for the first time-point.

Intuitively, for an online monitor to decide if Φ_n is satisfied at the first time-point of a trace from the family of traces, it must remember the exact subset $\mathcal{X}$ to check if the set X of atomic propositions, which eventually appear with the atomic proposition e, belongs to $\mathcal{X}$. As there are 2^{2^n} different sets $\mathcal{X}$, we derive a lower bound of 2^n bits to store $\mathcal{X}$.

We remark that the top-level operator Next in the formula Φ_n is used to make the formula trivially false on the worst-case traces described above at all time-points but the first one (recall that all evaluated monitors produce a sequence of Boolean verdicts at each position in the trace).

To benchmark monitoring Φ_n, for increasing $n \in \mathbb{N}$, we use traces of a fixed length obtained by concatenating worst-case traces with an increasing base time-stamp τ_0 into a single trace of a fixed length (independent of n). This way, we benchmark the time complexity to process a time-point rather than the time complexity to process an increasing number of time-points. We benchmark monitoring Φ_n using the tools Hydra, Vydra, Aerial, VeriMon supporting MTL with future temporal operators.

Configuration	pMTL	MTL	MDL
Formulas	past-only MTL	MTL	MDL
Max. interval bounds $b_{\max}$	50	50	50
Trace length ℓ	100 000	100 000	1 000
Event rate er	1	10	10
Max. time-stamp difference Δ	1	4	4

Figure 3.26. Summary of configurations for the experiment "Large Formulas".

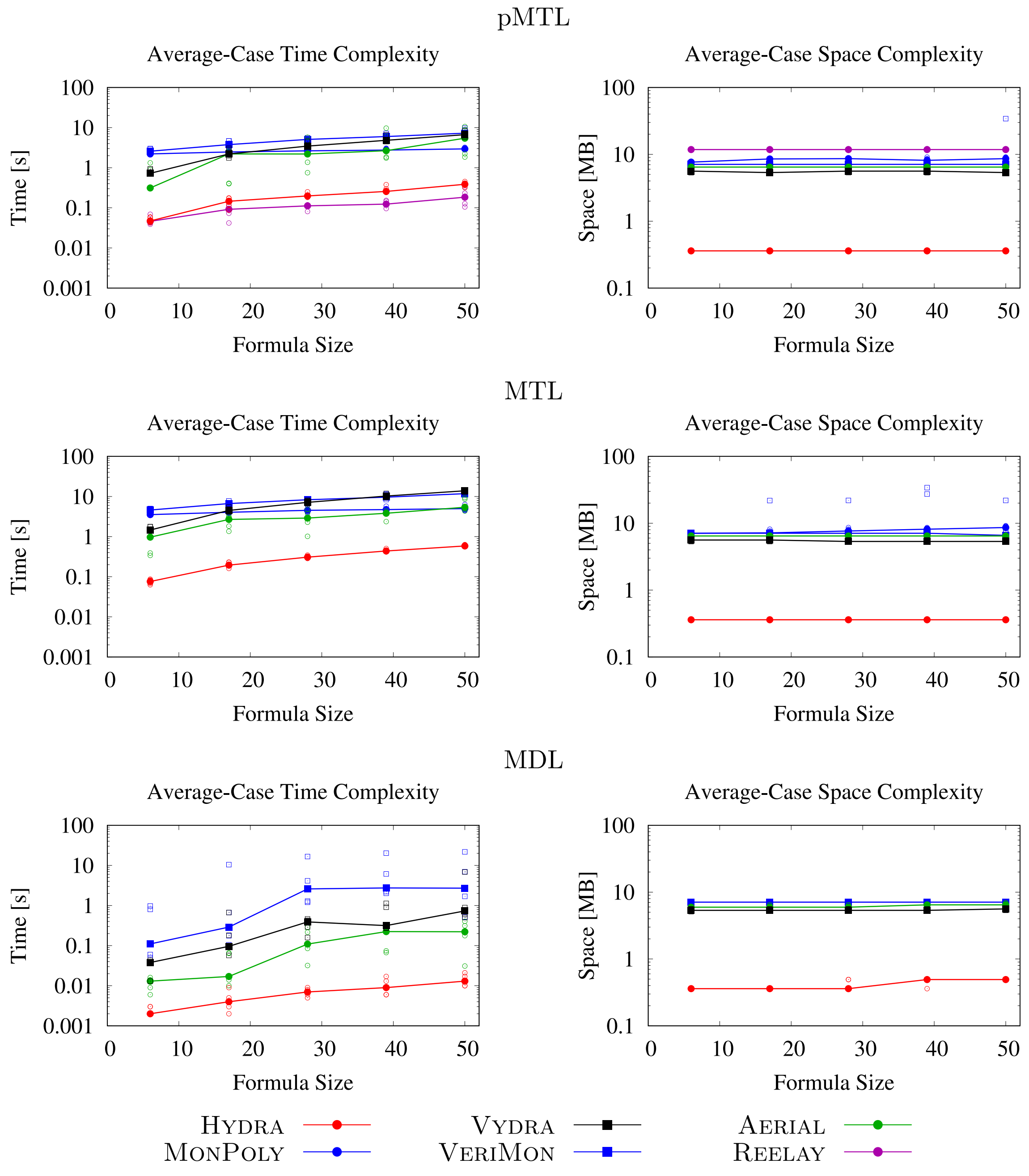


Figure 3.27. Evaluation results for the experiment "Large Formulas".

Figure 3.28 summarizes the evaluation results. We observe that HYDRA's and VYDRA's time complexity is polynomial, whereas AERIAL's and VERIMON's time complexity is exponential. (Recall that all y-axes are in logarithmic scale.)

Worst-case Regular Expressions for MDL$^{\mathsf{Aerial}}$ Finally, we carry out a benchmark on the MDL regular expressions from Section 2.2, for which we conjecture that the equivalent MDL$^{\mathsf{Aerial}}$ regular expressions are of quadratic size. The well-formed MDL regular expressions r_i, $i \in \mathbb{N}$, are defined as:

$$r_i = \begin{cases} \phi_0^* & \text{if } i = 0, \\ (\phi_i \cdot r_{i-1})^* & \text{if } i > 0. \end{cases}$$

The MDL regular expressions r_i have the following simple form: $(\phi_0)^*, (\phi_1 \cdot (\phi_0)^*)^*, (\phi_2 \cdot (\phi_1 \cdot (\phi_0)^*)^*)^*, \ldots$ and the equivalent MDL$^{\mathsf{Aerial}}$ regular expressions $\tilde{r}_i \coloneqq \mathsf{embed}'(\mathsf{rderive}(r_i))$ of quadratic size are obtained using the functions defined in Figure 2.7 (Section 2.2).

In this experiment, we benchmark the MDL formulas $\langle r_i|_{[0,0]}$ and $|r_i\rangle_{[0,0]}$ and the equivalent MDL$^{\mathsf{Aerial}}$ formulas $\langle \tilde{r}_i|_{[0,0]}$ and $|\tilde{r}_i\rangle_{[0,0]}$ on a pseudorandom trace of a fixed length and high event-rate, where every atomic proposition p_i is included in the set of atomic propositions Γ_i at a time-point i independently with probability $\frac{1}{2}$. We choose the direct subformulas of the MDL regular expressions as follows: $\phi_i \coloneqq p_i$, where $p_i \in \Sigma$ are atomic propositions.

Figure 3.28 summarizes the evaluation results. We observe that HYDRA significantly outperforms all other tools in terms of time complexity, but HYDRA's space usage temporarily exceeds AERIAL's space usage. This is because HYDRA stores all deterministic states of the automaton for the MDL regular expressions r_i ever encountered during monitoring. On the other hand, AERIAL does not buffer any Boolean expressions for the MDL$^{\mathsf{Aerial}}$ formulas $\langle \tilde{r}_i|_{[0,0]}$ and $|\tilde{r}_i\rangle_{[0,0]}$ and thus its space usage is linear in their size, i.e., quadratic in $|r_i|$.

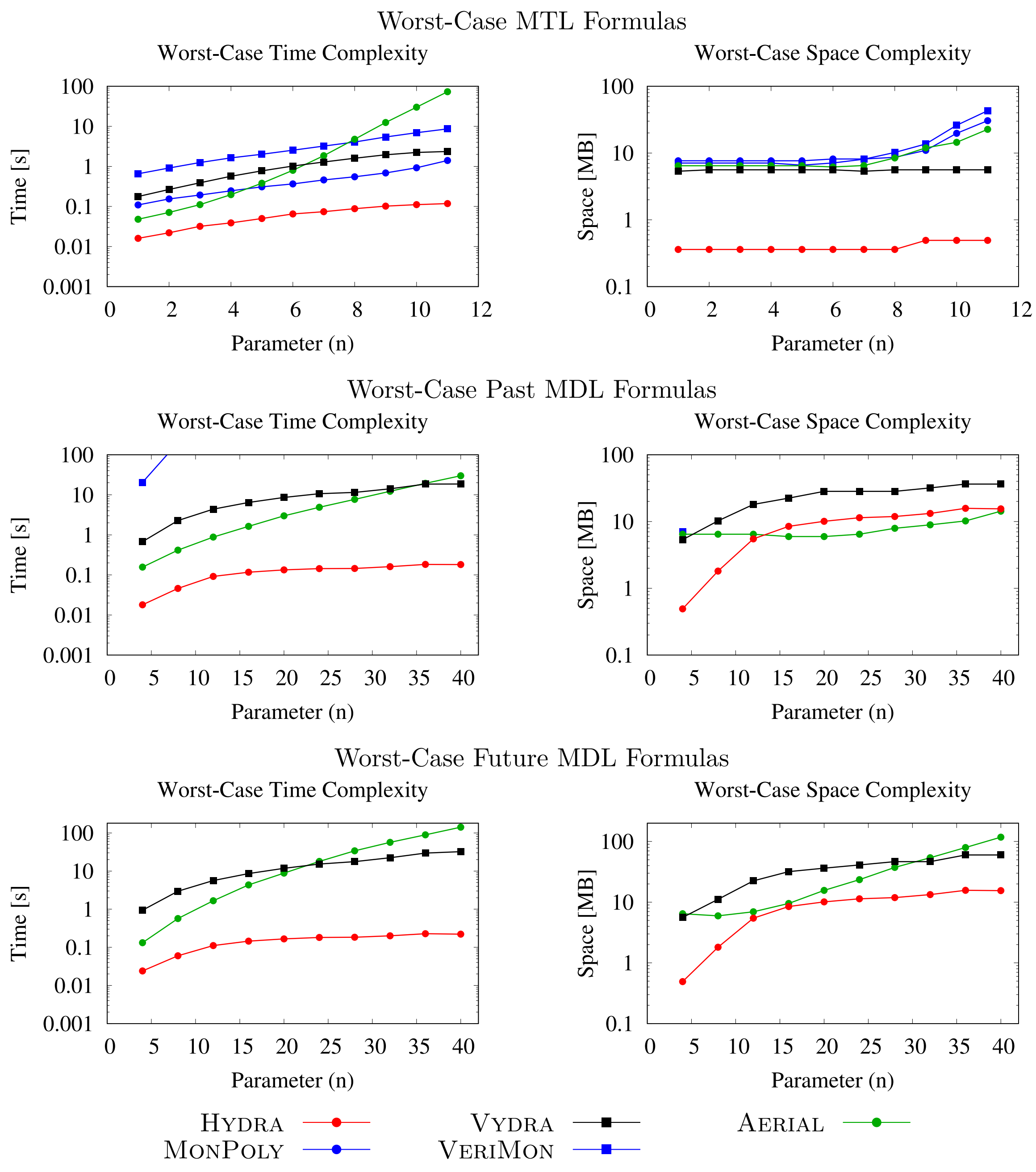


Figure 3.28. Evaluation results for the worst-case experiments.

Chapter 4

First-Order Monitoring

4.1 Introduction

Propositional temporal specification languages can express properties about a finite (a priori fixed) set of events. In this setting, we proposed a multi-head monitoring algorithm (Chapter 3) that can efficiently handle an arbitrary specification formulated in metric dynamic logic—a propositional temporal logic. In particular, there are no restrictions on the use of negation and other logical operators. On the other hand, fixing a set of events for a system a priori might be infeasible: Consider, for instance, a system managing user authentication, where new user accounts can be created while the system is running. And even if the system did not permit to create new accounts, enumerating a large number of users in a specification explicitly might be tedious and error-prone. Hence, a more flexible solution is to only fix a small set of event types and let the actual events be parameterized by data values, e.g., the users of the system. A monitor for a parameterized property decides if the property holds for the individual values of the parameters, i.e., the monitor computes a set of the parameter valuations satisfying the property (violations can also be computed as satisfactions of the property's negation).

We first consider the case of parameterized properties without temporal operators that can also be interpreted as database queries, with a well-established theory and practice. To benefit from this theory and its efficient implementations in the form of standard database management systems (e.g., PostgreSQL), the parameterized property must be expressed in a database query language which typically imposes several restrictions on the shape of the query. For instance, the database query language SQL restricts the use of negation in queries to make sure that (potentially multiple) ways of evaluating an SQL query can be automatically derived from its syntactic structure. Consequently, we could easily express the query from Example 1.2 in relational calculus, but not in SQL.

Codd's theorem states that all domain-independent queries of the relational calculus can be expressed in relational algebra (RA) [21]. Yet, the proof of Codd's theorem does not address the efficiency of evaluating the resulting RA expression. Moreover, it does not cover queries that are not domain-independent in general, but that could still be evaluated using RA over specific database instances with additional properties. An example of such a query is a variation of the query from Example 1.3:

$$Q^{susp}_{user} \coloneqq \mathsf{B}(b) \land \exists s.\, \forall p.\, \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s).$$

This query might be relevant for a shop in which brands (unary finite relation B of brands) sell products (binary finite relation P relating brands and products) and products are reviewed by users with a score (ternary finite relation S relating products, users, and scores). It is satisfied by all brands and users for which there exists a score that the user assigned to all the brand's products. If a brand has no product, then every user satisfies this condition and the query is satisfied by an infinite set of tuples. However, if every brand has at least one product, then the query is only satisfied by a finite number of tuples that can be computed by the following RA expression (where $-$ is the set difference operator and $\triangleright$ is the anti-join):

$$\pi_{brand,user}((\pi_{user,score}(\mathsf{S}) \times \mathsf{B}) - \pi_{brand,user,score}((\pi_{user,score}(\mathsf{S}) \times \mathsf{P}) \triangleright \mathsf{S})).$$

Van Gelder and Topor [33,34] present a translation from a decidable class of domain-independent RC queries, called *evaluable*, to RA expressions with the goal of improving the performance of evaluating the resulting RA expression. Still, they do not cover queries like Q^{susp}_{user} and the performance of evaluating the RA expression produced by their translation can also be further improved.

RC Query Translation In this thesis, we translate arbitrary RC queries to RA expressions under the assumption of an infinite domain. To deal with queries that are domain-dependent, our translation produces two RA expressions, instead of a single equivalent one. The first RA expression characterizes the original RC query's relative safety, the decidable question of whether the query evaluates to a finite relation for a given database, which can be the case even for a domain-dependent query, e.g., Q^{susp}_{user}. If the original query is relatively safe on a given database, i.e., produces some finite result, then the second RA expression evaluates to the same finite result. Taken together, the two RA expressions solve the *query capturability* problem [3]: they allow us to enumerate the original RC query's finite evaluation result, or to learn that it would be infinite using RA operations on the unmodified database.

MFOTL Query Translation Next we generalize our translation of RC queries to all MFOTL queries, i.e., first-order queries with temporal operators. Our translation improves upon the translation by Basin et al. [10] who describe a heurestic to bring a given MFOTL query to RANF. Their heuristic is incomplete, i.e., it fails to translate many MFOTL queries to equivalent queries in RANF. This issue also prevented Havelund et al. [42] from using MonPoly [10] on a query formalizing data races in a concurrent system.

Unlike for RC, where we produce a pair of RA expressions, we translate an MFOTL query Q into a single MFOTL query in RANF that can be evaluated at every time-point by state-of-the-art first-order monitoring algorithms, e.g., VeriMon [8] and MonPoly [10]. This single MFOTL query uses a fresh variable $f \notin \mathsf{fv}(Q)$ to signal if the evaluation result at a time-point is infinite. If it is infinite, then the RANF query is satisfied by a single tuple with a special value $\mathsf{c_{inf}}$ of the variable f. Otherwise, the RANF query is satisfied by the same set of tuples as Q, with each tuple extended with another special value $\mathsf{c_{fin}}$ of the variable f. The advantage of having a pair of queries is that the finiteness check is decoupled from the computation of the finite set of satisfying tuples. However, for MFOTL, one would need to run two monitors for the two queries and synchronize their results at the individual time-points.

MFOTL Query Evaluation Once we have an MFOTL query in RANF, we can evaluate it by following its syntactic structure using operations on finite tables. To optimize this evaluation, we propose, implement, and formally verify [24] using the Isabelle/HOL proof assistant an efficient algorithm for the temporal operators S_I and U_I whose total time complexity for processing a sequence of time-points depends linearly on the number of these time-points and on the total number of tuples in the input relations for the arguments of the temporal operators. In contrast, MonPoly's time complexity can be quadratic in the number of processed time-points in the worst-case.

Chapter Outline Our translation of an RC query to two RA expressions proceeds in several steps. We presented standard algorithms for translating an arbitrary safe-range RC query to an equivalent RANF query that can be directly mapped to an equivalent RA expression in Section 2.3.1. In this section, we focus on translating an RC query to two safe-range RC queries (Section 4.2). Afterwards, we generalize our translation of RC queries to MFOTL queries (Section 4.3). Finally, we present our optimized monitoring algorithm for the temporal operators S_I and U_I (Section 4.4). We also carry out empirical evaluations of our translations of RC (Section 4.2.7) and MFOTL (Section 4.3.2) queries as well as our optimized monitoring algorithm for S_I and U_I (Section 4.4.3). For RC queries, we also provide a theoretical complexity analysis (Section 4.2.4) and develop a method called Data Golf that produces hard database instances used to evaluate pseudorandom RC queries (Section 4.2.5).

4.2 Relational Calculus Query Translation

Our approach to evaluating an arbitrary RC query Q over a fixed structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ proceeds by translating Q into a pair of safe-range queries $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ such that

(FV) $\mathsf{fv}(Q_{\mathit{fin}}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to *false*; $\mathsf{fv}(Q_{\mathit{inf}}) = \emptyset$;

(EVAL) $[\![Q]\!]$ is an infinite set if Q_{inf} holds; otherwise $[\![Q]\!] = [\![Q_{\mathit{fin}}]\!]$ is a finite set.

Since the queries Q_{fin} and Q_{inf} are safe-range, they are domain-independent and thus $[\![Q_{\mathit{fin}}]\!]$ is a finite set of tuples. In particular, $[\![Q]\!]$ is a finite set of tuples if Q_{inf} does not hold. Our translation generalizes Hull and Su's case distinction that restricts bound variables [44] to restrict all variables. Moreover, we use Van Gelder and Topor's idea to replace the active domain by a smaller set (generator) specific to each variable [34] while further improving the generators.

4.2.1 Restricting One Variable

Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables. This assumption on $\tilde{Q}$ will be established by translating an arbitrary query Q bottom-up (Section 4.2.2). In this section, we develop a translation of $\tilde{Q}$ into an equivalent query $\tilde{Q}'$ that satisfies the following:

- $\tilde{Q}'$ has range-restricted bound variables;
- $\tilde{Q}'$ is a disjunction and x is range-restricted in all but the last disjunct.

The disjunct in which x is not range-restricted has a special form that is central to our translation: it is the conjunction of a query in which x does not occur and a query that is satisfied by infinitely many values of x. From the case distinction "for the corresponding variable: in or out of *adom*, and equality or inequality to other 'previous' variables if out of *adom*" [44], we translate $\tilde{Q}$ into the following equivalent query:

$$\begin{aligned}\tilde{Q} \equiv\, & (\tilde{Q} \land x \in \mathsf{adom}(\tilde{Q})) \lor \textstyle\bigvee_{y \in \mathsf{fv}(\tilde{Q}) - \{x\}} (\tilde{Q}[x \mapsto y] \land x \approx y) \lor \\ & (\tilde{Q}[x/\mathit{false}] \land \neg(x \in \mathsf{adom}(\tilde{Q}) \lor \textstyle\bigvee_{y \in \mathsf{fv}(\tilde{Q}) - \{x\}} x \approx y)).\end{aligned}$$

Here, $x \in \mathsf{adom}(\tilde{Q})$ stands for an RC query with a single free variable x that is satisfied by an assignment α if and only if $\alpha(x) \in \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$. The translation distinguishes the following three cases for a fixed assignment α:

- if $\alpha(x) \in \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ holds, then we do not alter the query $\tilde{Q}$;
- if $x \approx y$ holds for some free variable $y \in \mathsf{fv}(\tilde{Q}) - \{x\}$, then x can be replaced by y in $\tilde{Q}$;
- otherwise, $\tilde{Q}$ is equivalent to $\tilde{Q}[x/\mathit{false}]$, i.e., all atomic predicates with a free occurrence of x can be replaced by *false* (because $\alpha(x) \notin \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$), all equalities $x \approx y$ and $y \approx x$ for $y \in \mathsf{fv}(\tilde{Q}) - \{x\}$ can be replaced by *false* (because $\alpha(x) \neq \alpha(y)$), and all equalities $x \approx z$ for a bound variable z can be replaced by *false* (because $\alpha(x) \notin \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ and z is range-restricted in its subquery $\exists z.\, Q_z$, by assumption, i.e., $\mathsf{gen}(z, Q_z)$ holds and thus, for all α', we have $\alpha' \models \exists z.\, Q_z$ if and only if there exists $d \in \mathsf{adom}^{\mathcal{S}}(Q_z) \subseteq \mathsf{adom}^{\mathcal{S}}(\tilde{Q})$ such that $\alpha'[z \mapsto d] \models Q_z$).

Note that $\exists \vec{\mathsf{fv}}(Q) - \{x\}.\, Q$ is the query in which all free variables of Q except x are existentially quantified. Given a set of quantified predicates $\mathcal{G}$, we write $\exists \vec{\alpha}.\, \mathcal{G}$ for $\bigvee_{Q_{qp} \in \mathcal{G}} \exists \vec{\alpha}.\, Q_{qp}$. To avoid enumerating the entire active domain $\mathsf{adom}^{\mathcal{S}}(Q)$ of the query Q and a structure $\mathcal{S}$, Van Gelder and Topor [34] replace the condition $x \in \mathsf{adom}(Q)$ in their translation by $\exists \vec{\mathsf{fv}}(\mathcal{G}) - \{x\}.\, \mathcal{G}$, where the generator set $\mathcal{G}$ is a subset of atomic predicates. Because their translation [34] must yield an equivalent query (for every finite or infinite domain), $\mathcal{G}$ must satisfy, for all α,

$$\begin{array}{lll} \alpha \models \neg\exists \vec{\mathsf{fv}}(\mathcal{G}) - \{x\}.\, \mathcal{G} \Longrightarrow (\alpha \models Q \Longleftrightarrow \alpha \models Q[x/\mathit{false}]) & (\textsc{vgt}_1) & \text{and} \\ \alpha \models Q[x/\mathit{false}] \qquad\quad \Longrightarrow \alpha \models \forall x.\, Q & (\textsc{vgt}_2). & \end{array}$$

Note that ($\textsc{vgt}_2$) does not hold for the query $Q := \neg\mathsf{B}(x)$ and thus a generator set $\mathcal{G}$ of atomic predicates satisfying ($\textsc{vgt}_2$) only exists for a proper subset of all RC queries. In contrast, we only require that $\mathcal{G}$ satisfies ($\textsc{vgt}_1$) in our translation. To this end, we define a *covered* relation $\mathsf{cov}(x, Q, \mathcal{G})$ (in contrast to Van Gelder and Topor's *constrained* relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ defined in Figure 2.11) such that, for every variable x and query $\tilde{Q}$ with range-restricted bound variables, there exists at least one set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ and ($\textsc{vgt}_1$) holds. Figure 4.1 shows the definition of this relation. Unlike the generator set $\mathcal{G}$ in $\mathsf{gen}(x, Q, \mathcal{G})$, the *cover* set $\mathcal{G}$ in $\mathsf{cov}(x, Q, \mathcal{G})$ may also contain equalities between two variables. Hence, we define a function $\mathsf{qps}(\mathcal{G})$ that collects all *generators*, i.e., quantified predicates, and a function $\mathsf{eqs}(x, \mathcal{G})$ that collects all *variables* y distinct from x occurring in equalities of the form $x \approx y$. We use $\mathsf{qps}^{\vee}(\mathcal{G})$ to denote the query $\bigvee_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} Q_{qp}$. We state the soundness and completeness of the relation $\mathsf{cov}(x, Q, \mathcal{G})$ in the next lemma, which follows by induction on the derivation of $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

Lemma 4.1. *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$. Then there exists a set $\mathcal{G}$ of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds and, for any such $\mathcal{G}$ and all α,*

$$\alpha \models \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x, \mathcal{G})} x \approx y) \Longrightarrow (\alpha \models \tilde{Q} \Longleftrightarrow \alpha \models \tilde{Q}[x/\mathit{false}]).$$

Finally, to preserve the dependencies between the variable x and the remaining free variables of Q occurring in the quantified predicates from $\mathsf{qps}(\mathcal{G})$, we do not project $\mathsf{qps}(\mathcal{G})$ on the single variable x, i.e., we restrict x by $\mathsf{qps}^{\vee}(\mathcal{G})$ instead of $\exists \vec{\mathsf{fv}}(Q) - \{x\}.\, \mathsf{qps}(\mathcal{G})$. From Lemma 4.1, we derive our optimized translation characterized by the following lemma.

Lemma 4.2. *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$, and $\mathcal{G}$ be a set of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Then $x \in \mathsf{fv}(Q_{qp})$ and $\mathsf{fv}(Q_{qp}) \subseteq \mathsf{fv}(\tilde{Q})$, for every $Q_{qp} \in \mathsf{qps}(\mathcal{G})$, and*

$$\begin{aligned} \tilde{Q} \equiv{} & (\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x, \mathcal{G})} (\tilde{Q}[x \mapsto y] \wedge x \approx y) \vee \\ & (\tilde{Q}[x/\mathit{false}] \wedge \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x, \mathcal{G})} x \approx y)). \end{aligned} \qquad (\bigstar)$$

$$
\begin{array}{ll}
\mathsf{cov}(x, x \approx x, \emptyset); & \\
\mathsf{cov}(x, Q, \emptyset) & \text{if } x \notin \mathsf{fv}(Q); \\
\mathsf{cov}(x, x \approx y, \{x \approx y\}) & \text{if } x \neq y; \\
\mathsf{cov}(x, y \approx x, \{x \approx y\}) & \text{if } x \neq y; \\
\mathsf{cov}(x, Q, \{Q\}) & \text{if } \mathsf{ap}(Q) \text{ and } x \in \mathsf{fv}(Q); \\
\mathsf{cov}(x, \neg Q, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q, \mathcal{G}); \\
\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2); \\
\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}) \text{ and } Q_1[x/\mathit{false}] = \mathit{true}; \\
\mathsf{cov}(x, Q_1 \vee Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\mathit{false}] = \mathit{true}; \\
\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G}_1 \cup \mathcal{G}_2) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2); \\
\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}) \text{ and } Q_1[x/\mathit{false}] = \mathit{false}; \\
\mathsf{cov}(x, Q_1 \wedge Q_2, \mathcal{G}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\mathit{false}] = \mathit{false}; \\
\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, \mathcal{G}) & \text{if } x \neq y \text{ and } \mathsf{cov}(x, Q_y, \mathcal{G}) \text{ and } (x \approx y) \notin \mathcal{G}; \\
\mathsf{cov}(x, \exists y.\, Q_y, \tilde{\exists} y.\, (\mathcal{G} - \{x \approx y\}) \cup \mathcal{G}_y[y \mapsto x]) & \text{if } x \neq y \text{ and } \mathsf{cov}(x, Q_y, \mathcal{G}) \text{ and } \mathsf{gen}(y, Q_y, \mathcal{G}_y).
\end{array}
$$

Figure 4.1. The *covered* relation.

Note that x is not guaranteed to be range-restricted in (★)'s last disjunct. However, it occurs only in the negation of a disjunction of quantified predicates with a free occurrence of x and equalities of the form $x \approx y$. We will show how to handle such occurrences in Sections 4.2.2 and 4.2.3. Moreover, the negation of the disjunction can be omitted if (VGT$_2$) holds.

4.2.2 Restricting Bound Variables

Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables. Suppose that the variable x is not range-restricted, i.e., $\mathsf{gen}(x, \tilde{Q})$ does not hold. To translate $\exists x.\, \tilde{Q}$ into an inf-equivalent query with range-restricted bound variables ($\exists x.\, \tilde{Q}$ does not have range-restricted bound variables precisely because x is not range-restricted in $\tilde{Q}$), we first apply (★) to $\tilde{Q}$ and distribute the existential quantifier binding x over disjunction. Next we observe that

$$\exists x.\, (\tilde{Q}[x \mapsto y] \wedge x \approx y) \equiv \tilde{Q}[x \mapsto y] \wedge \exists x.\, (x \approx y) \equiv \tilde{Q}[x \mapsto y],$$

where the first equivalence follows because x does not occur free in $\tilde{Q}[x \mapsto y]$ and the second equivalence follows from the straightforward validity of $\exists x.\, (x \approx y)$. Moreover, we observe that

$$\exists x.\, (\tilde{Q}[x/\mathit{false}] \wedge \neg(\mathsf{qps}^{\vee}(\mathcal{G}) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x, \mathcal{G})} x \approx y)) \overset{\infty}{\equiv} \tilde{Q}[x/\mathit{false}]$$

because x is not free in $\tilde{Q}[x/\mathit{false}]$ and there exists a value d for x in the infinite domain $\mathcal{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathsf{eqs}(x, \mathcal{G})$ and d is not among the finitely many values interpreting the quantified predicates in $\mathsf{qps}(\mathcal{G})$. Altogether, we obtain the following lemma.

Lemma 4.3. *Let $\tilde{Q}$ be a query with range-restricted bound variables, $x \in \mathsf{fv}(\tilde{Q})$, and $\mathcal{G}$ be a set of quantified predicates and equalities such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Then*

$$\exists x.\, \tilde{Q} \overset{\infty}{\equiv} (\exists x.\, \tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee \textstyle\bigvee_{y \in \mathsf{eqs}(x, \mathcal{G})} (\tilde{Q}[x \mapsto y]) \vee \tilde{Q}[x/\mathit{false}]. \qquad (★\exists)$$

Our approach for restricting all bound variables recursively applies Lemma 4.3. Because the set $\mathcal{G}$ such that $\mathsf{cov}(x, Q, \mathcal{G})$ holds is not necessarily unique, we introduce the following (general) notation. We denote the non-deterministic choice of an object X from a non-empty set $\mathcal{X}$ as

```
     input:  An RC query $Q$.
     output: A query $\tilde{Q}$ with range-restricted bound variables such that $Q \overset{\infty}{\equiv} \tilde{Q}$.
  1  function $\mathsf{fixbound}(\mathcal{Q}, x) = \{Q_{\mathit{fix}} \in \mathcal{Q} \mid x \in \mathsf{nongens}(Q_{\mathit{fix}})\}$;
  2  function $\mathsf{rb}(Q) =$
  3      switch $Q$ do
  4          case $\neg Q'$ do return $\neg\mathsf{rb}(Q')$;
  5          case $Q'_1 \vee Q'_2$ do return $\mathsf{rb}(Q'_1) \vee \mathsf{rb}(Q'_2)$;
  6          case $Q'_1 \wedge Q'_2$ do return $\mathsf{rb}(Q'_1) \wedge \mathsf{rb}(Q'_2)$;
  7          case $\exists x.\, Q_x$ do
  8              $\mathcal{Q} := \mathsf{flat}^{\vee}(\mathsf{rb}(Q_x))$;
  9              while $\mathsf{fixbound}(\mathcal{Q}, x) \neq \emptyset$ do
 10                  $Q_{\mathit{fix}} \leftarrow \mathsf{fixbound}(\mathcal{Q}, x)$;
 11                  $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
 12                  $\mathcal{Q} := (\mathcal{Q} - \{Q_{\mathit{fix}}\}) \cup \{Q_{\mathit{fix}} \wedge \mathsf{qps}^{\vee}(\mathcal{G})\} \cup \bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{Q_{\mathit{fix}}[x \mapsto y]\} \cup$
                     $\{Q_{\mathit{fix}}[x/\mathit{false}]\}$;
 13              return $\bigvee_{\tilde{Q} \in \mathcal{Q}} \tilde{\exists} x.\, \tilde{Q}$;
 14          otherwise do return $Q$;
```

Figure 4.2. Restricting bound variables.

$X \leftarrow \mathcal{X}$. We define the recursive function $\mathsf{rb}(Q)$ in Figure 4.2, where rb stands for *range-restrict bound* (variables). The function converts an arbitrary RC query Q into an inf-equivalent query with range-restricted bound variables. We proceed by describing the case $\exists x.\, Q_x$. First, $\mathsf{rb}(Q_x)$ is recursively applied on Line 8 to establish the precondition of Lemma 4.3 that the translated query has range-restricted bound variables. Because existential quantification distributes over disjunction, we flatten disjunction in $\mathsf{rb}(Q_x)$ and process the individual disjuncts independently. We apply $(\bigstar\exists)$ to every disjunct Q_{fix} in which the variable x is not already range-restricted. For every Q'_{fix} added to $\mathcal{Q}$ after applying $(\bigstar\exists)$ to Q_{fix} the variable x is either range-restricted or does not occur in Q'_{fix}, i.e., $x \notin \mathsf{nongens}(Q'_{\mathit{fix}})$. This entails the termination of the loop on Lines 9–12.

Example 4.4. Consider the query $Q^{\mathit{susp}}_{\mathit{user}} := \mathsf{B}(b) \wedge \exists s.\, \forall p.\, \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s)$ from Section 4.1. Restricting its bound variables yields the query

$$\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) = \mathsf{B}(b) \wedge ((\exists s.\, (\neg\exists p.\, \mathsf{P}(b,p) \wedge \neg\mathsf{S}(p,u,s)) \wedge (\exists p.\, \mathsf{S}(p,u,s))) \vee (\neg\exists p.\, \mathsf{P}(b,p))).$$

The bound variable p is already range-restricted in $Q^{\mathit{susp}}_{\mathit{user}}$ and thus only s must be restricted. Applying $(\bigstar)$ to restrict s in $\neg\exists p.\, \mathsf{P}(b,p) \wedge \neg\mathsf{S}(p,u,s)$, then existentially quantifying s, and distributing the existential over disjunction yields the first disjunct in $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})$ above and $\exists s.\, (\neg\exists p.\, \mathsf{P}(b,p)) \wedge \neg(\exists p.\, \mathsf{S}(p,u,s))$ as the second disjunct. Because there exists some value in the infinite domain $\mathcal{D}$ that does not belong to the finite interpretation of the atomic predicate $\mathsf{S}(p,u,s)$, the query $\exists s.\, \neg(\exists p.\, \mathsf{S}(p,u,s))$ is a tautology over $\mathcal{D}$. Hence, $\exists s.\, (\neg\exists p.\, \mathsf{P}(b,p)) \wedge \neg(\exists p.\, \mathsf{S}(p,u,s))$ is inf-equivalent to $\neg\exists p.\, \mathsf{P}(b,p)$, i.e., the second disjunct in $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})$. This reasoning justifies applying $(\bigstar\exists)$ to restrict s in $\exists s.\, \neg\exists p.\, \mathsf{P}(b,p) \wedge \neg\mathsf{S}(p,u,s)$. ◇

4.2.3 Restricting Free Variables

Given an arbitrary query Q, we translate the inf-equivalent query $\mathsf{rb}(Q)$ with range-restricted bound variables into a pair of safe-range queries $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ such that our translation's main

```
input:  An RC query $Q$.
output: Safe-range query pair $(Q_{\mathit{fin}}, Q_{\mathit{inf}})$ for which (FV) and (EVAL) hold.
1  function $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) = \{(Q_{\mathit{fix}}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}} \mid \mathsf{nongens}(Q_{\mathit{fix}}) \neq \emptyset\}$;
2  function $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) = \{(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}} \mid \mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) \neq \emptyset \lor$
     $\mathsf{fv}(Q_{\not\infty} \land Q^{\approx}) \neq \mathsf{fv}(Q)\}$;
3  function $\mathsf{split}(Q) =$
4      $\mathcal{Q}_{\mathit{fin}} := \{(\mathsf{rb}(Q), \mathit{true})\}; \mathcal{Q}_{\mathit{inf}} := \emptyset$;
5      while $\mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}}) \neq \emptyset$ do
6          $(Q_{\mathit{fix}}, Q^{\approx}) \leftarrow \mathsf{fixfree}(\mathcal{Q}_{\mathit{fin}})$;
7          $x \leftarrow \mathsf{nongens}(Q_{\mathit{fix}})$;
8          $\mathcal{G} \leftarrow \{\mathcal{G} \mid \mathsf{cov}(x, Q_{\mathit{fix}}, \mathcal{G})\}$;
9          $\mathcal{Q}_{\mathit{fin}} := (\mathcal{Q}_{\mathit{fin}} - \{(Q_{\mathit{fix}}, Q^{\approx})\}) \cup \{(Q_{\mathit{fix}} \land \mathsf{qps}^{\lor}(\mathcal{G}), Q^{\approx})\} \cup$
             $\bigcup_{y \in \mathsf{eqs}(x,\mathcal{G})} \{(Q_{\mathit{fix}}[x \mapsto y], Q^{\approx} \land x \approx y)\}$;
10         $\mathcal{Q}_{\mathit{inf}} := \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\mathit{fix}}[x/\mathit{false}]\}$;
11     while $\mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q) \neq \emptyset$ do
12         $(Q_{\not\infty}, Q^{\approx}) \leftarrow \mathsf{inf}(\mathcal{Q}_{\mathit{fin}}, Q)$;
13         $\mathcal{Q}_{\mathit{fin}} := \mathcal{Q}_{\mathit{fin}} - \{(Q_{\not\infty}, Q^{\approx})\}$;
14         $\mathcal{Q}_{\mathit{inf}} := \mathcal{Q}_{\mathit{inf}} \cup \{Q_{\not\infty} \land Q^{\approx}\}$;
15     return $(\bigvee_{(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}} (Q_{\not\infty} \land Q^{\approx}), \mathsf{rb}(\bigvee_{Q_\infty \in \mathcal{Q}_{\mathit{inf}}} \exists \vec{\mathsf{fv}}(Q_\infty).\, Q_\infty))$;
```

Figure 4.3. Restricting free variables.

properties FV and EVAL hold. Our translation is based on the following lemma.

Lemma 4.5. *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables and let $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ for a set of quantified predicates and equalities $\mathcal{G}$. If $\tilde{Q}[x/\mathit{false}]$ is not satisfied by any tuple, then*

$$[\![\tilde{Q}]\!] = \left[\!\!\left[(\tilde{Q} \land \mathsf{qps}^{\lor}(\mathcal{G})) \lor \bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y] \land x \approx y)\right]\!\!\right]. \qquad (☆)$$

If $\tilde{Q}[x/\mathit{false}]$ is satisfied by some tuple, then $[\![\tilde{Q}]\!]$ is an infinite set.

Proof. If $\tilde{Q}[x/\mathit{false}]$ is not satisfied by any tuple, then (☆) follows from (★). If $\tilde{Q}[x/\mathit{false}]$ is satisfied by some tuple, then the last disjunct in (★) applied to $\tilde{Q}$ is satisfied by infinitely many tuples obtained by assigning x some value from the infinite domain $\mathcal{D}$ such that $x \neq y$ holds for all finitely many $y \in \mathsf{eqs}(x, \mathcal{G})$ and x does not appear among the finitely many values interpreting the quantified predicates from $\mathsf{qps}(\mathcal{G})$. □

We remark that $[\![\tilde{Q}]\!]$ might be an infinite set of tuples even if $\tilde{Q}[x/\mathit{false}]$ is never satisfied, for some x. This is because $\tilde{Q}[y/\mathit{false}]$ might be satisfied by some tuple, for some y, in which case Lemma 4.5 (for y) implies that $[\![\tilde{Q}]\!]$ is an infinite set of tuples. Still, (☆) can be applied to $\tilde{Q}$ for x resulting in a query satisfied by the same infinite set of tuples.

Our approach is implemented by the function $\mathsf{split}(Q)$ defined in Figure 4.3. In the following, we describe this function and informally justify its correctness, formalized by the input/output specification. In $\mathsf{split}(Q)$, we represent the queries Q_{fin} and Q_{inf} using a set $\mathcal{Q}_{\mathit{fin}}$ of query pairs and a set $\mathcal{Q}_{\mathit{inf}}$ of queries such that

$$Q_{\mathit{fin}} := \bigvee_{(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}} (Q_{\not\infty} \land Q^{\approx}), \qquad Q_{\mathit{inf}} := \bigvee_{Q_\infty \in \mathcal{Q}_{\mathit{inf}}} \exists \vec{\mathsf{fv}}(Q_\infty).\, Q_\infty,$$

and, for every $(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}$, $Q^{\approx}$ is a conjunction of equalities. As long as there exists some $(Q_{\mathit{fix}}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}$ such that $\mathsf{nongens}(Q_{\mathit{fix}}) \neq \emptyset$, we apply (✩) to Q_{fix} and add the query $Q_{\mathit{fix}}[x/\mathit{false}]$ to $\mathcal{Q}_{\mathit{inf}}$. We remark that if we applied (✩) to the entire disjunct $Q_{\mathit{fix}} \wedge Q^{\approx}$, the loop on Lines 5–10 might not terminate. Note that, for every $(Q'_{\mathit{fix}}, Q'^{\approx})$ added to $\mathcal{Q}_{\mathit{fin}}$ after applying (✩) to Q_{fix}, $\mathsf{nongens}(Q'_{\mathit{fix}})$ is a proper subset of $\mathsf{nongens}(Q_{\mathit{fix}})$. This entails the termination of the loop on Lines 5–10. Finally, if $[\![Q_{\mathit{fix}}]\!]$ is an infinite set of tuples, then $[\![Q_{\mathit{fix}} \wedge Q^{\approx}]\!]$ is an infinite set of tuples, too. This is because the equalities in $Q^{\approx}$ merely duplicate columns of the query Q_{fix}. Hence, it indeed suffices to apply (✩) to Q_{fix} instead of $Q_{\mathit{fix}} \wedge Q^{\approx}$.

After the loop on Lines 5–10 in Figure 4.3 terminates, for every $(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}$, $Q_{\not\infty}$ is a safe-range query and $Q^{\approx}$ is a conjunction of equalities such that $\mathsf{fv}(Q_{\not\infty} \wedge Q^{\approx}) = \mathsf{fv}(Q)$. However, the query $Q_{\not\infty} \wedge Q^{\approx}$ does not have to be safe-range, e.g., if $Q_{\not\infty} := \mathsf{B}(x)$ and $Q^{\approx} := (x \approx y \wedge u \approx v)$. Given a set of equalities $\mathcal{Q}^{\approx}$, let $\mathsf{classes}(\mathcal{Q}^{\approx})$ be the set of equivalence classes of free variables $\mathsf{fv}(\mathcal{Q}^{\approx})$ with respect to $\mathcal{Q}^{\approx}$. For instance, $\mathsf{classes}(\{x \approx y, y \approx z, u \approx v\}) = \{\{x, y, z\}, \{u, v\}\}$. Let $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) := \bigcup_{V \in \mathsf{classes}(\mathsf{flat}^{\wedge}(Q^{\approx})), V \cap \mathsf{fv}(Q_{\not\infty}) = \emptyset} V$ be the set of all variables in equivalence classes from $\mathsf{classes}(\mathsf{flat}^{\wedge}(Q^{\approx}))$ that are disjoint from $Q_{\not\infty}$'s free variables. Then, $Q_{\not\infty} \wedge Q^{\approx}$ is safe-range if and only if $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) = \emptyset$ (recall the definition of safe-range).

Now if $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) \neq \emptyset$ and $Q_{\not\infty} \wedge Q^{\approx}$ is satisfied by some tuple, then $[\![Q_{\not\infty} \wedge Q^{\approx}]\!]$ is an infinite set of tuples because all equivalence classes of variables in $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) \neq \emptyset$ can be assigned arbitrary values from the infinite domain $\mathcal{D}$. In our example with $Q_{\not\infty} := \mathsf{B}(x)$ and $Q^{\approx} := (x \approx y \wedge u \approx v)$, we have $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) = \{u, v\} \neq \emptyset$. Moreover, if $\mathsf{fv}(Q_{\not\infty} \wedge Q^{\approx}) \neq \mathsf{fv}(Q)$ and $Q_{\not\infty} \wedge Q^{\approx}$ is satisfied by some tuple, then this tuple can be extended to infinitely many tuples over $\mathsf{fv}(Q)$ by choosing arbitrary values from the infinite domain $\mathcal{D}$ for the variables in the non-empty set $\mathsf{fv}(Q) - \mathsf{fv}(Q_{\not\infty} \wedge Q^{\approx})$. Hence, for every $(Q_{\not\infty}, Q^{\approx}) \in \mathcal{Q}_{\mathit{fin}}$ with $\mathsf{disjointvars}(Q_{\not\infty}, Q^{\approx}) \neq \emptyset$ or $\mathsf{fv}(Q_{\not\infty} \wedge Q^{\approx}) \neq \mathsf{fv}(Q)$, we remove $(Q_{\not\infty}, Q^{\approx})$ from $\mathcal{Q}_{\mathit{fin}}$ and add $Q_{\not\infty} \wedge Q^{\approx}$ to $\mathcal{Q}_{\mathit{inf}}$. Note that we only remove pairs from $\mathcal{Q}_{\mathit{fin}}$, hence, the loop on Lines 11–14 terminates. Afterwards, the query Q_{fin} is safe-range. However, the query Q_{inf} does not have to be safe-range. Indeed, every query $Q_{\infty} \in \mathcal{Q}_{\mathit{inf}}$ has range-restricted bound variables, but not all the free variables of Q_{∞} need be range-restricted and thus the query $\exists \vec{\mathsf{fv}}(Q_{\infty}).\, Q_{\infty}$ does not have to be safe-range. But the query Q_{inf} is closed and thus the inf-equivalent query $\mathsf{rb}(Q_{\mathit{inf}})$ with range-restricted bound variables is safe-range.

Lemma 4.6. *Let Q be an RC query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. Then the queries Q_{fin} and Q_{inf} are safe-range; $\mathsf{fv}(Q_{\mathit{fin}}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to false; and $\mathsf{fv}(Q_{\mathit{inf}}) = \emptyset$.*

Lemma 4.7. *Let a structure $\mathcal{S}$ with an infinite domain $\mathcal{D}$ be fixed. Let Q be an RC query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. If $\models Q_{\mathit{inf}}$, then $[\![Q]\!]$ is an infinite set. Otherwise, then $[\![Q]\!] = [\![Q_{\mathit{fin}}]\!]$ is a finite set.*

By Lemma 4.6, Q_{fin} is a safe-range (and thus also domain-independent) query. Hence, for a fixed structure $\mathcal{S}$, the tuples in $[\![Q_{\mathit{fin}}]\!]$ only contain elements in the active domain $\mathsf{adom}(Q_{\mathit{fin}})$, i.e., $[\![Q_{\mathit{fin}}]\!] = [\![Q_{\mathit{fin}}]\!] \cap \mathsf{adom}(Q_{\mathit{fin}})^{|\mathsf{fv}(Q_{\mathit{fin}})|}$. Our translation does not introduce new constants in Q_{fin} and thus $\mathsf{adom}(Q_{\mathit{fin}}) \subseteq \mathsf{adom}(Q)$. Hence, by Lemma 4.7, if $\not\models Q_{\mathit{inf}}$, then $[\![Q_{\mathit{fin}}]\!]$ is equal to the "output-restricted unlimited interpretation" [44] of Q, i.e., $[\![Q_{\mathit{fin}}]\!] = [\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$. In contrast, if $\models Q_{\mathit{inf}}$, then $[\![Q_{\mathit{fin}}]\!] = [\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$ does not necessarily hold. For instance, for $Q := \neg\mathsf{B}(x)$, our translation yields $\mathsf{split}(Q) = (\mathit{false}, \mathit{true})$. In this case, we have $Q_{\mathit{inf}} = \mathit{true}$ and thus $\models Q_{\mathit{inf}}$ because $\neg\mathsf{B}(x)$ is satisfied by infinitely many tuples over an infinite domain. However, if $\mathsf{B}(x)$ is never satisfied, then $[\![Q_{\mathit{fin}}]\!] = \emptyset$ is not equal to $[\![Q]\!] \cap \mathsf{adom}(Q)^{|\mathsf{fv}(Q)|}$.

Example 4.8. Consider the query $Q \coloneqq \mathsf{B}(x) \lor \mathsf{P}(x,y)$. The variable y is not range-restricted in Q and thus $\mathsf{split}(Q)$ restricts y by a conjunction of Q with $\mathsf{P}(x,y)$. However, if $Q[y/\mathit{false}] = \mathsf{B}(x)$ is satisfied by some tuple, then $[\![Q]\!]$ contains infinitely many tuples. Hence, $\mathsf{split}(Q) = ((\mathsf{B}(x) \lor \mathsf{P}(x,y)) \land \mathsf{P}(x,y), \exists x.\, \mathsf{B}(x))$. Because $Q_{\mathit{fin}} = (\mathsf{B}(x) \lor \mathsf{P}(x,y)) \land \mathsf{P}(x,y)$ is only used if $\not\models Q_{\mathit{inf}}$, i.e., if $\mathsf{B}(x)$ is never satisfied, we could simplify Q_{fin} to $\mathsf{P}(x,y)$. However, our translation does not implement such heuristic simplifications. $\diamond$

Example 4.9. Consider the query $Q \coloneqq \mathsf{B}(x) \land u \approx v$. The variables u and v are not range-restricted in Q and thus $\mathsf{split}(Q)$ chooses one of these variables (e.g., u) and restricts it by splitting Q into $Q_{\not\approx} = \mathsf{B}(x)$ and $Q^{\approx} = u \approx v$. Now, all variables are range-restricted in $Q_{\not\approx}$, but the variables in $Q_{\not\approx}$ and $Q^{\approx}$ are disjoint. Hence, $[\![Q]\!]$ contains infinitely many tuples whenever $Q_{\not\approx}$ is satisfied by some tuple. In contrast, $[\![Q]\!] = \emptyset$ if $Q_{\not\approx}$ is never satisfied. Hence, we have $\mathsf{split}(Q) = (\mathit{false}, \exists x.\, \mathsf{B}(x))$. $\diamond$

Example 4.10. Consider the query $Q^{\mathit{susp}}_{\mathit{user}} \coloneqq \mathsf{B}(b) \land \exists s.\, \forall p.\, \mathsf{P}(b,p) \longrightarrow \mathsf{S}(p,u,s)$ from Section 4.1. Restricting its bound variables yields the query $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) = \mathsf{B}(b) \land ((\exists s.\, (\neg\exists p.\, \mathsf{P}(b,p) \land \neg \mathsf{S}(p,u,s)) \land (\exists p.\, \mathsf{S}(p,u,s))) \lor (\neg\exists p.\, \mathsf{P}(b,p)))$ derived in Example 4.4. Splitting $Q^{\mathit{susp}}_{\mathit{user}}$ yields

$$\mathsf{split}(Q^{\mathit{susp}}_{\mathit{user}}) = (\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) \land (\exists s,p.\, \mathsf{S}(p,u,s)), \exists b.\, \mathsf{B}(b) \land \neg\exists p.\, \mathsf{P}(b,p)).$$

To understand $\mathsf{split}(Q^{\mathit{susp}}_{\mathit{user}})$, we apply (★) to $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})$ for the free variable u:

$$\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) \equiv (\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) \land (\exists s,p.\, \mathsf{S}(p,u,s))) \lor (\mathsf{B}(b) \land (\neg\exists p.\, \mathsf{P}(b,p)) \land \neg\exists s,p.\, \mathsf{S}(p,u,s)).$$

If the subquery $\mathsf{B}(b) \land (\neg\exists p.\, \mathsf{P}(b,p))$ from the second disjunct is satisfied for some b, then $Q^{\mathit{susp}}_{\mathit{user}}$ is satisfied by infinitely many values for u from the infinite domain $\mathcal{D}$ that do not belong to the finite interpretation of $\mathsf{S}(p,u,s)$ and thus satisfy the subquery $\neg\exists s,p.\, \mathsf{S}(p,u,s)$. Hence, $[\![Q^{\mathit{susp}}_{\mathit{user}}]\!]^{\mathcal{S}} = [\![\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}})]\!]^{\mathcal{S}}$ is an infinite set of tuples whenever $\mathsf{B}(b) \land \neg\exists p.\, \mathsf{P}(b,p)$ is satisfied for some b. In contrast, if $\mathsf{B}(b) \land \neg\exists p.\, \mathsf{P}(b,p)$ is not satisfied for any b, then $Q^{\mathit{susp}}_{\mathit{user}}$ is equivalent to $\mathsf{rb}(Q^{\mathit{susp}}_{\mathit{user}}) \land (\exists s,p.\, \mathsf{S}(p,u,s))$ obtained also by applying (☆) to $Q^{\mathit{susp}}_{\mathit{user}}$ for the free variable u. $\diamond$

Definition 4.11. *Let Q be a query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. Let $\hat{Q}_{\mathit{fin}} \coloneqq \mathsf{sr2ranf}(Q_{\mathit{fin}})$ and $\hat{Q}_{\mathit{inf}} \coloneqq \mathsf{sr2ranf}(Q_{\mathit{inf}})$ be the equivalent RANF queries. We define $\mathsf{rw}(Q) \coloneqq (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$.*

4.2.4 Complexity Analysis

In this section, we analyze the time complexity of capturing a query Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ if it is finite. To bound the asymptotic time complexity of capturing a fixed query Q, we ignore the (constant) time complexity of computing $\mathsf{rw}(Q) = (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$ and focus on the time complexity of evaluating the RANF queries $\hat{Q}_{\mathit{fin}}$ and $\hat{Q}_{\mathit{inf}}$, i.e., the query cost of $\hat{Q}_{\mathit{fin}}$ and $\hat{Q}_{\mathit{inf}}$. Without loss of generality, we assume that the input query Q has pairwise distinct (free and bound) variables to derive a set of quantified predicates from Q's atomic predicates and formulate our time complexity bound. Still, the RANF queries $\hat{Q}_{\mathit{fin}}$ and $\hat{Q}_{\mathit{inf}}$ computed by our translation do not have to have pairwise distinct (free and bound) variables.

Let $\mathsf{av}(Q)$ be the set of all (free and bound) variables in a query Q. We define the relation $\lesssim_Q$ on $\mathsf{av}(Q)$ such that $x \lesssim_Q y$ iff the scope of an occurrence of $x \in \mathsf{av}(Q)$ is contained in the scope of an occurrence of $y \in \mathsf{av}(Q)$. Formally, we define $x \lesssim_Q y$ iff $y \in \mathsf{fv}(Q)$ or $\exists x.\, Q_x \sqsubseteq \exists y.\, Q_y \sqsubseteq Q$ for some Q_x and Q_y. Note that $\lesssim_Q$ is a preorder on all variables and a partial order on the bound variables for every query with pairwise distinct (free and bound) variables.

Let $\mathsf{aps}(Q)$ be the set of all atomic predicates in a query Q. We denote by $\overline{\mathsf{qps}}(Q)$ the set of quantified predicates obtained from $\mathsf{aps}(Q)$ by performing the variable substitution $x \mapsto y$, where x and y are related by equalities in Q and $x \lesssim_Q y$, and existentially quantifying from a quantified predicate Q_{qp} the innermost bound variable x in Q that is free in Q_{qp}. Let $\mathsf{eqs}^*(Q)$ be the transitive closure of equalities occurring in Q. Formally, we define $\overline{\mathsf{qps}}(Q)$ by:

- $Q_{ap} \in \overline{\mathsf{qps}}(Q)$ if $Q_{ap} \in \mathsf{aps}(Q)$;
- $Q_{qp}[x \mapsto y] \in \overline{\mathsf{qps}}(Q)$ if $Q_{qp} \in \overline{\mathsf{qps}}(Q)$, $(x, y) \in \mathsf{eqs}^*(Q)$, and $x \lesssim_Q y$;
- $\exists x.\, Q_{qp} \in \overline{\mathsf{qps}}(Q)$ if $Q_{qp} \in \overline{\mathsf{qps}}(Q)$, $x \in \mathsf{fv}(Q_{qp}) - \mathsf{fv}(Q)$, and $x \lesssim_Q y$ for all $y \in \mathsf{fv}(Q_{qp})$.

When restricting a variable by a disjunction of quantified predicates $\mathsf{qps}^\vee(\mathcal{G})$ for some $\mathcal{G}$, we only introduce quantified predicates $Q_{qp} \in \overline{\mathsf{qps}}(Q)$.

We bound the time complexity of capturing Q by considering subsets $\mathcal{Q}_{qps}$ of quantified predicates $\overline{\mathsf{qps}}(Q)$ that are *minimal* in the sense that every quantified predicate in $\mathcal{Q}_{qps}$ contains a unique free variable that is not free in any other quantified predicate in $\mathcal{Q}_{qps}$. Formally, we define $\mathsf{minimal}(\mathcal{Q}_{qps}) \coloneqq \forall Q_{qp} \in \mathcal{Q}_{qps}.\, \mathsf{fv}(\mathcal{Q}_{qps} - \{Q_{qp}\}) \neq \mathsf{fv}(\mathcal{Q}_{qps})$. Every minimal subset $\mathcal{Q}_{qps}$ of quantified predicates $\overline{\mathsf{qps}}(Q)$ contributes the product of the numbers of tuples satisfying each quantified predicate $Q_{qp} \in \mathcal{Q}_{qps}$ to the overall bound (that product is an upper bound on the number of tuples satisfying the join over all $Q_{qp} \in \mathcal{Q}_{qps}$). Similarly to Ngo et al. [55], we use the notation $\tilde{\mathcal{O}}(\cdot)$ to hide logarithmic factors incurred by set operations.

Theorem 4.12. *Let Q be a fixed RC query with pairwise distinct (free and bound) variables. The time complexity of capturing Q, i.e., checking if $[\![Q]\!]$ is finite and enumerating $[\![Q]\!]$ if it is finite, is in* $\tilde{\mathcal{O}}\Big(\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|\Big)$.

We prove Theorem 4.12 at the end of this section. Before proving the theorem, we present a few examples. Examples 4.13 and 4.14 show that the time complexity from Theorem 4.12 cannot be achieved by the translation of Van Gelder and Topor [34] or over finite domains. Example 4.15 shows how equalities affect the bound in Theorem 4.12.

Example 4.13. Consider the query $Q \coloneqq \mathsf{B}(b) \wedge \exists u, s.\, \neg\exists p.\, \mathsf{P}(b, p) \wedge \neg\mathsf{S}(p, u, s)$, equivalent to Q^{susp} from Example 1.2. Then $\mathsf{aps}(Q) = \{\mathsf{B}(b), \mathsf{P}(b,p), \mathsf{S}(p,u,s)\}$ and $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(b), \mathsf{P}(b,p), \exists p.\, \mathsf{P}(b,p), \mathsf{S}(p,u,s), \exists p.\, \mathsf{S}(p,u,s), \exists s,p.\, \mathsf{S}(p,u,s), \exists u,s,p.\, \mathsf{S}(p,u,s)\}$. The translated query Q_{vgt} by Van Gelder and Topor [34] restricts the variables r and s by $\exists s,p.\, \mathsf{S}(p,u,s)$ and $\exists u,p.\, \mathsf{S}(p,u,s)$, respectively. For an interpretation of B by $\{(\mathsf{c}') \mid \mathsf{c}' \in \{1,\dots,m\}\}$, P by $\{(\mathsf{c}',\mathsf{c}') \mid \mathsf{c}' \in \{1,\dots,m\}\}$, and S by $\{(\mathsf{c},\mathsf{c}',\mathsf{c}') \mid \mathsf{c} \in \{1,\dots,n\}, \mathsf{c}' \in \{1,\dots,m\}\}$, $n, m \in \mathbb{N}$, computing the join of $\mathsf{P}(b,p)$, $\exists s,p.\, \mathsf{S}(p,u,s)$, and $\exists u,p.\, \mathsf{S}(p,u,s)$, which is a Cartesian product, results in a time complexity in $\Omega(n \cdot m^2)$ for Q_{vgt}. In contrast, Theorem 4.12 yields an asymptotically better time complexity in $\tilde{\mathcal{O}}(n + m + n \cdot m)$ for our translation:

$$\tilde{\mathcal{O}}(|[\![\mathsf{B}(b)]\!]| + |[\![\mathsf{P}(b,p)]\!]| + |[\![\mathsf{S}(p,u,s)]\!]| + (|[\![\mathsf{B}(b)]\!]| + |[\![\mathsf{P}(b,p)]\!]|) \cdot |[\![\mathsf{S}(p,u,s)]\!]|). \qquad \diamond$$

Example 4.14. The query $\neg\mathsf{S}(x,y,z)$ is satisfied by a finite set of tuples over a finite domain $\mathcal{D}$ (as is every query over a finite domain). For an interpretation of S by $\{(\mathsf{c},\mathsf{c},\mathsf{c}) \mid \mathsf{c} \in \mathcal{D}\}$, the equality $|\mathcal{D}| = |[\![\mathsf{S}(x,y,z)]\!]|$ holds and the number of satisfying tuples is

$$|[\![\neg\mathsf{S}(x,y,z)]\!]| = |\mathcal{D}|^3 - |[\![\mathsf{S}(x,y,z)]\!]| = |[\![\mathsf{S}(x,y,z)]\!]|^3 - |[\![\mathsf{S}(x,y,z)]\!]| \in \Omega(|[\![\mathsf{S}(x,y,z)]\!]|^3),$$

which exceeds the bound $\tilde{\mathcal{O}}(|[\![\mathsf{S}(x,y,z)]\!]|)$ of Theorem 4.12. Hence, our infinite domain assumption is crucial for achieving the better complexity bound. $\diamond$

Example 4.15. Consider the following query over the infinite domain $\mathcal{D} = \mathbb{N}$ of natural numbers:

$$\begin{aligned} Q \coloneqq\ & \forall u.\, (u \approx 0 \lor u \approx 1 \lor u \approx 2) \longrightarrow \\ & \quad (\exists v.\, \mathsf{B}(v) \land (u \approx 0 \longrightarrow x \approx v) \land (u \approx 1 \longrightarrow y \approx v) \land (u \approx 2 \longrightarrow z \approx v)). \end{aligned}$$

Note that this query is equivalent to $Q \equiv \mathsf{B}(x) \land \mathsf{B}(y) \land \mathsf{B}(z)$ and thus it is satisfied by a finite set of tuples of size $|[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^3$. The set of atomic predicates of Q is $\mathsf{aps}(Q) = \{\mathsf{B}(v)\}$ and it must be closed under the equalities occurring in Q to yield a valid bound in Theorem 4.12. In this case, $\overline{\mathsf{qps}}(Q) = \{\mathsf{B}(v), \exists v.\, \mathsf{B}(v), \mathsf{B}(x), \mathsf{B}(y), \mathsf{B}(z)\}$ and the bound in Theorem 4.12 is $|[\![\mathsf{B}(v)]\!]| \cdot |[\![\mathsf{B}(x)]\!]| \cdot |[\![\mathsf{B}(y)]\!]| \cdot |[\![\mathsf{B}(z)]\!]| = |[\![\mathsf{B}(x)]\!]|^4$. In particular, this bound is not tight, but it still reflects the complexity of evaluating the RANF queries produced by our translation as our translation does not derive the equivalence $Q \equiv \mathsf{B}(x) \land \mathsf{B}(y) \land \mathsf{B}(z)$. ◇

Guard Query Given a RANF query $\hat{Q}$, we define a *guard* query $\mathsf{guard}(\hat{Q})$ that is implied by $\hat{Q}$, i.e., $\mathsf{guard}(\hat{Q})$ can be used to over-approximate the set of satisfying tuples for $\hat{Q}$. We use this over-approximation in our proof of Theorem 4.12 and also in our algorithm for monitoring safe-range MFOTL queries (Section 4.3). The guard query $\mathsf{guard}(\hat{Q})$ has a simple structure: it is the disjunction of conjunctions of quantified predicates and equalities.

We now define the set of quantified predicates $\mathsf{qps}(Q)$ occurring in the guard query $\mathsf{guard}(Q)$. For an atomic predicate $Q_{ap} \in \mathsf{aps}(Q)$, let $\mathcal{B}_Q(Q_{ap})$ be the set of sequences of bound variables for all occurrences of Q_{ap} in Q. For example, let $Q_{ex} \coloneqq ((\exists z.\, (\exists y, z.\, \mathsf{P}_3(x, y, z)) \land \mathsf{P}_2(y, z)) \land \mathsf{P}_1(z)) \lor \mathsf{P}_3(x, y, z)$. Then $\mathsf{aps}(Q_{ex}) = \{\mathsf{P}_1(z), \mathsf{P}_2(y, z), \mathsf{P}_3(x, y, z)\}$ and $\mathcal{B}_{Q_{ex}}(\mathsf{P}_3(x, y, z)) = \{[y, z], []\}$, where $[]$ denotes the empty sequence corresponding to the occurrence of $\mathsf{P}_3(x, y, z)$ in Q_{ex} for which the variables x, y, z are all free in Q_{ex}. Note that the variable z in the other occurrence of $\mathsf{P}_3(x, y, z)$ in Q_{ex} is bound to the innermost quantifier. Hence, neither $[z, y]$ nor $[z, y, z]$ are in $\mathcal{B}_{Q_{ex}}(\mathsf{P}_3(x, y, z))$. Furthermore, let $\mathsf{qps}(Q)$ be the set of the quantified predicates obtained by existentially quantifying sequences of bound variables in $\mathcal{B}_{Q'}(Q_{ap})$ from the atomic predicates $Q_{ap} \in \mathsf{aps}(Q')$ in all subqueries Q' of Q. Formally, $\mathsf{qps}(Q) \coloneqq \bigcup_{Q' \sqsubseteq Q, Q_{ap} \in \mathsf{aps}(Q')} \{\exists \vec{v}.\, Q_{ap} \mid \vec{v} \in \mathcal{B}_{Q'}(Q_{ap})\}$. For instance, $\mathsf{qps}(Q_{ex}) = \{\mathsf{P}_3(x, y, z), \exists z.\, \mathsf{P}_3(x, y, z), \exists yz.\, \mathsf{P}_3(x, y, z), \mathsf{P}_2(y, z), \exists z.\, \mathsf{P}_2(y, z), \mathsf{P}_1(z)\}$.

Let a structure $\mathcal{S}$ be fixed. We observe that every tuple satisfying a RANF query $\hat{Q}$ belongs to the set of tuples satisfying the join over some minimal subset $\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q})$ of quantified predicates and also satisfying equalities duplicating some columns from $\mathcal{Q}_{qps}$. Hence, we define the guard query $\mathsf{guard}(\hat{Q})$ as follows:

$$\mathsf{guard}(\hat{Q}) \coloneqq \bigvee_{\substack{\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q}), \mathsf{minimal}(\mathcal{Q}_{qps}), \\ \mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \land y \in \mathsf{fv}(\hat{Q})\}, \\ \mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q})}} \left(\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \land \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx} \right).$$

Note that $\{x \approx y \mid x \in V \land y \in V'\}$ denotes the set of all equalities $x \approx y$ between variables $x \in V$ and $y \in V'$. We express the correctness of the guard query in the following lemma.

Lemma 4.16. *Let $\hat{Q}$ be a RANF query. Then, for all variable assignments α,*

$$\alpha \models \hat{Q} \Longrightarrow \alpha \models \mathsf{guard}(\hat{Q})$$

holds. Moreover, $\mathsf{fv}(\mathsf{guard}(\hat{Q})) = \mathsf{fv}(\hat{Q})$ *unless* $\mathsf{guard}(\hat{Q}) = \mathit{false}$. *Hence,* $[\![\hat{Q}]\!]$ *satisfies*

$$[\![\hat{Q}]\!] \subseteq [\![\mathsf{guard}(\hat{Q})]\!].$$

Proof. The statement is proved by well-founded induction over the inductive definition of $\mathsf{ranf}(\hat{Q})$. □

Proof of Theorem 4.12 A crucial property of our translation that is central for the proof of Theorem 4.12 is a relationship between the quantified predicates $\mathsf{qps}(\hat{Q})$ for a RANF query $\hat{Q}$ produced by our translation and the original query Q. The relationship is formalized in the following lemma.

Lemma 4.17. *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$. Let $\hat{Q} \in \{\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}}\}$. Then $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$.*

Proof. Let $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. We observe that $\mathsf{aps}(Q_{\mathit{fin}}) \subseteq \overline{\mathsf{qps}}(Q)$, $\mathsf{eqs}^*(Q_{\mathit{fin}}) \subseteq \mathsf{eqs}^*(Q)$, $\lesssim_{Q_{\mathit{fin}}} \subseteq \lesssim_Q$, $\mathsf{aps}(Q_{\mathit{inf}}) \subseteq \overline{\mathsf{qps}}(Q)$, $\mathsf{eqs}^*(Q_{\mathit{inf}}) \subseteq \mathsf{eqs}^*(Q)$, and $\lesssim_{Q_{\mathit{inf}}} \subseteq \lesssim_Q$. Hence, $\overline{\mathsf{qps}}(Q_{\mathit{fin}}) \subseteq \overline{\mathsf{qps}}(Q)$ and $\overline{\mathsf{qps}}(Q_{\mathit{inf}}) \subseteq \overline{\mathsf{qps}}(Q)$.

Next we observe that $\mathsf{qps}(Q') \subseteq \overline{\mathsf{qps}}(Q')$ for every query Q'. Finally, we show that $\mathsf{qps}(\hat{Q}_{\mathit{fin}}) \subseteq \mathsf{qps}(Q_{\mathit{fin}})$ and $\mathsf{qps}(\hat{Q}_{\mathit{inf}}) \subseteq \mathsf{qps}(Q_{\mathit{inf}})$. We observe that $\mathcal{B}_{\mathsf{cp}(Q')}(Q_{ap}) \subseteq \mathcal{B}_{Q'}(Q_{ap})$, $\mathcal{B}_{\mathsf{srnf}(Q')}(Q_{ap}) \subseteq \mathcal{B}_{Q'}(Q_{ap})$, and then $\mathsf{qps}(\mathsf{cp}(Q')) \subseteq \mathsf{qps}(Q')$, $\mathsf{qps}(\mathsf{srnf}(Q')) \subseteq \mathsf{qps}(Q')$, for every query Q'.

Suppose that $Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ is a safe-range query in which no variable occurs both free and bound, no bound variables shadow each other, i.e., there are no subqueries $\exists x.\, Q_x \sqsubseteq Q'_x$ and $\exists x.\, Q'_x \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$, and every two subqueries $\exists x.\, Q_x \sqsubseteq Q_1$ and $\exists x.\, Q'_x \sqsubseteq Q_2$ such that $Q_1 \wedge Q_2 \sqsubseteq Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ have the property that $\exists x.\, Q_x$ or $\exists x.\, Q'_x$ is a quantified predicate. Then the free variables in $\bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ never clash with the bound variables in Q', i.e., Line 24 in Figure 2.16 is never executed. Next we observe that $\mathcal{B}_{\mathsf{sr2ranf}(Q',\mathcal{Q})}(Q_{ap}) \subseteq \mathcal{B}_{Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}}(Q_{ap})$ and then $\mathsf{qps}(\mathsf{sr2ranf}(Q', \mathcal{Q})) \subseteq \mathsf{qps}(Q' \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q})$. Because Q_{fin}, Q_{inf} have the properties from the beginning of this paragraph and $\mathsf{qps}(\mathsf{srnf}(Q')) \subseteq \mathsf{qps}(Q')$, for every query Q', we get $\mathsf{qps}(\hat{Q}_{\mathit{fin}}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{\mathit{fin}})) \subseteq \mathsf{qps}(Q_{\mathit{fin}})$ and $\mathsf{qps}(\hat{Q}_{\mathit{inf}}) = \mathsf{qps}(\mathsf{sr2ranf}(Q_{\mathit{inf}})) \subseteq \mathsf{qps}(Q_{\mathit{inf}})$. □

Recall Example 4.13. The query $\exists u, p.\, \mathsf{S}(p, u, s)$ is in $\mathsf{qps}(Q_{vgt})$, but not in $\overline{\mathsf{qps}}(Q)$. Hence, $\mathsf{qps}(Q_{vgt}) \subseteq \overline{\mathsf{qps}}(Q)$, i.e., an analogue of Lemma 4.17 for Van Gelder and Topor's translation, does not hold.

We now derive a bound on $\left|\left[\!\left[\hat{Q}'\right]\!\right]\right|$, for an arbitrary RANF subquery $\hat{Q}' \sqsubseteq \hat{Q}$, $\hat{Q} \in \{\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}}\}$.

Lemma 4.18. *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$. Let $\hat{Q}' \sqsubseteq \hat{Q}$ be a RANF subquery of $\hat{Q} \in \{\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}}\}$. Then*

$$\left|\left[\!\left[\hat{Q}'\right]\!\right]\right| \leq \sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} 2^{|\mathsf{av}(\hat{Q})|} \cdot \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|.$$

Proof. Applying Lemma 4.16 to the RANF query $\hat{Q}'$ yields

$$\left[\!\left[\hat{Q}'\right]\!\right] \subseteq \left[\!\left[\mathsf{guard}(\hat{Q}')\right]\!\right] = \bigcup_{\substack{\mathcal{Q}_{qps} \subseteq \mathsf{qps}(\hat{Q}'), \mathsf{minimal}(\mathcal{Q}_{qps}), \\ \mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\}, \\ \mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q}')}} \left[\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx}\right]\!\right].$$

We observe that $\left|\left[\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp} \wedge \bigwedge_{Q^{\approx} \in \mathcal{Q}^{\approx}} Q^{\approx}\right]\!\right]\right| \leq \left|\left[\!\left[\bigwedge_{Q_{qp} \in \mathcal{Q}_{qps}} Q_{qp}\right]\!\right]\right| \leq \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|$ where the first inequality follows from the fact that equalities $Q^{\approx} \in \mathcal{Q}^{\approx}$ can only restrict a set of tuples and duplicate columns. Because $\hat{Q}'$ is a subquery of $\hat{Q}$, it follows that $\mathsf{qps}(\hat{Q}') \subseteq \mathsf{qps}(\hat{Q})$. Lemma 4.17 yields $\mathsf{qps}(\hat{Q}) \subseteq \overline{\mathsf{qps}}(Q)$. Hence, we derive $\mathsf{qps}(\hat{Q}') \subseteq \overline{\mathsf{qps}}(Q)$.

The number of equalities in $\{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\}$ is at most

$$|\mathsf{fv}(\mathcal{Q}_{qps})| \cdot \left|\mathsf{fv}(\hat{Q}')\right| \le \left|\mathsf{fv}(\hat{Q}')\right|^2 \le \left|\mathsf{av}(\hat{Q})\right|^2,$$

where the first inequality holds because $\mathsf{fv}(\mathcal{Q}_{qps}) \cup \mathsf{fv}(\mathcal{Q}^{\approx}) = \mathsf{fv}(\hat{Q}')$ and thus $\mathsf{fv}(\mathcal{Q}_{qps}) \subseteq \mathsf{fv}(\hat{Q}')$ and the second inequality holds because the variables in a subquery $\hat{Q}'$ of $\hat{Q}$ are included in the set of all variables in $\hat{Q}$. Hence, the number of subsets $\mathcal{Q}^{\approx} \subseteq \{x \approx y \mid x \in \mathsf{fv}(\mathcal{Q}_{qps}) \wedge y \in \mathsf{fv}(\hat{Q}')\}$ is at most $2^{\left|\mathsf{av}(\hat{Q})\right|^2}$. □

Next we bound the query cost of a RANF query $\hat{Q} \in \{\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}}\}$ over the structure $\mathcal{S}$.

Lemma 4.19. *Let Q be an RC query with pairwise distinct (free and bound) variables and let $\mathsf{rw}(Q) = (\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}})$. Let $\hat{Q} \in \{\hat{Q}_{\mathit{fin}}, \hat{Q}_{\mathit{inf}}\}$. Then*

$$\mathsf{cost}^{\mathcal{S}}(\hat{Q}) \le \left|\mathsf{sub}(\hat{Q})\right| \cdot \left|\mathsf{av}(\hat{Q})\right| \cdot 2^{\left|\mathsf{av}(\hat{Q})\right|} \cdot \textstyle\sum_{\mathcal{Q}_{qps} \subseteq \overline{\mathsf{qps}}(Q), \mathsf{minimal}(\mathcal{Q}_{qps})} \prod_{Q_{qp} \in \mathcal{Q}_{qps}} |[\![Q_{qp}]\!]|.$$

Proof. Recall that $\left|\mathsf{sub}(\hat{Q})\right|$ denotes the number of subqueries of the query $\hat{Q}$ and thus bounds the number of RANF subqueries $\hat{Q}'$ of the query $\hat{Q}$. For every subquery $\hat{Q}'$ of $\hat{Q}$, we first use the fact that $\left|\mathsf{fv}(\hat{Q}')\right| \le \left|\mathsf{av}(\hat{Q})\right|$ to bound $\left|[\![\hat{Q}']\!]\right| \cdot \left|\mathsf{fv}(\hat{Q}')\right| \le \left|[\![\hat{Q}']\!]\right| \cdot \left|\mathsf{av}(\hat{Q})\right|$. Then we use the estimation of $\left|[\![\hat{Q}']\!]\right|$ by Lemma 4.18. □

Finally, we prove Theorem 4.12.

Proof (Theorem 4.12). We derive Theorem 4.12 from Lemma 4.19 and the fact that the quantities $\left|\mathsf{sub}(\hat{Q})\right|$, $\left|\mathsf{av}(\hat{Q})\right|$, and $2^{\left|\mathsf{av}(\hat{Q})\right|^2}$ only depend on the query Q and thus they do not contribute to the asymptotic time complexity of capturing a fixed query Q. □

4.2.5 Data Golf

In this section, we devise the *Data Golf* benchmark for generating structures used in our empirical evaluation (Section 4.2.6). Given an RC query, we seek a structure that results in a nontrivial evaluation result for the overall query and for all its subqueries. Intuitively, the resulting structure makes query evaluation potentially more challenging compared to the case where some subquery results in a trivial (e.g., empty) evaluation result. More specifically, Data Golf has two objectives. The first resembles the *regex golf* game's objective [26] (hence the name) and aims to find a structure on which the result of a given query contains a given *positive* set of tuples and does not contain any tuples from another given *negative* set. The second objective is to ensure that all the query's subqueries evaluate to a non-trivial result.

Formally, given a query Q and two sets of tuples $\mathcal{T}^+$ and $\mathcal{T}^-$ over a fixed domain $\mathcal{D}$, representing assignments of $\mathsf{av}(Q)$ and satisfying further assumptions on their values, Data Golf produces a structure $\mathcal{S}$ (represented as a partial mapping from predicate symbols to their interpretations) such that the projections of tuples in $\mathcal{T}^+$ ($\mathcal{T}^-$) to $\vec{\mathsf{fv}}(Q)$ are in $[\![Q]\!]$ (disjoint from $[\![Q]\!]$) and $|[\![Q']\!]|$ and $|[\![\neg Q']\!]|$ contain at least $\min\{|\mathcal{T}^+|, |\mathcal{T}^-|\}$ tuples, for every $Q' \sqsubseteq Q$. To be able to produce such a structure $\mathcal{S}$, we make the following assumptions on Q:

```
input:  An RC query $Q$ satisfying CON, CST, VAR, REP, a sequence of pairwise distinct
        variables $\vec{v}$, $\mathsf{av}(Q) \subseteq \vec{v}$, sets of tuples $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ over $\vec{v}$ such that all values of
        variables from $\mathsf{av}(Q)$ in these tuples are pairwise distinct (also across tuples)
        except that, in every tuple in $\mathcal{T}^+_{\vec{v}}$ ($\mathcal{T}^-_{\vec{v}}$), the variables in $\mathcal{V}^+$ ($\mathcal{V}^-$) have the same
        value (which is different across tuples), where $\mathsf{dg}^{\approx}(Q,\gamma) = (\mathcal{V}^+,\mathcal{V}^-)$, $\gamma \in \{0,1\}$.
output: A structure $\mathcal{S}$ such that $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$, $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$, and $|[\![Q']\!]|$ and
        $|[\![\neg Q']\!]|$ contain at least $\min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ tuples, for every $Q' \sqsubseteq Q$.
1  function $\mathsf{dg}^{\approx}(Q,\gamma) =$
2      switch $Q$ do
3          case $r(t_1,\dots,t_{\iota(r)})$ do return $(\emptyset,\emptyset)$;
4          case $x \approx y$ do return $(\{x,y\},\emptyset)$;
5          case $\neg Q'$ do
6              $(\mathcal{V}^+,\mathcal{V}^-) := \mathsf{dg}^{\approx}(Q',\gamma)$;
7              return $(\mathcal{V}^-,\mathcal{V}^+)$;
8          case $Q'_1 \lor Q'_2$ or $Q'_1 \land Q'_2$ do
9              $(\mathcal{V}^+_1,\mathcal{V}^-_1) := \mathsf{dg}^{\approx}(Q'_1,\gamma)$;
10             $(\mathcal{V}^+_2,\mathcal{V}^-_2) := \mathsf{dg}^{\approx}(Q'_2,\gamma)$;
11             if $\gamma = 0$ then return $(\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^-_1 \cup \mathcal{V}^-_2)$;
12             else if $Q = Q'_1 \lor Q'_2$ then return $(\mathcal{V}^+_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^-_2)$;
13             else if $Q = Q'_1 \land Q'_2$ then return $(\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^+_1 \cup \mathcal{V}^-_2)$;
14         case $\exists y.\, Q_y$ do return $\mathsf{dg}^{\approx}(Q_y,\gamma)$;
15 function $\mathsf{dg}(Q,\vec{v},\mathcal{T}^+_{\vec{v}},\mathcal{T}^-_{\vec{v}},\gamma) =$
16     switch $Q$ do
17         case $r(t_1,\dots,t_{\iota(r)})$ do return $\{r^{\mathcal{S}} \mapsto \mathcal{T}^+_{\vec{v}}[t_1,\dots,t_{\iota(r)}]\}$;
18         case $x \approx y$ do return $\emptyset$;
19         case $\neg Q'$ do return $\mathsf{dg}(Q',\vec{v},\mathcal{T}^-_{\vec{v}},\mathcal{T}^+_{\vec{v}},\gamma)$;
20         case $Q'_1 \lor Q'_2$ or $Q'_1 \land Q'_2$ do
21             $(\mathcal{V}^+_1,\mathcal{V}^-_1) := \mathsf{dg}^{\approx}(Q'_1,\gamma)$; $(\mathcal{V}^+_2,\mathcal{V}^-_2) := \mathsf{dg}^{\approx}(Q'_2,\gamma)$;
22             if $\gamma = 0$ then $(\mathcal{V}^1,\mathcal{V}^2) := (\mathcal{V}^+_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
23             else if $Q = Q'_1 \land Q'_2$ then $(\mathcal{V}^1,\mathcal{V}^2) := (\mathcal{V}^-_1 \cup \mathcal{V}^-_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
24             else if $Q = Q'_1 \lor Q'_2$ then $(\mathcal{V}^1,\mathcal{V}^2) := (\mathcal{V}^+_1 \cup \mathcal{V}^+_2, \mathcal{V}^-_1 \cup \mathcal{V}^+_2)$;
25             $(\mathcal{T}^1_{\vec{v}},\mathcal{T}^2_{\vec{v}}) \leftarrow \{(\mathcal{T}^1_{\vec{v}},\mathcal{T}^2_{\vec{v}}) \mid |\mathcal{T}^1_{\vec{v}}| = |\mathcal{T}^2_{\vec{v}}| = \min\{\left|\mathcal{T}^+_{\vec{v}}\right|,\left|\mathcal{T}^-_{\vec{v}}\right|\}$, all values in tuples in
                $\mathcal{T}^+_{\vec{v}},\mathcal{T}^-_{\vec{v}},\mathcal{T}^1_{\vec{v}},\mathcal{T}^2_{\vec{v}}$ are pairwise distinct (also across tuples) except that,
                in every tuple in $\mathcal{T}^1_{\vec{v}}$ ($\mathcal{T}^2_{\vec{v}}$), the variables in $\mathcal{V}^1$ ($\mathcal{V}^2$) have the same value
                (which is different across tuples)$\}$;
26             if $\gamma = 0$ then return
                $\mathsf{dg}(Q'_1,\vec{v},\mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}},\mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\gamma) \cup \mathsf{dg}(Q'_2,\vec{v},\mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}},\gamma)$;
27             else if $Q = Q'_1 \lor Q'_2$ then return
                $\mathsf{dg}(Q'_1,\vec{v},\mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}},\mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\gamma) \cup \mathsf{dg}(Q'_2,\vec{v},\mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^+_{\vec{v}},\gamma)$;
28             else if $Q = Q'_1 \land Q'_2$ then return
                $\mathsf{dg}(Q'_1,\vec{v},\mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^-_{\vec{v}},\mathcal{T}^1_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\gamma) \cup \mathsf{dg}(Q'_2,\vec{v},\mathcal{T}^+_{\vec{v}} \cup \mathcal{T}^2_{\vec{v}},\mathcal{T}^-_{\vec{v}} \cup \mathcal{T}^1_{\vec{v}},\gamma)$;
29         case $\exists y.\, Q_y$ do return $\mathsf{dg}(Q_y,\vec{v},\mathcal{T}^+_{\vec{v}},\mathcal{T}^-_{\vec{v}},\gamma)$;
```

Figure 4.4. Computing the Data Golf structure.

CON the bound variable y in every subquery $\exists y.\, Q_y$ of Q satisfies $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{G})$ (Figure 2.11) for some set $\mathcal{G}$ such that $\mathsf{eqs}(y, \mathcal{G}) = \emptyset$ and, for every $Q_{qp} \in \mathcal{G}$, $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$ holds; this avoids subqueries like $\exists y.\, \neg \mathsf{P}_2(x, y)$ and $\exists y.\, (\mathsf{P}_2(x, y) \vee \mathsf{P}_1(y))$;

CST Q contains no subquery of the form $x \approx \mathsf{c}$, which is satisfied by exactly one tuple;

VAR Q contains no closed subqueries, e.g., $\mathsf{P}_1(42)$, because a closed subquery is either satisfied by all possible tuples or no tuple at all; and

REP Q contains no repeated predicate symbols and no equalities $x \approx y$ in Q share a variable; this avoids subqueries like $\mathsf{P}_1(x) \wedge \neg \mathsf{P}_1(x)$ and $x \approx y \wedge \neg x \approx y$.

Given a sequence of pairwise distinct variables $\vec{v}$ and a tuple $\vec{d}$ of the same length, we may interpret the tuple $\vec{d}$ as a *tuple over* $\vec{v}$, denoted as $\vec{d}(\vec{v})$. Given a sequence $t_1, \ldots, t_k \in \vec{v} \cup \mathcal{C}$ of terms, we denote by $\vec{d}(\vec{v})[t_1, \ldots, t_k]$ the tuple obtained by evaluating the terms $t_1, \ldots, t_k$ over $\vec{d}(\vec{v})$. Formally, we define $\vec{d}(\vec{v})[t_1, \ldots, t_k] \coloneqq (d'_i)_{i=1}^k$, where $d'_i = \vec{d}_j$ if $t_i = \vec{v}_j$ and $d'_i = t_i$ if $t_i \in \mathcal{C}$. We lift this notion to sets of tuples over $\vec{v}$ in the standard way.

Data Golf is formalized by the function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$, defined in Figure 4.4, where $\vec{v}$ is a sequence of pairwise distinct variables containing all variables in Q, i.e., $\mathsf{av}(Q) \subseteq \vec{v}$, $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ are sets of tuples over $\vec{v}$, and $\gamma \in \{0, 1\}$ is a *strategy*. In the case of a conjunction or a disjunction, we add disjoint sets $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$ of tuples over $\vec{v}$ to $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ so that the intermediate results for the subqueries are neither equal nor disjoint. We implement two strategies (parameter γ) to choose these sets $\mathcal{T}^1_{\vec{v}}$, $\mathcal{T}^2_{\vec{v}}$. To reflect Q's equalities in the sets $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$, given a strategy γ, we define the function $\mathsf{dg}^{\approx}(Q, \gamma) = (\mathcal{V}^+, \mathcal{V}^-)$ that computes two sets of variables $\mathcal{V}^+$ and $\mathcal{V}^-$ whose values must be equal in every tuple in $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$, respectively. The values of the remaining variables ($\vec{v} - \mathcal{V}^+$ and $\vec{v} - \mathcal{V}^-$, respectively) must be pairwise distinct and also different from the value of the variables in $\mathcal{V}^+$ and $\mathcal{V}^-$, respectively.

Finally, we justify why a Data Golf structure $\mathcal{S}$ computed by $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ satisfies $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$. We proceed by induction on the query Q. Because of REP, the Data Golf structures for the subqueries Q_1, Q_2 of a binary query $Q_1 \vee Q_2$ or $Q_1 \wedge Q_2$ can be combined using the union operator. The only case that does not follow immediately is that $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$ for a query Q of the form $\exists y.\, Q_y$. We prove this case by contradiction. Without loss of generality we assume that $\vec{\mathsf{fv}}(Q_y) = \vec{\mathsf{fv}}(Q) \cdot y$. Suppose that $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$ and $\vec{d} \in [\![Q]\!]$. Because $\vec{d} \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)]$, there exists some d such that $\vec{d} \cdot d \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \in [\![Q]\!]$, there exists some d' such that $\vec{d} \cdot d' \in [\![Q_y]\!]$. By the induction hypothesis, $\vec{d} \cdot d \notin [\![Q_y]\!]$ and $\vec{d} \cdot d' \notin \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\mathsf{con}_{\mathsf{vgt}}(y, Q_y, \mathcal{G})$ holds for some $\mathcal{G}$ satisfying CON, the query Q_y is equivalent to $(Q_y \wedge \mathsf{qps}^{\vee}(\mathcal{G})) \vee Q_y[y/\bot]$. We have $\vec{d} \cdot d' \in [\![Q_y]\!]$. If the tuple $\vec{d} \cdot d'$ satisfies $Q_y[y/\bot]$, then $\vec{d} \cdot d \in [\![Q_y]\!]$ (contradiction) because the variable y does not occur in the query $Q_y[y/\bot]$ and thus its assignment in $\vec{d} \cdot d'$ can be arbitrarily changed. Otherwise, the tuple $\vec{d} \cdot d'$ satisfies some quantified predicate $Q_{qp} \in \mathsf{qps}(\mathcal{G})$ and (CON) implies $\{y\} \subsetneq \mathsf{fv}(Q_{qp})$. Hence, the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the assignment of a variable $x \in \mathsf{fv}(Q_{qp}) - \{y\}$. Let $\overline{\mathcal{T}}^+_{\vec{v}}$ and $\overline{\mathcal{T}}^-_{\vec{v}}$ be the sets in the recursive call of dg on the atomic predicate from Q_{qp}. Because $\vec{d} \cdot d \in \mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \subseteq \overline{\mathcal{T}}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \cup \overline{\mathcal{T}}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$, the tuple $\vec{d} \cdot d$ is in $\overline{\mathcal{T}}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)] \cup \overline{\mathcal{T}}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Because $\vec{d} \cdot d'$ satisfies the quantified predicate Q_{qp}, the tuple $\vec{d} \cdot d'$ is in $\overline{\mathcal{T}}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q_y)]$. Next we observe that the assignments of every variable (in particular, x) in the tuples from the sets $\overline{\mathcal{T}}^+_{\vec{v}}$, $\overline{\mathcal{T}}^-_{\vec{v}}$ are pairwise distinct (there can only be equal values of variables within one tuple). Because the tuples $\vec{d} \cdot d$ and $\vec{d} \cdot d'$ agree on the assignment of x, they must be equal, i.e., $\vec{d} \cdot d = \vec{d} \cdot d'$ (contradiction).

The sets $\mathcal{T}^+_{\vec{v}}$, $\mathcal{T}^-_{\vec{v}}$ only grow in dg's recursion and the properties CON, CST, VAR, REP imply that Q has no closed subquery. Hence, $\mathcal{T}^+_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \subseteq [\![Q]\!]$ and $\mathcal{T}^-_{\vec{v}}[\vec{\mathsf{fv}}(Q)] \cap [\![Q]\!] = \emptyset$ imply that $|[\![Q']\!]|$ and $|[\![\neg Q']\!]|$ contain at least $\min\{\left|\mathcal{T}^+_{\vec{v}}\right|, \left|\mathcal{T}^-_{\vec{v}}\right|\}$ tuples, for every $Q' \sqsubseteq Q$.

Example 4.20. Consider the query $Q \coloneqq \neg\exists y.\, \mathsf{P}_2(x, y) \wedge \neg \mathsf{P}_3(x, y, z)$. This query Q satisfies the assumptions CON, CST, VAR, REP. In particular, $\mathsf{con}_{\mathsf{vgt}}(y, \mathsf{P}_2(x, y) \wedge \neg \mathsf{P}_3(x, y, z), \mathcal{G})$ holds for $\mathcal{G} = \{\mathsf{P}_2(x, y)\}$ with $\{y\} \subsetneq \mathsf{fv}(\mathsf{P}_2(x, y))$. We choose $\vec{v} = (x, z, y)$, $\mathcal{T}^+_{\vec{v}} = \{(0, 4, 8), (2, 6, 10)\}$, and $\mathcal{T}^-_{\vec{v}} = \{(12, 16, 20), (14, 18, 22)\}$. The function $\mathsf{dg}(Q, \vec{v}, \mathcal{T}^+_{\vec{v}}, \mathcal{T}^-_{\vec{v}}, \gamma)$ first flips $\mathcal{T}^+_{\vec{v}}$ and $\mathcal{T}^-_{\vec{v}}$ because Q's main operator is negation. For conjunction (a binary operator), two additional sets of tuples are computed: $\mathcal{T}^1_{\vec{v}} = \{(24, 28, 32), (26, 30, 34)\}$ and $\mathcal{T}^2_{\vec{v}} = \{(36, 40, 44), (38, 42, 46)\}$. Depending on the strategy ($\gamma = 0$ or $\gamma = 1$), one of the following structures is computed: $\mathcal{S}_0 = \{\mathsf{P}_2 \mapsto \{(12, 20), (14, 22), (24, 32), (26, 34)\}, \mathsf{P}_3 \mapsto \mathcal{T}^+_{xyz}\}$, or $\mathcal{S}_1 = \{\mathsf{P}_2 \mapsto \{(12, 20), (14, 22), (0, 8), (2, 10)\}, \mathsf{P}_3 \mapsto \mathcal{T}^+_{xyz}\}$, where $\mathcal{T}^+_{xyz} = \{(0, 8, 4), (2, 10, 6), (24, 32, 28), (26, 34, 30)\}$.

The query $\mathsf{P}_1(x) \wedge Q$ is satisfied by the finite set of tuples $\mathcal{T}^+_{\vec{v}}[x, z]$ under the structure $\mathcal{S}_1 \cup \{\mathsf{P}_1 \mapsto \{(0), (2)\}\}$ obtained by extending $\mathcal{S}_1$ ($\gamma = 1$). In contrast, the same query $\mathsf{P}_1(x) \wedge Q$ is satisfied by an infinite set of tuples including $\mathcal{T}^+_{\vec{v}}[x, z]$ and disjoint from $\mathcal{T}^-_{\vec{v}}[x, z]$ under the structure $\mathcal{S}_0 \cup \{\mathsf{P}_1 \mapsto \{(0), (2)\}\}$ obtained by extending $\mathcal{S}_0$ ($\gamma = 0$). $\diamond$

4.2.6 Implementation

We have implemented our translation RC2SQL consisting of roughly 1000 lines of OCaml code [60]. Overall, the translation is defined as

$$\mathsf{RC2SQL}(Q) \coloneqq (Q'_{\mathit{fin}}, Q'_{\mathit{inf}})$$

where

$$\begin{aligned} Q'_{\mathit{fin}} &\coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(Q_{\mathit{fin}})),\\ Q'_{\mathit{inf}} &\coloneqq \mathsf{ranf2sql}(\mathsf{optcnt}(Q_{\mathit{inf}})),\\ (Q_{\mathit{fin}}, Q_{\mathit{inf}}) &\coloneqq \mathsf{rw}(Q). \end{aligned}$$

The function $\mathsf{rw}(\cdot)$ is defined in Section 4.2.4 as a composition of the functions $\mathsf{split}(\cdot)$ and $\mathsf{sr2ranf}(\cdot)$ functions, defined in Sections 4.2.3 and 2.3.1, respectively.

Although our translation satisfies the worst-case complexity bound (Theorem 4.12), we further improve its average-case complexity by implementing the following optimizations:

- We use a sample structure of constant size, called a *training database*, to estimate the query cost when resolving the nondeterministic choices in our algorithms. All our experiments used a Data Golf structure with $|\mathcal{T}^+| = |\mathcal{T}^-| = 2$ as the training database. Still, our translation satisfies the correctness and worst-case complexity claims (Section 4.2.3 and 4.2.4) for every choice of the training database.

- We use the function optcnt optimizing RANF subqueries of the form $\exists \vec{y}.\, Q^+ \wedge \bigwedge_{i=1}^{k} \neg Q_i^-$ using the count aggregation operator. Inspired by Claußen et al. [20], we compare the number of assignments of $\vec{y}$ that satisfy Q^+ and $\bigvee_{i=1}^{k}(Q^+ \wedge Q_i^-)$, respectively.

- To compute an SQL query from a RANF query, we define the function $\mathsf{ranf2sql}(\cdot)$. We first obtain an equivalent RA expression using the standard approach [1] but adjusting the case of closed queries [19]. To translate RA expressions into SQL, we reuse a publicly available RA interpreter RADB [78]. We modify its implementation to improve the performance of the resulting SQL query. We map the anti-join operator $\hat{Q}_1 \triangleright \hat{Q}_2$ to a more efficient LEFT JOIN, if $\mathsf{fv}(\hat{Q}_2) \subsetneq \mathsf{fv}(\hat{Q}_1)$, and we perform common subquery elimination.

Nondeterministic Choices To resolve the nondeterministic choices in our algorithms, we suppose that the algorithms have access to a *training database* $\mathcal{T}$ of constant size. The training database is used to compare the cost of queries over the actual database and thus it should preserve the relative ordering of queries by their cost over the actual database as much as possible. Still, our translation satisfies the correctness and worst-case complexity claims (Section 4.2.3 and 4.2.4) for every choice of the training database. The training databases used in our empirical evaluation are obtained using the function dg (Section 4.2.5) with $|\mathcal{T}^+| = |\mathcal{T}^-| = 2$. Because of its constant size, the complexity of evaluating a query over the training database is constant and does not impact the asymptotic time complexity of evaluating the query over the actual database using our translation. There are three types of nondeterministic choices to be resolved in our algorithms:

- Choosing some $X \in \mathcal{X}$ in a while-loop. As the while-loops always update $\mathcal{X}$ with $\mathcal{X} := (\mathcal{X} - \{X\}) \cup f(X)$ for some f, the order in which the elements of $\mathcal{X}$ are chosen does not matter.

- Choosing a subset of queries $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ in the function $\mathsf{sr2ranf}(Q, \mathcal{Q})$. Because $\mathsf{sr2ranf}(Q, \mathcal{Q})$ yields a RANF query, we enumerate all *minimal* subsets (a subset $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ is minimal if there exists no proper subset $\overline{\mathcal{Q}}' \subsetneq \overline{\mathcal{Q}}$ that could be used instead of $\overline{\mathcal{Q}}$) and choose one that minimizes the query cost of the RANF query.

- Choosing a variable $x \in V$ and a set $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, where $\tilde{Q}$ is a query with range-restricted bound variables and $V \subseteq \mathsf{fv}(\tilde{Q})$ is a subset of its free variables. Observe that the measure $\mathsf{measure}(Q)$ on queries, defined in Figure 2.12, decreases for the queries in the premises of the rules for $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$, defined in Figure 2.10 and 4.1. Hence, deriving $\mathsf{gen}(x, \tilde{Q}, \mathcal{G})$ and $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ either succeeds or gets stuck after at most $\mathsf{measure}(\tilde{Q})$ steps. In particular, we can enumerate all sets $\mathcal{G}$ such that $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ holds. Because we derive one additional query $\tilde{Q}[x \mapsto y]$ for every $y \in \mathsf{eqs}(x, \mathcal{G})$ and a single query $\tilde{Q} \wedge \mathsf{qps}^{\vee}(\mathcal{G})$, we choose $x \in V$ and $\mathcal{G}$ minimizing $|\mathsf{eqs}(x, \mathcal{G})|$ as the first objective and $\sum_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} \mathsf{cost}^{\mathcal{T}}(Q_{qp})$ as the second objective. Our particular choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ is merely a heuristic and does not provide any additional guarantees compared to every other choice of $\mathcal{G}$ with $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$.

Optimization using Count Aggregations In this section, we introduce count aggregations and describe a generalization of Claußen et al. [20]'s approach to evaluate RANF queries using count aggregations. Consider the query

$$Q_x \wedge \neg\exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy}),$$

where $\mathsf{fv}(Q_x) = \{x\}$, $\mathsf{fv}(Q_y) = \{y\}$, and $\mathsf{fv}(Q_{xy}) = \{x, y\}$. This query is obtained by applying our translation to the query $Q_x \wedge \forall y.\, (Q_y \longrightarrow Q_{xy})$. The cost of the translated query is dominated by the cost of the Cartesian product $Q_x \wedge Q_y$. Consider the subquery $Q' \coloneqq \exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy})$. A assignment α satisfies Q' iff α satisfies Q_x and there exists a value d such that $\alpha[y \mapsto d]$ satisfies Q_y, but not Q_{xy}, i.e., the number of values d such that $\alpha[y \mapsto d]$ satisfies Q_y is not equal to the number of values d such that $\alpha[y \mapsto d]$ satisfies both Q_y and Q_{xy}. An alternative evaluation of Q' evaluates the queries Q_x, Q_y, $Q_y \wedge Q_{xy}$ and computes the numbers of values d such that $\alpha[y \mapsto d]$ satisfies Q_y and $Q_y \wedge Q_{xy}$, respectively, i.e., computes count aggregations. These count aggregations are then used to filter assignments α satisfying Q_x to get assignments α satisfying Q'. The asymptotic time complexity of the alternative evaluation never exceeds that of the evaluation computing the Cartesian product $Q_x \wedge Q_y$ and asymptotically improves if $|[\![Q_x]\!]| + |[\![Q_y]\!]| + |[\![Q_{xy}]\!]| \ll |[\![Q_x \wedge Q_y]\!]|$. Furthermore, we observe that a assignment α satisfies $Q_x \wedge \neg Q'$ if α satisfies Q_x, but not Q', i.e., the number of values d such that $\alpha[y \mapsto d]$ satisfies Q_y is equal to the number of values d such that $\alpha[y \mapsto d]$ satisfies $Q_y \wedge Q_{xy}$.

Next we introduce the syntax and semantics of count aggregations. We extend RC's syntax by $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$, where Q is a query, c is a variable representing the result of the count aggregation, and $\vec{v}$ is a sequence of variables that are bound by the aggregation operator. The semantics of the count aggregation is defined as follows:

$$(\mathcal{S}, \alpha) \models [\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c) \text{ iff } (M = \emptyset \longrightarrow \mathsf{fv}(Q) \subseteq \vec{v}) \,\text{and}\, \alpha(c) = |M|,$$

where $M = \{\vec{d} \in \mathcal{D}^{|\vec{v}|} \mid (\mathcal{S}, \alpha[\vec{v} \mapsto \vec{d}]) \models Q\}$. We use the condition $M = \emptyset \longrightarrow \mathsf{fv}(Q) \subseteq \vec{v}$ instead of $M \neq \emptyset$ to set c to a zero count if the group M is empty and there are no group-by variables (like in SQL). The set of free variables in a count aggregation is $\mathsf{fv}([\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)) = (\mathsf{fv}(Q) - \vec{v}) \cup \{c\}$. Finally, we extend the definition of $\mathsf{ranf}(Q)$ with the case of a count aggregation:

$$\mathsf{ranf}([\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)) \text{ iff } \mathsf{ranf}(Q) \,\text{and}\, \vec{v} \subseteq \mathsf{fv}(Q) \,\text{and}\, c \notin \mathsf{fv}(Q).$$

We formulate translations introducing count aggregations in the following two lemmas.

Lemma 4.21. *Let $\exists \vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}$, $\mathcal{Q} \neq \emptyset$, be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}})$. Then*

$$\begin{aligned}(\exists \vec{v}.\, Q_{\vec{v}} \wedge \textstyle\bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}) \equiv {} & ((\exists \vec{v}.\, Q_{\vec{v}}) \wedge \textstyle\bigwedge_{\overline{Q} \in \mathcal{Q}} \neg (\exists \vec{v}.\, Q_{\vec{v}} \wedge \overline{Q})) \vee \\ & (\exists c, c'.\, [\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c) \wedge \\ & \quad [\mathsf{CNT}\, \vec{v}.\, \textstyle\bigvee_{\overline{Q} \in \mathcal{Q}} (Q_{\vec{v}} \wedge \overline{Q})](c') \wedge \neg (c = c')).\end{aligned} \tag{\#}$$

Moreover, the right-hand side of (#) is in RANF.

Lemma 4.22. *Let $\hat{Q} \wedge \neg \exists \vec{v}.\, Q_{\vec{v}} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}$, $\mathcal{Q} \neq \emptyset$, be a RANF query. Let c, c' be fresh variables that do not occur in $\mathsf{fv}(\hat{Q}) \cup \mathsf{fv}(Q_{\vec{v}})$. Then*

$$\begin{aligned}(\hat{Q} \wedge \neg \exists \vec{v}.\, Q_{\vec{v}} \wedge \textstyle\bigwedge_{\overline{Q} \in \mathcal{Q}} \neg \overline{Q}) \equiv {} & (\hat{Q} \wedge \neg (\exists \vec{v}.\, Q_{\vec{v}})) \vee \\ & (\exists c, c'.\, \hat{Q} \wedge [\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c) \wedge \\ & \quad [\mathsf{CNT}\, \vec{v}.\, \textstyle\bigvee_{\overline{Q} \in \mathcal{Q}} (Q_{\vec{v}} \wedge \overline{Q})](c') \wedge (c = c')).\end{aligned} \tag{\#\#}$$

Moreover, the right-hand side of (##) is in RANF.

Note that the query cost does not decrease after applying the translation (#) or (##) because of the subquery $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$ in which $Q_{\vec{v}}$ is evaluated before the count aggregation is computed. For the query $\exists y.\, ((Q_x \wedge Q_y) \wedge \neg Q_{xy})$ from before, we would compute $[\mathsf{CNT}\, y.\, Q_x \wedge Q_y](c)$, i.e., we would not (yet) avoid computing the Cartesian product $Q_x \wedge Q_y$. However, we could reduce the scope of the bound variable y by further translating

$$[\mathsf{CNT}\, y.\, Q_x \wedge Q_y](c) \equiv Q_x \wedge [\mathsf{CNT}\, y.\, Q_y](c).$$

This technique called *mini-scoping* can be applied to a count aggregation $[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}}](c)$ if the aggregated query $Q_{\vec{v}}$ is a conjunction that can be split into two RANF conjuncts and the variables $\vec{v}$ do not occur free in one of the conjuncts (that conjunct can be pulled out of the count aggregation). Mini-scoping can be analogously applied to queries of the form $\exists \vec{v}.\, Q_{\vec{v}}$.

Moreover, we can split a count aggregation over a conjunction $Q_{\vec{v}} \wedge Q'_{\vec{v}}$ into a product of count aggregations if the conjunction can be split into two RANF conjuncts with disjoint sets of bound variables, i.e., $\vec{v} \cap \mathsf{fv}(Q_{\vec{v}}) \cap \mathsf{fv}(Q'_{\vec{v}}) = \emptyset$:

$$\begin{aligned}[\mathsf{CNT}\, \vec{v}.\, Q_{\vec{v}} \wedge Q'_{\vec{v}}](c) \equiv (\exists c_1, c_2.\ & [\mathsf{CNT}\, \vec{v} \cap \mathsf{fv}(Q_{\vec{v}}).\, Q_{\vec{v}}](c_1) \wedge \\ & [\mathsf{CNT}\, \vec{v} \cap \mathsf{fv}(Q'_{\vec{v}}).\, Q'_{\vec{v}}](c_2) \wedge \\ & c = c_1 \cdot c_2).\end{aligned}$$

Here c_1, c_2 are fresh variables that do not occur in $\mathsf{fv}(Q_{\vec{v}}) \cup \mathsf{fv}(Q'_{\vec{v}}) \cup \{c\}$. Note that mini-scoping is only a heuristic and it can both improve and harm the time complexity of query evaluation. We implement the translations from Lemmas 4.21 and 4.22 and mini-scoping in a function called $\mathsf{optcnt}(\cdot)$. Given a RANF query $\hat{Q}$, $\mathsf{optcnt}(\hat{Q})$ is an equivalent RANF query after introducing count aggregations and performing mini-scoping. The function $\mathsf{optcnt}(\hat{Q})$ uses a training database to decide how to apply the translations from Lemmas 4.21 and 4.22 and mini-scoping. More specifically, the function $\mathsf{optcnt}(\hat{Q})$ tries several possibilities and chooses one that minimizes the query cost of the resulting RANF query.

Example 4.23. We show how we introduce count aggregations into the RANF query

$$\hat{Q} := Q_x \wedge \neg\exists y.\, (Q_x \wedge Q_y \wedge \neg Q_{xy}).$$

After applying the translation (##) and mini-scoping to this query, we obtain the following equivalent RANF query:

$$\begin{aligned}\mathsf{optcnt}(\hat{Q}) := {} & (Q_x \wedge \neg(Q_x \wedge \exists y.\, Q_y)) \vee \\ & (\exists c, c'.\, Q_x \wedge\ [\mathsf{CNT}\, y.\, Q_y](c) \wedge \\ & \qquad [\mathsf{CNT}\, y.\, Q_y \wedge Q_{xy}](c') \wedge (c = c')).\end{aligned} \qquad \diamond$$

Translating RANF to SQL Our translation of a RANF query into SQL has two steps: we first translate the query to an equivalent RA expression, which we then translate to SQL using a publicly available RA interpreter RADB [78].

We define the function $\mathsf{ranf2ra}(\hat{Q})$ translating RANF queries $\hat{Q}$ into equivalent RA expressions $\mathsf{ranf2ra}(\hat{Q})$. The translation is based on Algorithm 5.4.8 by Abiteboul et al. [1], which we modify as follows. We adjust the way closed RC queries are handled. Chomicki and Toman [19] observed that closed RC queries cannot be handled by SQL, since SQL allows neither empty projections nor 0-ary relations. They propose to use a unary auxiliary predicate $\mathsf{A} \in \mathcal{R}$ whose interpretation $\mathsf{A}^{\mathcal{S}} = \{\mathsf{t}\}$ always contains exactly one tuple t. Every closed query $\exists x.\, Q_x$ is then translated into $\exists x.\, \mathsf{A}(t) \wedge Q_x$ with an auxiliary free variable t. Every other closed query $\hat{Q}$ is translated into $\mathsf{A}(t) \wedge \hat{Q}$, e.g., $\mathsf{B}(42)$ is translated into $\mathsf{A}(t) \wedge \mathsf{B}(42)$. We also use the auxiliary predicate A to translate queries of the form $x \approx \mathsf{c}$ and $\mathsf{c} \approx x$ because the single tuple (t) in $\mathsf{A}^{\mathcal{S}}$ can be mapped to any constant c. Finally, we extend [1, Algorithm 5.4.8] with queries of the form $[\mathsf{CNT}\,\vec{v}.\,Q_{\vec{v}}](c)$.

The RADB interpreter, abbreviated here by the function $\mathsf{ra2sql}(\cdot)$, translates a RA expression into SQL, by simply mapping the RA operators into their SQL counterparts. The function $\mathsf{ra2sql}(\cdot)$ is primitive recursive on RA expressions. We modify RADB to further improve performance of the query evaluation as follows.

A RANF query $Q_1 \wedge \neg Q_2$, where $\mathsf{ranf}(Q_1)$, $\mathsf{ranf}(Q_2)$, and $\mathsf{fv}(Q_2) \subsetneq \mathsf{fv}(Q_1)$ is translated into RA expression $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$, where $\triangleright$ denotes the anti-join operator and $\mathsf{ranf2ra}(Q_1)$, $\mathsf{ranf2ra}(Q_2)$ are the equivalent relational algebra expressions for Q_1, Q_2, respectively. The RADB interpreter only supports the anti-join operator $\mathsf{ranf2ra}(Q_1) \triangleright \mathsf{ranf2ra}(Q_2)$ expressed as $\mathsf{ranf2ra}(Q_1) - (\mathsf{ranf2ra}(Q_1) \bowtie \mathsf{ranf2ra}(Q_2))$, where $-$ denotes the set difference operator and $\bowtie$ denotes the natural join. Alternatively, the anti-join operator can be directly mapped to `LEFT JOIN` in SQL. We generalize RADB to use `LEFT JOIN` since it performs better in our empirical evaluation [60].

The RADB interpreter introduces a separate SQL subquery in a `WITH` clause for every subexpression in the RA expression. We extend RADB to additionally perform common subquery elimination, i.e., to merge syntactically equal subqueries. Common subquery elimination is also assumed in our query cost (Section 2.3.1).

Finally, the function $\mathsf{ranf2sql}(\hat{Q})$ (Figure 2.13) is defined as $\mathsf{ranf2sql}(\hat{Q}) := \mathsf{ra2sql}(\mathsf{ranf2ra}(\hat{Q}))$, i.e., as a composition of the two translations from RANF to RA and from RA to SQL.

4.2.7 Empirical Evaluation

We empirically assess the time complexity of evaluating the queries produced by our translation RC2SQL by answering the following three research questions:

RQ1: How does RC2SQL perform compared to the state-of-the-art?

RQ2: How does RC2SQL scale on large databases?

RQ3: How does RC2SQL perform on real-world databases?

To answer RQ1, we compare RC2SQL with the translation by Van Gelder and Topor [34] (VGT), our formally verified implementation [61] of the algorithm by Ailamazyan et al. [2] that uses an extended active domain as the generators, and the DDD [52,53], LDD [17], and MonPoly-reg [10] tools that support direct RC query evaluation using binary decision diagrams. We could not find a publicly available implementation of Van Gelder and Topor's translation. Therefore, the tool VGT for evaluable RC queries is derived from our implementation by modifying the function $\mathsf{rb}(\cdot)$ in Figure 4.2 to use the relation $\mathsf{con}_{\mathsf{vgt}}(x, Q, \mathcal{G})$ (Figure 2.11) instead of $\mathsf{cov}(x, Q, \mathcal{G})$ (Figure 4.1) and to use the generator $\exists \vec{\mathsf{fv}}(Q) - \{x\}.\, \mathsf{qps}^{\vee}(\mathcal{G})$ instead of $\mathsf{qps}^{\vee}(\mathcal{G})$. Evaluable queries Q are always translated into $(Q_{\mathit{fin}}, \bot)$ by $\mathsf{rw}(\cdot)$ because all of Q's free variables are range-restricted. We also consider translation variants that omit the count aggregation optimization $\mathsf{optcnt}(\cdot)$, marked with a minus $(^{-})$.

SQL queries computed by the translations are evaluated using the PostgreSQL database engine. We have also used the MySQL database engine but omit its timings from our evaluation after discovering that it computed incorrect results for some queries. This issue was reported and subsequently confirmed by MySQL developers. We run our experiments on an Intel Core i5-4200U CPU computer with 8 GB RAM. The relations in PostgreSQL are recreated before each invocation to prevent optimizations based on caching recent query evaluation results. We provide all our experiments in an easily reproducible and publicly available artifact [60].

To answer RQ2, we use Data Golf structures $\mathcal{S}$ of growing size. We control the size of the Data Golf structure $\mathcal{S}$ in our experiments using a parameter $n = |\mathcal{T}^+| = |\mathcal{T}^-|$. Because the sets $\mathcal{T}^+$ and $\mathcal{T}^-$ grow in the recursion on subqueries, relations in a Data Golf structure typically have even more than n tuples.

In the Small, Medium, and Large experiments, we generate ten pseudorandom queries with a fixed size 14 and Data Golf structures $\mathcal{S}$ (strategy $\gamma = 1$). The queries satisfy the Data Golf assumptions along with a few additional ones: the queries are in SRNF, but they are not safe-range, every bound variable actually occurs in its scope, disjunction only appears at the top-level, and only pairwise distinct variables appear as terms in predicates. The queries have 2 free variables and every subquery has at most 4 free variables. The values of the parameter n for Data Golf structures are summarized in Figure 4.5.

The Infinite experiment consists of five pseudorandom queries Q that are *not* evaluable and $\mathsf{rw}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$, where $Q_{\mathit{inf}} \neq \bot$. Specifically, the queries are of the form $Q_1 \land \forall x, y.\ Q_2 \longrightarrow Q_3$, where Q_1, Q_2, and Q_3 are either atomic predicates or equalities. We choose the queries so that the number of their satisfying tuples is not too high, e.g., quadratic in the parameter n, because no tool can possibly enumerate so many tuples within the timeout. For each query Q, we compare the performance of our tool to tools that directly evaluate Q on structures generated by the two Data Golf strategies (parameter γ), which trigger infinite or finite evaluation results on the considered queries. For infinite results, our tool outputs this fact (by evaluating Q_{inf}), whereas the other tools also output a finite representation of the infinite result. For finite results, all tools produce the same output.

Figure 4.5 shows the empirical evaluation results for the experiments Small, Medium, Large, and Infinite. All entries are execution times in seconds, TO is a timeout, and RE is a runtime error. In the experiments Small, Medium, and Large, the columns correspond to ten unique pseudorandom queries (the same queries are used in all the three experiments). In the Infinite experiment, we use five unique pseudorandom queries and two Data Golf strategies. The time it takes for our translation rc2sql to translate each query is shown in the first line for the experiments Small and Infinite because the queries in the experiments Medium and Large are the same as in Small. The remaining lines show evaluation times with the lowest time for a query typeset in bold. We omit the rows for tools that time out or crash on all queries of an experiment, e.g., Ailamazyan et al. [2]. We conclude that our translation rc2sql significantly outperforms all other tools on all queries (except vgt on the second query, but rc2sql still outperforms vgt) and scales well to higher values of n, i.e., larger relations in the Data Golf structures, on all queries.

To answer RQ3, we also use real-world structures obtained from the Amazon review dataset [56] and evaluate the tools on the queries Q^{susp} and Q^{susp}_{user} from the introduction and on the more challenging query $Q^{susp}_{text} := \mathsf{B}(b) \land \exists u, s, t.\ \forall p.\ \mathsf{P}(b, p) \longrightarrow \mathsf{S}(p, u, s) \lor \mathsf{T}(p, u, t)$ with an additional relation T that relates user's review text (variable t) to a product. The query Q^{susp}_{text} computes all brands for which there is a user, a score, and a review text such that all the brand's products were reviewed by that user with that score or by that user with that text. We use both Data Golf structures (strategy $\gamma = 1$) and real-world structures obtained from the Amazon review dataset [56]. The real-world relations P, S, and T are obtained by projecting the respective tables from the Amazon review dataset for some chosen product categories (abbreviated GC and MI in Figure 4.6) and the relation B contains all brands from P that have at least three products.

Experiment SMALL, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, $n = 500$:

RC2SQL(Q)	1.30	0.00	0.20	0.00	0.00	0.00	0.00	0.10	0.00	0.00
RC2SQL	**0.30**	0.30	**0.30**	0.30	**0.30**	**0.30**	**0.20**	0.30	**0.30**	**0.30**
RC2SQL$^-$	**0.30**	**0.20**	**0.30**	**0.20**	**0.30**	**0.30**	**0.20**	**0.20**	**0.30**	**0.30**
VGT	2.60	0.30	3.00	2.30	3.20	16.80	4.40	2.60	29.30	9.10
VGT$^-$	35.70	21.90	37.70	22.90	TO	13.20	7.00	4.70	TO	19.00
DDD	5.10	3.20	7.80	RE	1.70	5.70	8.30	3.30	25.80	8.40
LDD	59.10	25.20	72.20	23.50	16.70	44.40	22.70	30.20	276.30	49.20
MONPOLY-REG	60.80	22.00	56.70	19.90	19.40	61.90	20.50	41.50	200.00	47.20

Experiment MEDIUM, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, $n = 20000$:

RC2SQL	4.10	2.00	3.90	2.10	2.10	2.10	**0.90**	1.60	4.50	**1.70**
RC2SQL$^-$	**2.80**	**1.40**	**2.60**	**1.30**	**1.30**	**1.80**	**0.90**	**1.30**	**4.00**	**1.70**
VGT	5.40	1.80	5.50	3.80	5.00	TO	TO	4.40	TO	TO
VGT$^-$	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Experiment LARGE, Evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 14$, tool = RC2SQL:

$n = 40000$	8.00	4.00	8.00	4.20	4.00	4.10	1.60	3.10	8.80	3.60
$n = 80000$	16.30	8.30	16.30	8.50	7.90	8.50	3.20	6.30	18.90	7.20
$n = 120000$	26.20	12.40	23.40	13.00	12.20	12.60	4.90	10.00	32.60	10.20

Experiment INFINITE, Non-evaluable pseudorandom queries Q, $|\mathsf{sub}(Q)| = 7$, $n = 4000$:

	Infinite results ($\gamma = 0$)					Finite results ($\gamma = 1$)				
RC2SQL(Q)	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.10	0.00
RC2SQL	0.80	0.80	0.80	0.80	0.80	0.90	**2.40**	**1.10**	1.10	**2.20**
RC2SQL$^-$	**0.50**	**0.50**	**0.50**	**0.50**	**0.50**	**0.60**	TO	1.60	**0.70**	TO
DDD	45.00	122.90	45.20	81.20	114.10	44.50	96.30	45.00	83.30	101.00
LDD	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
MONPOLY-REG	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Figure 4.5. Experiments SMALL, MEDIUM, LARGE, and INFINITE. We use the following abbreviations: TO = Timeout of 300s, RE = Runtime Error.

Because the tool by Ailamazyan et al., DDD, LDD, and MONPOLY-REG only support integer data, we injectively remap the string and floating-point values from the Amazon review dataset to integers.

Figure 4.6 shows the empirical evaluation results: the time it takes for our translation RC2SQL to translate each query is shown in the first line and the execution times on Data Golf structures (top) and on structures derived from the real-world dataset for two specific product categories (bottom) are shown in the remaining lines. We remark that VGT cannot handle the query Q^{susp}_{user} as it is not evaluable [34]. Our translation RC2SQL significantly outperforms all other tools (except VGT on Q^{susp}, but RC2SQL still outperforms VGT) on both Data Golf and real-world structures. VGT$^-$ translates Q^{susp} into a RANF query with a higher query cost than RC2SQL$^-$. However, the optimization $\mathsf{optcnt}(\cdot)$ manages to rectify this inefficiency and thus VGT exhibits a comparable performance as RC2SQL. Specifically, the factor of $80\times$ in query cost between VGT$^-$ and RC2SQL$^-$ improves to $1.1\times$ in query cost between VGT and RC2SQL on a Data Golf structure with $n = 20$ [60]. VGT does not finish evaluating the query Q^{susp}_{text} on GC and MI datasets within 10 minutes, unlike RC2SQL.

Query	Q^{susp}		Q^{susp}_{user}		Q^{susp}_{text}	
Param. n	10^3	10^4	10^3	10^4	10^3	10^4
RC2SQL(Q)	0.00 s		0.00 s		0.60 s	
RC2SQL	**2.50**	**2.50**	**3.10**	**3.60**	**6.00**	**7.40**
RC2SQL$^-$	61.80	TO	62.20	TO	545.60	TO
VGT	3.10	2.90	–	–	TO	TO
VGT$^-$	TO	TO	–	–	TO	TO
DDD	5.80	TO	5.60	TO	25.60	TO
LDD	33.30	TO	33.90	TO	216.10	TO
MonPoly-reg	44.50	TO	45.10	TO	164.40	TO

Dataset	GC	MI	GC	MI	GC	MI
RC2SQL	**2.90**	**16.20**	**4.20**	**21.40**	**8.90**	**91.30**
RC2SQL$^-$	273.90	TO	270.10	TO	TO	TO
VGT	3.50	18.90	–	–	TO	TO
VGT$^-$	TO	TO	–	–	TO	TO
DDD	93.30	TO	90.10	TO	178.50	TO
LDD	TO	TO	TO	TO	TO	TO
MonPoly-reg	TO	TO	TO	TO	TO	TO

Figure 4.6. Experiment with the queries Q^{susp}, Q^{susp}_{user}, Q^{susp}_{text}. We use the following abbreviations: GC = Gift Cards dataset, MI = Musical Instruments dataset, TO = Timeout of 600s.

4.3 MFOTL Query Translation

In this section, we generalize our translation of RC queries from Section 4.2 to all MFOTL queries, implement the translation for MFOTL queries, and empirically evaluate the performance of monitoring MFOTL queries using our translation compared to alternative approaches. We develop a translation that takes an arbitrary MFOTL query and produces a pair of MFOTL queries in RANF: one characterizes the original query's relative safety and the other one is equivalent to the original query if the original query is relatively safe. Then we combine these two queries into a single query. This way, we obtain a monitor for an arbitrary MFOTL query that decides for every time-point if it evaluates to a finite relation and computes the relation if it is finite.

We assume that all MFOTL queries are bounded-future, i.e., the intervals I of their U_I operators satisfy $\mathsf{right}(I) \in \mathbb{T}_{\text{fin}}$. Because we apply the function $\mathsf{flipL}(I)$, defined in Section 2.1, to intervals of MFOTL temporal operators, we assume that the time domain $\mathbb{T}$ is total in this section. This means that intervals I such that $\mathsf{memL}(0, 0, I)$ and $\mathsf{right}(I) \notin \mathbb{T}_{\text{fin}}$ are full intervals.

4.3.1 Generalizing Relational Calculus Query Translation

To obtain a translation for MFOTL queries, we add the cases in Figure 4.7 to the definition of the relation $\mathsf{cov}(x, Q, \mathcal{G})$ (Figure 4.1). Here, $\tilde{\mathsf{op}}\ Q'$, $\mathsf{op} \in \{\bullet_I, \bigcirc_I, \blacklozenge_I, \lozenge_I\}$, stands for $\mathsf{op}\ Q'$ if Q' is a temporal quantified predicate and for Q' if Q' is an equality between two variables. This way, the set $\mathcal{G}$ such that $\mathsf{cov}(x, Q, \mathcal{G})$ consists of temporal quantified predicates and equalities between variables.

Next we extend the definition of the function $\mathsf{rb}(Q)$, which restricts bound variables in a

$$\begin{array}{ll}
\mathsf{cov}(x, \bullet_I\, Q, \{\tilde{\bullet}_I\, Q' \mid Q' \in \mathcal{G}\}) & \text{if } \mathsf{cov}(x, Q, \mathcal{G});\\
\mathsf{cov}(x, \bigcirc_I\, Q, \{\tilde{\bigcirc}_I\, Q' \mid Q' \in \mathcal{G}\}) & \text{if } \mathsf{cov}(x, Q, \mathcal{G});\\
\mathsf{cov}(x, Q_1 \,\mathsf{S}_I\, Q_2, \{\tilde{\blacklozenge}_I\, Q' \mid Q' \in \mathcal{G}_1 \cup \mathcal{G}_2\}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2);\\
\mathsf{cov}(x, Q_1 \,\mathsf{S}_I\, Q_2, \{\tilde{\blacklozenge}_I\, Q' \mid Q' \in \mathcal{G}\}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\mathit{false}] = \mathit{false};\\
\mathsf{cov}(x, Q_1 \,\mathsf{U}_I\, Q_2, \{\tilde{\lozenge}_I\, Q' \mid Q' \in \mathcal{G}_1 \cup \mathcal{G}_2\}) & \text{if } \mathsf{cov}(x, Q_1, \mathcal{G}_1) \text{ and } \mathsf{cov}(x, Q_2, \mathcal{G}_2);\\
\mathsf{cov}(x, Q_1 \,\mathsf{U}_I\, Q_2, \{\tilde{\lozenge}_I\, Q' \mid Q' \in \mathcal{G}\}) & \text{if } \mathsf{cov}(x, Q_2, \mathcal{G}) \text{ and } Q_2[x/\mathit{false}] = \mathit{false};
\end{array}$$

Figure 4.7. Additional cases for MFOTL queries in the definition of the relation $\mathsf{cov}(x, Q, \mathcal{G})$.

query Q (Figure 4.2), to include cases for MFOTL temporal operators. We call a query Q S-*restricted* if the subqueries Q_1' (or Q_1'' if $Q_1' = \neg Q_1''$) and Q_2' in every subquery $Q_1' \,\mathsf{S}_I\, Q_2'$ of Q with $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$ are safe-range MFOTL queries and $\mathsf{fv}(Q_1') \subseteq \mathsf{fv}(Q_2')$. Because the query $(\neg Q_1) \,\mathsf{S}_I\, Q_2$ is in RANF if Q_1 and Q_2 are in RANF and $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2)$ (although $\neg Q_1$ might not be in RANF), we distinguish the case when the left-hand side of the S_I operator is a negation. For the sake of a subsequent translation to a bounded-future MFOTL query in RANF, the translated query $\mathsf{rb}(Q)$ is S-restricted. We extend the function $\mathsf{rb}(Q)$ to MFOTL queries in Figure 4.8. The cases of RC operators are unchanged and the cases of MFOTL temporal operators except for $Q_1' \,\mathsf{S}_I\, Q_2'$ with $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$ proceed by simple recursion.

When encoutering a subquery $Q_1' \,\mathsf{S}_I\, Q_2'$ with $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$, we first recursively compute $\mathsf{rb}(Q_1')$ and $\mathsf{rb}(Q_2')$ to obtain S-restricted queries with range-restricted bound variables. If $\mathsf{memL}(0, 0, I)$, then we have a S_I operator with a full interval I (because $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$). If not $\mathsf{memL}(0, 0, I)$, then we further translate the S_I operator using the following equivalence:

$$Q_1 \,\mathsf{S}_I\, Q_2 \equiv (\blacklozenge_I\, \mathit{true}) \wedge \neg \blacklozenge_{\mathsf{flipL}(I)} (\neg Q_1 \vee \neg \bullet_{\mathsf{dropL}(I)} (Q_1 \,\mathsf{S}_{\mathsf{dropL}(I)}\, Q_2)).$$

Afterwards, we only have two S_I operators with $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$: $\blacklozenge_I\, \mathit{true}$, which is already in RANF, and $Q_1 \,\mathsf{S}_{\mathsf{dropL}(I)}\, Q_2$, whose interval $\mathsf{dropL}(I)$ is full. As long as $Q_1 \,\mathsf{S}_I\, Q_2$ (or $Q_1 \,\mathsf{S}_{\mathsf{dropL}(I)}\, Q_2$, respectively) is not S-restricted, we apply the translation formalized in the following lemma. Given a set of temporal quantified predicates and equalities $\mathcal{G}$, $\mathsf{tqps}(\mathcal{G})$ denotes the subset of $\mathcal{G}$ containing all temporal quantified predicates in $\mathcal{G}$ and $\mathsf{eqs}(x, \mathcal{G})$ denotes all variables y distinct from the variable x occurring in equalities of the form $x \approx y$ in $\mathcal{G}$. Let $\mathsf{tqps}^{\vee}(\mathcal{G})$ denote the query $\bigvee_{Q_{tqp} \in \mathsf{tqps}(\mathcal{G})} Q_{tqp}$ and let $\mathsf{tqps}^{\vee \blacklozenge_I}(\mathcal{G})$ denote the query $\bigvee_{Q_{tqp} \in \mathsf{tqps}(\mathcal{G})} \blacklozenge_I\, Q_{tqp}$.

Lemma 4.24. *Let Q_1 and Q_2 be MFOTL queries with range-restricted bound variables, let $I \in \mathbb{I}$ be a full interval, let $x \in \mathsf{fv}(Q_1) \cup \mathsf{fv}(Q_2)$, and let $\mathcal{G}_1$ and $\mathcal{G}_2$ be sets of temporal quantified predicates and equalities such that $\mathsf{cov}(x, Q_1, \mathcal{G}_1)$ and $\mathsf{cov}(x, Q_2, \mathcal{G}_2)$ hold. Let $\mathcal{G} := \mathcal{G}_2$ if $Q_2[x/\mathit{false}] = \mathit{false}$ and $\mathcal{G} := \mathcal{G}_1 \cup \mathcal{G}_2$ otherwise. Let $Q_1' := Q_1 \wedge \mathsf{tqps}^{\vee \blacklozenge_I}(\mathcal{G})$ if $x \in \mathsf{fv}(Q_1)$ and not $\mathsf{gen}(x, Q_1)$ and $Q_1' := Q_1$ otherwise. Let f be a function mapping MFOTL queries to MFOTL queries such that (i) $f(Q) = Q$ for all Q, or (ii) $f(Q) = \neg Q$ for all Q. Then*

$$\begin{aligned}
f(Q_1) \,\mathsf{S}_I\, Q_2 \equiv {} & (\textstyle\bigwedge_{y \in \mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge f(Q_1') \,\mathsf{S}_I\, (Q_2 \wedge \mathsf{tqps}^{\vee \blacklozenge_I}(\mathcal{G}))) \vee \\
& (\textstyle\bigwedge_{y \in \mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge f(Q_1') \,\mathsf{S}_I\, (f(Q_1) \wedge \mathsf{tqps}^{\vee}(\mathcal{G}) \wedge \\
& \bullet_I\, ((\neg \mathsf{tqps}^{\vee \blacklozenge_I}(\mathcal{G})) \wedge (f(Q_1) \,\mathsf{S}_I\, Q_2)[x/\mathit{false}]))) \vee \\
& (\textstyle\bigwedge_{y \in \mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge (\neg \mathsf{tqps}^{\vee \blacklozenge_I}(\mathcal{G})) \wedge (f(Q_1) \,\mathsf{S}_I\, Q_2)[x/\mathit{false}]) \vee \\
& \textstyle\bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} ((f(Q_1) \,\mathsf{S}_I\, Q_2)[x \mapsto y] \wedge x \approx y).
\end{aligned} \qquad (\bigstar^{\mathrm{FO}})$$

Let i be the time-point at which $Q_1 \,\mathsf{S}_I\, Q_2$ is evaluated (under a fixed temporal structure $\bar{\mathcal{S}}$ and a fixed assignment α). Let j be the past time-point at which Q_2 might be satisfied. The

input: A bounded-future MFOTL query Q.
output: A bounded-future S-restricted MFOTL query $\tilde{Q}$ with range-restricted bound variables such that $Q \overset{\infty}{\equiv} \tilde{Q}$.

```
function rbSince(Q_1, I, Q_2, f) =
  if $\mathsf{nongens}(Q_1) \cup \mathsf{nongens}(Q_2) = \emptyset$ and $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2)$ then
    return $f(Q_1)\ \mathsf{S}_I\ Q_2$;
  else
    $x \leftarrow \mathsf{nongens}(Q_1) \cup \mathsf{nongens}(Q_2) \cup (\mathsf{fv}(Q_1) - \mathsf{fv}(Q_2))$;
    $\mathcal{G}_1 \leftarrow \{\mathcal{G}_1 \mid \mathsf{cov}(x, Q_1, \mathcal{G}_1)\}$; $\mathcal{G}_2 \leftarrow \{\mathcal{G}_2 \mid \mathsf{cov}(x, Q_2, \mathcal{G}_2)\}$;
    if $Q_2[x/\mathit{false}] = \mathit{false}$ then $\mathcal{G} := \mathcal{G}_2$;
    else $\mathcal{G} := \mathcal{G}_1 \cup \mathcal{G}_2$;
    $I' := \mathsf{dropL}(I)$;
    if $x \in \mathsf{fv}(Q_1)$ and not $\mathsf{gen}(x, Q_1)$ then $Q_1' := Q_1 \wedge \mathsf{tqps}^{\vee\blacklozenge_{I'}}(\mathcal{G})$;
    else $Q_1' := Q_1$;
    $\tilde{Q} := \mathsf{rb}((\bigwedge_{y\in\mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge f(Q_1')\ \mathsf{S}_{I'}\ (Q_2 \wedge \mathsf{tqps}^{\vee\blacklozenge_{I'}}(\mathcal{G}))) \vee$
              $(\bigwedge_{y\in\mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge f(Q_1')\ \mathsf{S}_{I'}\ (f(Q_1) \wedge \mathsf{tqps}^{\vee}(\mathcal{G}) \wedge$
                $\bullet_{I'}\ ((\neg \mathsf{tqps}^{\vee\blacklozenge_{I'}}(\mathcal{G})) \wedge (f(Q_1)\ \mathsf{S}_{I'}\ Q_2)[x/\mathit{false}]))) \vee$
              $(\bigwedge_{y\in\mathsf{eqs}(x,\mathcal{G})} (\neg x \approx y) \wedge (\neg \mathsf{tqps}^{\vee\blacklozenge_{I'}}(\mathcal{G})) \wedge (f(Q_1)\ \mathsf{S}_{I'}\ Q_2)[x/\mathit{false}]) \vee$
              $\bigvee_{y\in\mathsf{eqs}(x,\mathcal{G})} ((f(Q_1)\ \mathsf{S}_{I'}\ Q_2)[x \mapsto y] \wedge x \approx y))$;
    if $\mathsf{memL}(0, 0, I)$ then return $\tilde{Q}$;
    else return $(\blacklozenge_I\ \mathit{true}) \wedge \neg\blacklozenge_{\mathsf{flipL}(I)}\ (\neg f(Q_1) \vee \neg\bullet_{I'}\ \tilde{Q})$;
function rb(Q) =
  switch $Q$ do
    case $\bullet_I\ Q'$ do return $\bullet_I\ \mathsf{rb}(Q')$;
    case $\bigcirc_I\ Q'$ do return $\bigcirc_I\ \mathsf{rb}(Q')$;
    case $Q_1'\ \mathsf{S}_I\ Q_2'$ do
      if $\mathsf{right}(I) \in \mathbb{T}_{\mathrm{fin}}$ then return $\mathsf{rb}(Q_1')\ \mathsf{S}_I\ \mathsf{rb}(Q_2')$;
      else
        switch $\mathsf{rb}(Q_1')$ do
          case $\neg Q_1''$ do return $\mathsf{rbSince}(Q_1'', I, \mathsf{rb}(Q_2'), (\lambda Q.\, \neg Q))$;
          otherwise do return $\mathsf{rbSince}(\mathsf{rb}(Q_1'), I, \mathsf{rb}(Q_2'), (\lambda Q.\, Q))$;
    case $Q_1'\ \mathsf{U}_I\ Q_2'$ do return $\mathsf{rb}(Q_1')\ \mathsf{U}_I\ \mathsf{rb}(Q_2')$;
    ...
```

Figure 4.8. Extension of $\mathsf{rb}(Q)$ to MFOTL queries.

translation ($\bigstar^{\mathrm{FO}}$) proceeds by a case distinction if $x \approx y$ holds for some $y \in \mathsf{eqs}(x, \mathcal{G})$ (the last line) or not (all remaining lines). The case distinction further proceeds based on whether

- $\mathsf{tqps}^{\vee}(\mathcal{G})$ holds at some time-point $k \leq j$, then $\mathsf{tqps}^{\vee\blacklozenge_I}(\mathcal{G})$ holds at all time-points between j and i (first disjunct);
- $\mathsf{tqps}^{\vee}(\mathcal{G})$ holds at some time-point $k > j$, but does not hold before, then $\mathsf{tqps}^{\vee\blacklozenge_I}(\mathcal{G})$ holds at all time-points between k and i, but not strictly before k (second disjunct);
- $\mathsf{tqps}^{\vee}(\mathcal{G})$ holds at no time-point before and including i (third disjunct).

We apply Lemma 4.24 with (ii) $f(Q) = \neg Q$ if the subquery Q_1 of $Q_1 \mathsf{S}_I Q_2$ has the form $Q_1 = \neg Q_1'$ for some Q_1'. Otherwise, we apply Lemma 4.24 with (i) $f(Q) = Q$. After applying the translation ($\bigstar^{\mathrm{FO}}$) in the function $\mathsf{rb}(Q)$, the lexicographic measure $(\mathsf{fv}(Q_1) \cup \mathsf{fv}(Q_2), \mathsf{nongens}(Q_1) \cup \mathsf{nongens}(Q_2) \cup (\mathsf{fv}(Q_1) - \mathsf{fv}(Q_2)))$ decreases for each of the four S_I operators of the form $f(Q_1) \mathsf{S}_I Q_2$ after applying ($\bigstar^{\mathrm{FO}}$):

- $f(Q_1') \mathsf{S}_I (Q_2 \wedge \mathsf{tqps}^{\vee\blacklozenge_I}(\mathcal{G}))$: no new free variables, all variables that were range-restricted are still range-restricted, x is range-restricted or does not occur, all variables that occur in Q_1', but not in $Q_2 \wedge \mathsf{tqps}^{\vee\blacklozenge_I}(\mathcal{G})$, are also in $\mathsf{fv}(Q_1) - \mathsf{fv}(Q_2)$, and x occurs at the left-hand side and right-hand side;
- $f(Q_1') \mathsf{S}_I (f(Q_1) \wedge \mathsf{tqps}^{\vee}(\mathcal{G}) \wedge \CIRCLE_I ((\neg \mathsf{tqps}^{\vee\blacklozenge_I}(\mathcal{G})) \wedge (f(Q_1) \mathsf{S}_I Q_2)[x/\mathit{false}]))$: no new free variables, all variables that were range-restricted are still range-restricted, x is range-restricted or does not occur, and the free variables of the left-hand side are a subset of the free variables of the right-hand side;
- $(f(Q_1) \mathsf{S}_I Q_2)[x/\mathit{false}]$: the set of all free variables shrinks;
- $(f(Q_1) \mathsf{S}_I Q_2)[x \mapsto y]$: the set of all free variables shrinks.

Hence, the function $\mathsf{rb}(Q)$ applying ($\bigstar^{\mathrm{FO}}$) exhaustively terminates.

The function $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$, defined in Figure 4.3, immediately extends to MFOTL. Moreover, the queries Q_{fin} and Q_{inf} are S-restricted. Indeed, if $Q_1 \mathsf{S}_I Q_2$ is S-restricted, then Q_2 is safe-range and $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2)$. Then $(Q_1 \mathsf{S}_I Q_2)[x \mapsto y]$ is clearly S-restricted. If $x \in \mathsf{fv}(Q_2)$, then $(Q_1 \mathsf{S}_I Q_2)[x/\mathit{false}] = \mathit{false}$ because Q_2 is safe-range. Otherwise, $Q_2[x/\mathit{false}] = Q_2$ and $(Q_1 \mathsf{S}_I Q_2)[x/\mathit{false}]$ is S-restricted. Lemmas 4.1, 4.2, 4.3 immediately extend to MFOTL. The remaining lemmas on RC query translation are generalized to MFOTL in a straightforward way as follows:

Lemma 4.25. *Let a temporal structure $\bar{S}$ with an infinite domain $\mathcal{D}$ and a time-point i be fixed. Let x be a free variable in a query $\tilde{Q}$ with range-restricted bound variables and let $\mathsf{cov}(x, \tilde{Q}, \mathcal{G})$ for a set of temporal quantified predicates and equalities $\mathcal{G}$. If $\tilde{Q}[x/\mathit{false}]$ is not satisfied by any tuple at the time-point i, then*

$$[\![\tilde{Q}]\!] = \left[\!\!\left[(\tilde{Q} \wedge \mathsf{tqps}^{\vee}(\mathcal{G})) \vee \bigvee_{y \in \mathsf{eqs}(x,\mathcal{G})} (\tilde{Q}[x \mapsto y] \wedge x \approx y)\right]\!\!\right].$$

If $\tilde{Q}[x/\mathit{false}]$ is satisfied by some tuple, then $[\![\tilde{Q}]\!]$ is an infinite set.

```
function srnf(Q) =
    switch Q do
        case $\bullet_I\ Q'$ do return $\bullet_I$ srnf($Q'$);
        case $\bigcirc_I\ Q'$ do return $\bigcirc_I$ srnf($Q'$);
        case $Q'_1\ \mathsf{S}_I\ Q'_2$ do
            switch $Q'_1$ do
                case $\neg Q''_1$ do return $(\neg\mathsf{srnf}(Q''_1))\ \mathsf{S}_I\ \mathsf{srnf}(Q'_2)$;
                otherwise do return $\mathsf{srnf}(Q'_1)\ \mathsf{S}_I\ \mathsf{srnf}(Q'_2)$;
        case $Q'_1\ \mathsf{U}_I\ Q'_2$ do
            switch $Q'_1$ do
                case $\neg Q''_1$ do return $(\neg\mathsf{srnf}(Q''_1))\ \mathsf{U}_I\ \mathsf{srnf}(Q'_2)$;
                otherwise do return $\mathsf{srnf}(Q'_1)\ \mathsf{U}_I\ \mathsf{srnf}(Q'_2)$;
        ...
```

Figure 4.9. Extension of $\mathsf{srnf}(Q)$ to MFOTL queries.

```
function measure(Q) =
    switch Q do
        case $\bullet_I\ Q'$ do return measure($Q'$);
        case $\bigcirc_I\ Q'$ do return measure($Q'$);
        case $Q'_1\ \mathsf{S}_I\ Q'_2$ do
            if $Q'_1 = \mathit{true}$ then return measure($Q'_2$);
            else return $1 + \mathsf{measure}(Q'_1) + \mathsf{measure}(Q'_2)$;
        case $Q'_1\ \mathsf{U}_I\ Q'_2$ do
            if $Q'_1 = \mathit{true}$ then return measure($Q'_2$);
            else return $1 + \mathsf{measure}(Q'_1) + \mathsf{measure}(Q'_2)$;
        ...
```

Figure 4.10. Extension of $\mathsf{measure}(Q)$ to MFOTL.

Lemma 4.26. *Let Q be a bounded-future MFOTL query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. Then the queries Q_{fin} and Q_{inf} are bounded-future, S-restricted, and safe-range; $\mathsf{fv}(Q_{\mathit{fin}}) = \mathsf{fv}(Q)$ unless Q_{fin} is syntactically equal to false; and $\mathsf{fv}(Q_{\mathit{inf}}) = \emptyset$.*

Lemma 4.27. *Let a temporal structure $\bar{S}$ with an infinite domain $\mathcal{D}$ and a time-point i be fixed. Let Q be an MFOTL query and $\mathsf{split}(Q) = (Q_{\mathit{fin}}, Q_{\mathit{inf}})$. If $i \models Q_{\mathit{inf}}$, then $[\![Q]\!]_i^{\bar{S}}$ is an infinite set. Otherwise, $[\![Q]\!] = [\![Q_{\mathit{fin}}]\!]$ is a finite set.*

It remains to extend the functions $\mathsf{srnf}(Q)$ (Figure 2.14) and $\mathsf{sr2ranf}(Q, \mathcal{Q})$ (Figure 2.16) to MFOTL. We extend $\mathsf{srnf}(Q)$ to MFOTL in Figure 4.9 by simple recursion on MFOTL temporal operators. Note that we do not push negation if the left-hand side of S_I or U_I operators is a negation. This is because in general we could not distribute the S_I or U_I operator over its left-hand side, e.g., if the left-hand side is a disjunction. The termination of the function $\mathsf{srnf}(Q)$ follows using the lexicographic measure $(\mathsf{measure}(Q), \mathsf{tdepth}(Q))$, where $\mathsf{measure}(Q)$ is defined in Figure 2.12 and extended to MFOTL in Figure 4.10 and $\mathsf{tdepth}(Q)$ is defined in Figure 4.11. We also generalize Lemma 2.17 to MFOTL.

```
function tdepth($Q$) =
    switch $Q$ do
        case $\neg Q'$ do return tdepth($Q'$);
        case $Q'_1 \lor Q'_2$ do return $\max\{$tdepth($Q'_1$), tdepth($Q'_2$)$\}$;
        case $Q'_1 \land Q'_2$ do return $\max\{$tdepth($Q'_1$), tdepth($Q'_2$)$\}$;
        case $\exists x.\, Q_x$ do return tdepth($Q_x$);
        case $\bullet_I\ Q'$ do return $1 +$ tdepth($Q'$);
        case $\bigcirc_I\ Q'$ do return $1 +$ tdepth($Q'$);
        case $Q'_1\ \mathsf{S}_I\ Q'_2$ do return $1 + \max\{$tdepth($Q'_1$), tdepth($Q'_2$)$\}$;
        case $Q'_1\ \mathsf{U}_I\ Q'_2$ do return $1 + \max\{$tdepth($Q'_1$), tdepth($Q'_2$)$\}$;
        otherwise do return $1$;
```

Figure 4.11. The measure $\mathsf{tdepth}(Q)$ on MFOTL queries.

Lemma 4.28. *Let Q be a bounded-future S-restricted safe-range MFOTL query. Then $\mathsf{srnf}(Q)$ is a bounded-future S-restricted safe-range MFOTL query in SRNF such that $\mathsf{gen}(x, \neg Q')$ does not hold for any variable x and subquery $\neg Q'$ of $\mathsf{srnf}(Q)$.*

We extend $\mathsf{sr2ranf}(Q, \mathcal{Q})$ to MFOTL in Figure 4.12. The operators $\mathsf{S}_I^{\neg}$ and $\mathsf{U}_I^{\neg}$ are defined as: $Q_1\ \mathsf{S}_I^{\neg}\ Q_2 := (\neg Q_1)\ \mathsf{S}_I^{\neg}\ Q_2$ and $Q_1\ \mathsf{U}_I^{\neg}\ Q_2 := (\neg Q_1)\ \mathsf{U}_I^{\neg}\ Q_2$. The termination of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ follows from the lexicographic measure $(2 \cdot \mathsf{measure}(Q) + \mathsf{eqneg}(Q) + 2 \cdot \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{measure}(\overline{Q}) + 2 \cdot |\mathcal{Q}|$, $\mathsf{measure}(Q) + \sum_{\overline{Q} \in \mathcal{Q}} \mathsf{measure}(\overline{Q}), \mathsf{tdepth}(Q))$, where $\mathsf{measure}(Q)$ is defined in Figure 2.12 and extended to MFOTL in Figure 4.10, $\mathsf{tdepth}(Q)$ is defined in Figure 4.11, and $\mathsf{eqneg}(Q) := 1$ if Q is an equality between two variables or the negation of a query, and $\mathsf{eqneg}(Q) := 0$ otherwise. Suppose that we translate a temporal operator op and this operator is evaluated at a time-point i. When moving across op, we extend the queries from $\mathcal{Q}$ with a temporal operator covering the time-point i from any time-point that is relevant for op's satisfaction, e.g., we use $\bullet_I$ for $\bigcirc_I$ and vice-versa, we use $\Diamond_{\mathsf{dropL}(I)}$ for S_I, and we use $\blacklozenge_{\mathsf{dropL}(I)}$ for U_I. If $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$ for S_I, then $\Diamond_{\mathsf{dropL}(I)}$ would introduce an unbounded U_I into the translated query. Hence, we cannot propagate any query from $\mathcal{Q}$ to the subqueries of S_I with $\mathsf{right}(I) \notin \mathbb{T}_{\mathrm{fin}}$. This is possible because S_I is S-restricted. For the temporal operators S_I and U_I, $\mathsf{sr2ranf}(Q, \mathcal{Q})$ cannot return any subset $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ of queries, where $\{\Diamond_{\mathsf{dropL}(I)}\ \overline{Q} \mid \overline{Q} \in \overline{\mathcal{Q}}\}$ or $\{\blacklozenge_{\mathsf{dropL}(I)}\ \overline{Q} \mid \overline{Q} \in \overline{\mathcal{Q}}\}$ was propagated to the subqueries, because we cannot guarantee that the queries from $\overline{\mathcal{Q}}$ are satisfied at the time-point i at which S_I or U_I is evaluated. Note that none of the conditions "$Q'_1 \land Q'_2 \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range" and "$Q'_2 \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range" in doBin implies the other one. For instance, if $Q'_1 := \neg \mathsf{B}(x)$, $Q'_2 := \mathsf{B}(y)$, and $\mathcal{Q} = \emptyset$, then only $Q'_2 \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range and if $Q'_1 := \mathsf{B}(y)$, $Q'_2 := \neg \mathsf{B}(y)$, and $\mathcal{Q} = \emptyset$, then only $Q'_1 \land Q'_2 \land \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range.

Finally, $\mathsf{sr2ranf}(Q) := \hat{Q}$, where $(\hat{Q}, \_) := \mathsf{sr2ranf}(\mathsf{srnf}(Q), \emptyset)$, yields an MFOTL query $\hat{Q}$ in RANF that is equivalent to Q and Definition 4.3 immediately extends to MFOTL.

4.3.2 Implementation and Empirical Evaluation

Implementation We have extended our translation RC2SQL for RC queries (Section 4.2.6) to the translation MFOTL2RANF [62] for MFOTL queries. Overall, the translation is defined as

$$\mathsf{MFOTL2RANF}(Q) := (Q_{\mathit{inf}} \land (f \approx \mathsf{c}_{\mathsf{inf}}) \land \textstyle\bigwedge_{x \in \mathsf{fv}(Q)} x \approx \mathsf{c}_{\mathsf{inf}}) \lor ((\neg Q_{\mathit{inf}}) \land (f \approx \mathsf{c}_{\mathsf{fin}}) \land Q_{\mathit{fin}}),$$

```
input:  A bounded-future S-restricted safe-range MFOTL query $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$ such that
        $\mathsf{gen}(x, \neg Q')$ does not hold for any variable $x$ and subquery $\neg Q'$.
output: A bounded-future MFOTL query $\hat{Q}$ in RANF and a subset $\overline{\mathcal{Q}} \subseteq \mathcal{Q}$ such that
        $Q \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q} \equiv \hat{Q} \wedge \bigwedge_{\overline{Q} \in \mathcal{Q}} \overline{Q}$; $(\bar{\mathcal{S}}, \alpha, i) \models \hat{Q} \Longrightarrow (\bar{\mathcal{S}}, \alpha, i) \models \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ holds for all
        $\bar{\mathcal{S}}$, $\alpha$, $i$; $\hat{Q} = \mathsf{cp}(\hat{Q})$; and $\mathsf{fv}(Q) \subseteq \mathsf{fv}(\hat{Q}) \subseteq \mathsf{fv}(Q) \cup \mathsf{fv}(\mathcal{Q})$, unless $\hat{Q} = \mathit{false}$.
function $\mathsf{doBin}(Q_1, \mathsf{op}_1, Q_2, \mathcal{Q}, \mathsf{op}_2, \mathsf{op}_3) =$
    $\overline{\mathcal{Q}} \leftarrow \{\overline{\mathcal{Q}} \subseteq \mathcal{Q} \mid Q_1 \wedge Q_2 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range, $Q_2 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range,
                and $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q})\}$;
    if not $\mathsf{ranf}(Q_1)$ then
        if $Q_1 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \overline{Q}$ is safe-range then
            $(Q_1', \_) \leftarrow \{\mathsf{sr2ranf}(Q_1 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \mathsf{op}_2\ \overline{Q}, \emptyset),$
                        $\mathsf{sr2ranf}(Q_1 \wedge (\mathsf{op}_3\ Q_2) \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \mathsf{op}_2\ \overline{Q}, \emptyset)\}$;
        else $(Q_1', \_) \coloneqq \mathsf{sr2ranf}(Q_1 \wedge (\mathsf{op}_3\ Q_2) \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \mathsf{op}_2\ \overline{Q}, \emptyset)$;
    $(Q_2', \_) \coloneqq \mathsf{sr2ranf}(Q_2 \wedge \bigwedge_{\overline{Q} \in \overline{\mathcal{Q}}} \mathsf{op}_2\ \overline{Q}, \emptyset)$;
    return $(\mathsf{cp}(Q_1'\ \mathsf{op}_1\ Q_2'), \emptyset)$;
function $\mathsf{sr2ranfSince}(Q_1, \mathsf{op}_1, I, Q_2, \mathcal{Q}) =$
    if $\mathsf{right}(I) \in \mathbb{T}_{\mathrm{fin}}$ then return $\mathsf{doBin}(Q_1, \mathsf{op}_1, Q_2, \mathcal{Q}, \Diamond_{\mathsf{dropL}(I)}, \blacklozenge_{\mathsf{dropL}(I)})$;
    else
        $(Q_1', \_) \coloneqq \mathsf{sr2ranf}(Q_1, \emptyset)$;
        $(Q_2', \_) \coloneqq \mathsf{sr2ranf}(Q_2, \emptyset)$;
        return $(\mathsf{cp}(Q_1'\ \mathsf{op}_1\ Q_2'), \emptyset)$
function $\mathsf{sr2ranf}(Q, \mathcal{Q}) =$
    if $\mathsf{ranf}(Q)$ then return $(\mathsf{cp}(Q), \emptyset)$;
    switch $Q$ do
        case $\bullet_I\ Q'$ do
            $(Q', \overline{\mathcal{Q}}) \coloneqq \mathsf{sr2ranf}(Q', \{\bigcirc_I\ \overline{Q} \mid \overline{Q} \in \mathcal{Q}\})$;
            return $(\mathsf{cp}(\bullet_I\ Q'), \{\overline{Q} \mid \bigcirc_I\ \overline{Q} \in \overline{\mathcal{Q}}\})$;
        case $\bigcirc_I\ Q'$ do
            $(Q', \overline{\mathcal{Q}}) \coloneqq \mathsf{sr2ranf}(Q', \{\bullet_I\ \overline{Q} \mid \overline{Q} \in \mathcal{Q}\})$;
            return $(\mathsf{cp}(\bigcirc_I\ Q'), \{\overline{Q} \mid \bullet_I\ \overline{Q} \in \overline{\mathcal{Q}}\})$;
        case $Q_1'\ \mathsf{S}_I\ Q_2'$ do
            switch $Q_1'$ do
                case $\neg Q_1''$ do return $\mathsf{sr2ranfSince}(Q_1'', \mathsf{S}_I^{\neg}, I, Q_2', \mathcal{Q})$;
                otherwise do return $\mathsf{sr2ranfSince}(Q_1', \mathsf{S}_I, I, Q_2', \mathcal{Q})$;
        case $Q_1'\ \mathsf{U}_I\ Q_2'$ do
            switch $Q_1'$ do
                case $\neg Q_1''$ do return $\mathsf{doBin}(Q_1'', \mathsf{U}_I^{\neg}, Q_2', \mathcal{Q}, \blacklozenge_{\mathsf{dropL}(I)}, \Diamond_{\mathsf{dropL}(I)})$;
                otherwise do return $\mathsf{doBin}(Q_1', \mathsf{U}_I, Q_2', \mathcal{Q}, \blacklozenge_{\mathsf{dropL}(I)}, \Diamond_{\mathsf{dropL}(I)})$;
        ...
```

Figure 4.12. Extension of $\mathsf{sr2ranf}(Q, \mathcal{Q})$ to MFOTL queries.

where $(Q_{\mathit{fin}}, Q_{\mathit{inf}}) := \mathsf{rw}(Q)$; $f \notin \mathsf{fv}(Q)$ is a fresh variable signaling if the result is infinite ($f \approx \mathsf{c}_{\mathsf{inf}}$); and $\mathsf{c}_{\mathsf{fin}}, \mathsf{c}_{\mathsf{inf}} \in \mathcal{D}$ are arbitrary constants (preferably $\mathsf{c}_{\mathsf{fin}} \neq \mathsf{c}_{\mathsf{inf}}$ so that the value of the variable f uniquely expresses if the query Q evaluates to an infinite result). In our implementation, we use $\mathsf{c}_{\mathsf{fin}} = 0$ and $\mathsf{c}_{\mathsf{inf}} = 1$. Unlike RC2SQL, which produces a pair of SQL queries, the translation MFOTL2RANF produces a single MFOTL query in RANF that can be evaluated at every time-point by state-of-the-art first-order monitoring algorithms, e.g., VeriMon [8] and MonPoly [10].

As in RC2SQL, we improve the average-case complexity of evaluating the MFOTL query $\mathsf{MFOTL2RANF}(Q)$ by implementing the following optimization: We use a sample trace of constant length with databases Γ_k of constant size, called a *training trace*, to estimate the temporal query cost when resolving the nondeterministic choices in our algorithms. The training trace is used to compare the temporal query cost over the actual observed trace and thus it should preserve the relative ordering of queries by their temporal query cost over the actual observed trace as much as possible. Still, our translation satisfies the correctness claims (Section 4.3.1) for every choice of the training trace. Because the temporal query cost of a query Q with a low progress $\mathsf{prog}(Q, \overline{\tau})$ could be low simply because there are only a few time-points at which Q can be evaluated on the given training trace, we always maximize $\mathsf{prog}(Q, \overline{\tau})$ as the first objective and $\mathsf{cost}^{\bar{\mathcal{S}}}_{<\ell}(Q)$ as the second objective. When choosing a variable $x \in V$ and a set $\mathcal{G}$ such that $\mathsf{cov}(x, Q, \mathcal{G})$, we maximize the minimal progress of a query $\overline{Q} \in \mathcal{G}$, i.e., we maximize the quantity $\min_{\overline{Q} \in \mathcal{G}}\{\mathsf{prog}(\overline{Q}, \overline{\tau})\}$, as the first objective. Then we proceed analogously to RC2SQL: minimizing $|\mathsf{eqs}(x, \mathcal{G})|$ as the second objective and minimizing $\sum_{Q_{qp} \in \mathsf{qps}(\mathcal{G})} \mathsf{cost}^{\mathcal{T}}(Q_{qp})$ as the third objective.

Empirical Evaluation We empirically evaluate our translation's performance on MFOTL queries by answering the following two research questions:

RQ1: How does MFOTL2RANF perform compared to the state-of-the-art?

RQ2: How does MFOTL2RANF scale with respect to the trace length?

To answer RQ1, we compare the time complexity of evaluating the query $\mathsf{MFOTL2RANF}(Q)$, produced by our translation, with the time complexity of evaluating the original MFOTL query Q using state-of-the-art tools supporting MFOTL temporal operators and having an available implementation: DejaVu [41, 42] and MonPoly-reg [10]. We use CppMon [39], VeriMon [8], and MonPoly [10] to evaluate the MFOTL query $\mathsf{MFOTL2RANF}(Q)$ in RANF. In the following, we refer to the tools CppMon, VeriMon, and MonPoly as *MFOTL monitors.* Our empirical evaluation can be reproduced using a publicly available artifact [62]. The experimental setup is as described in Section 3.3.

To answer RQ2, we generate traces of increasing trace length that doubles in every step. Then we measure evaluation times on these traces of increasing trace length and compute the ratios of evaluation times between consecutive trace lengths. The ratios are useful to assess the asymptotics of the evaluation times.

Queries We use the DataRace query from Havelund et al. [42] capturing data races: "A data race occurs when two threads access (read or write) the same shared variable simultaneously, and at least one of the threads writes to the variable. The property states that in this case there must exist a lock, which both threads hold whenever they access the variable." Formally, it is expressed by the following MFOTL query, where $I \in \mathbb{I}$ is an interval, the atomic predicates *read*(t, x) and *write*(t, x) express that the thread t reads (writes to) a shared variable x, and the atomic predicates *acq*(t, l) and *rel*(t, l) express that the thread t acquires (releases) a lock l:

$$\begin{aligned}\textsc{Datarace}^I_{\blacklozenge} \coloneqq \forall t_1, t_2, x.\ ((\blacklozenge_I\ (\mathit{read}(t_1, x) \lor \mathit{write}(t_1, x))) \land (\blacklozenge_I\ \mathit{write}(t_2, x))) \longrightarrow\\ \exists l.\ (\blacksquare_I\ ((\mathit{read}(t_1, x) \lor \mathit{write}(t_1, x)) \longrightarrow ((\neg \mathit{rel}(t_1, l))\ \mathsf{S}_I\ \mathit{acq}(t_1, l)))) \land\\ (\blacksquare_I\ ((\mathit{read}(t_2, x) \lor \mathit{write}(t_2, x)) \longrightarrow ((\neg \mathit{rel}(t_2, l))\ \mathsf{S}_I\ \mathit{acq}(t_2, l))))\end{aligned}$$

The MFOTL query $\textsc{Datarace}^I_{\blacklozenge}$ is not in RANF and MonPoly's heurestics [10] fail to bring it into RANF. Hence, Havelund et al. [42] could not use MonPoly to evaluate $\textsc{Datarace}^I_{\blacklozenge}$. We use the MFOTL monitors on the MFOTL query $\mathsf{MFOTL2RANF}(\textsc{Datarace}^I_{\blacklozenge})$ in RANF produced by our translation. We also use the MFOTL monitors on the MFOTL query

$$\widehat{\textsc{Datarace}}^I_{\blacklozenge} \coloneqq \neg \exists t_1, t_2, x.\ (\blacklozenge_I\ A(t_1, x)) \land (\blacklozenge_I\ \mathit{write}(t_2, x)) \land \neg(\exists l.\, H(t_1, x, l) \land H(t_2, x, l)),$$

where we use the following abbreviations:

$$\begin{aligned}A(t, x) &\coloneqq \mathit{read}(t, x) \lor \mathit{write}(t, x),\\ L(t, l) &\coloneqq (\neg \mathit{rel}(t, l))\ \mathsf{S}_I\ \mathit{acq}(t, l),\\ G(t, x, l) &\coloneqq A(t, x) \land L(t, l) \land \neg \CIRCLE_I\ (\blacklozenge_I\ A(t, x)),\\ H(t, x, l) &\coloneqq (\neg(A(t, x) \land (\blacklozenge_I\ G(t, x, l)) \land \neg L(t, l)))\ \mathsf{S}_I\ G(t, x, l).\end{aligned}$$

The query $G(t, x, l)$ is satisfied when a thread t accesses (reads or writes to) a variable x for the first time and holds a lock l at that time. The query $H(t, x, l)$ is satisfied by threads t that accessed x at some past time-point and expresses that the thread t holds the lock l whenever it accesses the variable x (since t accessed x for the first time). The subquery $(\blacklozenge_I\ G(t, x, l))$ in $H(t, x, l)$ is only used to bring $H(t, x, l)$ into RANF without changing the semantics of $H(t, x, l)$. The query $\widehat{\textsc{Datarace}}^I_{\blacklozenge}$ is in RANF and equivalent to $\textsc{Datarace}^I_{\blacklozenge}$ for full intervals I. However, $\widehat{\textsc{Datarace}}^I_{\blacklozenge}$ is not necessarily equivalent to $\textsc{Datarace}^I_{\blacklozenge}$ if I is not full, e.g., if $I = [0, 100]$ for the time domain $\mathbb{T} = \mathbb{R}$ of real numbers, a thread t accesses a variable x at every time-point, and the time-stamp differences between consecutive time-points are less than 100. In that case, $G(t, x, l)$ only holds at the very first time-point and we could not come up with an alternative definition of $G(t, x, l)$ such that $G(t, x, l)$ holds at the time-point when the thread t accesses the variable x for the first time *within the interval* I and holds the lock l. Still, our translation produces an MFOTL query $\mathsf{MFOTL2RANF}(\textsc{Datarace}^I_{\blacklozenge})$ in RANF that is equivalent to $f \approx \mathsf{c_{fin}} \land \textsc{Datarace}^I_{\blacklozenge}$ for an *arbitrary* interval I (Q_{inf} is equivalent to *false* because $\textsc{Datarace}^I_{\blacklozenge}$ is a closed query and thus it cannot be satisfied by infinitely many tuples).

The query $\textsc{Datarace}^I_{\blacklozenge}$ could be violated at a time-point i after a data race although no data race actually happened at the time-point i. Hence, we also consider the queries $\textsc{Datarace}^I$ and $\widehat{\textsc{Datarace}}^I$ obtained from $\textsc{Datarace}^I_{\blacklozenge}$ and $\widehat{\textsc{Datarace}}^I_{\blacklozenge}$ by replacing $(\blacklozenge_I\ (\mathit{read}(t_1, x) \lor \mathit{write}(t_1, x))) \land (\blacklozenge_I\ \mathit{write}(t_2, x))$ with $(\mathit{read}(t_1, x) \land \blacklozenge_I\ \mathit{write}(t_2, x)) \lor (\mathit{write}(t_1, x) \land \blacklozenge_I\ (\mathit{read}(t_2, x) \lor \mathit{write}(t_2, x)))$. Then the queries $\textsc{Datarace}^I$ and $\widehat{\textsc{Datarace}}^I$ are only satisfied at time-points at which a datarace actually happened.

We also conduct a benchmark on the query $\textsc{BlindWrite}^I$ expressing that a thread t writes to a variable x without having read the variable x while holding a lock l:

$$\textsc{BlindWrite}^I \coloneqq \neg \exists t, x, l.\ \mathit{write}(t, x) \land ((\neg(\mathit{read}(t, x) \lor \mathit{rel}(t, l)))\ \mathsf{S}_I\ \mathit{acq}(t, l)).$$

Finally, we conduct a benchmark on the query $\textsc{NeedlessRead}^I$ expressing that a thread t writes to a variable x and subsequently reads the variable x while holding a lock l:

$$\textsc{NeedlessRead}^I \coloneqq \neg \exists t, x, l.\ \mathit{write}(t, x) \land ((\neg \mathit{rel}(t, l))\ \mathsf{U}_I\ \mathit{read}(t, x)) \land ((\neg \mathit{rel}(t, l))\ \mathsf{S}_I\ \mathit{acq}(t, l)).$$

Traces We use the traces from Havelund et al. [42] and also generate our own pseudorandom traces. Every time-point always carries a single event in these traces. A trace from Havelund et al. [42] consists of a sequence of blocks of time-points. Given a number of locks n_{lock} and a number of variables n_{var}, the time-points within a block contain events for a single thread t: $\mathit{acq}(t, 1), \mathit{acq}(t, 2), \ldots, \mathit{acq}(t, n_{\mathsf{lock}}), \mathit{rel}(t, n_{\mathsf{lock}}), \mathit{rel}(t, n_{\mathsf{lock}} - 1), \ldots, \mathit{rel}(t, 2), \mathit{read}(t, 1), \mathit{write}(t, 1),$ $\mathit{read}(t, 2), \mathit{write}(t, 2), \ldots, \mathit{read}(t, n_{\mathsf{var}}), \mathit{write}(t, n_{\mathsf{var}}), \mathit{rel}(t, 1)$. At the end of the trace, there are five time-points with the following events: $\mathit{acq}(1, 1), \mathit{read}(1, -1), \mathit{rel}(1, 1), \mathit{acq}(2, 2), \mathit{write}(2, -1)$.

Given a fixed trace length ℓ, a number of threads n_{th}, a number of locks n_{lock}, and a number of variables n_{var}, we generate a single event at every time-point in our pseudorandom traces. We first choose a pseudorandom function $\mathit{lock} : \{0, \ldots, n_{\mathsf{var}} - 1\} \to \{0, \ldots, n_{\mathsf{lock}} - 1\}$ assigning a lock to every variable. Then we choose the event at a time-point $i \in \{0, \ldots, \ell - 1\}$ with time-stamp i uniformly between the following possibilities (if at least one event satisfying the corresponding condition exists):

- $\mathit{acq}(t, l)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $l \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that no thread holds lock l;
- $\mathit{acq}(t, l)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $l \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that no thread holds lock l;
- $\mathit{rel}(t, l)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $l \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that thread t holds lock l;
- $\mathit{read}(t, x)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $x \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that thread t holds lock $\mathit{lock}(x)$;
- $\mathit{read}(t, x)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $x \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that thread t holds some lock l, but t does not hold lock $\mathit{lock}(x)$;
- $\mathit{write}(t, x)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $x \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that thread t holds lock $\mathit{lock}(x)$;
- $\mathit{write}(t, x)$, where $t \in \{0, \ldots, n_{\mathsf{th}} - 1\}$ and $x \in \{0, \ldots, n_{\mathsf{lock}} - 1\}$ are chosen uniformly at random so that thread t holds some lock l, but t does not hold lock $\mathit{lock}(x)$.

The case $\mathit{acq}(t, l)$ is deliberately listed twice so that it is more likely for the threads to acquire new locks rather than release them.

Evaluation Results We use our pseudorandom trace with trace length $\ell = 20$ and $n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}} = 20$ as the training trace. We do not include the translation time in the evaluation time on the actual traces because the query $\mathsf{MFOTL2RANF}(Q)$ is only computed once for every possible trace and computing the translated query $\mathsf{MFOTL2RANF}(Q)$ takes less than 0.1 seconds on all queries Q used in our empirical evaluation. We also do not include DejaVu's compilation time in the evaluation time. We noticed that MonPoly-reg crashes on $\widehat{\textsc{Datarace}}_{\blacklozenge}^{I}/\textsc{Datarace}_{\blacklozenge}^{I}$ as well as on $\widehat{\textsc{Datarace}}^{I}/\textsc{Datarace}^{I}$ and produces incorrect results on $\textsc{BlindWrite}^{I}$. Hence, we only use MonPoly-reg on $\textsc{NeedlessRead}^{I}$.

Table 4.1 contains evaluation times for the query $\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$ on the traces from the original experiment by Havelund et al. [42]. DejaVu outperforms MFOTL monitors significantly

$\ell =$	10 005	100 005	1 050 005
	$\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$		
CppMon	673.40 s	TO	TO
ratio			
VeriMon	TO	TO	TO
ratio			
MonPoly	TO	TO	TO
ratio			
DejaVu	TO	TO	TO
ratio			
	$\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$		
DejaVu	1.24 s	3.14 s	9.59 s
ratio		2.53×	3.05×

Table 4.1. The queries $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$/$\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$ and traces from Havelund et al. [42] The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

even when using the MFOTL monitors on the manually translated query $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$. This is because binary decision diagrams (BDDs) used in DejaVu are able to compress the intermediate results for the subqueries very well given the particular structure of the original traces by Havelund et al. [42]. Still, despite the particular trace structure, DejaVu times out on the manually translated query $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$.

Table 4.2 contains evaluation times for the query $\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$ on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. This time, CppMon outperforms DejaVu when using CppMon on the manually translated query $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$. After reordering the quantifiers in the query $\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$ so that the variable x comes first, DejaVu (denoted as $\textsc{DejaVu}_{\pi}$ in Table 4.2) runs faster than DejaVu on the original query $\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$. Still, CppMon on the manually translated query $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$ outperforms $\textsc{DejaVu}_{\pi}$ and has better asymptotics.

Table 4.3 contains evaluation times for the query $\textsc{Datarace}^{[0,\infty]}$ on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. Compared to the query $\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$, CppMon and VeriMon have better asymptotics of their evaluation times on the query $\mathsf{MFOTL2RANF}(\textsc{Datarace}^{[0,\infty]})$. Otherwise, the evaluation results in Table 4.3 are similar to the evaluation results in Table 4.2.

Table 4.4 contains evaluation times for the query $\textsc{Datarace}^{[0,\ell/10]}$ on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. This time, we only use the query produced automatically by our translation MFOTL2RANF because $\widehat{\textsc{Datarace}}^{[0,\infty]}$ does not generalize to proper metric time constraints. MFOTL monitors significantly outperform DejaVu (reordering the quantifies in the query does not help DejaVu in this case).

Table 4.5 contains evaluation times for the query $\textsc{BlindWrite}^{[0,\infty]}$ on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. DejaVu significantly outperforms MFOTL monitors and DejaVu's evaluation times also have better asymptotics than MFOTL monitors' evaluation times. On the other hand, DejaVu has a significantly worse performance for a different order of the quantifiers ($\textsc{DejaVu}_{\pi}$ in Table 4.5). We conjecture that a better representation of sets

$\ell =$	250	500	1 000	2 000	4 000	8 000	16 000
				$\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}$			
CppMon	0.00 s	0.20 s	0.80 s	3.50 s	13.90 s	64.50 s	276.70 s
ratio			4.00×	4.38×	3.97×	4.64×	4.29×
VeriMon	1.30 s	5.90 s	24.10 s	106.60 s	443.40 s	TO	TO
ratio		4.54×	4.08×	4.42×	4.16×		
MonPoly	0.10 s	0.90 s	6.60 s	55.10 s	436.80 s	TO	TO
ratio		9.00×	7.33×	8.35×	7.93×		
DejaVu	TO	TO	TO	TO	TO	TO	TO
ratio							
				$\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$			
CppMon	1.00 s	8.30 s	87.00 s	828.70 s	TO	TO	TO
ratio		8.30×	10.48×	9.53×			
VeriMon	10.60 s	69.70 s	469.90 s	TO	TO	TO	TO
ratio		6.58×	6.74×				
MonPoly	4.00 s	34.60 s	365.10 s	TO	TO	TO	TO
ratio		8.65×	10.55×				
DejaVu	0.39 s	1.84 s	18.18 s	174.13 s	TO	TO	TO
ratio		4.75×	9.89×	9.58×			
DejaVu$_\pi$	0.23 s	0.40 s	0.65 s	0.99 s	13.47 s	124.38 s	724.60 s
ratio		1.77×	1.62×	1.52×	13.65×	9.23×	5.83×

Table 4.2. The queries $\widehat{\textsc{Datarace}}_{\blacklozenge}^{[0,\infty]}/\textsc{Datarace}_{\blacklozenge}^{[0,\infty]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

in VeriMon would significantly improves its performance: currently a query like $acq(t, l) \land (\blacklozenge_{[0,\infty]}\ read(t, x))$ is evaluated by grouping together tuples with the same value for t and computing Cartesian products between matching groups of the two subqueries with the same value for t. Maintaining the groups (also known as *database index*) throughout monitoring might significantly improve VeriMon's performance on this query and also other queries.

Table 4.6 contains evaluation times for the queries $\textsc{BlindWrite}^{[0,\ell/10]}/\textsc{BlindWrite}^{[\ell/10,\infty]}$ with proper metric time constraints on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. Similarly to the query $\textsc{Datarace}^{[0,\ell/10]}$ with metric time constraints, MFOTL monitors significantly outperform DejaVu (reordering the quantifies in the query does not help DejaVu in this case).

Finally, Table 4.7 contains evaluation times for the query $\textsc{NeedlessRead}^{[0,\ell/10]}$ with a bounded future temporal operator on our pseudorandom traces with $\ell = n_{\mathsf{th}} = n_{\mathsf{lock}} = n_{\mathsf{var}}$. DejaVu only supports past temporal operators and thus it cannot be used on $\textsc{NeedlessRead}^{[0,\ell/10]}$. Although $\textsc{NeedlessRead}^{[0,\ell/10]}$ is the only MFOTL query (among those benchmarked in this section) on which MonPoly-reg can be successfully executed, MFOTL monitors significantly outperform MonPoly-reg.

Overall, we conclude that using MFOTL monitors on the query $\mathsf{MFOTL2RANF}(Q)$ produced by our translation yields a significantly better performance compared to the state-of-the-art on MFOTL queries with proper metric time constraints or future temporal operators. For past-only MFOTL queries with unbounded temporal operators ($I = [0, \infty]$), further optimizations

$\ell =$	250	500	1 000	2 000	4 000	8 000	16 000
				$\widehat{\text{DATARACE}}^{[0,\infty]}$			
CPPMON	0.00 s	0.10 s	0.70 s	3.00 s	12.10 s	57.00 s	240.30 s
ratio			7.00×	4.29×	4.03×	4.71×	4.22×
VERIMON	1.00 s	4.10 s	16.40 s	71.60 s	291.60 s	TO	TO
ratio		4.10×	4.00×	4.37×	4.07×		
MONPOLY	0.00 s	0.60 s	4.30 s	36.70 s	292.10 s	TO	TO
ratio			7.17×	8.53×	7.96×		
DEJAVU	TO	TO	TO	TO	TO	TO	TO
ratio							
				$\text{DATARACE}^{[0,\infty]}$			
CPPMON	0.10 s	0.70 s	3.10 s	13.50 s	55.20 s	238.50 s	TO
ratio		7.00×	4.43×	4.35×	4.09×	4.32×	
VERIMON	2.70 s	9.10 s	38.60 s	158.80 s	693.80 s	TO	TO
ratio		3.37×	4.24×	4.11×	4.37×		
MONPOLY	0.10 s	0.80 s	6.40 s	49.00 s	376.20 s	TO	TO
ratio		8.00×	8.00×	7.66×	7.68×		
DEJAVU	0.30 s	1.27 s	10.76 s	104.93 s	TO	TO	TO
ratio		4.26×	8.45×	9.75×			
DEJAVU_π	0.21 s	0.30 s	0.54 s	0.82 s	2.03 s	70.65 s	446.05 s
ratio		1.46×	1.80×	1.52×	2.46×	34.82×	6.31×

Table 4.3. The queries $\widehat{\text{DATARACE}}^{[0,\infty]}$/$\text{DATARACE}^{[0,\infty]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

$\ell =$	250	500	1 000	2 000	4 000	8 000
CPPMON	0.00 s	0.10 s	0.60 s	2.40 s	10.70 s	47.40 s
ratio			6.00×	4.00×	4.46×	4.43×
VERIMON	1.00 s	3.90 s	14.60 s	53.70 s	227.30 s	TO
ratio		3.90×	3.74×	3.68×	4.23×	
MONPOLY	0.00 s	0.20 s	1.20 s	7.30 s	52.40 s	442.00 s
ratio			6.00×	6.08×	7.18×	8.44×
DEJAVU	1.11 s	7.28 s	158.33 s	TO	TO	TO
ratio		6.59×	21.75×			

Table 4.4. The query $\text{DATARACE}^{[0,\ell/10]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

$\ell =$	4 000	8 000	16 000	32 000
CppMon	2.50 s	10.40 s	42.70 s	171.30 s
ratio		4.16×	4.11×	4.01×
VeriMon	6.00 s	27.50 s	125.00 s	598.90 s
ratio		4.58×	4.55×	4.79×
MonPoly	0.20 s	1.00 s	4.30 s	20.60 s
ratio		5.00×	4.30×	4.79×
DejaVu	0.76 s	1.12 s	1.85 s	2.84 s
ratio		1.46×	1.66×	1.54×
DejaVu$_\pi$	2.07 s	6.29 s	23.14 s	227.32 s
ratio		3.04×	3.68×	9.83×

Table 4.5. The query BlindWrite$^{[0,\infty]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

$\ell =$	250	500	1 000	2 000	4 000	8 000	16 000
	BlindWrite$^{[0,\ell/10]}$						
CppMon	0.00 s	0.00 s	0.00 s	0.10 s	0.60 s	2.70 s	11.10 s
ratio					6.00×	4.50×	4.11×
VeriMon	0.00 s	0.10 s	0.20 s	0.50 s	1.20 s	3.60 s	12.00 s
ratio			2.00×	2.50×	2.40×	3.00×	3.33×
MonPoly	0.00 s	0.00 s	0.00 s	0.00 s	0.10 s	0.60 s	2.70 s
ratio						6.00×	4.50×
DejaVu	0.44 s	1.04 s	24.50 s	324.33 s	TO	TO	TO
ratio		2.39×	23.58×	13.24×			
	BlindWrite$^{[\ell/10,\infty]}$						
CppMon	0.10 s	0.30 s	1.50 s	6.40 s	26.30 s	120.00 s	771.80 s
ratio		3.00×	5.00×	4.27×	4.11×	4.56×	6.43×
VeriMon	1.70 s	5.90 s	27.40 s	119.20 s	524.10 s	TO	TO
ratio		3.47×	4.64×	4.35×	4.40×		
MonPoly	0.00 s	0.20 s	1.10 s	6.50 s	41.20 s	329.40 s	TO
ratio			5.50×	5.91×	6.34×	8.00×	
DejaVu	0.47 s	2.08 s	27.96 s	352.46 s	TO	TO	TO
ratio		4.42×	13.42×	12.60×			
DejaVu$_\pi$	0.71 s	2.81 s	29.90 s	TO	TO	TO	TO
ratio		3.96×	10.64×				

Table 4.6. The queries BlindWrite$^{[0,\ell/10]}$/BlindWrite$^{[\ell/10,\infty]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

$\ell =$	4 000	8 000	16 000	32 000
CppMon	1.00 s	4.50 s	20.70 s	101.30 s
ratio		4.50×	4.60×	4.89×
VeriMon	10.50 s	53.20 s	244.80 s	TO
ratio		5.07×	4.60×	
MonPoly	0.90 s	4.50 s	24.10 s	140.60 s
ratio		5.00×	5.36×	5.83×
MonPoly-reg	716.90 s	TO	TO	TO
ratio				

Table 4.7. The query NeedlessRead$^{[0,\ell/10]}$ and our pseudorandom traces. The abbreviation TO denotes a timeout of 900 seconds. The ratios are computed between consecutive trace lengths in the table.

in MFOTL monitors' representation of sets is needed to improve their performance, but the asymptotics of VeriMon's evaluation times for increasing trace length suggest that using our translation mfotl2ranf and *further optimized* MFOTL monitors might yield a better performance compared to the state-of-the-art.

4.4 MFOTL Temporal Operator Evaluation

In this section, we present algorithms for efficiently evaluating the MFOTL temporal operators Since (S_I) and Until (U_I). The algorithm for S_I consists of the functions $\mathsf{init}_\mathsf{S} : \mathbb{A} \to \mathbb{M}_\mathsf{S}$ and $\mathsf{adv}_\mathsf{S} : \mathbb{M}_\mathsf{S} \times \mathbb{T} \times \mathcal{P}(\mathcal{D}^*) \times \mathcal{P}(\mathcal{D}^*) \to \mathbb{M}_\mathsf{S} \times \mathcal{P}(\mathcal{D}^*)$ that initialize and update a state $m \in \mathbb{M}_\mathsf{S}$ for evaluating S_I. The initialization function init_S takes fixed arguments *args* $\in \mathbb{A}$ of S_I (e.g., the interval I and the free variables of subqueries) as input and yields a state for evaluating S_I. We assume that the free variables of Q_1 are a subset of the free variables of Q_2 in $Q_1 \ \mathsf{S}_I \ Q_2$. The update function adv_S takes a state, a time-stamp of a time-point i, and two sets of tuples satisfying the subqueries Q_1 and Q_2 of $Q_1 \ \mathsf{S}_I \ Q_2$ at the time-point i, and yields an updated state and a set of tuples satisfying $Q_1 \ \mathsf{S}_I \ Q_2$ at the time-point i.

The algorithm for U_I consists of the functions $\mathsf{init}_\mathsf{U} : \mathbb{A} \to \mathbb{M}_\mathsf{U}$ and $\mathsf{adv}_\mathsf{U} : \mathbb{M}_\mathsf{U} \times \mathbb{T} \times \mathcal{P}(\mathcal{D}^*) \times \mathcal{P}(\mathcal{D}^*) \to \mathbb{M}_\mathsf{U} \times \mathcal{P}(\mathcal{D}^*)^*$ that initialize and update a state $m \in \mathbb{M}_\mathsf{U}$ for evaluating U_I. The functions init_U and adv_U for U_I are analogous to the corresponding functions for S_I, but while adv_S always yields a single set of tuples satisfying $Q_1 \ \mathsf{S}_I \ Q_2$ at the most recent time-point i, adv_U yields a sequence of sets of tuples satisfying $Q_1 \ \mathsf{U}_I \ Q_2$ at a sequence of consecutive time-points for which $Q_1 \ \mathsf{U}_I \ Q_2$ could be evaluated. This is because the set of tuples satisfying $Q_1 \ \mathsf{U}_I \ Q_2$ at a time-point i might depend on the sets of tuples satisfying the subqueries Q_1 and Q_2 at future time-points after i. Hence, the evaluation must be delayed until those sets of tuples are computed. But once they are received, sets of tuples satisfying $Q_1 \ \mathsf{U}_I \ Q_2$ at potentially multiple consecutive time-points can be computed.

We have implemented the functions init_S (init_U, respectively) and adv_S (adv_U, respectively) in VeriMon [9] using the Isabelle/HOL proof assistant [24] and thus their correctness is formally verified. Our implementation of the functions init_S (init_U, respectively) and adv_S (adv_U, respectively) improves upon the implementation of similar functions in MonPoly [10] and their formalization in VeriMon$^-$ [71] whose worst-case time complexity at a single time-point i depends on the total number of time-points processed so far. In contrast, the total time

data_in	list of $\mathbb{T} \times \mathcal{P}((\mathcal{D})^*)$
data_prev	list of $\mathbb{T} \times \mathcal{P}((\mathcal{D})^*)$
tuple_in	$(\mathcal{D})^* \to \mathbb{T} \cup \{\bot\}$
tuple_since	$(\mathcal{D})^* \to \mathbb{T} \cup \{\bot\}$
result	$\mathcal{P}((\mathcal{D})^*)$

Figure 4.13. The types of selected components in the optimized state for S_I.

complexity of adv_S (adv_U, respectively) to process i time-points is just linear in the total number of tuples in the two input sets of tuples at these i time-points. Hence, the amortized time complexity of adv_S (adv_U, respectively) at a single time-point i does not depend on the total number of time-points processed so far. The space complexity of adv_S (adv_U, respectively) is linear in the total number of tuples in the two input sets of tuples processed so far. The time and space complexity of init_S (init_U, respectively) is constant.

4.4.1 Since Operator

We first describe the evaluation of $Q_1\ \mathsf{S}_I\ Q_2$ in VeriMon$^-$. Suppose that the most recent time-point is i with time-stamp τ. The monitor's state for $Q_1\ \mathsf{S}_I\ Q_2$ consists of a list of sets of tuples $T_{\tau'}$ along with time-stamps τ' such that the tuples satisfy Q_2 at a time-point j with the time-stamp τ' and they satisfy Q_1 at all time-points k such that $j < k \leq i$. Note that VeriMon$^-$'s state also contains sets $T_{\tau'}$ of satisfying tuples for time-stamps τ' that are not yet in the interval, i.e., $\mathsf{mem}(\tau', \tau, I)$ does not hold. VeriMon$^-$'s state is updated for every new time-point with time-stamp τ'' for which we already know the two input tables R_{Q_1} and R_{Q_2} for the subqueries Q_1 and Q_2. The update consists of the following three steps:

(1) remove tables that fall out of the interval with respect to the new time-stamp τ'';

(2) evaluate the conjunction of each remaining table with R_{Q_1} using a relational join; and

(3) add the new tuples from R_{Q_2}, either by inserting them into the most recent table T_τ (if $\tau'' = \tau$) or by adding a new table $T_{\tau''}$ (otherwise).

Finally, we take the union of all tables within the interval (i.e., tables $T_{\tau'}$ such that $\mathsf{mem}(\tau', \tau'', I)$ holds) to obtain the set of tuples satisfying $Q_1\ \mathsf{S}_{[a,b]}\ Q_2$. As the collections of these tables often overlap between consecutive evaluation steps (invokations of adv_S), recomputing the union from scratch at every time-point is inefficient. Hence, we design an optimized state that represents the information in VeriMon$^-$'s state such that its update and evaluation can be performed more efficiently. In our optimized state, we partition the list of tables $T_{\tau'}$ into a list *data_prev* for time-stamps that are not yet in the interval and a list *data_in* for time-stamps that are already in the interval. The optimized state also contains a mapping *tuple_in* that assigns to each tuple occurring in some table $T_{\tau'}$ in the interval (i.e., in *data_in*) the *latest* time-stamp $\tau^\uparrow$ in the interval (i.e., such that $\mathsf{mem}(\tau^\uparrow, \tau, I)$ holds) for which this tuple occurs in the respective table $T_{\tau^\uparrow}$. Finally, the optimized state contains a mapping *tuple_since* that assigns to each tuple occurring in some table $T_{\tau'}$ in *data_in* or *data_prev* the *earliest* time-stamp $\tau^\downarrow$ for which this tuple occurs in the respective table $T_{\tau^\downarrow}$. The types of selected components in the optmized state for S_I are summarized in Figure 4.13.

For the sake of performance, we further implement the following adjustments: We split each table $T_{\tau'}$ for a time-stamp τ' with multiple time-points into a list of tables, one for each time-point.

(Overall, *data_prev* and *data_in* are still lists of tables and the corresponding time-stamps that might not be pairwise distinct.) We do not remove any tuples from the lists *data_in* and *data_prev* because their length can be as high as the number of time-points processed so far and traversing them might be expensive. Instead, we delete tuples from the two mappings *tuple_in* and *tuple_since*. Finally, we update tuples in *tuple_since* lazily, i.e., only at defined garbage collection points such that the mapping *tuple_since* may even contain tuples and time-stamps τ' from some table $T_{\tau'}$ that has already fallen out of the interval (i.e., $\mathsf{memR}(\tau', \tau, I)$ does not hold). To determine the garbage collection points, the optimized state stores the time-stamp of the last time-point at which garbage collection was performed (or 0 if no garbage collection has been performed so far) and the time-stamp of the most recent time-point. Garbage collection is performed whenever these two time-points do not satisfy the interval's upper bound condition.

The function $\mathsf{init_S}$ initializes the optimized state to consist of empty lists and empty mappings. The function $\mathsf{adv_S}$ updates the optimized state implementing the steps (1)–(3) outlined above:

(1)* we drop tables from *data_in* that fall out of the interval based on the newly received time-stamp τ''; we remove these tuples also from *tuple_in* if their latest occurrence (which is stored in this mapping) in *data_in* has fallen out of the interval; we move tables that newly enter the interval from *data_prev* to *data_in*, and update the tuples from these moved tables in *tuple_in* to the most recent time-stamp $\tau^{\uparrow}$ for which they now occur in the interval, but only if *tuple_since* maps such a tuple to a time-stamp $\tau^{\downarrow}$ such that $\tau^{\downarrow} \leq \tau^{\uparrow}$ (otherwise the tuple is actually not in the corresponding table $T_{\tau^{\uparrow}}$);

(2)* we delete the tuples that are not matched by any tuple in the given table R_{Q_1} from the mappings *tuple_since* and *tuple_in*; to efficiently determine the tuples to be deleted, we additionally store the tuples (over the free variables of Q_2) in the domains of the mappings *tuple_since* and *tuple_in* grouped by their projections to the free variables of Q_1, where $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2)$;

(3)* we append the new table R_{Q_2} to *data_prev* (or directly *data_in* if $\mathsf{mem}(\tau'', \tau'', I)$ holds), add the tuples from R_{Q_2} that were not in *tuple_since* to that mapping (with value τ''), and, if $\mathsf{mem}(\tau'', \tau'', I)$, update the tuples from R_{Q_2} in the mapping *tuple_in* to the current time-stamp τ''.

This way, $\mathsf{adv_S}$ has computed the updated optimized state and it only remains to compute the set of tuples satisfying $Q_1 \; \mathsf{U}_I \; Q_2$. To this end, VeriMon^{-} would compute the union of tables $T_{\tau'}$ with time-stamp τ' in the interval. In our optimized state, this union corresponds to the domain of the mapping *tuple_in*. We actually store the domain of the mapping *tuple_in* separately as a set of tuples *result* in our optimized state to avoid traversing the mapping *tuple_in* from scratch to fetch its domain. Crucially, and unlike in VeriMon^{-}'s state, the join operation does not change the tables $T_{\tau'}$, i.e., *data_in* and *data_prev*, in our optimized state. This functionality is implemented more efficiently by filtering the two mappings *tuple_since* and *tuple_in*.

Example 4.29. Figure 4.14 shows how the optimized state for the query $P(x) \; \mathsf{S}_{[2,4]} \; Q(x)$ is updated. In total, four time-points are processed. The first two columns show the time-stamp and database for each time-point. The last four columns show the optimized state after the step named in the third column. The satisfactions $\{\}, \{\}, \{b, c\}, \{a\}$ computed by $\mathsf{adv_S}$ can be read off directly from the domain of the mapping *tuple_in* after each time-point's last step (3)*. We omit steps that do not change the state.

time-stamp	database	step	*data_prev*	*data_in*	*tuple_in*	*tuple_since*
		init_S	$[]$	$[]$	$\{\}$	$\{\}$
1	$\{Q(a), Q(b), Q(c)\}$	(3)*	$[(1, \{a, b, c\})]$	$[]$	$\{\}$	$\{a \mapsto 1, b \mapsto 1, c \mapsto 1\}$
2	$\{P(b), P(c)\}$	(2)*	$[(1, \{a, b, c\})]$	$[]$	$\{\}$	$\{b \mapsto 1, c \mapsto 1\}$
		(3)*	$[(1, \{a, b, c\}), (2, \{\})]$	$[]$	$\{\}$	$\{b \mapsto 1, c \mapsto 1\}$
3	$\{P(b), P(c), Q(a), Q(b)\}$	(1)*	$[(2, \{\})]$	$[(1, \{a, b, c\})]$	$\{b \mapsto 1, c \mapsto 1\}$	$\{b \mapsto 1, c \mapsto 1\}$
		(3)*	$[(2, \{\}), (3, \{a, b\})]$	$[(1, \{a, b, c\})]$	$\{b \mapsto 1, c \mapsto 1\}$	$\{a \mapsto 3, b \mapsto 1, c \mapsto 1\}$
7	$\{P(a)\}$	(1)*	$[]$	$[(3, \{a, b\})]$	$\{a \mapsto 3, b \mapsto 3\}$	$\{a \mapsto 3, b \mapsto 1, c \mapsto 1\}$
		(2)*	$[]$	$[(3, \{a, b\})]$	$\{a \mapsto 3\}$	$\{a \mapsto 3\}$
		(3)*	$[(7, \{\})]$	$[(3, \{a, b\})]$	$\{a \mapsto 3\}$	$\{a \mapsto 3\}$

Figure 4.14. An example of updating the optimized state for the query $P(x) \mathsf{S}_{[2,4]} Q(x)$.

The first row shows the initial state. For the first time-point, the steps (1)* with time-stamp 1 and (2)* with the empty table $\{\}$ (as there are no P events) do not change the initial state. In step (3)*, the table $\{a, b, c\}$ with the parameters of the Q events is appended to *data_prev* (as $\mathsf{mem}(1, 1, [2, 4])$ does not hold) and its elements are added to *tuple_since*.

For the second time-point, the step (1)* with time-stamp 2 has again no effect: *data_prev*'s first entry is not moved to *data_in* as the initial time-point with time-stamp 1 is not in the interval yet ($\mathsf{mem}(1, 2, [2, 4])$ does not hold). There is also nothing to drop in *data_in* which is empty. In step (2)* with the table $\{b, c\}$ containing the parameters of the P events, the entry a is deleted from *tuple_since*, but not from *data_prev*. In step (3)*, the empty table $\{\}$ (as there are no Q events at the second time-point) is appended to *data_prev*.

For the third time-point with time-stamp 3, we move *data_prev*'s first entry to *data_in* in step (1)* because the initial time-point with time-stamp 1 is now in the interval ($\mathsf{mem}(1, 3, [2, 4])$ holds). The tuples b, c of that entry are added to *tuple_in* because *tuple_since* maps them to a time-stamp that is at most 1. Note that a is not added because it is not contained in *tuple_since*. Because the second time-point with time-stamp 2 is not yet in the interval ($\mathsf{mem}(2, 3, [2, 4])$ does not hold), the second entry in *data_prev* is not moved to *data_in* yet. Step (2)* with the table $\{b, c\}$ does not change the state because the domain of the mappings *tuple_in* and *tuple_since* only contains elements from this table $\{b, c\}$. In step (3)*, the table $\{a, b\}$ with the parameters of the Q events is appended to *data_prev*. Now, a is added to *tuple_since* because it was not contained in the mapping, but b is already contained in *tuple_since* and thus its value is not updated.

When the fourth time-point with time-stamp 7 is processed, the first two time-stamps fall out of the interval and their corresponding entries in *data_prev* and *data_in* are discarded in step (1)*. The tuples from these entries are also discarded from *tuple_in* because they are mapped to time-stamps that are not in the interval anymore, but not from *tuple_since* because *tuple_since* is updated lazily. As before, the last table $\{a, b\}$ in *data_prev* is moved to *data_in* and its elements are added to *tuple_in*. As the time from the last garbage collection (or the start of monitoring) has progressed by more than the upper bound of the interval $[2, 4]$, garbage collection

ts	$\mathbb{N} \to \mathbb{T} \cup \{\bot\}$
tables	list of $\mathcal{P}((\mathcal{D})^*) \times (\mathbb{T} + \mathbb{N})$
a_1	$(\mathcal{D})^* \to \mathbb{N} \cup \{\bot\}$
$\overline{a_2}$	list of $(\mathcal{D})^* \to (\mathbb{T} + \mathbb{N}) \cup \{\bot\}$
result	$\mathcal{P}((\mathcal{D})^*)$

Figure 4.15. The types of selected components in the optimized state for U_I.

is performed in step (2)*, which removes the key c from *tuple_since*. The join operation in step (2)* further removes b from *tuple_in* and *tuple_since* (as it is not contained in the table $\{a\}$ of parameters of the P events). Finally, the empty table $\{\}$ with the parameters of the Q events is appended to *data_prev* in step (3)*. $\diamond$

4.4.2 Until Operator

We first describe the evaluation of $Q_1 \mathsf{U}_I Q_2$ in VeriMon$^-$. Suppose that the most recent time-point is i with time-stamp τ. The monitor's state for $Q_1 \mathsf{U}_I Q_2$ consists of a list of entries for every time-point $j \le i$ with time-stamp τ' such that $\mathsf{memR}(\tau', \tau, I)$ holds, i.e., there is an entry for every time-point $j \le i$ at which the satisfaction of $Q_1 \mathsf{U}_I Q_2$ might still be influenced by time-points strictly beyond i (that we have not received yet). For every such time-point $j \le i$, VeriMon$^-$ stores its time-stamp τ', the set of tuples T_1 satisfying Q_1 at all time-points k such that $j \le k \le i$, and the set of tuples T_2 satisfying $Q_1 \mathsf{U}_I Q_2$ at the time-point j given only the time-points up to i. VeriMon$^-$'s state is evaluated and updated for every new time-point with time-stamp τ'' for which we already know the two input tables R_{Q_1} and R_{Q_2} for the subqueries Q_1 and Q_2. The evaluation and update consist of the following four steps:

(1) for every time-point j with time-stamp τ' such that $\mathsf{memR}(\tau', \tau'', I)$ does not hold (i.e., neither the new time-point nor any time-point afterwards can potentially influence the satisfaction of $Q_1 \mathsf{U}_I Q_2$ at the time-point j), we return the set of tuples T_2 as the set of tuples satisfying $Q_1 \mathsf{U}_I Q_2$ at the time-point j and drop the corresponding entry from the monitor's state;

(2) for every time-point j with time-stamp τ' such that $\mathsf{memL}(\tau', \tau'', I)$ holds, we use a relational join to compute the conjunction of R_{Q_2} with the set of tuples T_1 satisfying Q_1 and add the resulting tuples to the set of tuples T_2 satisfying $Q_1 \mathsf{U}_I Q_2$ at the time-point j;

(3) for every time-point j with a corresponding entry in the monitor's state, we update the set of tuples T_1 satisfying Q_1 by computing its conjunction with R_{Q_1} using a relational join;

(4) we create a new entry for the new time-point with time-stamp τ'', with the table R_{Q_1} as T_1 and, if $\mathsf{mem}(\tau'', \tau'', I)$, with the table R_{Q_2} as T_2, and otherwise with the empty table $\emptyset$ as T_2.

In VeriMon, we use an optimized state that represents the information in VeriMon$^-$'s state such that its evaluation and update can be performed more efficiently. We represent the tables T_1 by a single mapping a_1 of tuples satisfying Q_1 to the minimum time-point j such that the tuples satisfy Q_1 at all time-points k such that $j \le k \le i$. We represent the tables T_2 by a list $\overline{a_2}$ of mappings a_2 of tuples satisfying $Q_1 \mathsf{U}_I Q_2$ to the maximum time-point $k \le i$ (or its time-stamp if $\mathsf{memL}(0, 0, I)$ does not hold) at which they still satisfy $Q_1 \mathsf{U}_{\mathsf{dropL}(I)} Q_2$ (note that the interval I's lower bound condition is ignored). The optimized state also contains the time-point $tp := i + 1$, a list tss of time-stamps at time-points for which the set of tuples satisfying $Q_1 \mathsf{U}_I Q_2$ could not be computed yet, the number len of such time-points, and a mapping ts of time-points for which the set of tuples satisfying $Q_1 \mathsf{U}_I Q_2$ could not be computed yet to their time-stamps (that also occur in the list tss). The state contains both tss and ts to account for different access pattern to the time-stamps.

The function init_S initializes the optimized state to consist of empty lists, empty sets, and empty mappings (init_S also sets $tp := 0$). The function adv_S evaluates and updates the optimized state for a new time-point $tp = i + 1$ with time-stamp τ'' implementing the steps (1)–(4) outlined above:

(1)* for every time-point j with time-stamp τ' such that $\mathsf{memR}(\tau', \tau'', I)$ does not hold, we successively return the keys of the mapping a_2 for the time-point j as the set of tuples satisfying $Q_1 \mathsf{U}_I Q_2$ at the time-point j, drop the keys from the mapping a_2 that are assigned the time-point j (if $\mathsf{memL}(0, 0, I)$ holds) or that would not satisfy the interval I's lower bound condition at the next time-point $j + 1$ (if $\mathsf{memL}(0, 0, I)$ does not hold), and merge the mapping a_2 with the mapping a_2 for the next time-point $j + 1$ (if the list $\overline{a_2}$ contains such a mapping);

(2)* for every tuple in R_{Q_2}, we use the mapping a_1 to determine the earliest time-point j such that Q_1 is satisfied at all time-points k such that $j \leq k \leq i$ and, if $\mathsf{memL}(ts(j), \tau'', I)$ holds, we assign the new time-point tp (or its time-stamp τ'' if $\mathsf{memL}(0, 0, I)$ does not hold) to this tuple in the mapping a_2 at the time-point j;

(3)* we drop all tuples from the mapping a_1 that are not in the set R_{Q_1} and add all tuples in the set R_{Q_1} that are not in the mapping a_1 to the mapping a_1 (with values tp);

(4)* we create a new mapping a_2 for the new time-point tp and, if $\mathsf{mem}(\tau'', \tau'', I)$, add all tuples in R_{Q_2} as its keys (with values tp or τ'').

In step (1)*, for every time-point j with time-stamp τ' such that $\mathsf{memR}(\tau', \tau'', I)$ does not hold, we successively return the keys of the mapping a_2 for that time-point j. To efficiently determine these keys, the optimized state stores them explicitly as a set *result*. To efficiently determine the keys from this mapping a_2 to be dropped when combining the mapping a_2 for the time-point j with the mapping a_2 for the next time-point $j+1$, the optimized state also maintains a list *tables* of input tables R_{Q_2} (with their corresponding time-points, if $\mathsf{memL}(0, 0, I)$ holds, or time-stamps, if $\mathsf{memL}(0, 0, I)$ does not hold) because the tuples are dropped from the mappings $\overline{a_2}$ in this order. The types of selected components in the optmized state for U_I are summarized in Figure 4.15. Values that are either time-points (if $\mathsf{memL}(0, 0, I)$ holds) or time-stamps (if $\mathsf{memL}(0, 0, I)$ does not hold) are represented by the *sum* type $\mathbb{T} + \mathbb{N}$. The lists in Figure 4.15 are actually implemented as queues or mappings (of positions to the values in the list) to be able to perform lookups efficiently.

Example 4.30. Figure 4.16 shows how the optimized state for the query $P(x) \mathsf{U}_{[2,4]} Q(x)$ is updated. For the sake of simplicity, we do not show all components of the optimized state. In total, four time-points are processed. The first two columns show the time-stamp and database for each time-point. The third column contains an evaluation step. The next three columns show selected components of the optimized state after the step named in the third column. The

time-stamp	database	step	ts	a_1	$\overline{a_2}$	$done$
		init_S	$\{\}$	$\{\}$	$[\,]$	
1	$\{P(a)\}$	(3)*	$\{\}$	$\{a \mapsto 0\}$	$[\,]$	
		(4)*	$\{0 \mapsto 1\}$	$\{a \mapsto 0\}$	$[\{\}]$	
2	$\{P(a), P(b)\}$	(3)*	$\{0 \mapsto 1\}$	$\{a \mapsto 0, b \mapsto 1\}$	$[\{\}]$	
		(4)*	$\{0 \mapsto 1,$ $1 \mapsto 2\}$	$\{a \mapsto 0, b \mapsto 1\}$	$[\{\}, \{\}]$	
3	$\{P(b), Q(a)\}$	(2)*	$\{0 \mapsto 1,$ $1 \mapsto 2\}$	$\{a \mapsto 0, b \mapsto 1\}$	$[\{a \mapsto 3\}, \{\}]$	
		(3)*	$\{0 \mapsto 1,$ $1 \mapsto 2\}$	$\{b \mapsto 1\}$	$[\{a \mapsto 3\}, \{\}]$	
		(4)*	$\{0 \mapsto 1,$ $1 \mapsto 2,$ $2 \mapsto 3\}$	$\{b \mapsto 1\}$	$[\{a \mapsto 3\}, \{\}, \{\}]$	
7	$\{Q(b)\}$	(1)*	$\{2 \mapsto 3\}$	$\{b \mapsto 1\}$	$[\{\}]$	$[\{a\}, \{\}]$
		(2)*	$\{2 \mapsto 3\}$	$\{b \mapsto 1\}$	$[\{b \mapsto 7\}]$	
		(4)*	$\{2 \mapsto 3,$ $3 \mapsto 7\}$	$\{b \mapsto 1\}$	$[\{b \mapsto 7\}, \{\}]$	

Figure 4.16. An example of updating the optimized state for the query $P(x)\ \mathsf{U}_{[2,4]}\ Q(x)$.

satisfactions $\{a\}, \{\}$ computed by adv_U are listed in the last column *done* after each time-point's step (1)*. We omit steps that do not change the state.

The first row shows the initial state. For the first time-point, the steps (1)* and (2)* do not change the initial state. In step (3)*, the table $\{a\}$ with the parameters of the P events is added to the mapping a_1 with values 0 corresponding to the current time-point 0. In step (4)*, the time-stamp 1 of the current time-point is added to the mapping ts of time-points to their time-stamps and a new mapping a_2 (without any keys because $\mathsf{mem}(1, 1, [2, 4])$ does not hold) is added to the list of mappings $\overline{a_2}$.

For the second time-point, the steps (1)* and (2)* again do not change the state. In step (3)*, the table $\{a, b\}$ with the parameters of the P events is added to the mapping a_1 with values 1 corresponding to the current time-point 1. The tuple a with value 0 is not updated in the mapping a_1 because it was already contained in it. In step (4)*, the time-stamp 2 of the current time-point is added to the mapping ts of time-points to their time-stamps and a new empty mapping a_2 is added to the list of mappings $\overline{a_2}$.

For the third time-point, in step (2)*, we use the mapping a_1 to determine that the tuple a with a corresponding Q event in the database satisfies the left-hand side of the Until operator since the time-point 0. Because $\mathsf{memL}(ts(0), 3, [2, 4])$ holds, we add a with value 3 corresponding to the time-stamp (because $\mathsf{memL}(0, 0, [2, 4])$ does not hold) of the current time-point to the mapping a_2 for the time-point 0 (the first entry in the list of mappings $\overline{a_2}$). In step (3)*, the tuple a is dropped from the mapping a_1 because it has no corresponding P event in the current database. In step (4)*, the time-stamp 3 of the current time-point is added to the mapping ts of time-points to their time-stamps and a new empty mapping a_2 is added to the list of mappings $\overline{a_2}$.

For the fourth time-point, in step (1)*, evaluation results for the query $P(x)\ \mathsf{U}_{[2,4]}\ Q(x)$ at the first two time-points are produced (because $\mathsf{memR}(1, 7, [2, 4])$ and $\mathsf{memR}(2, 7, [2, 4])$ do not

Name	Query
Once	$q(x, y) \wedge (\blacklozenge_I\ r(x, y))$
Since	$q(x, y) \wedge (s(x)\ \mathsf{S}_I\ r(x, y))$
NotSince	$q(x, y) \wedge (\neg s(x)\ \mathsf{S}_I\ r(x, y))$
Eventually	$q(x, y) \wedge (\lozenge_I\ r(x, y))$
Until	$q(x, y) \wedge (s(x)\ \mathsf{U}_I\ r(x, y))$
NotUntil	$q(x, y) \wedge (\neg s(x)\ \mathsf{U}_I\ r(x, y))$

Figure 4.17. MFOTL queries for benchmarking MFOTL temporal operator evaluation.

hold). The set $\{a\}$ of satisfactions at the first time-point is obtained directly from the keys of the mapping a_2 for the first time-point. Then this mapping $\{a \mapsto 3\}$ is combined with the empty mapping a_2 for the second time-point. Because the lower bound condition $\mathsf{memL}(2, 3, [2, 4])$ does not hold for the second time-point with time-stamp $ts(1) = 2$ and the third time-point with time-stamp 3 from the mapping $\{a \mapsto 3\}$, we get an empty mapping a_2 for the second time-point. Consequently, the set of satisfactions at the second time-point is empty and combining the empty mapping a_2 for the second and third time-point yields an empty mapping a_2 for the third time-point. In step (2)*, we use the mapping a_1 to determine that the tuple b with a corresponding Q event in the database satisfies the left-hand side of the Until operator since the time-point 1. Because the list $\overline{a_2}$ contains no a_2 for the time-point 1 and $\mathsf{memL}(ts(2), 7, [2, 4])$ holds for the first time-point 2 with time-stamp 3 in the mapping ts and the current time-point with time-stamp 7, we add b with value 7 corresponding to the time-stamp of the current time-point to the mapping a_2 for the time-point 2 (the first entry in the list $\overline{a_2}$). In step (4)*, the time-stamp 7 of the current time-point is added to the mapping ts of time-points to their time-stamps and a new empty mapping a_2 is added to the list of mappings a_2. $\diamond$

4.4.3 Implementation and Evaluation

We have integrated our optimized state and update and evaluation functions for the Since and Until temporal operators into VeriMon and exported verified OCaml code from VeriMon's formalization using Isabelle/HOL. VeriMon reuses MonPoly's unverified OCaml code for parsing the query and trace file and outputting verdicts. Several optimizations of VeriMon's algorithm for evaluating the Since and Until temporal operators have been implemented and formally verified in Isabelle/HOL by Emanuele Marsicano [51].

We empirically evaluate the time and space complexity of VeriMon's Since and Until temporal operators by answering the following two research questions:

RQ1: *How does VeriMon scale with respect to the event rate?*

RQ2: *How does VeriMon scale with respect to the magnitude of time constaints?*

To answer these research questions, we conduct a series of experiments measuring the time and space usage of VeriMon and the state-of-the-art tool MonPoly [10]. We also include CppMon [39] in our empirical evaluation because it is supposed to implement VeriMon's algorithm in C++. Our empirical evaluation can be reproduced using a publicly available artifact [62]. The experimental setup is as described in Section 3.3.

Queries Our implementation of the algorithms for efficiently evaluating S_I and U_I also supports queries of the form $(\neg Q_1)\ \mathsf{S}_I\ Q_2$ and $(\neg Q_1)\ \mathsf{U}_I\ Q_2$, where Q_1 and Q_2 are MFOTL queries in RANF with finite sets of tuples satisfying Q_1 and Q_2 at every time-point. Figure 4.17 lists the MFOTL queries used in our experiments. We use the interval $I = [10, 20]$ to answer RQ1 and $I = [n, 2 \cdot n]$, where n is a parameter, to answer RQ2.

The subquery $q(x, y)$ is used to check if the temporal operator is satisfied by a fixed tuple (x, y). The traces in our experiments contain exactly one tuple at every time-point that satisfies $q(x, y)$. This way, the overall query's output at a time-point is limited to at most a single tuple. If the subquery $q(x, y)$ was omitted, the monitor would have to output all tuples satisfying the temporal operator. In that case, the time complexity of evaluating VeriMon's Since and Until temporal operators would be dominated by the time complexity of outputting all their satisfying tuples.

Traces Given a fixed trace length ℓ and a query Q, we generate a trace with $\mathcal{O}(\ell)$ events in total such that the temporal operator is satisfied by as many tuples as there are time-points within its interval (up to a constant factor) and the overall query is satisfied at a constant fraction of all time-points. To this end, we generate the following events at every time-point $i \in \{0, \ldots, \ell - 1\}$:

- one event $r(x, y)$, where

  $$x \in \begin{cases} \{0, \ldots, 9\} & \text{if } Q \in \{\mathsf{Since}, \mathsf{Until}\}, \\ \{0, \ldots, \ell - 1\} & \text{otherwise,} \end{cases}$$

  and $y \in \{0, \ldots, \ell - 1\}$ are chosen uniformly at random;
- if $Q \in \{\mathsf{Since}, \mathsf{Until}\}$, events $s(x)$, for all x satisfying $r(x, y)$ at some past (future, respectively) time-point, independently with probability $1 - \frac{1}{\ell}$;
- if $Q \in \{\mathsf{NotSince}, \mathsf{NotUntil}\}$, one event $s(x)$, where we randomly choose between (i) x satisfying $r(x, y)$ at some past (future, respectively) time-point chosen uniformly at random, (ii) $x \in \{0, \ldots, \ell - 1\}$ chosen uniformly at random;
- one event $q(x, y)$, where we randomly choose between (i) x, y satisfying $r(x, y)$ at some past (future, respectively) time-point within the interval I chosen uniformly at random (if at least one such time-point exists), (ii) $x, y \in \{0, \ldots, \ell - 1\}$ chosen uniformly at random.

We use consecutive time-stamps $0, 1, \ldots$ with the same time-stamp for every block of er consecutive time-points with the same time-stamp.

Increasing Event Rate We now answer RQ1 by validating that the time complexity of evaluating VeriMon's Since and Until temporal operators is event-rate independent. To this end, we increase the event rate $er \in \{20, \ldots, 200\}$. We use a fixed trace length $\ell = 20\,000$.

Figure 4.18 contains the evaluation results for the six queries Once, Since, NotSince, Eventually, Until, and NotUntil when increasing the event rate. We omit the space usage because we expect the space usage of all tools to depend on the event rate. The evaluation results confirm that the time complexity of evaluating VeriMon's Since and Until temporal operators is event-rate independent. In contrast, the time complexity of evaluating MonPoly's Since and Until temporal operators depends on the event-rate except for the two queries Once and Eventually whose evaluation is optimized in MonPoly and MonPoly also performs significantly better in those cases. CppMon's time complexity of evaluating Since and Until temporal operators depends on the event rate for all the six queries. This shows that CppMon does not faithfully implement all VeriMon's optimizations.

Increasing Interval Bounds We now answer RQ2 by validating that the time complexity of evaluating VeriMon's Since and Until temporal operators does not depend on the magnitude of time constraints, i.e., that its time complexity is *interval-oblivious*. To this end, we increase the parameter $n \in \{200, \dots, 2\,000\}$ inducing the interval bounds $I = [n, 2 \cdot n]$. We use a fixed trace length $\ell = 20\,000$ and event rate $er = 1$.

Figure 4.19 contains the evaluation results for the six queries Once, Since, NotSince, Eventually, Until, and NotUntil when increasing the parameter n and thus the interval bounds. We omit the space usage because we expect the space usage of all tools to depend on the interval bounds. The evaluation results confirm that the time complexity of evaluating VeriMon's Since and Until temporal operators is interval-oblivious. In contrast, the time complexity of evaluating MonPoly's (and CppMon's) Since and Until temporal operators depends on the interval bounds for all the six queries.

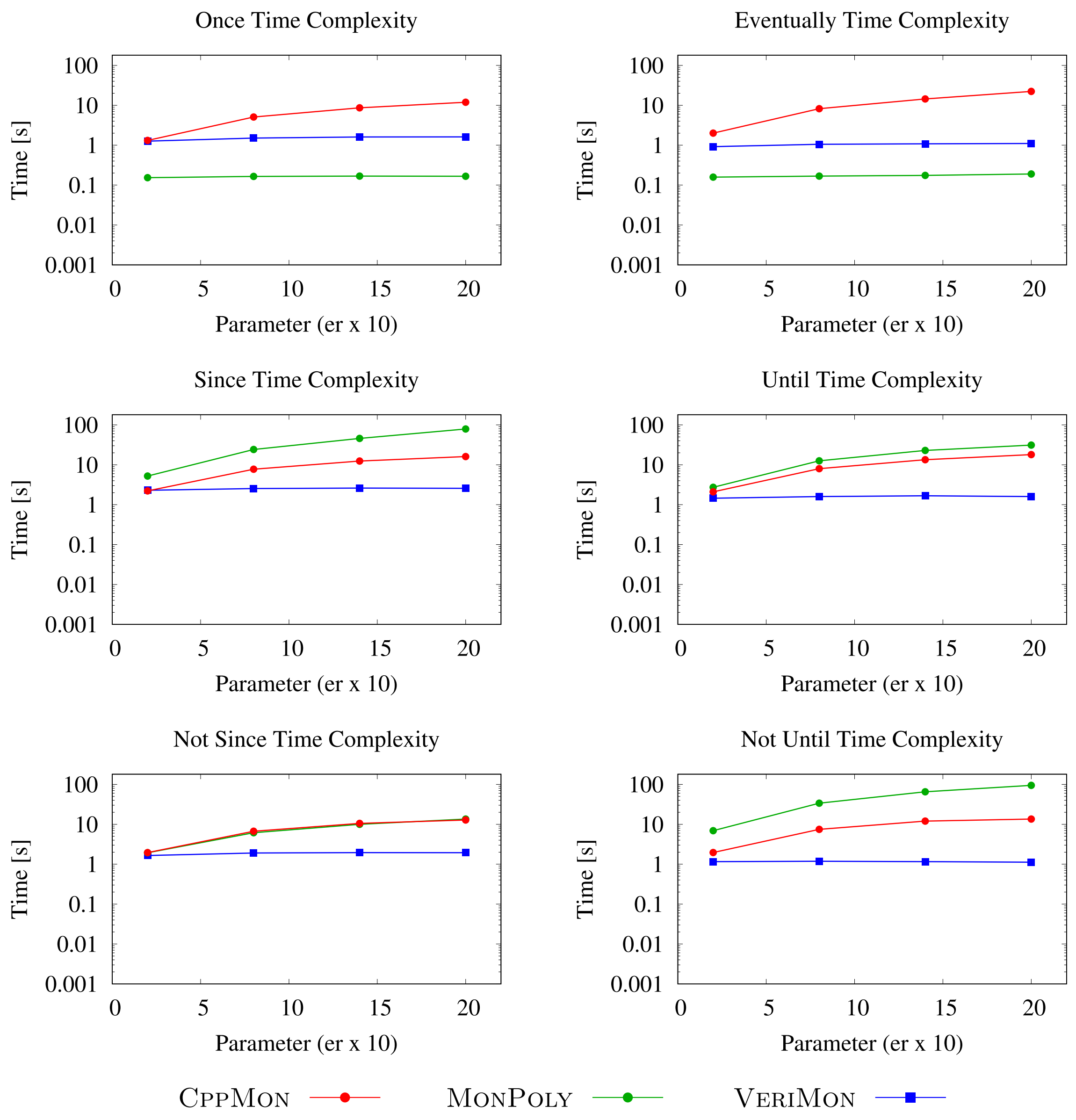


Figure 4.18. Evaluation results for increasing event rate.

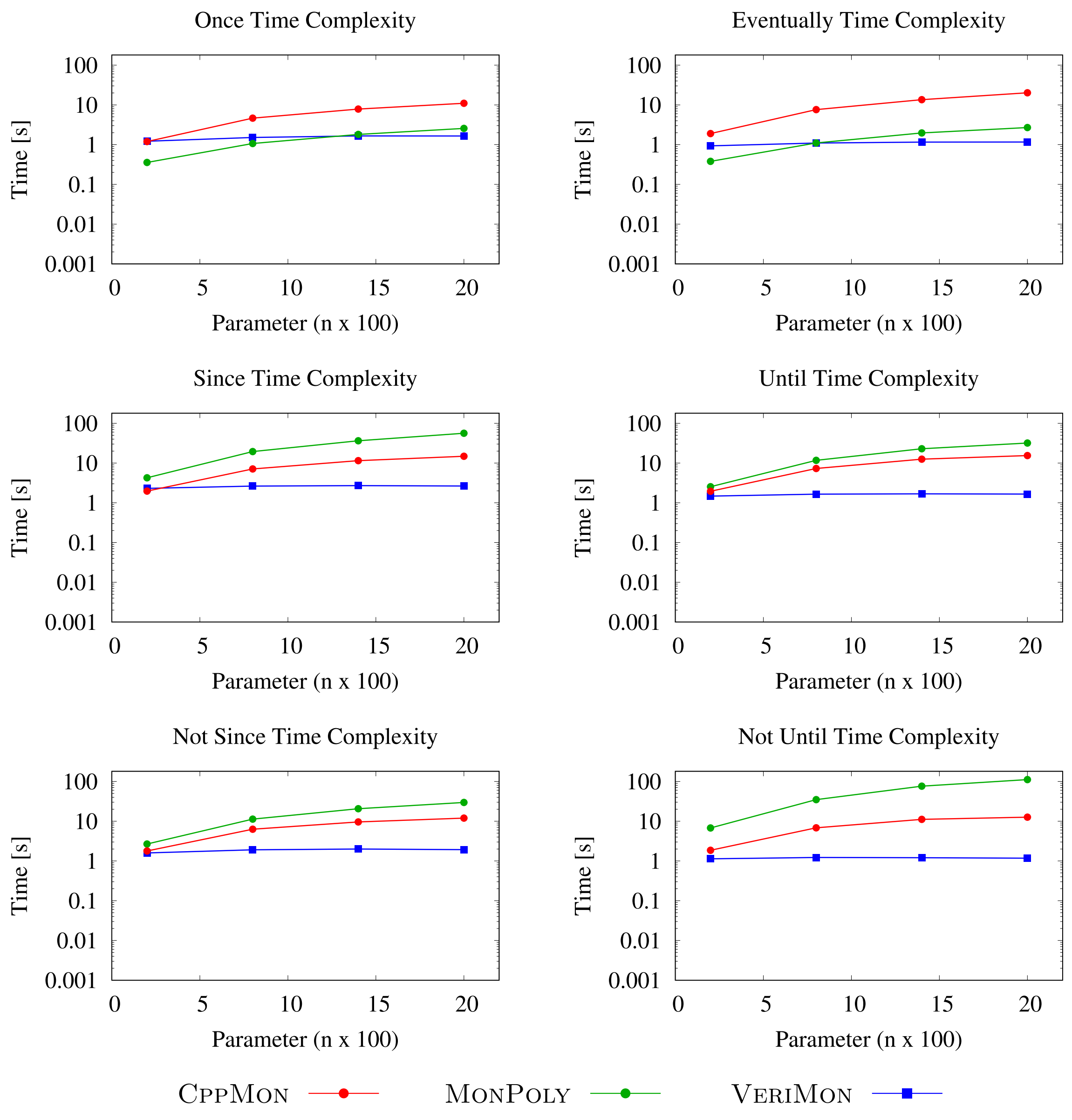


Figure 4.19. Evaluation results for increasing interval bounds.

Chapter 5

Conclusion

We conclude this thesis with a summary of our main results and outline possible directions for future work.

5.1 Summary

We proposed multi-head monitoring as a novel approach to analyzing traces. A multi-head monitor reads an input trace simultaneously at multiple positions and its reading heads move asynchronously. Hence, multi-head monitoring fills a middle-ground between online and offline monitoring. We developed, implemented, and formally verified the correctness of multi-head monitors for metric temporal logic (MTL) and metric dynamic logic (MDL). Our multi-head monitors for MTL and MDL support time-stamps from an abstract time domain satisfying certain algebraic properties. Instances of the abstract time domain include natural and real numbers as well as products of them ordered either by the direct product order or lexicographically. Our monitors are the first event-rate independent monitors for MTL and MDL that produce a stream of Boolean verdicts. This is a significant improvement over the event-rate independent monitor Aerial in terms of the monitor's interface: Boolean verdicts are much easier for humans to understand than Aerial's non-standard equivalence verdicts. Additionally, our monitor is interval-oblivious: The constants occurring in the formula's metric constraints have no impact on the monitor's time- and memory consumption. To our knowledge, this property is unprecedented for monitors for metric specification languages in the point-based setting.

Metric first-order temporal logic (MFOTL) generalizes MTL with parametric events and first-order variables ranging over an arbitrary domain. This makes MFOTL queries substantially more expressive than MTL queries. VeriMon is an online monitor for MFOTL developed by formally verifying a monitoring algorithm for MFOTL from previous work. We optimized the time complexity of evaluating VeriMon's Since and Until MFOTL temporal operators so that the amortized time complexity to process a new time-point is linear in the size of the input relations for the subqueries, but event-rate independent and interval-oblivous.

We bridge the gap between declarative first-order logic queries and procedural (and efficient) relational algebra normal form (RANF) query evaluation by developing and implementing a novel approach to relational calculus (RC) and MFOTL query evaluation over an infinite domain via translation to RANF queries that can be evaluated using relational algebra operations on finite tables. We translate an arbitrary RC query into a pair of RANF queries: one characterizes the original query's relative safety (i.e., whether it evaluates to a finite relation) and the other one is equivalent to the original query if the original query is relatively safe. We translate an arbitrary MFOTL query into a single RANF query that can be evaluated at a sequence of time-points by a single instance of a monitoring algorithm (e.g., VeriMon) using relational algebra operations on finite tables. This way, the monitor decides for an arbitrary MFOTL query and every time-point if the query evaluates to a finite relation at that time-point and computes the relation if it is finite.

5.2 Future Work

In the following, we identify and outline possible directions for follow-up projects.

Multi-Head Monitoring for First-Order Logics We developed multi-head monitors for propositional temporal logics, namely MTL and MDL. Generalizing these multi-head monitors to first-order temporal logics, i.e., MFOTL and MFODL [8] (combining MFOTL and MDL), is an interesting direction for future work. We conjecture that a generalization is feasible for past-only MFOTL (i.e., MFOTL without future temporal operators) queries in RANF because the movement pattern of the multi-head monitor's reading heads for past-only MTL does not depend on the content of the databases in the trace. Instead of a Boolean value, the generalized multi-head monitor would return a finite set of tuples satisfying the first-order temporal query. The resulting multi-head monitor for past-only MFOTL queries in RANF could have both time and *space* complexity depending linearly on the amount of data in the trace, but event-rate independent and interval-oblivious. Currently, the MFOTL monitor described in Section 4.4 only achieves this objective for the time complexity, but not the space complexity. For MFOTL and MFODL with future temporal operators, the number of reading heads used by a generalization of our multi-head monitors for MTL and MDL would likely depend on the content of the databases in the trace. This would defeat the core idea of a multi-head finite transducer using only a bounded number of reading heads that does not depend on the input word (the input trace in the case of monitoring). Still, it might be feasible to generalize our multi-head monitor to a monitor for MFOTL and MFODL queries in RANF with future temporal operators whose space complexity depends linearly on the amount of data in the trace, but is event-rate independent and interval-oblivious.

Extending RC Query Translation We defined RC as a first-order query language with equality. Integrating additional features into our base language could be an interesting direction for future work. Such features include order relations (inequalities), bag semantics, or aggregations (count, sum, minimum, etc.). We conjecture that it is impossible to derive a computable query translation deciding relative safety for a query language with a total order relation and count aggregation simultaneously. Our conjecture is based on a reduction from the undecidable Hilbert's tenth problem of Diophantine equations' solvability (i.e., whether a multivariate polynomial with integer coefficients has integer solutions) to the relative safety of RC queries with integer inequalities and count aggregations. The reduction proceeds as follows:

- for every variable x, perform a case distinction whether its value is negative ($x = -x'$, for some positive integer x'), zero ($x = 0$), or positive ($x = x'$, for some positive integer x'), and substitute x by $-x'$, 0, or x', respectively;
- write the equation (with positive integer variables) so that all coefficients of its left-hand side and right-hand side are positive, e.g., write $3x - 2y + xy = 0$ as $3x + xy = 2y$;
- express every term of the form $c \cdot z_1 \cdots z_k$, where c is a positive integer coefficient and $z_1, \dots, z_k$ are positive integer variables, as

  $$[\mathsf{CNT}\, z'_0, z'_1, \dots, z'_k.\, 0 \le z'_0 < c \land 0 \le z'_1 < z_1 \land \cdots \land 0 \le z'_k < z_k](t),$$

  where t is a fresh variable introduced for the term $c \cdot z_1 \cdots z_k$;

- express every sum $t_1 + t_2$, where t_1 and t_2 are positive integer variables, as

  $$[\mathsf{CNT}\, t'_1, t'_2.\, (0 \leq t'_1 < t_1 \wedge t'_2 = t_2) \vee (t'_1 = t_1 \wedge 0 \leq t'_2 < t_2)](t),$$

  where t is a fresh variable introduced for the sum $t_1 + t_2$;
- define the RC query $Q^\wedge$ as the conjunction of the above count aggregations, an equality $t_l = t_r$ between the variables t_l and t_r representing the sum of the equation's left-hand side and right-hand side;
- define the resulting RC query of the reduction as $(\exists \vec{\mathsf{fv}}(Q^\wedge).\ Q^\wedge) \wedge w \geq 0$, where w is a fresh variable; this RC query is relatively safe if and only if the equation has no solution.

Still, the relative safery of RC queries with a total order relation over integers or real numbers is decidable by using difference decision diagrams (DDD) [53] or linear decision diagrams (LDD) [17]. We also conjecture that our query translation for RC queries can be extended to RC with count aggregation (without any order relation).

Efficient Table Representations in Monitoring Algorithms MFOTL monitors MonPoly and VeriMon, which use relational algebra operations on finite tables, use balanced binary search trees to represent the tables (sets of satisfying tuples) during query evaluation. The tuples in the search trees are ordered lexicographically for a fixed variable order derived from the syntactic structure of the query. DejaVu also uses a fixed variable order in the binary decision diagrams representing sets of satisfying tuples. However, for a query like

$$(q_1(x, y) \wedge \blacklozenge\, r_1(x, z)) \vee (q_2(y, z) \wedge \blacklozenge\, r_2(y, x)) \vee (q_3(z, x) \wedge \blacklozenge\, r_3(z, y)),$$

no fixed variable order for the entire query yields optimal performance. Indeed, the first variable in the fixed variable order to efficiently evaluate each of the three disjuncts should be x, y, and z, respectively. To control the variable order in the tables for the individual parts of a query without changing the built-in set representations (in OCaml or Isabelle/HOL), we propose to group the sets of tuples according to their projections to a subset of variables. In other words, we replace simple tables by database indices. Formally, a database index has the type $(\mathcal{D})^* \to \mathcal{P}((\mathcal{D})^*) \cup \{\bot\}$, i.e., it is a mapping of tuples (the projections of the actual tuples to a subset of variables) to subsets of the actual set of tuples. We have already used such mappings in VeriMon's optimized state for the Since operator (step (2)* in Section 4.4.1). Hence, the next step is to use the mappings for all sets of satisfying tuples during query evaluation and also determine the actual subsets of variables to which the tuples are projected. For the Since operator $Q_1 \mathsf{S}_I Q_2$, which must satisfy $\mathsf{fv}(Q_1) \subseteq \mathsf{fv}(Q_2)$, the evaluation benefits from projecting on $\mathsf{fv}(Q_1)$. It seems challenging to generalize this (local) decision for the Since operator to all operators so that the overall query evaluation (globally) benefits from the choices of variable subsets for the database indices.

Verifying RC and MFOTL Query Translation We have formally verified the correctness of the multi-head monitors for MTL and MDL (Section 3.2) as well as the correctness of VeriMon (Section 4.4)—the monitor for MFOTL—using the Isabelle/HOL proof assistant. The query translation for arbitrary RC and MFOTL queries has been implemented in the tools rc2sql and mfotl2ranf using the functional programming language OCaml without a formal verification of their correctness. The correctness of RC2SQL has been extensively tested against the formally verified approach to query evaluation by Ailamazyan et al. [61] using small Data Golf structures (Section 4.2.5). The correctness of MFOTL2RANF has only been tested on the examples from Section 4.3.2. Hence, formalizing the correctness proofs for our query traslation in a proof assistant would significantly increase the translation's trustworthiness.

Query Progress of MFOTL Query Translation The *progress* (Section 2.3.2) of the query $\mathsf{MFOTL2RANF}(Q)$ produced by our translation can be lower (i.e., worse) than the progress of the original MFOTL query Q. This means that an MFOTL monitor might not be able to evaluate the query $\mathsf{MFOTL2RANF}(Q)$ at all time-points at which the original query Q could be potentially evaluated using a more optimized translation. As an example, we can consider the query $Q \coloneqq q(x, y) \land \neg \blacklozenge_{[0,100]} \neg r(x, y)$ that is translated to the query

$$Q' \coloneqq \mathsf{MFOTL2RANF}(Q) = (q(x, y) \land \neg(\blacklozenge_{[0,100]} ((\Diamond_{[0,100]} q(x, y)) \land \neg r(x, y))) \land \_\_\_inf = 0)$$

by our translation, where $\_\_\_inf$ is a special variable indicating if the set of satisfying tuples is infinite at a time-point (which cannot be the case for the query Q and thus we have the conjunct $\_\_\_inf = 0$ in Q'). For example, given the sequence of time-stamps $\overline{\tau} \coloneqq [0, 10, 20, 101]$, the original query Q could be evaluated (e.g., using DEJAVU) at all four time-points over a trace with four time-points and the time-stamps $\overline{\tau}$. Formally, we have $\mathsf{prog}(Q, \overline{\tau}) = 4$. However, the translated RANF query Q' can only be evaluated at the first time-point by MFOTL monitors, e.g., VERIMON, that only evaluate $\mathsf{prog}(Q', \overline{\tau}) = 1$ time-point of a trace with the time-stamps $\overline{\tau}$.

We leave it as an open question if an arbitrary MFOTL query Q can be translated to a single RANF query Q' that is equivalent to $\mathsf{MFOTL2RANF}(Q)$ and additionally satisfies $\mathsf{prog}(Q', \overline{\tau}) \geq \mathsf{prog}(Q, \overline{\tau})$ for every sequence $\overline{\tau}$ of time-stamps, i.e., the progress for Q' can only improve.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. URL: `http://webdam.inria.fr/Alice/`.

[2] Alfred K. Ailamazyan, Mikhail M. Gilula, Alexei P. Stolboushkin, and Grigorii F. Schwartz. Reduction of a relational model with infinite domains to the case of finite domains. *Dokl. Akad. Nauk SSSR*, 286:308–311, 1986. URL: `http://mi.mathnet.ru/dan47310`.

[3] Arnon Avron and Yoram Hirshfeld. On first order database query languages. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 226–231. IEEE Computer Society, 1991. `doi:10.1109/LICS.1991.151647`.

[4] Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012. `doi:10.1007/978-3-642-32759-9\_9`.

[5] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018. `doi:10.1007/978-3-319-75632-5`.

[6] David A. Basin, Bhargav Nagaraja Bhatt, Srdan Krstic, and Dmitriy Traytel. Almost event-rate independent monitoring. *Formal Methods Syst. Des.*, 54(3):449–478, 2019. `doi:10.1007/s10703-018-00328-3`.

[7] David A. Basin, Bhargav Nagaraja Bhatt, and Dmitriy Traytel. Almost event-rate independent monitoring of metric temporal logic. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 94–112, 2017. `doi:10.1007/978-3-662-54580-5\_6`.

[8] David A. Basin, Thibault Dardinier, Lukas Heimes, Srdan Krstic, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *IJCAR 2020*, volume 12166 of *Lecture Notes in Computer Science*, pages 432–453. Springer, 2020. `doi:10.1007/978-3-030-51074-9\_25`.

[9] David A. Basin, Thibault Dardinier, Lukas Heimes, Srdan Krstic, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. VeriMon: Optimized monitoring algorithm for metric first-order dynamic logic with aggregations, 2022. `https://bitbucket.org/jshs/monpoly`.

[10] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015. `doi:10.1145/2699444`.

[11] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Algorithms for monitoring real-time properties. *Acta Informatica*, 55(4):309–338, 2018. `doi:10.1007/s00236-017-0295-4`.

[12] David A. Basin, Srdan Krstic, and Dmitriy Traytel. Almost event-rate independent monitoring of metric dynamic logic. In Shuvendu K. Lahiri and Giles Reger, editors, *Runtime Verification - 17th International Conference, RV 2017, Seattle, WA, USA, September 13-16, 2017, Proceedings*, volume 10548 of *Lecture Notes in Computer Science*, pages 85–102. Springer, 2017. `doi:10.1007/978-3-319-67531-2\_6`.

[13] Jan Baumeister, Bernd Finkbeiner, Matthis Kruse, and Maximilian Schwenger. Automatic optimizations for stream-based monitoring languages. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 451–461. Springer, 2020. `doi:10.1007/978-3-030-60508-7\_25`.

[14] Michael Benedikt and Leonid Libkin. Relational queries over interpreted structures. *J. ACM*, 47(4):644–680, 2000. `doi:10.1145/347476.347477`.

[15] Paul Bernays. Alonzo church. an unsolvable problem of elementary number theory. american journal of mathematics, vol. 58 (1936), pp. 345–363. *The Journal of Symbolic Logic*, 1(2):73–74, 1936.

[16] Achim Blumensath and Erich Grädel. Finite presentations of infinite structures: Automata and interpretations. *Theory Comput. Syst.*, 37(6):641–674, 2004. `doi:10.1007/s00224-004-1133-y`.

[17] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Decision diagrams for linear arithmetic. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 53–60. IEEE, 2009. `doi:10.1109/FMCAD.2009.5351143`.

[18] Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2009. `doi:10.1007/978-3-642-00768-2\_23`.

[19] Jan Chomicki and David Toman. Implementing temporal integrity constraints using an active DBMS. *IEEE Trans. Knowl. Data Eng.*, 7(4):566–582, 1995. `doi:10.1109/69.404030`.

[20] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 286–295. Morgan Kaufmann, 1997. URL: `http://www.vldb.org/conf/1997/P286.PDF`.

[21] E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

[22] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: Temporal stream-based specification language. In Tiago Massoni and Mohammad Reza Mousavi, editors, *Formal Methods: Foundations and Applications - 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26-30, 2018, Proceedings*, volume 11254 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2018. `doi:10.1007/978-3-030-03044-5\_10`.

[23] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*, pages 166–174. IEEE Computer Society, 2005. `doi:10.1109/TIME.2005.26`.

[24] Thibault Dardinier, Lukas Heimes, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. Formalization of an optimized monitoring algorithm for metric first-order dynamic logic with aggregations. *Arch. Formal Proofs*, 2020, 2020. URL: `https://www.isa-afp.org/entries/MFODL_Monitor_Optimized.html`.

[25] Johann C. Dauer, Bernd Finkbeiner, and Sebastian Schirmer. Monitoring with verified guarantees. In Lu Feng and Dana Fisman, editors, *Runtime Verification - 21st International Conference, RV 2021, Virtual Event, October 11-14, 2021, Proceedings*, volume 12974 of *Lecture Notes in Computer Science*, pages 62–80. Springer, 2021. `doi:10.1007/978-3-030-88494-9\_4`.

[26] Erling Ellingsen. Regex golf, 2013. `https://alf.nu/RegexGolf`.

[27] Martha Escobar-Molano, Richard Hull, and Dean Jacobs. Safety and translation of calculus queries with scalar functions. In Catriel Beeri, editor, *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 25-28, 1993, Washington, DC, USA*, pages 253–264. ACM Press, 1993. `doi:10.1145/153850.153909`.

[28] Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021. `doi:10.1007/s10009-021-00609-z`.

[29] Peter Faymonville, Bernd Finkbeiner, Sebastian Schirmer, and Hazem Torfah. A stream-based specification language for network monitoring. In Yliès Falcone and César Sánchez, editors, *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, volume 10012 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2016. `doi:10.1007/978-3-319-46982-9\_10`.

[30] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. Streamlab: Stream-based monitoring of cyber-physical systems. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 421–431. Springer, 2019. `doi:10.1007/978-3-030-25540-4\_24`.

[31] Bernd Finkbeiner, Stefan Oswald, Noemi Passing, and Maximilian Schwenger. Verified rust monitors for lola specifications. In Jyotirmoy Deshmukh and Dejan Nickovic, editors, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings*, volume 12399 of *Lecture Notes in Computer Science*, pages 431–450. Springer, 2020. `doi:10.1007/978-3-030-60508-7\_24`.

[32] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Formal Methods Syst. Des.*, 24(2):101–127, 2004. `doi:10.1023/B:FORM.0000017718.28096.48`.

[33] Allen Van Gelder and Rodney W. Topor. Safety and correct translation of relational calculus formulas. In Moshe Y. Vardi, editor, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, USA*, pages 313–327. ACM, 1987. `doi:10.1145/28659.28693`.

[34] Allen Van Gelder and Rodney W. Topor. Safety and translation of relational calculus queries. *ACM Trans. Database Syst.*, 16(2):235–278, 1991. `doi:10.1145/114325.103712`.

[35] Giuseppe De Giacomo, Antonio Di Stasio, Francesco Fuggitti, and Sasha Rubin. Pure-past linear temporal and dynamic logic on finite traces. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4959–4965. ijcai.org, 2020. `doi:10.24963/ijcai.2020/690`.

[36] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In Francesca Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 854–860. IJCAI/AAAI, 2013. URL: `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6997`.

[37] Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on finite traces. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1558–1564. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/223`.

[38] Felipe Gorostiaga and César Sánchez. Striver: Stream runtime verification for real-time event-streams. In Christian Colombo and Martin Leucker, editors, *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, volume 11237 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2018. `doi:10.1007/978-3-030-03769-7\_16`.

[39] Matthieu Gras and Srđan Krstić. Cppmon: Efficient c++ monitor for metric first-order temporal logic, 2022. Private repository.

[40] Klaus Havelund. Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transf.*, 17(2):143–170, 2015. `doi:10.1007/s10009-014-0309-2`.

[41] Klaus Havelund and Doron Peled. First-order timed runtime verification using BDDs. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23,*

2020, Proceedings, volume 12302 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2020. doi:10.1007/978-3-030-59152-6_1.

[42] Klaus Havelund, Doron Peled, and Dogan Ulus. First-order temporal logic monitoring with BDDs. *Formal Methods Syst. Des.*, 56(1):1–21, 2020. doi:10.1007/s10703-018-00327-4.

[43] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi:10.1007/3-540-46002-0_24.

[44] Richard Hull and Jianwen Su. Domain independence and the relational calculus. *Acta Informatica*, 31(6):513–524, 1994. doi:10.1007/BF01213204.

[45] Aaron Kane, Omar Chowdhury, Anupam Datta, and Philip Koopman. A case study on runtime monitoring of an autonomous research vehicle (ARV) system. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, volume 9333 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2015. doi:10.1007/978-3-319-23820-3_7.

[46] Michael Kifer. On safety, domain independence, and capturability of database queries (preliminary report). In Catriel Beeri, Joachim W. Schmidt, and Umeshwar Dayal, editors, *Proceedings of the Third International Conference on Data and Knowledge Bases: Improving Usability and Responsiveness, June 28-30, 1988, Jerusalem, Israel*, pages 405–415. Morgan Kaufmann, 1988. doi:10.1016/b978-1-4832-1313-2.50037-8.

[47] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990. doi:10.1007/BF01995674.

[48] Jianwen Li, Moshe Y. Vardi, and Kristin Y. Rozier. Satisfiability checking for mission-time LTL. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, volume 11562 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2019. doi:10.1007/978-3-030-25543-5_1.

[49] Leonid Libkin. *Elements of Finite Model Theory.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. URL: http://www.cs.toronto.edu/%7Elibkin/fmt, doi:10.1007/978-3-662-07003-1.

[50] Hong-Cheu Liu, Jeffrey Xu Yu, and Weifa Liang. Safety, domain independence and translation of complex value database queries. *Inf. Sci.*, 178(12):2507–2533, 2008. doi:10.1016/j.ins.2008.02.005.

[51] Emanuele Marsicano. Verified incremental evaluation of aggregation operators in metric first-order temporal logic, 2021. Bachelor's thesis.

[52] Jesper B. Møller. DDDLIB: A library for solving quantified difference inequalities. In Andrei Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference*

on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings, volume 2392 of *Lecture Notes in Computer Science*, pages 129–133. Springer, 2002. `doi:10.1007/3-540-45620-1\_9`.

[53] Jesper B. Møller, Jakob Lichtenberg, Henrik Reif Andersen, and Henrik Hulgaard. Difference decision diagrams. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Computer Science Logic, 13th International Workshop, CSL '99, 8th Annual Conference of the EACSL, Madrid, Spain, September 20-25, 1999, Proceedings*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1999. `doi:10.1007/3-540-48168-0\_9`.

[54] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods Syst. Des.*, 51(1):31–61, 2017. `doi:10.1007/s10703-017-0275-x`.

[55] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013. `doi:10.1145/2590989.2590991`.

[56] Jianmo Ni, Jiacheng Li, and Julian J. McAuley. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, pages 188–197. Association for Computational Linguistics, 2019. `doi:10.18653/v1/D19-1018`.

[57] Robert A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *J. ACM*, 16(2):324–327, 1969. `doi:10.1145/321510.321524`.

[58] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977. `doi:10.1109/SFCS.1977.32`.

[59] Martin Raszyk. HYDRA: Monitoring tool for MTL and MDL, 2021. `https://github.com/mraszyk/hydra`.

[60] Martin Raszyk. RC2SQL: Translation of relational calculus queries to SQL, 2021. `https://github.com/mraszyk/rc2sql`.

[61] Martin Raszyk. First-order query evaluation. *Arch. Formal Proofs*, 2022, 2022. URL: `https://www.isa-afp.org/entries/Eval_FO.html`.

[62] Martin Raszyk. MFOTL2RANF: Translation of MFOTL queries to RANF, 2022. `https://github.com/mraszyk/mfotl2ranf`.

[63] Martin Raszyk. Multi-head monitoring of metric dynamic logic. *Arch. Formal Proofs*, 2022, 2022. URL: `https://www.isa-afp.org/entries/VYDRA_MDL.html`.

[64] Martin Raszyk, David A. Basin, Srdan Krstic, and Dmitriy Traytel. Multi-head monitoring of metric temporal logic. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *ATVA 2019*, volume 11781 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2019. `doi:10.1007/978-3-030-31784-3\_9`.

[65] Martin Raszyk, David A. Basin, Srdan Krstic, and Dmitriy Traytel. Practical relational calculus query evaluation. In *24th International Conference on Database Theory, ICDT 2022*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[66] Martin Raszyk, David A. Basin, and Dmitriy Traytel. From nondeterministic to multi-head deterministic finite-state transducers. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 127:1–127:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ICALP.2019.127`.

[67] Martin Raszyk, David A. Basin, and Dmitriy Traytel. Multi-head monitoring of metric dynamic logic. In Dang Van Hung and Oleg Sokolsky, editors, *ATVA 2020*, volume 12302 of *Lecture Notes in Computer Science*, pages 233–250. Springer, 2020. `doi:10.1007/978-3-030-59152-6\_13`.

[68] Peter Z. Revesz. *Introduction to Constraint Databases*. Texts in Computer Science. Springer, 2002. `doi:10.1007/b97430`.

[69] Grigore Rosu and Klaus Havelund. Synthesizing dynamic programming algorithms from linear temporal logic formulae. Technical report, 2001.

[70] Grigore Rosu and Klaus Havelund. Rewriting-based techniques for runtime verification. *Autom. Softw. Eng.*, 12(2):151–197, 2005. `doi:10.1007/s10515-005-6205-y`.

[71] Joshua Schneider, David A. Basin, Srdan Krstic, and Dmitriy Traytel. A formally verified monitor for metric first-order temporal logic. In Bernd Finkbeiner and Leonardo Mariani, editors, *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 310–328. Springer, 2019. `doi:10.1007/978-3-030-32079-9\_18`.

[72] Prasanna Thati and Grigore Rosu. Monitoring algorithms for metric temporal logic specifications. *Electron. Notes Theor. Comput. Sci.*, 113:145–162, 2005. `doi:10.1016/j.entcs.2004.01.029`.

[73] Boris A Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *Doklady Akademii Nauk SSSR*, 70(4):569–572, 1950.

[74] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[75] Dogan Ulus. Online monitoring of metric temporal logic using sequential networks. *CoRR*, abs/1901.00175, 2019. URL: `http://arxiv.org/abs/1901.00175`, `arXiv:1901.00175`.

[76] Moshe Y. Vardi. The decision problem for database dependencies. *Inf. Process. Lett.*, 12(5):251–254, 1981. `doi:10.1016/0020-0190(81)90025-9`.

[77] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 137–146. ACM, 1982. `doi:10.1145/800070.802186`.

[78] Jun Yang. radb, 2019. `https://github.com/junyang/radb`.

[79] Sheila Zingg, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. Verified first-order monitoring with recursive rules, 2022.