




Randomized First-Order Monitoring with Hashing

Joshua Schneider^(✉) 

Institute of Information Security, Department of Computer Science, ETH Zürich,
Zürich, Switzerland
joshua.schneider@inf.ethz.ch

Abstract. Online monitors for first-order specifications may need to store many domain values in their state, requiring significant memory. We propose an approach that compresses the monitor’s state using randomized hash functions. Unlike input sampling, our approach does not require the knowledge of distributions over traces to achieve low error probability. We develop algorithms that insert hash functions into temporal–relational algebra specifications and compute upper bounds on the resulting error probability. We employ a special hashing scheme that allows us to merge values across attributes, which further reduces memory usage. We evaluated our implementation and achieved memory reductions up to 33% when monitoring traces with large domain values, with error probability less than two in a million.

Keywords: Online monitoring · Temporal–relational algebra · Hashing

1 Introduction

Online monitors must run in diverse environments that possibly offer limited computing resources. For instance, the monitoring of operating system kernels [37] competes with the user’s applications, embedded hardware is often underpowered and not easily extendable (e.g., in unmanned aerial systems [43]), and so forth. However, online monitors for first-order specification languages may use a significant amount of memory, which hampers their applicability in such environments. One reason is that they store domain values from the trace in their internal state. These values can be large in some applications (consider events that are parametrized by natural language texts or URLs).

To address this problem, we develop a monitoring algorithm that compresses the domain values using randomized hash functions. Large values such as strings are replaced by hash values which have more compact in-memory representations. Hashing may incur a loss of accuracy: because of collisions, the algorithm’s

output may be incorrect with non-zero probability. Our monitor therefore satisfies a probabilistic correctness property. Thanks to the randomization, collisions are independent of the domain values in the trace. Crucially, we demonstrate how to compute an *a priori* upper bound on the error probability for any given specification from the hash functions’ collision probability.

Simple specification languages permit very efficient monitors, e.g., every past-time LTL specification can be monitored in constant space [24]. Some applications demand more complex specifications. We focus on a temporal-relational algebra (TRA, see Sect. 2), which corresponds to a fragment of metric first-order temporal logic monitorable using finite relations. First-order languages such as TRA are more concise and, assuming an infinite domain of event parameters, more expressive than propositional languages such as LTL. However, monitoring a fixed TRA specification requires space polynomial in the size of the trace [6]. Specifically, the space usage depends linearly on the size of the domain values’ representation. We show that hashing removes this dependency for a nontrivial fragment of TRA and traces whose rate is bounded by a constant.

Randomization and hashing are well-established in algorithm design. Previous research has mostly focused on simpler problems, such as approximate set membership [9, 35, 40]. First-order monitors often operate on *structured* sets, which cannot be easily encoded using the existing space-optimal data structures. We present compact data structures that provide the operations needed by monitors for expressive languages, but we do not aim at optimality. Moreover, hashing has been used successfully in model checkers to reduce space [13, 27]. In that domain, the error analysis generally depends on the number of reachable states, whereas in our case the specification’s structure is significant.

Core Ideas. Let us illustrate our approach on the example specification, “every $a(x, y)$ event must be preceded by a corresponding $b(x, y)$ event not more than 10 time units ago.” We assume that simultaneous events are possible. We represent, at every point in time recorded in the trace, the a events at that time by a relation that stores tuples (x, y) with the events’ parameters; the b events are handled similarly. To evaluate the specification, our monitor maintains in its memory another relation R that is equal to the union of all b relations from the past 10 time units, relative to the trace position that the monitor is currently processing. The set difference of the current a relation and R yields (x, y) tuples that witness violations of the specification.

We reduce a specific factor of the memory usage: the representation of domain values in the monitor’s state (for our example, in the R relation). This representation usually adds a factor linear in the binary size of the largest value. The basic idea is to replace the domain values x with their images $h(x)$ under a hash function h , which is sampled randomly from a suitable collection at the start of monitoring. As an additional optimization, we merge multiple attributes (tuple elements) into a single hash value whenever the attributes are used consistently as a group. Merged hashes over disjoint attribute sets can be combined in an order-independent way. This hashing scheme repurposes a well-known construction that extends the domain of universal hash functions [49]. To the best of

our knowledge, the construction’s merging property has not been used before to compress relations.

Hashing is possible because many of the TRA’s operators continue to give correct results up to h . For instance, hashing the domain values in a relation commutes with the union operation. However, some operations may randomly introduce errors with a small probability, which manifest themselves as added or missing tuples. Such errors occur whenever hashes $h(x)$ are compared for equality, e.g., in an intersection. It is important to bound the error probability; otherwise, the monitor would be of little use. Therefore, we develop an algorithm that computes upper bounds for these probabilities by taking the TRA expression’s structure and information about the trace into account. Specifically, the algorithm expects as inputs upper bounds on the cardinalities of the relations consumed and computed by the monitor. For the above example and a hash size of 63 bits, the error probability is computed to be at most $3.3 \cdot 10^{-19} \cdot n_a \cdot 10n_b$ per time-point (position) in the trace, where n_a and n_b are the maximum number of a and b events per time unit. This is the only information we require about the trace; in particular, we do not assume a probabilistic model explaining the trace distribution. Our approach and the error analysis are independent of implementation details, such as the data structure used to implement the 10 time unit sliding window.

Some operators are incompatible with hashing, specifically order comparisons. We handle them on a best-effort basis: values are not hashed if they are used by the problematic operators. However, we still demonstrate an overall reduction of memory usage on relevant traces empirically. For TRA expressions with bounded intervals, no functions, no aggregations except for counting, and no order comparisons, we show that the linear factor can indeed be eliminated if the number of events per time unit is bounded by a constant.

Summary. We make the following contributions.

- Our space-efficient monitor (Sect. 3) relies on the key observation that many temporal–relational operations can be computed on relations containing hashed values, with low error probability. As an additional space optimization, we provide a hashing scheme that merges hashes from different attributes within a relation.
- We analyze the error probabilities of TRA expressions with hashing operators (Sect. 4). Our analysis is compositional and yields upper bounds. One can thus estimate the error probability for every hash size by observing or estimating these cardinalities, or alternatively, minimize the error given a space constraint.
- We implement our space-efficient monitor as an extension of MonPoly (Sect. 5). The extended tool automatically inserts hash operators into user-provided specifications and it outputs error bounds for individual input traces. We discuss the specifications most amenable to our optimization and evaluate the memory usage and accuracy of our tool. In a case study with real data, we could reduce the memory usage by 33% with an error probability

below $2 \cdot 10^{-6}$. Our evaluation demonstrates that the technique is particularly effective for traces with large domain values, such as long strings.

This paper is accompanied by an artifact that consists of the monitor implementation and the evaluation scripts. The artifact is available at <https://bitbucket.org/jshs/hashmon>. Our extended report [42] provides the proofs for all lemmas and theorems.

2 Temporal–Relational Algebra

Our monitoring algorithm extends MonPoly’s, which has been designed for specifications expressed in metric first-order temporal logic (MFOTL) with aggregations [5, 6]. MonPoly translates MFOTL to a temporal–relational algebra (TRA), which it then evaluates using finite relations over an infinite domain. To simplify the presentation, we focus on the TRA, as our algorithms work directly with its operators, which do not map one-to-one to MFOTL’s operators. We note that other variants of TRA exist in the literature [38, 46].

We assume a countably infinite set \mathcal{A} of attributes and a domain \mathcal{D} of constants totally ordered by \leq . A tuple u over a finite set $U \subset \mathcal{A}$ of attributes is a mapping from U to \mathcal{D} . We write $att(u)$ for u ’s attributes U , and $u(a)$ for u ’s value at $a \in att(u)$. A relation R over U is a finite set of tuples over U ; overloading notation, we define $att(R) = U$. A schema S is a collection of relation names r with associated attribute sets $att_S(r)$. A database D over S is a mapping from $r \in S$ to relations $D(r)$ over $att_S(r)$.

The following grammar defines the expressions e of TRA. We write \bar{z} for a list of elements derived from nonterminal z . The nonterminals a and a' range over attributes; c is a constant in \mathcal{D} ; and I and I^∞ are finite and infinite intervals over \mathbb{N} , respectively.¹

$$\begin{aligned}
 t ::= & a \mid c \mid f(\bar{t}) & \circ ::= & = \mid \neq \mid \leq \mid < & \omega ::= & \text{COUNT} \mid \text{SUM} \mid \text{MIN} \mid \text{MAX} \\
 e ::= & R \mid r \mid \pi(\bar{a})e \mid \varrho(\bar{a} \leftarrow \bar{a}')e \mid \sigma(t_1 \circ t_2)e \mid \eta(a \mapsto t)e \\
 & \mid e_1 \bowtie e_2 \mid e_1 \triangleright e_2 \mid e_1 \cup e_2 \mid \mathbf{Y}_{I^\infty} e \mid e_1 \mathbf{S}_{I^\infty}^m e_2 \mid \mathbf{X}_{I^\infty} e \mid e_1 \mathbf{U}_I^m e_2 \mid \omega(a' \mapsto t; \bar{a})e
 \end{aligned}$$

Terms t are either attributes, constants, or function applications; we do not further specify the available function symbols f . An expression can be a constant relation R , a relation name $r \in S$, or a compound expression. We sometimes write $r(\bar{a})$ to indicate that $att_S(r) = \bar{a}$. The projection operator $\pi(\bar{a})$ preserves only the attributes \bar{a} and removes all other attributes. The renaming operator $\varrho(\bar{a} \leftarrow \bar{a}')$ replaces the attributes in the list \bar{a}' simultaneously by the corresponding attributes in \bar{a} . The selection operator $\sigma(t_1 \circ t_2)$ filters tuples according to the condition $t_1 \circ t_2$. The assignment operator $\eta(a \mapsto t)$ computes a new attribute a from the term t . The natural join $e_1 \bowtie e_2$ contains exactly those tuples that are in e_1 and e_2 when restricted to e_1 ’s and e_2 ’s attributes, respectively. The anti-join $e_1 \triangleright e_2$ is similar, except that the restrictions to e_2 ’s attributes must not be in e_2 .

¹ The meta-variable I will later be used for both types of intervals.

Table 1. Syntax, well-formedness, attributes, and semantics of TRA expressions

e	e is well-formed iff	$att(e)$	$\forall u, i. u \in \llbracket e \rrbracket_i$ iff $att(u) = att(e)$ and
R	no restriction	$att(R)$	$u \in R$
r	$r \in S$	$att_S(r)$	$u \in \xi_i(r)$
$\pi(\bar{a})e_1$	$\bar{a} \subseteq att(e_1)$	\bar{a}	$\exists u' \in \llbracket e_1 \rrbracket_i. u = u' _{\bar{a}}$
$\varrho(\bar{a} \leftarrow a')e_1$	$\bar{a}' = att(e_1)$	\bar{a}	$\exists u' \in \llbracket e_1 \rrbracket_i. \bigwedge_k u(a_k) = u'(a'_k)$
$\sigma(t_1 \circ t_2)e_1$	$att(t_1) \subseteq att(e_1), att(t_2) \subseteq att(e_1)$	$att(e_1)$	$u \in \llbracket e_1 \rrbracket_i$ and $t_1(u) \circ t_2(u)$
$\eta(a \mapsto t)e_1$	$att(t) \subseteq att(e_1)$	$att(e_1) \cup \{a\}$	$\exists u' \in \llbracket e_1 \rrbracket_i. u = u'[a \mapsto t(u')]$
$e_1 \bowtie e_2$	no restriction	$att(e_1) \cup att(e_2)$	$u _{att(e_1)} \in \llbracket e_1 \rrbracket_i$ and $u _{att(e_2)} \in \llbracket e_2 \rrbracket_i$
$e_1 \triangleright e_2$	$att(e_1) \supseteq att(e_2)$	$att(e_1)$	$u \in \llbracket e_1 \rrbracket_i$ and $u _{att(e_2)} \notin \llbracket e_2 \rrbracket_i$
$e_1 \cup e_2$	$att(e_1) = att(e_2)$	$att(e_1)$	$u \in \llbracket e_1 \rrbracket_i \cup \llbracket e_2 \rrbracket_i$
$\mathbf{Y}_I e_1$	no restriction	$att(e_1)$	$i > 0$ and $\tau_i - \tau_{i-1} \in I$ and $u \in \llbracket e_1 \rrbracket_{i-1}$
$e_1 \mathbf{S}_I^m e_2$	$att(e_1) \subseteq att(e_2)$	$att(e_2)$	$\exists j \leq i. \tau_i - \tau_j \in I$ and $u \in$ $\llbracket e_2 \rrbracket_j$ and $\forall k. j < k \leq i \Rightarrow$ $u _{att(e_1)} \in_m \llbracket e_1 \rrbracket_k$
$\mathbf{X}_I e_1$	no restriction	$att(e_1)$	$\tau_{i+1} - \tau_i \in I$ and $u \in \llbracket e_1 \rrbracket_{i+1}$
$e_1 \mathbf{U}_I^m e_2$	$att(e_1) \subseteq att(e_2)$	$att(e_2)$	$\exists j \geq i. \tau_j - \tau_i \in I$ and $u \in$ $\llbracket e_2 \rrbracket_j$ and $\forall k. j > k \geq i \Rightarrow$ $u _{att(e_1)} \in_m \llbracket e_1 \rrbracket_k$
$\omega(a' \mapsto t; \bar{a})e_1$	$att(t) \subseteq att(e_1), \bar{a} \subseteq att(e_1), a' \notin \bar{a}$	$\bar{a} \cup \{a'\}$	$M \neq \{\!\!\}\}$ and $u(a') = \omega(M)$, where $M = \{\!\!\}t(u') \mid u' \in$ $\llbracket e_1 \rrbracket_i, u _{\bar{a}} = u' _{\bar{a}}$

The metric *previously* (**Y**) and *since* (**S**) operators are as in MTL [29]. We write $[l, u]$ for the interval $\{x \in \mathbb{N} \mid l \leq x \leq u\}$. The superscript $m \in \{\triangleleft, \bowtie\}$ indicates whether **S**'s left operand is negated or not. We also support the future counterparts *next* (**X**) and *until* (**U**) with finite intervals. The derived connective $\mathbf{O}_I e$ abbreviates $\{\!\!\}\mathbf{S}_{I \infty}^{\bowtie} e$, where $\{\!\!\}$ is the unique tuple with an empty domain. Finally, $\omega(a' \mapsto t; \bar{a})$ is an aggregation of type ω over t with grouping attributes \bar{a} and result in a' . For SUM aggregations, we assume that \mathcal{D} is equipped with an associative and commutative addition operator. We usually omit the term t in COUNT aggregations as it is not used.

Table 1 defines the well-formedness, attributes, and semantics of expressions. The semantics, which implicitly depends on the trace ξ , assigns to every well-formed expression e an infinite sequence of relations $\llbracket e \rrbracket_i$, where $i \in \mathbb{N}$. A trace is an infinite sequence of time-stamped databases over the schema associated with e . We model time-stamps as natural numbers and make the standard assumption that the time-stamps are non-strictly increasing and always eventually increasing. The attributes $att(e)$ of e coincide with $att(\llbracket e \rrbracket_i)$ for all i . The following notation is used in Table 1: We write ξ_i for the i th database in the trace, which additionally carries the time-stamp $\tau_i \in \mathbb{N}$. We write $u|_U$ for the tuple u restricted to the attributes U . Terms t are interpreted as mappings $t(u)$ from tuples u over supersets of $att(t)$, the attributes occurring in t , to \mathcal{D} . By $x \in_m A$ we mean $x \in A$ if $m = \bowtie$ and $x \notin A$ if $m = \triangleleft$. The notation $\{\!\!\}\dots\}$ denotes a multiset. The aggregation operators COUNT and SUM account for multiplicities of tuples.

Example 1. Suppose that the trace ξ describes product reviews submitted by customers to a webshop, with time-stamps expressed in days. The trace is over the schema $S = \{p, r\}$, where $\text{att}(p) = \{pid, b\}$ and $\text{att}(r) = \{rid, pid, rating\}$. The relations p contain products with identifier pid and brand b whenever they are first added to the webshop. The relations r contain the reviews of product pid by a reviewer rid with the given $rating$. We want to detect review spam campaigns that target specific brands. The expression $e_{rb} \equiv r \bowtie (\mathbf{O}_{\mathbb{N}}p)$ augments each review with the brand, using the $\mathbf{O}_{\mathbb{N}}$ operator as the reviewed product must have been added before the review; we have $\text{att}(e_{rb}) = \{rid, pid, rating, b\}$. The expression $e_{ex} \equiv \pi(b)\sigma(n_1 \geq 3n_2)((\text{COUNT}(n_1; b)\mathbf{O}_{[0,6]}e_{rb}) \bowtie (\text{COUNT}(n_2; b)\mathbf{O}_{[7,27]}e_{rb}))$ obtains the set of brands that received at least three times as many reviews in the previous week than in all of the three weeks before. These brands are possible targets of spam.

The evaluation of individual TRA operators is described elsewhere [6]. We reuse their implementation from MonPoly and briefly explain the evaluation of $e_1 \mathbf{S}_I^{\bowtie} e_2$. For this operator, the algorithm stores a list containing pairs of time-stamps and relations. This list is continuously updated so that at time-point i , $\llbracket e_1 \mathbf{S}_I^{\bowtie} e_2 \rrbracket_i$ is equal to the union of the relations in the list whose time-stamp difference to the current time-stamp τ_i is in the interval I . For every new time-point j , the algorithm first intersects all relations in the list with $\llbracket e_1 \rrbracket_j$ and then adds $(\tau_j, \llbracket e_2 \rrbracket_j)$ to it. Elements that are too old with respect to I are removed. We note that there exists a faster algorithm [4], but we believe that it has little or no advantage in terms of space as it stores more (redundant) information to obtain a better time complexity. Confirming this intuition empirically is left as future work.

3 Algorithmic Details

Our monitoring algorithm has two phases. In the initialization phase, the monitor randomly chooses a hash function, and it rewrites the TRA expression to introduce explicit hashing operators. The rewriting will be explained in Sect. 3.2. In the main phase, the rewritten expression is evaluated over the incoming trace. The main phase is mostly the same as MonPoly’s algorithm and we will only discuss how our modifications affect it.

3.1 Hash Abstractions

We begin by describing the hashing operators that we added to the TRA and how they are evaluated. We first focus on the simpler case where attributes are hashed individually before we generalize to merged attributes. Our monitoring algorithm is parametrized by a family \mathcal{H} of hash functions from \mathcal{D} to 2^k , where $k \in \mathbb{N}$ is the hash size in bits. The monitor samples a single function $h \in \mathcal{H}$ uniformly at random in the initialization phase. We assume a set $\mathcal{A}_{\#}$ denoting hashed attributes, disjoint from \mathcal{A} . The set contains an attribute named $\#a$ for

every $a \in \mathcal{A}$. Let u be a tuple and $X \subseteq \text{att}(u) - \mathcal{A}_\#$. We define $h_X(u)$ as the tuple over $(\text{att}(u) - X) \cup \{\#v \mid v \in X\}$ satisfying

$$h_X(u)(a) = \text{if } a = \#b \text{ for } b \in X \text{ then } h(u(b)) \text{ else } u(a).$$

For a relation R and $X \subseteq \text{att}(R) - \mathcal{A}_\#$, let $h_X(R)$ be the image of R under h_X . We call $h_X(u)$ and $h_X(R)$ *hash abstractions*, as many different tuples and relations map to the same value. We could now add h_X as a new operator to TRA, with the semantics just described. All other operators would be extended to attributes from the set $\mathcal{A} \cup \mathcal{A}_\#$.

Consider the example $p(a, b) \mathbf{S}_{[0,9]}^\times ((\mathbf{O}_{[0,9]}q(a)) \bowtie (\mathbf{O}_{[0,9]}r(b)))$. Intuitively, all atomic expressions should be hashed, as the temporal operators \mathbf{O} and \mathbf{S} store their results for some time. Therefore, we would monitor $h_{\{a,b\}}p(a, b) \mathbf{S}_{[0,9]}^\times ((\mathbf{O}_{[0,9]}h_{\{a\}}q(a)) \bowtie (\mathbf{O}_{[0,9]}h_{\{b\}}r(b)))$. Observe that both arguments to the top-level \mathbf{S} operator have attributes $\{\#a, \#b\}$. The operator compares the equality of tuples over these attributes, i.e., it always compares the values of both $\#a$ and $\#b$ simultaneously. If we hashed these values again into a single k -bit hash, we would only need half the number of bits to store a tuple in \mathbf{S} 's state while still being able to correctly test equality with high probability. However, the following example shows that generalizing this idea is not straightforward.

Example 2. In the expression

$$(p(a) \bowtie \mathbf{O}_{\mathbb{N}}q(b, c)) \mathbf{S}_{[0,9]}^\times ((\varrho(c \leftarrow a)p(a)) \bowtie \mathbf{O}_{\mathbb{N}}\varrho(a \leftarrow b, b \leftarrow c)q(b, c)),$$

it is impossible to hash every atomic expression into a single attribute *and* to do the same with the relations going into the \mathbf{S} operator. The reason is that the left operand's values would have the shape $h(a, h(b, c))$ whereas for the right operand it is $h(c, h(a, b))$.

We solve this problem by employing special hash functions on tuples. Functions from this family have the property that the hash of a tuple u over U can be computed even if for some disjoint subsets $U' \subseteq U$ only the hashes of $u|_{U'}$ are available. Specifically, it is possible to merge hashes of tuples over disjoint attribute sets such that the result is independent of the merging order. The construction works in two steps: First, a single hash function as above compresses the values of each attribute to k bits. Second, we combine these “pre-hashes” using a linear form over the finite field $GF(2^k)$, whose elements are in a bijection with k -bit strings. The coefficients of the linear form, one for every attribute, are chosen uniformly at random in the initialization phase.

The second (combining) step is a well-known method [34] for extending the domain of universal hash functions that was proposed by Wegman and Carter [49]. Accordingly, we assume that the “pre-hash” family \mathcal{H} is ϵ' -almost universal:

Definition 1 [45]. *A finite family \mathcal{H} of functions $\mathcal{D} \rightarrow 2^k$ is ϵ' -almost universal iff $|\{h \in \mathcal{H} \mid h(x) = h(y)\}|/|\mathcal{H}| \leq \epsilon'$ for all distinct $x, y \in \mathcal{D}$.*

We generalize the combining step to tuples over any subset of attributes. The resulting hashes for different attribute sets are not related in a meaningful way. This is not an issue: the monitoring algorithm always compares hashes over the same attributes. To merge hashes for disjoint attribute sets, we simply add them in $GF(2^k)$. As a further modification, pre-hashes are not multiplied with their coefficients until they are about to be merged for the first time. Thus, the hash values for different *nonmerged* attributes remain comparable. This allows us to evaluate a selection operator that compares two different hashed attributes, for example. We arrive at the following definition for our hash family derived from \mathcal{H} . Fix a finite set of attributes $\mathcal{A}_e \subset \mathcal{A}$, which will be instantiated with the set of all attributes that occur in the monitored expression e .

Definition 2. *The distribution \mathcal{H}^* is obtained by sampling $h \in \mathcal{H}$ and $f \in \mathcal{A}_e \rightarrow GF(2^k)$ uniformly and independently at random, then mapping (h, f) to the function*

$$h^*(u) = \mathbf{if} \text{ att}(u) = \{a\} \text{ for some } a \mathbf{ then } h(u(a)) \mathbf{ else } \sum_{a \in \text{att}(u)} f(a) \cdot h(u(a))$$

defined on tuples u over subsets of \mathcal{A}_e . All arithmetic is over $GF(2^k)$.

Lemma 1. *Suppose that \mathcal{H} is ϵ' -almost universal. Define $\epsilon = \epsilon' + 2^{-k}$. For all tuples $u_1 \neq u_2$ over the same attributes in \mathcal{A}_e , $\Pr_{h^* \in \mathcal{H}^*} [h^*(u_1) = h^*(u_2)] \leq \epsilon$.*

As before, we would like to control which attributes are hashed and also how they are merged. We generalize $\mathcal{A}_\#$ to attributes of the form $\#X$, where X is a finite, nonempty subset of \mathcal{A} . A *hash specifier* Y is a set of disjoint and nonempty subsets of $\mathcal{A}_e \subset \mathcal{A}$. We then generalize hash abstractions as follows, where u is a tuple with $\bigcup Y \subseteq \text{att}(u)$:

$$h_Y^*(u)(a) = \mathbf{if} \ a = \#X \text{ for } X \in Y \mathbf{ then } h^*(u|_X) \mathbf{ else } u(a).$$

The bound on the collision probability from Lemma 1 carries over to hash abstractions: $h_Y^*(u_1)$ is equal to $h_Y^*(u_2)$ for two distinct tuples u_1 and u_2 with probability at most ϵ .

The next lemma is a key property of \mathcal{H}^* . It allows us to extend the domain of h_Y^* to tuples that already contain hashed values, as long as they are compatible with Y . This restriction is captured by the relation $Y_1 \sqsubseteq Y_2$ defined by $\forall X_1 \in Y_1. \exists X_2 \in Y_2. X_1 \subseteq X_2$.

Lemma 2. *Let $Y_1 \sqsubseteq Y_2$ be hash specifiers. Then $h_{Y_2}^*(u)$ can be computed from $h_{Y_1}^*(u)$.*

We conclude with a summary of the augmented TRA. We add the hashing operator h_Y^*e to the syntax introduced in Sect. 2. It is well-formed iff Y is a hash specifier satisfying $\text{hsp}(e) \sqsubseteq Y$, where $\text{hsp}(e) = \{X \mid \#X \in \text{att}(e)\}$ is the unique hash specifier induced by e 's attributes. We have $\text{att}(h_Y^*e) = (\text{att}(e) - \mathcal{A}_\# - \bigcup Y) \cup \{\#X \mid X \in Y\}$. There are new well-formedness requirements for the other operators: Hashed attributes may be renamed only if they have not

been merged. If a hashed attribute occurs in a selection or assignment operator, it must be nonmerged and both terms must consist of a single hashed attribute or a constant; the selection must be of type $=$ or \neq . Hashed attributes must not occur in an aggregation’s term. For binary operators, the operands’ induced hash specifiers must be equal on the operands’ common attributes. The semantics $\llbracket h_Y^* e \rrbracket_i$ is obtained by applying the computation from Lemma 2 to each tuple in the input relation $\llbracket e \rrbracket_i$, which is interpreted as a hash abstraction over $hsp(e)$. The hash function h^* used in these computations is sampled from the distribution \mathcal{H}^* during the monitor’s initialization phase.

3.2 Expression Rewriting

We now describe how the hash operators are inserted in the initialization phase. Ideally, this transformation should result in a space-optimal evaluation while keeping the worst-case error probability below a user-defined threshold (or vice versa). Achieving this objective is a hard optimization problem. For example, it might not be optimal to hash a temporal operator’s operand that always evaluates to a small relation. The error incurred by later operators may be comparatively large and it would be more effective to spend the error budget elsewhere. It is impossible to compute exact bounds on the relations’ sizes because the satisfiability of relational algebra queries is already undecidable [1]. Therefore, one must relax the optimization, and it is not clear how to do that in a principled manner. We defer the analysis of this problem to future work and instead rely on a heuristic to rewrite the expression. The heuristic is based on the following principles:

- The expression’s structure does not change except that hash operators are inserted.
- Every attribute is hashed greedily in the operands of temporal connectives, as these connectives contribute the most to the monitor’s state. Operands of the other binary connectives may be hashed so that they have the same hashed attributes. An attribute cannot be hashed if any operator on the path to the expression’s root performs an operation other than equality testing with the attribute’s value. The equality test may be implicit, e.g., as part of a join. The reason is that other operations, such as orderings, cannot be evaluated on hashed values (which incidentally hampers the expressiveness of BDD-based monitors [21, 22]). Order-preserving perfect hash functions are not suited for our purpose because of their superlinear space lower bound [17].
- Any set of hashed attributes is merged greedily whenever it is used homogeneously by all operators on the path to the root, i.e., all attributes’ values are always compared together. For instance, a and b cannot be merged in the join’s operands in $p(a, b) \bowtie q(a)$ because a ’s but not b ’s values are compared.
- In general, the greedy approach assumes that it is better to hash and merge than not. Other objectives may be more appropriate in specific applications. For example, if specification violations must always be detected but false alerts are acceptable, a different heuristic taking the predicted error (Sect. 4.1) into account should be used.

Algorithm 1. Expression rewriting

```

let addHash((e, Y), Y') = if Y' = Y then e else hY',e*
let rec rw(apx, Y, e) = match e with
| π(ā)e1 ⇒ let (ē1, Y1) = rw(apx, Y ∪ {att(e1) - ā}, e1) in
    let ā' = {if a ∈ ∪ Y1 then #X for unique X ∈ Y1 s.t. a ∈ X else a | a ∈ ā} in
    (π(ā')ē1, {X ∩ ā | X ∈ Y1})
| e1 SIm e2 ⇒ let K = att(e1), apx' = apx ∧ (|I| < ∞) in
    let Y' = ({X - K | X ∈ Y} ∪ (if apx' then {X ∩ K | X ∈ Y} else ∅)) in
    (addHash(rw(apx', Y', e1), Y') SIm addHash(rw(apx', Y', e2), Y'), Y')
| ...

```

We implemented the heuristic as a bottom-up rewriting procedure (Algorithm 1). We only show the π and \mathbf{S} cases due to space constraints; see the extended report [42] for the full algorithm. The projection case illustrates how the connectives' parameters are adjusted, and \mathbf{S} imposes the most interesting constraints on its operands.

The main function $\text{rw}(apx, Y, e)$ transforms the expression e . Its result is a pair (\tilde{e}, Y') , where \tilde{e} is the rewritten expression and $Y' = \text{hsp}(\tilde{e})$ is the hash specifier induced by \tilde{e} . The constraints apx and Y represent the restrictions imposed by the operations on the path from the root to the current expression e . The boolean apx indicates whether \tilde{e} may introduce errors. There can be hash operators in \tilde{e} even if errors are disallowed, but the hashed attributes must not be tested for equality. The hash specifier Y partitions the sub-expression's attributes. Attributes not in $\bigcup Y$ are excluded from hashing, and the partitioning in Y indicates which attributes may be merged. For the root expression, we set apx to true, and the specifier Y is the empty set: as the relations computed for the root are output to the user, there should not be any hashed values in that output.

For projections $\pi(\bar{a})e_1$, the sub-expression e_1 is rewritten using the same constraints, except that the removed attributes $\text{att}(e_1) - \bar{a}$ can be hashed and merged (but not with other attributes). The rewriting function computes a new list \bar{a}' of projected attributes to account for the new names of the hashed attributes. The order of this list does not matter, hence we define it using set notation. Note that the heuristic is not greedy for projections: no hash abstraction is inserted if $\{X \cap \bar{a} \mid X \in Y_1\}$ differs from the constraint Y .

For $e_1 \mathbf{S}_I^m e_2$, the key K consists of the attributes $\text{att}(e_1)$ that the connective tests for equality internally. The operands are rewritten recursively. The apx flag is propagated unless I is unbounded. In this case, it could be possible to force an error at a sufficiently large time-point if the operands are not exact. The specifier Y' is derived from Y : If errors are allowed, the sets in Y are split into key and non-key attributes. Otherwise, all key attributes are removed, as the equality test on hashed keys might introduce errors. Finally, the rewritten operands are wrapped in hashing operations so that both operands have compatible hashed attributes (namely Y').

Example 3. The expressions from Example 1 are rewritten to $\tilde{e}_{rb} \equiv h_{Y_2}^*(h_{Y_1}^* r) \bowtie (\mathbf{O}_{\mathbb{N}} h_{Y_1}^* p)$ and $\tilde{e}_{ex} \equiv \pi(b)\sigma(n_1 \geq 3n_2)((\text{COUNT}(n_1; b)\mathbf{O}_{[0,6]}\tilde{e}_{rb}) \bowtie (\text{COUNT}(n_2; b)\mathbf{O}_{[7,27]}\tilde{e}_{rb}))$. The attribute b representing the brand is never

hashed because it is part of the monitor’s output, which evaluates \tilde{e}_{ex} . At first only $Y_1 = \{pid\}$ is hashed, as pid is the only attribute apart from b exposed to a temporal connective in \tilde{e}_{rb} . After the join in \tilde{e}_{rb} , $Y_2 = \{pid, rid, rating\}$ can be merged, as all three attributes are discarded by the aggregations.

4 Analysis of the Algorithm

In this section, we analyze the error probability of our monitor and comment on its space complexity. Our analysis relates the error probability to the size of the hash values, which affects the algorithm’s space complexity. Specifically, we show how to compute an upper bound on the error probability for a given expression. This results in a symbolic expression whose variables refer to the collision probability ϵ , the maximum number of time-points per unit of time, and the maximum relation sizes that may occur during the expression’s evaluation. Based on this information, the user can adjust the hash size to achieve the desired level of accuracy. Additionally, we show that a concrete error bound can be computed by the monitor for a particular trace. This may provide a more precise error estimate.

4.1 Error Bounds

We first establish a formal framework in which we carry out our analysis. To this end, we introduce the notion of randomized monitoring, which allows us to quantify the monitor’s accuracy in terms of its worst-case error probability. A *randomized monitor* M is modeled as a mapping from finite trace prefixes to discrete probability distributions over finite sequences of relations. We assume that M satisfies the following completeness property. For every (infinite) trace ξ there exists a look-ahead function ℓ_ξ , which maps any desired length of the monitor’s output to a sufficient length of the monitor’s input. More precisely, the length of the sequences in the support² of $M(x)$ is at least n for every prefix x of ξ with length $|x| \geq \ell_\xi(n)$. In other words, outputs may be delayed, but the monitor must always eventually compute a verdict. Our monitor inherits its look-ahead function from MonPoly; it depends only on the upper bounds of the future operator’s intervals and on the time-stamps in ξ .

We parametrize the monitoring problem by a nonempty, possibly infinite set X of traces, which represents the application-specific knowledge about the possible inputs to the monitor. We also fix a TRA expression e and perform the following random experiment: for any trace $\xi \in X$ and time-point $i \in \mathbb{N}$, the monitor is run on a sufficiently long prefix of ξ using fresh randomness. We are interested in the worst-case probability (over the choice of ξ and i) of the i th output deviating from the correct relation. Let $x \ll \xi$ denote that x is a finite prefix of ξ , and let x_i denote the i th element in the sequence x . We make the

² The support of a discrete probability distribution is the set of values with nonzero probability.

semantics' dependency on the trace explicit: from now on, we write $\llbracket e \rrbracket_i^\xi$ instead of $\llbracket e \rrbracket_i$, where ξ is the trace.

Definition 3. *The error probability of M on X is*

$$\text{err}_X(M) = \sup_{\xi \in X, i \in \mathbb{N}, x \ll \xi, |x| \geq \ell_\xi(i)} \text{Pr}_M[M(x)_i \neq \llbracket e \rrbracket_i^\xi].$$

Similarly, the false-positive probability $fp_X(M)$ and the false-negative probability $fn_X(M)$ are defined by

$$\begin{aligned} fp_X(M) &= \sup_{\xi \in X, i \in \mathbb{N}, x \ll \xi, |x| \geq \ell_\xi(i)} \text{Pr}_M[M(x)_i \not\subseteq \llbracket e \rrbracket_i^\xi] \\ fn_X(M) &= \sup_{\xi \in X, i \in \mathbb{N}, x \ll \xi, |x| \geq \ell_\xi(i)} \text{Pr}_M[M(x)_i \not\supseteq \llbracket e \rrbracket_i^\xi]. \end{aligned}$$

Lemma 3. $\max\{fp_X(M), fn_X(M)\} \leq \text{err}_X(M) \leq fp_X(M) + fn_X(M)$.

The error probability is our measure of the monitor's accuracy. The false-positive and false-negative probabilities provide more information about the nature of the errors. In some applications it may be more tolerable to have errors of the one kind than of the other. No probability distribution is associated with X ; the probabilities are taken solely with respect to the internal coin flips of the algorithm implementing M .

According to Definition 3, the trace ξ cannot depend on the randomness of M . Such a dependency would be incompatible with our hashing approach, as one could construct an adversarial input that causes an error with certainty at sufficiently large time-points (by trying different values until a hash collision is found). However, the probability of an error at *some* time-point can be 1 even if $\text{err}_X(M) < 1$, as for many specifications it is unavoidable that a collision occurs somewhere in an infinite trace if the domain is large enough. Therefore, we consider the probability for each time-point in isolation in Definition 3.

The following design decisions guide our error analysis: (1) It should be compositional so that the bounds can be computed by recursion over the expression's structure. (2) The set of traces is parameterized by the maximum rate and the maximum relation sizes for each sub-expression, as defined below. We need to bound these quantities because the worst-case error probability would otherwise be 1 for most expressions. Moreover, relying on concrete numeric upper bounds makes the analysis more precise. (3) We analyze the false-positive and false negative probabilities separately; by Lemma 3, this allows us to approximate the overall error probability within a factor of 2.

The first step is to adapt the notions of false-positive and false-negative probabilities to rewritten expressions \tilde{e} . To this end, we recover the original, exact expression e from \tilde{e} by removing all hash operators. We perform the analysis on \tilde{e} instead of e to decouple it from the heuristic used by Algorithm 1.

Definition 4. *Let X be a set of traces. Suppose that e is the unique expression obtained by removing all hash operators from \tilde{e} and flattening attributes of the form $\#X$ into an enumeration of X (see the extended report [42] for details).*

Table 2. Upper bounds on false-positive and false-negative error probabilities, per time-point

\tilde{e}	$fp(\tilde{e}) \leq \dots$	$fn(\tilde{e}) \leq \dots$	
R, r	0	0	
$h_Y^* \tilde{e}_1, \pi(_) \tilde{e}_1, \varrho(_) \tilde{e}_1, \eta(_) \tilde{e}_1, \sigma(t_1 \circ t_2) \tilde{e}_1, \mathbf{Y}_I \tilde{e}_1, \mathbf{X}_I \tilde{e}_1$	$fp(\tilde{e}_1)$	$fn(\tilde{e}_1)$	(1)
$\sigma(\#\{a\} = t) \tilde{e}_1$	$fp(\tilde{e}_1) + \epsilon e_1 $	$fn(\tilde{e}_1)$	
$\sigma(\#\{a\} \neq t) \tilde{e}_1$	$fp(\tilde{e}_1)$	$fn(\tilde{e}_1) + \epsilon e_1 $	
$\tilde{e}_1 \bowtie \tilde{e}_2$	$fp(\tilde{e}_1) + fp(\tilde{e}_2) + \epsilon e_1 e_2 $	$fn(\tilde{e}_1) + fn(\tilde{e}_2)$	(2)
$\tilde{e}_1 \triangleright \tilde{e}_2$	$fp(\tilde{e}_1) + fn(\tilde{e}_2)$	$fn(\tilde{e}_1) + fp(\tilde{e}_2) + \epsilon e_1 e_2 $	(2)
$\tilde{e}_1 \cup \tilde{e}_2$	$fp(\tilde{e}_1) + fp(\tilde{e}_2)$	$fn(\tilde{e}_1) + fn(\tilde{e}_2)$	
$\tilde{e}_1 \mathbf{S}_I^{\bowtie} \tilde{e}_2, \tilde{e}_1 \mathbf{U}_I^{\bowtie} \tilde{e}_2$	$a_I \cdot fp(\tilde{e}_1) + b_I \cdot fp(\tilde{e}_2) + \epsilon \cdot b_I e_1 e_2 $	$a_I \cdot fn(\tilde{e}_1) + b_I \cdot fn(\tilde{e}_2)$	(2)
$\tilde{e}_1 \mathbf{S}_I^{\triangleleft} \tilde{e}_2, \tilde{e}_1 \mathbf{U}_I^{\triangleleft} \tilde{e}_2$	$a_I \cdot fn(\tilde{e}_1) + b_I \cdot fp(\tilde{e}_2)$	$a_I \cdot fp(\tilde{e}_1) + b_I \cdot fn(\tilde{e}_2) + \epsilon \cdot c_I e_1 e_2 $	(2)
$\omega(a' \mapsto t; \bar{a}) \tilde{e}_1$	$fp(\tilde{e}_1) + fn(\tilde{e}_1) + \epsilon (e_1 ^2 - e_1)/2$	$fp(\tilde{e}_1) + fn(\tilde{e}_1) + \epsilon (e_1 ^2 - e_1)/2$	(3)

$a_I = (\maxRate \cdot u) - 1$ and $b_I = \maxRate \cdot (u - l)$ and $c_I = b_I \cdot (\maxRate \cdot l + (b_I + 1)/2)$ for any half-open interval $I = \{x \in \mathbb{N} \mid l \leq x < u\}$. Set $a_I = b_I = c_I = \infty$ if I is unbounded.

Side conditions and remarks: (1) no hashed attribute in t_1 nor in t_2 ; (2) replace ϵ by 0 if there is no hashed attribute in $att(\tilde{e}_1) \cap att(\tilde{e}_2)$; (3) replace ϵ by 0 if: $\omega \in \{\text{MIN}, \text{MAX}\}$ and no hashed attribute in \bar{a} , or $\omega \in \{\text{COUNT}, \text{SUM}\}$ and no hashed attribute in e_1 .

Writing $h_Y^*(R)$ for the image of R under h_Y^* , the false-positive and false-negative probabilities of \tilde{e} are

$$fp(\tilde{e}) = \sup_{\xi \in X, i \in \mathbb{N}} \Pr[\llbracket \tilde{e} \rrbracket_i^\xi \not\subseteq h_{hsp(\tilde{e})}^*(\llbracket e \rrbracket_i^\xi)], \quad fn(\tilde{e}) = \sup_{\xi \in X, i \in \mathbb{N}} \Pr[h_{hsp(\tilde{e})}^*(\llbracket e \rrbracket_i^\xi) \not\subseteq \llbracket \tilde{e} \rrbracket_i^\xi].$$

The applications of $h_{hsp(\tilde{e})}^*$ ensure that the relations are over the same attributes. It may be surprising that a hash collision in $\llbracket e \rrbracket_i^\xi$ (i.e., two tuples $u, v \in \llbracket e \rrbracket_i^\xi$ such that $h_Y^*(u) = h_Y^*(v)$) does *not* count as an error. Definition 4 is nonetheless useful as $hsp(\tilde{e})$ is forced to be \emptyset at the monitored expression's root. Our main result follows.

Theorem 1. *Suppose that \mathcal{H} is an ϵ' -almost universal hash family with k bits. Then the bounds in Table 2 follow, where $\epsilon = \epsilon' + 2^{-k}$, $|e| = \sup_{\xi \in X, i \in \mathbb{N}} |\llbracket e \rrbracket_i^\xi|$ is the maximum size of the relations computed for e , and $\maxRate = \sup_{\xi \in X, x \in \mathbb{N}} |\{i \mid \tau_i = x\}|$ is the traces' maximum rate per time unit. If upper bounds on \maxRate and on $|e|$ for every sub-expression e of e_0 are given, one can compute constants c and c' in polynomial time such that $fp_X(\tilde{M}) \leq \epsilon \cdot c$ and $fn_X(\tilde{M}) \leq \epsilon \cdot c'$, where \tilde{M} is our monitor for e_0 and \mathcal{H} .*

The factors a_I , b_I , and c_I in Table 2 are estimates of the number of time-points or pair of time-points that may be the source of errors of a particular kind. The derivation of the factors is explained in the extended report [42]. The asymmetry between $fp(\tilde{e}_1 \mathbf{S}_I^{\bowtie} \tilde{e}_2)$ and $fn(\tilde{e}_1 \mathbf{S}_I^{\triangleleft} \tilde{e}_2)$ and similarly for \mathbf{U} is noteworthy. It is possible to construct examples that show that the false-negative probability of the \triangleleft operators may exceed the tighter bound that uses b_I instead of c_I .

Table 2 can be used to calculate error bounds given ϵ , or to calculate the largest ϵ such that the error is below a given threshold. The collision probability ϵ is a proxy for the hash values’ size, and thereby a factor of the randomized monitor’s space complexity. Although the bounds in Table 2 are not tight, our empirical evaluation (Sect. 5) shows that they are useful. Moreover, as the size of the hash values is logarithmically related to ϵ , achieving a tight bound is not critical in practice.

Example 4. For e_{ex} from Example 1, we compute using Table 2 that $fp(\tilde{e}_{ex}), fn(\tilde{e}_{ex}) \leq \epsilon \cdot (28 \cdot \text{maxRate} \cdot |r| \cdot |\mathbf{O}_{\text{INP}}| + (|\mathbf{O}_{[0,6]e_{rb}}|^2 + |\mathbf{O}_{[7,27]e_{rb}}|^2 - |\mathbf{O}_{[0,6]e_{rb}}| - |\mathbf{O}_{[7,27]e_{rb}}|)/2)$. Our implementation (Sect. 5) achieves a collision probability of $\epsilon \approx 2^{-61}$ with 63-bit hashes. Now assume that there are 10^6 products in total ($|\mathbf{O}_{\text{INP}}| = 10^6$), and at most 10^5 reviews are received per day ($\text{maxRate} \cdot |r| \leq 10^5$, $|\mathbf{O}_{[0,6]e_{rb}}| \leq 7 \cdot 10^5$, etc.). This yields an upper bound of around $2.3 \cdot 10^{-6}$ for each error type, or $4.6 \cdot 10^{-6}$ for the probability of any error occurring, per time-point. Conversely, we can compute that $\epsilon \leq 9.6 \cdot 10^{-17}$, which roughly corresponds to 55 hash bits, is sufficient to achieve an error rate below 0.1%.

The size bounds $|e|$ referred to by Theorem 1 are for the original, non-rewritten sub-expressions. Nonetheless, extensive domain knowledge might be necessary to obtain such bounds prior to monitoring, e.g., to choose the hash size appropriately. There is an alternative use of Table 2: The monitor may compute estimates of $fp(e_0)$ and $fn(e_0)$ for the specific trace it monitors. These can be more precise than the *a priori* estimates using Theorem 1. Our implementation computes trace-specific error bounds and presents them to the user. This can aid the user in judging the reliability of these verdicts. However, one challenge is that the observed relation sizes may be smaller than the bounds $|e|$, namely if there are false negatives, or hash collisions such that $|h_Y^*(\llbracket e \rrbracket_i^\xi)| < |\llbracket e \rrbracket_i^\xi|$. Calculating with observed sizes could result in estimates that are too small. (Larger observed sizes do not affect correctness because all our error bounds are monotonic.) We circumvent this by falling back to conservative upper bounds (e.g., the sum of the operands’ sizes for a union) for those sub-expressions with hashed attributes and/or possible false negatives.

4.2 Space Complexity

We focus on data complexity [47] and characterize a subset of expressions on which our approach works best, in that the monitor state contains only hashed values. An expression e is called *simple* if it is closed (i.e., $\text{att}(e) = \emptyset$), all intervals are finite, no functions appear in terms, all selections have the form $\sigma(t_1 = t_2)$ or $\sigma(t_1 \neq t_2)$, and all aggregations have type COUNT. Then the temporal connectives in the rewritten expression involve only hashed (not necessarily merged) attributes, which eliminates the influence of the domain value’s encoding as follows. Let $X_{m,n}$ be the class of traces for which $\text{maxRate} \leq m$, $|r| \leq m$ for every relation name $r \in S$, and all domain values are represented by at most n bits. It

is known that e can be monitored over prefixes of $X_{m,n}$ using polynomially many (in m) relations [6]. These relations have polynomially bounded cardinality as every interval in e is finite. A typical monitor for e would store all domain values that occur in the relations, and therefore the space complexity is multiplied by n . In contrast, our monitor works exclusively with relations over k -bit hashes. The polynomial bound on the number and cardinality of these relations persists, but it suffices to choose k on the order of $\log(\text{poly}(m)) - \log x = O(\log m - \log x)$ to achieve an error probability below x . This follows from Theorem 1 and the fact that every sub-expression of e has polynomially bounded size. Therefore, k is independent of n .

Theorem 2. *Simple expressions can be monitored over traces in $X_{m,n}$ in $O(\text{poly}(m) + n)$ space (with a fixed error bound).*

5 Implementation and Evaluation

We implemented our randomized monitor as an extension of the MonPoly tool [6], written in OCaml. The extension is transparent to the user: hashing can be enabled by setting a single command-line option. We performed experiments, using both Amazon review data [36] and randomly generated data, to answer the following questions: (Q1) Are there non-trivial specifications and data for which monitoring benefits from our approach? (Q2) How much does it reduce the monitor’s peak memory usage in practice? (Q3) How do our theoretical error bounds compare to the empirically observed error probability?

We added a module to MonPoly that implements the rewriting algorithm described in Sect. 3.2. Merging of attributes can be disabled to study its impact. For \mathcal{H} we use the CLHASH family [32] truncated to $k = 63$ bits, the size of native integers in OCaml. Truncated CLHASH is ϵ' -almost universal with $\epsilon' = 2.004/2^{63}$ for strings up to 2^{64} bytes [32, Lemmas 1 and 9], requiring only around 1 KiB to represent an element of \mathcal{H} . We modified MonPoly’s relation data type to keep track of the error and size bounds as described in Sect. 4. OCaml programs rely on a garbage collector (GC), which makes it difficult to measure peak memory usage in a meaningful way. However, we found OCaml’s GC to be conservative. Measured differences above a few MB were generally robust.

Our experiments were performed on two groups of expressions and data. The first group focused on a realistic use case, specifically the detection of fraudulent customer reviews. We used review data from Amazon spanning a period of over 20 years [36]. We restricted our attention to the “gift cards” category, which had the smallest number of products (1548) and a moderate number of reviews (147 194). We monitored Example 1 (adjusted to ignore additional attributes) and a formalization e_{frd} of the first stage of Heydari et al.’s fake review detection system [25], shown in the extended report [42]. The latter detects weeks and product brands with suspicious review counts. (The second stage would require some natural language processing, which is outside of our scope.) We modified the fake detection example to use a one-year sliding window for the review average per brand, whereas the original uses a global average, which would require offline monitoring.

The second group was based on the expressions $e_1 \equiv \pi()(p(a) \bowtie \mathbf{O}_{[0,9]}q(a))$, $e_2 \equiv \pi()(p(a) \mathbf{S}_{[0,9]}^{\downarrow} q(a))$, $e_3 \equiv \pi()(p(a) \mathbf{U}_{[0,9]}^{\bowtie} q(a, b) \triangleright q(a, b))$, and the one from Example 2 with all attributes projected away as e_4 . In e_3 , the hash abstraction of q is computed twice: once for each of q 's occurrences. The expression e_3' is a modification of e_3 in which the hash abstraction is shared by both occurrences. We generated pseudorandom trace prefixes with consecutive, non-repeating time-stamps over 100 time units with 20 000 tuples each, which were assigned randomly and uniformly to the relation names in the expression. For e_4 , we generated only 50 tuples per time-point because the formula computes a Cartesian product, resulting in a large blow-up. The domain values were random alphanumeric strings with exactly 100 characters. The second group's purpose was to determine the impact of the expressions' structure on the memory usage. It is clear that hashing is less effective for smaller values, so we did not perform further experiments with such values.

We performed additional experiments with data suitable for the DejaVu tool [21, 22]. DejaVu is a monitor for first-order past LTL with time constraints, implemented using binary decision diagrams (BDDs) instead of finite relations. DejaVu is the only other tool handling a large subset of TRA that we are aware of. Of our expressions, only e_1 , e_2 , and e_4 are supported by DejaVu because it lacks aggregation and future operators. It also cannot process simultaneous events. Therefore, we generated a separate set of traces (the ‘‘thin’’ set) with 2 000 time-points per time-stamp (50 for e_4), each consisting of a single tuple.

Table 3. Performance evaluation (B = baseline, Hm = merged hashes; percentages relative to B)

	Memory (MiB)				Runtime (s)		Max. error bound	
	B	ID	H	Hm	B	Hm	fp	fn
e_{ex}	13.7	13.7 (−0%)	12.9 (−6%)	12.0 (−12%)	20.5	17.2 (−16%)	$1 \cdot 10^{-5}$	$1 \cdot 10^{-5}$
e_{frd}	35.4	39.6 (+12%)	28.8 (−19%)	23.9 (−33%)	22.7	22.2 (−2%)	$2 \cdot 10^{-10}$	$2 \cdot 10^{-10}$
e_1	81.0	74.7 (−8%)	63.1 (−22%)	63.3 (−22%)	12.5	14.3 (+14%)	$3 \cdot 10^{-10}$	0
e_2	56.0	56.0 (+0%)	44.5 (−21%)	44.6 (−20%)	28.9	26.1 (−9%)	0	$1 \cdot 10^{-9}$
e_3	80.9	104.2 (+29%)	81.0 (+0%)	81.1 (+0%)	9.3	15.9 (+70%)	$1 \cdot 10^{-9}$	$3 \cdot 10^{-10}$
e_3'	80.8	81.0 (+0%)	56.1 (−31%)	56.4 (−30%)	9.3	12.7 (+31%)	$1 \cdot 10^{-9}$	$3 \cdot 10^{-10}$
e_4	30.3	47.3 (+56%)	50.0 (+65%)	37.7 (+24%)	12.0	14.3 (+19%)	$2 \cdot 10^{-9}$	0

Memory Usage and Runtime. We measured the peak memory usage and runtime using MonPoly's original algorithm (B), a special mode (ID) where the hash function is replaced by identity, and nonmerged (H) and merged (Hm) hashing. The purpose of ID was to determine whether our expression rewriting, the added operators, and the error tracking code had any effect on their own. Measurements were obtained on a laptop with an Intel i5-7200U CPU (2.5 GHz, Turbo Boost disabled) and 8 GB RAM (no swap) under Linux 5.15.13. MonPoly was compiled with OCaml 4.12.0 and default GC settings. We used the UNIX time command to measure elapsed real time (‘‘runtime’’) and the maximum resident

set size (“memory”). We computed the arithmetic mean over 3 repetitions. We compared against DejaVu revision 1e1f4eb0, running under OpenJDK 11.0.13 with an initial heap size of 8 MB. The BDD size was set to 15 bits based on the expected number of distinct domain values within the expressions’ intervals.

Table 3 shows the results. The percentages are relative to the baseline B . Hashing reduced the amount of memory needed for all formulas except e_3 and e_4 , and merging reduced it further for the Amazon examples. The effect was small for e_{ex} because there the relevant domain values were fairly short (at most 14 bytes each). The memory for e_3 increased under ID because the added hashing operators prevent the sharing of nodes in the immutable AVL trees that MonPoly uses to represent relations. Under B , the two occurrences of q share these trees. They are hashed twice in e_3 , resulting in independent copies, but not in e'_3 , where memory improved. This sensitivity to the expression structure demonstrates the complexity of the optimization problem from Sect. 3.2. We conjecture that the generally bad behavior of e_4 is also due to the loss of sharing, specifically in the O operators.

We could not draw definite conclusions about the impact on runtime. Computing hashes and transforming the relations obviously incurs some overhead, whereas comparing hash values in the search tree implementation might be faster than comparing long strings. The last two columns of Table 3 show the largest error bound output by the monitor (maximum across time-points and repetitions). For our test data, we find that the accuracy loss is very small and errors are highly unlikely. For example, the error bounds for the e_{frd} experiment correspond to a probability of less than $2 \cdot 10^{-6}$ for an error occurring anywhere in the trace prefix, which consisted of 2889 time-points.

DejaVu generally used much more memory than MonPoly on the “thin” traces: 984 MiB for e_1 (MonPoly Hm : 26.2 MiB) and 972 MiB for e_2 (Hm : 12.1 MiB). Memory usage exceeded 2 GiB for e_4 , hence we decided to exclude this expression from the experiments. Further research is necessary to determine whether the large memory footprint is due to the implementation or a fundamental consequence of using BDDs. We note that the runtimes of DejaVu and MonPoly are highly incomparable, the latter being more than 6 times faster on e_1 but 81 times slower on e_2 . Hashing is ineffective on the “thin” traces, resulting in an increase by 1% over B for e_1 and a decrease by 8% for e_2 , which are likely just noise. However, these traces are one order of magnitude smaller than those used for Table 3, so factors that are independent of the domain values dominate.

Error Probabilities and Bounds. We artificially truncated hash values to simulate the impact of their size. The left plot in Fig. 1 shows the error probability observed over 100 repetitions with different hash function seeds. We computed the midpoint of a Wilson score interval [50] at 95% confidence for every time-point, and took the maximum over all time-points. The right plot in Fig. 1 shows the corresponding error bounds output by the monitor (fp and fn added together, mean across all repetitions, then the maximum over time-points). The error bounds are almost tight for e_1 : as true positives are extremely unlikely for our pseudorandom traces, every collision in e_1 ’s join is observed as a false

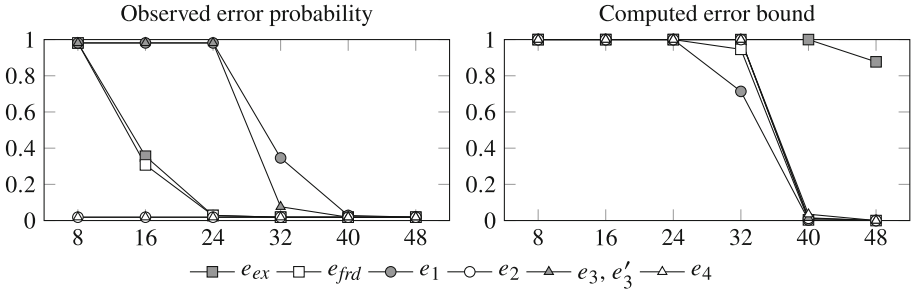


Fig. 1. Error probabilities and bounds for truncated hashes (x = number of bits)

positive. For e_{ex} , the bounds overestimate the observed error by a large margin. This is partially due to the projection operator, which hides deviations in the count aggregations as long as they do not affect the selection, and the fact that our worst-case analysis for aggregations holds for one large group, whereas the Amazon data has many groups. For e_2 and e_4 , we never observed any error because uniformly random traces are not the worst case for these expressions. For example, we can trigger errors for e_2 by generating much fewer q than p tuples.

6 Related Work

Approaches for monitoring parametrized events can be classified into several categories [23]. We focus on the bottom-up evaluation of specifications using finite relations [10, 20], which has been implemented in the MonPoly tool [6] for a fragment of metric first-order temporal logic (MFOTL) formulas. Our temporal-relational algebra (TRA) described in Sect. 2 is a direct encoding of that fragment. The principle of hashing event parameters could be applied to some of the other monitoring approaches as well, e.g., parametric trace slicing [41] and automatic structures [22]. Our error probability analysis is specific to TRA.

Some specifications can be monitored in constant space even on streams with parametrized events and unbounded rate [12, 16, 33]. Hashing allows us to reduce the memory needed for a class of specifications that falls outside of the constant-space fragment. Alternatively, monitoring performance can be improved by sampling from the input trace and interpolating over the gaps [3, 28]; a hidden Markov model represents the prior knowledge about the monitored system and it plays an important role for achieving high accuracy. Unlike our approach, sampling usually reduces the time overhead of monitoring. Grigore and Kiefer [18] studied optimal event sampling strategies for systems modeled as Markov chains. In contrast to these works that rely on sampling, we do not require any prior knowledge about the monitored system. Only for bounding the error probability we assume that certain trace statistics are available.

Statistical model checking [31, 44] is concerned with the verification of stochastic systems. The checked properties are quantitative: they express con-

straints on probabilities. Statistical model checking uses randomized simulations and thus yields approximate results. It is different from the randomized monitoring we consider, as our monitor checks individual traces, not system models, for safety properties with nonprobabilistic semantics.

There exists an extensive body of research on randomized algorithms and lower bounds related to data storage and retrieval. The standard example is the Bloom filter [8], which approximately answers set membership queries for static sets. There are variants supporting deletion [15] and dynamic resizing [2]. Set membership queries on dynamic sets can be reduced to the monitoring problem for sufficiently expressive specifications, but the latter is clearly more general. Intersections [19] and Cartesian products [48] of Bloom filters do not scale well to complex queries over relations with varying attributes, which frequently occur in first-order monitoring.

Bloom filters have been used successfully to save space in model checking algorithms, e.g., in the bitstate method [13,27], where the filters are used to track visited states. The only operation performed on the filter is a membership test, whereas first-order monitors apply complex transformations to their state. Therefore, we cannot hash the relations in the state as a whole, and the error analysis becomes more intricate. A different line of work in the model checking domain uses lossless compression schemes for states [7,26,30]. To our knowledge, such schemes have not yet been applied to monitoring, with the exception of BDDs [22]. Some of the ideas could prove fruitful, e.g., the work by Laarman et al. [30], which enforces sharing in a systematic way.

Another family of probabilistic data structures [14,39] represents the elements of a finite set using compact hash values. The work by Naor and Yogev [35] on membership queries over sliding windows is perhaps the closest to monitoring. Unlike them, we do not aim at achieving close-to-optimal memory usage, but rather we consider more richly structured sets (relations). Probabilistic data structures have been analyzed in adversarial environments [11], which can be relevant in the context of monitoring security policies. We do not consider an adversary model in this paper. Instead, we assume that there is no feedback from the monitor’s output to a possible adversary who could influence the trace adaptively.

7 Conclusion

We presented a randomized monitoring algorithm that compresses domain values using hash functions. We analysed its error probability and showed that useful upper bounds can be obtained in practice. Based on the evaluation results, we believe that hashing is a useful optimization in space-constrained applications where a small error probability can be tolerated. There are three main limitations: First, the focus on domain values means that the approach is ineffective for traces with small domain values. Second, the current implementation within MonPoly is not optimized for memory usage, and its immutable data structures sometimes exhibit unpredictable behavior. We plan to reimplement and optimize the monitor using imperative data structures. Third, the structure of the

specification may prevent hashing of some or all attributes, e.g., if functions are computed over the attributes. Open questions include: Can expressions be rewritten to allow more hashing, and how could this optimization problem be solved? What are space lower bounds for the operations relevant to first-order monitoring, going beyond basic set membership and sliding windows?

Acknowledgement. The author thanks David Basin, Srđan Krstić, Dmitriy Traytel, and the anonymous reviewers for their helpful comments and suggestions. This research was supported by the US Air Force grant “Monitoring at Any Cost” (FA9550-17-1-0306).

References

1. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*. Addison-Wesley, Boston (1995)
2. Almeida, P.S., Baquero, C., Preguiça, N.M., Hutchison, D.: Scalable Bloom filters. *Inf. Process. Lett.* **101**(6), 255–261 (2007)
3. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S.A., Stoller, S.D., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) *RV 2012*. LNCS, vol. 7687, pp. 168–182. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_18
4. Basin, D., Dardinier, T., Heimes, L., Krstić, S., Raszyk, M., Schneider, J., Traytel, D.: A formally verified, optimized monitor for metric first-order dynamic logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) *IJCAR 2020*. LNCS (LNAI), vol. 12166, pp. 432–453. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51074-9_25
5. Basin, D., Klaedtke, F., Marinovic, S., Zălinescu, E.: Monitoring of temporal first-order properties with aggregations. *Formal Methods Syst. Des.* **46**(3), 262–285 (2015). <https://doi.org/10.1007/s10703-015-0222-7>
6. Basin, D., Klaedtke, F., Müller, S., Zălinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 15:1–15:45 (2015)
7. Berg, F.I.: Recursive variable-length state compression for multi-core software model checking. In: Dutle, A., Moscato, M.M., Titolo, L., Muñoz, C.A., Perez, I. (eds.) *NFM 2021*. LNCS, vol. 12673, pp. 340–357. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-76384-8_21
8. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13**(7), 422–426 (1970)
9. Carter, L., Floyd, R.W., Gill, J., Markowsky, G., Wegman, M.N.: Exact and approximate membership testers. In: *STOC 1978*, pp. 59–65. ACM (1978)
10. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. *ACM Trans. Database Syst.* **20**(2), 149–186 (1995)
11. Clayton, D., Patton, C., Shrimpton, T.: Probabilistic data structures in adversarial environments. In: *CCS 2019*, pp. 1317–1334. ACM (2019)
12. D’Angelo, B., et al.: LOLA: runtime monitoring of synchronous systems. In: *TIME 2005*, pp. 166–174. IEEE Computer Society (2005)
13. Dillinger, P.C., Manolios, P.: Bloom filters in probabilistic verification. In: Hu, A.J., Martin, A.K. (eds.) *FMCAD 2004*. LNCS, vol. 3312, pp. 367–381. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30494-4_26

14. Fan, B., Andersen, D.G., Kaminsky, M., Mitzenmacher, M.: Cuckoo filter: practically better than bloom. In: CoNEXT 2014, pp. 75–88. ACM (2014)
15. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* **8**(3), 281–293 (2000)
16. Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L., Torfah, H.: StreamLAB: stream-based monitoring of cyber-physical systems. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 421–431. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_24
17. Fox, E.A., Chen, Q.F., Daoud, A.M., Heath, L.S.: Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Syst.* **9**(3), 281–308 (1991)
18. Grigore, R., Kiefer, S.: Selective monitoring. In: CONCUR 2018. LIPIcs, vol. 118, pp. 20:1–20:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
19. Guo, D., Wu, J., Chen, H., Yuan, Y., Luo, X.: The dynamic bloom filters. *IEEE Trans. Knowl. Data Eng.* **22**(1), 120–133 (2010)
20. Havelund, K.: Rule-based runtime verification revisited. *Int. J. Softw. Tools Technol. Transf.* **17**(2), 143–170 (2014). <https://doi.org/10.1007/s10009-014-0309-2>
21. Havelund, K., Peled, D.: First-order timed runtime verification using BDDs. In: Hung, D.V., Sokolsky, O. (eds.) ATVA 2020. LNCS, vol. 12302, pp. 3–24. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_1
22. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. *Formal Methods Syst. Des.* **56**(1), 1–21 (2020)
23. Havelund, K., Reger, G., Thoma, D., Zălinescu, E.: Monitoring events that carry data. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 61–102. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_3
24. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_24
25. Heydari, A., Ali Tavakoli, M., Salim, N.: Detection of fake opinions using time series. *Expert Syst. Appl.* **58**, 83–92 (2016)
26. Holzmann, G.J.: State compression in SPIN: recursive indexing and compression training runs. In: SPIN Workshop 1997 (1997)
27. Holzmann, G.J.: An analysis of bitstate hashing. *Formal Methods Syst. Des.* **13**(3), 289–307 (1998)
28. Kalajdzic, K., Bartocci, E., Smolka, S.A., Stoller, S.D., Grosu, R.: Runtime verification with particle filtering. In: Legay, A., Bensalem, S. (eds.) RV 2013. LNCS, vol. 8174, pp. 149–166. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40787-1_9
29. Koymans, R.: Specifying real-time properties with metric temporal logic. *Real Time Syst.* **2**(4), 255–299 (1990)
30. Laarman, A., van de Pol, J., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22306-8_4
31. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
32. Lemire, D., Kaser, O.: Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptogr. Eng.* **6**(3), 171–185 (2015). <https://doi.org/10.1007/s13389-015-0110-5>

33. Mamouras, K., Raghathanan, M., Alur, R., Ives, Z.G., Khanna, S.: StreamQRE: modular specification and efficient evaluation of quantitative queries over streaming data. In: PLDI 2017, pp. 693–708. ACM (2017)
34. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis, 2nd edn. Cambridge University Press, Cambridge (2017)
35. Naor, M., Yegorov, E.: Tight bounds for sliding bloom filters. *Algorithmica* **73**(4), 652–672 (2015)
36. Ni, J., Li, J., McAuley, J.J.: Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In: EMNLP 2019, pp. 188–197. Association for Computational Linguistics (2019). Dataset: <https://nijianmo.github.io/amazon/index.html>
37. de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S.: Efficient formal verification for the Linux kernel. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 315–332. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_17
38. Orgun, M.A., Wadge, W.W.: A relational algebra as a query language for temporal DATALOG. In: Tjoa, A., Ramos, I. (eds.) DEXA 1992, pp. 276–281. Springer, Vienna (1992). https://doi.org/10.1007/978-3-7091-7557-6_48
39. Pagh, A., Pagh, R., Rao, S.S.: An optimal Bloom filter replacement. In: SODA 2005, pp. 823–829. SIAM (2005)
40. Pagh, R., Segev, G., Wieder, U.: How to approximate a set without knowing its size in advance. In: FOCS 2013, pp. 80–89. IEEE Computer Society (2013)
41. Roşu, G., Chen, F.: Semantics and algorithms for parametric monitoring. *Log. Methods Comput. Sci.* **8**(1) (2012)
42. Schneider, J.: Randomized first-order monitoring with hashing (extended report) (2022). <https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/publications/pub2022/rv22-extended.pdf>
43. Schumann, J., Rozier, K.Y., Reinbacher, T., Mengshoel, O.J., Mbaya, T., Ippolito, C.: Towards real-time, on-board, hardware-supported sensor and software health management for unmanned aerial systems. *Int. J. Progn. Health Manag.* **6**(1), 1–27 (2015)
44. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_16
45. Stinson, D.R.: Universal hashing and authentication codes. *Des. Codes Cryptogr.* **4**(4), 369–380 (1994)
46. Tuzhilin, A., Clifford, J.: A temporal relational algebra as basis for temporal relational completeness. In: VLDB Conference 1990, pp. 13–23. Morgan Kaufmann (1990)
47. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: STOC 1982, pp. 137–146. ACM (1982)
48. Wang, Z., Luo, T., Xu, G., Wang, X.: The application of cartesian-join of Bloom filters to supporting membership query of multidimensional data. In: 2014 IEEE International Congress on Big Data, pp. 288–295. IEEE Computer Society (2014)
49. Wegman, M.N., Carter, L.: New hash functions and their use in authentication and set equality. *J. Comput. Syst. Sci.* **22**(3), 265–279 (1981)
50. Wilson, E.B.: Probable inference, the law of succession, and statistical inference. *J. Am. Stat. Assoc.* **22**(158), 209–212 (1927)