ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verified Incremental Evaluation of Aggregation Operators in Metric First-Order Temporal Logic

Bachelor Thesis

Emanuele Marsicano

September 12, 2021

Supervisor: Prof. Dr. David Basin
Advisors: Martin Raszyk, Joshua Schneider

Department of Computer Science, ETH Zürich

Abstract

Runtime verification is the act of observing a running system in order to make sure that it satisfies certain properties. Typically this is done using specialized tools, called monitors. A monitor takes as input a property we want to verify, expressed in some policy specification language, and a log file describing a run of the system we are monitoring. It then checks if the given property is always satisfied, and if not, outputs its violations.

A major challenge of monitoring tools is constructing correct monitors, with one solution being formally verifying, via proof-assistants, their behaviour. VeriMon is one such formally verified monitor which supports properties formulated in metric first-order temporal logic extended with aggregations. Although VeriMon monitors properties containing aggregations, the way they are computed is highly inefficient. For instance, when repeatedly computing them over incrementally updated sets where most of the underlying elements remain the same and only some elements are added or removed, they are computed from scratch every time leading to a lot of redundant calculations.

In this thesis, we first provide a formally verified algorithm for efficiently maintaining aggregations over incrementally updated sets. We then integrate it into the existing monitoring algorithm in order to efficiently compute aggregations over a temporal operator, and formally prove that the output of our new optimized algorithm is equivalent to the previous unoptimized one. Moreover, we provide some additional optimizations to the existing algorithm for evaluating the Until temporal operator, in order to efficiently integrate our optimized aggregation algorithm into it.

Finally, we empirically evaluate our new algorithm and compare it to the previous version of VeriMon and the unverified MonPoly. Our improved algorithm always outperforms the previous version, and either is on par or even outperforms its unverified counterpart, MonPoly.

Contents

Chapter 1

Introduction

We consider the monitoring problem: we are given a collection of timestamped events and some property over these events, expressed in some policy specification language, and we want to check if the property is always satisfied, and if not, output a list of all violations. Two aspects are key for any monitoring algorithm: correctness and efficiency.

Various policy specification languages for expressing properties have been proposed, such as first-order logic or linear temporal logic; and several tools have been designed for monitoring properties expressed in them. One of these is the MonPoly [3] monitoring tool, an efficient OCaml implementation for monitoring properties expressed in metric first-order temporal logic (MFOTL) [2], which is standard first-order logic enhanced by temporal operators to additionally express time-based constraints between events. MonPoly additionally supports aggregation operators that aggregate over sets of data values, for instance by computing their average or finding their minimum/maximum. One major shortcoming of MonPoly is the lack of reliable evidence for its correctness: although extensive testing can be used to give us some trust that the program behaves as expected, it is not feasible to account for every possible input, particularly with the expressiveness provided by MFOTL.

One approach to increase the trustworthiness is formally verifying the correctness of the algorithms in a proof assistant, such as Coq [13] or Isabelle/HOL [11]: using these tools we can directly provide statements about the semantics of our program, and then provide proofs that our implementation indeed satisfies those statements. Afterwards, using their code generator, we can obtain executable code in some target language that satisfies the properties formulated and proved in the proof assistant. In this way the amount of code that we need to trust to be correct is significantly reduced: as long as the core parts of the proof assistant and the compiler for the code generator's target language behave correctly, we are guaranteed that our exported code

behaves exactly as our proved statements describe.

The VeriMon [12] monitoring tool is such a formally verified implementation of the core parts of MonPoly in Isabelle/HOL. It supports an enhanced version of MFOTL, called metric first-order dynamic logic (MFODL) [1], which additionally adds support for regular expressions. It provides efficient algorithms for monitoring temporal operators, but not for aggregations over them. Aggregations over temporal operators allow us to aggregate over all events within a specific time-frame, and if consecutive time-frames overlap, so do the corresponding sets we aggregate over. When VeriMon computes the aggregation results, instead of storing the results obtained previously and updating them according to the elements that were added/removed, the aggregations are computed from scratch every time. If the underlying set is large, this becomes highly inefficient.

In the unverified MonPoly monitoring tool, aggregations have already been optimized when the updates on the underlying set we are aggregating over satisfy some specific properties: specifically, the elements have to be removed from the set in the same order as they were added, like the elements of a FIFO queue. When considering aggregations over temporal subformulas, these restrictions are satisfied only for the Once and Eventually temporal operator, and MonPoly implements the optimization only for the Once operator.

In this work we implement and verify MonPoly's optimizations described in the previous paragraph in VeriMon, and additionally extend the scope of optimizations by supporting arbitrary updates to the underlying sets we are aggregating over, without any restrictions on the order in which elements are added or removed. Our improved algorithm allows us to update the results of the aggregation in time linear in the number of elements that have been added or removed [1]. We then show that evaluating the Since and Until temporal operators give rise to incrementally updated sets, and therefore computing aggregations over them can be efficiently done using our new aggregation algorithm. After integrating these optimizations into the algorithms for evaluating Since and Until, we have provided a proved to be correct monitoring tool that supports a highly expressive language (MFOTL with aggregations), which even outperform the state-of-the-art tool MonPoly in some cases.

We first give a brief overview of MFOTL (Section 1.1) and the current implementation of VeriMon (Section 1.2), in particular the parts related to temporal operators and aggregations, and an overview of related work (Section 1.3). We then describe the new data structure and its operations used to efficiently maintain the result of the aggregation over an incrementally updated set

[1] The minimum, maximum and median aggregations have an additional logarithmic factor in the set size.

datatype event_data = EInt *integer* | EFloat *double* | EString *string8*
type_synonym db = (string × event_data list) set
type_synonym ts = nat
typedef trace = $\{s ::$ (db × ts) stream. trace_props s$\}$

Figure 1.1: VeriMon's model of traces

(Chapter 2), and how to update the existing algorithms for temporal operators in order to make use of this new data structure (Chapter 3). Finally, we evaluate our new implementation and compare its running time with the old version of VeriMon and with the unverified MonPoly monitoring tool (Chapter 4).

Every part of the improved algorithm has been formally proven correct in Isabelle/HOL. The whole formalization is available at [8].

1.1 Metric first-order temporal logic

The policy specification language used by VeriMon is based on MFODL, and expresses properties over infinite sequences of time-stamped collections of events. These sequences are called traces. Figure 1.1 shows how these infinite traces are modeled in VeriMon. A single event is defined by its name (a string), and a collection of parameters for that event, taken from the domain D of valid values (supported by VeriMon are integers, floats, and strings, as shown by the event_data datatype in the figure). A database is a collection of events that all occur at the same timestamp. A stream is a polymorphic infinite sequence, which we use in order to model traces as a stream of databases paired together with their corresponding timestamp.

Additionally, we require some specific properties on the timestamps of a trace, abbreviated in the figure as trace_props for simplicity. Let σ denote a trace. We denote the timestamp of the i-th database as $\tau\,\sigma\,i$. In order for σ to be a trace, we require two properties on the timestamps: the timestamp sequence has to be monotone ($\forall i.\ \tau\,\sigma\,i \leq \tau\,\sigma\,(i+1)$) and unbounded ($\forall t.\ \exists i.\ t < \tau\,\sigma\,i$). Note that the monitoring algorithm does not work directly on these infinite traces but rather on a finite prefix of them, for which we only require the monotone property to hold. Different databases can have the same timestamp: in particular, we distinguish between the time-point of a database (which is simply its position i in the trace σ), and its associated timestamp $\tau\,\sigma\,i$. Figure 1.2 shows an example of a prefix, encoded using MonPoly's syntax. We have two events, Request and Accept, each of which takes a single integer as a parameter. The first Accept event occurs at the same timestamp (1) as the first two Request events, but at a later time-point. For the purpose of temporal operators, an event occurring at the same timestamp but a later time-point

```
@1 Request(132) Request(24)
@1 Accept(24)
@4 Accept(132)
```

Figure 1.2: An example prefix of a trace in VeriMon syntax

is considered to have happened later (the Accept event happens after the Request event even though the timestamp did not increase).

Figure 1.3 shows the VeriMon implementation of terms and formulas. A term is either a constant of the types supported by VeriMon described before, a variable, or a combination of multiple terms using some operation, such as addition or subtraction. Note that variables are represented by De Bruijn indices, and thus take natural numbers in their constructor: the variable represented by the number i is the variable bound by the i-th quantifier, starting from the innermost one. For instance, consider the formula

$$\exists y.\ P(y)\ \wedge\ (\exists z.\ P(x) \wedge P(y,\ z))$$

Using VeriMon's representation, this formula becomes:

$$\text{Exists } P(\text{V } 0)\ \wedge\ (\text{Exists } P(\text{V } 2) \wedge P(\text{V } 1,\ \text{V } 0))$$

Note that the variable with number 0 always refers to the innermost bound variable, so that the first V 0 refers to the variable y, while the second one to the variable z. Free variables, such as x in the above formula, are treated as as being bound outside of the formula, in an arbitrary order decided when parsing the formula.

Formulas are constructed recursively from other formulas and from terms. MFOTL supports the operations from first-order logic such as the unary and binary logical operations, existential and universal [2] quantification, and binary predicates on terms such as order relations. The semantics of MFOTL is defined by the sat function (Fig. 1.3): it takes an infinite trace of events σ as described earlier, a list v defining the assignments of free variables appearing in the formula to values in the domain D, an integer i representing the time-point of the trace at which we want to evaluate the formula, the actual formula φ we are monitoring, and checks the satisfaction of the formula. MFOTL also supports the Previous, Next, Since and Until temporal operators, and the aggregation operator. Of particular interest to us are the aggregation operator, which we will come back to in section 1.2, and the Since and Until temporal operators, shown in Figure 1.3.

When monitoring temporal operators, we might not always want the temporal operator to extend over the whole log, but limit it to only a certain range of

[2] Not directly, but via its equivalent formulation using the existential quantifier

datatype trm = V *nat* | C *event_data* | trm + trm | trm - trm | ...
typedef I = $\{(a :: nat, b :: enat).\ a \leq b\}$
datatype formula = ... | formula S_I formula | formula U_I formula

fun sat :: *trace* $\Rightarrow$ *event_data list* $\Rightarrow$ *nat* $\Rightarrow$ *frm* $\Rightarrow$ *bool* where
  sat σ v i (α S_I β) = ($\exists j \leq i.\ T\,\sigma\,i - T\,\sigma\,j \in I \wedge$ sat σ v j $\beta \wedge$
    ($\forall k \in \{j <..\ i\}$. sat σ v k α)) |
  sat σ v i (α U_I β) = ($\exists j \geq i.\ T\,\sigma\,j - T\,\sigma\,i \in I \wedge$ sat σ v j $\beta \wedge$
    ($\forall k \in i\ ..<j$. sat σ v k α)) |
  ...

type_synonym tuple = *event_data option list*
type_synonym table = *tuple set*

Figure 1.3: VeriMon implementation of formulas

timestamps. To this end, we additionally provide an interval over which the temporal operators are evaluated: the intervals are represented by a pair of a natural number and an extended natural number which can additionally be equal to ∞, denoting the left and right boundaries of the interval respectively. In the actual monitoring algorithm, we only allow the upper bound of the interval to be unbounded for past temporal operators (Since), but not for future ones (Until), as evaluating unbounded future operators cannot always be done in finite time.

The formula $\varphi = \alpha\, S_I\, \beta$ is satisfied at time-point i if there is a previous time-point j such that the difference between the timestamp at i and the one at j is inside the interval I, the subformula β is satisfied at time-point i, and the subformula α holds for all time-points after j up to i. The formula $\varphi = \alpha\, U_I\, \beta$ is satisfied at time-point i if there is a future time-point j such that the difference between the timestamp at j and the one at i is inside the interval I, the subformula α holds for all time-points between i up to but excluding j, and the subformula β is satisfied at time-point j. Additionally, VeriMon supports standard linear-temporal logic abbreviations such as Once ($\blacklozenge_I\, \beta = \mathrm{True}\, S_I\, \beta$) and Eventually ($\Diamond_I\, \beta = \mathrm{True}\, U_I\, \beta$).

Consider for instance the formula $P(x,y) U_{[3,5]} Q(x,y)$, monitored on the prefix shown below:

$$\begin{aligned} &@1 \quad P(1,2)\, P(2,3) \\ &@2 \quad P(1,2) \\ &@4 \quad P(1,2)\, Q(1,2)\, Q(2,3) \end{aligned}$$

At time-point 0, the given formula holds for the assignments $x = 1$ and $y = 2$: there is a later point (namely, time-point 2) where $Q(1,2)$ holds, $P(1,2)$ holds

at time-points 0 and 1, and the difference between the timestamps at time-point 2 and 0 is inside the interval ($2 - 0 = 3 \in [3, 5]$). On the other hand, at time-point 2 with the same assignment the formula does not hold: while all other conditions are still satisfied, the timestamp difference i not in the interval anymore ($4 - 2 = 2 \notin [3, 5]$). Finally, for the assignments $x = 2$ and $y = 3$ at time-point 1 the formula does not hold either, as we are missing the event $P(2, 3)$ at time-point 2.

Note that in the list v we only need to provide assignments for variables that are free in the formula φ. When running the actual monitoring algorithm, as the set of free variables could possibly change when recursively evaluating subformulas, we represent the assignment to each variable as an option datatype instead of storing a list with assignments for only the free variables. The assignments for the variables can be set to either None, indicating that the variable does not appear free in φ, or to Some k, indicating that it appears free and the value assigned to the variable is k.

A list of optional variable assignments t as just described is called a tuple, and a set of such tuples is called a table. We say that the tuple t is a satisfaction for φ at time-point i if the formula φ, with the free variable assignments induced by t, is satisfied at time-point i, that is, if sat σ (map the t) i ω holds, where the "the" function returns the value of the option, if it is set.

The VeriMon monitoring algorithm takes as inputs a formula φ and a prefix of a trace σ, and outputs for each time-point i of the prefix that it was able to evaluate (as future temporal operators cannot be immediately evaluated), the finite table containing all satisfactions of φ at i, if any. Note that in the original formulation of the monitoring problem we were interested in violations of a formula φ, not its satisfactions. This is easily fixed by passing as inputs to our monitoring algorithm the negated version of the formula we want to monitor $\varphi' = \neg\varphi$, or a formula that is logically equivalent to it. Finally, note that some formulas or their subformulas result in an infinite set of satisfactions, which are not supported by the VeriMon algorithm. Thus, VeriMon can only monitor a specific subset of formulas, called "safe" formulas, which satisfy certain syntactic conditions that guarantee that their evaluation at any step always returns a finite set of satisfactions. Upon receiving a formula as an input, VeriMon first checks if it is a safe formula, and only if it is continues with the actual monitoring algorithm.

1.2 Aggregation operator

We now describe the aggregation operator and its semantics and implementation in MFOTL. The generalized aggregation operator has the form $\varphi = y \leftarrow \omega\, t; b.\ \varphi'$. Here y denotes the variable we are storing the aggregation result in, ω denotes the type and default value of the aggregation we are

```
fun sat :: trace ⇒ data list ⇒ nat ⇒ frm ⇒ bool where
  sat σ v i (y ← ω t; b. φ) =
    let M = {(x, ecard Z) | x Z.
    Z = {z. length z = b ∧ sat σ (z @ v) i φ ∧ etrm (z @ v) t = x} ∧ Z ≠ {}}
    in (M = {} → fvφ ⊆ {0..<b}) ∧ v ! y = eval_agg_op ω M)

  ...
```

Figure 1.4: Semantics of aggregation operator

computing, t denotes the aggregation term, b is the number of variables that are bound by the aggregation operator, and finally φ' is the subformula we are aggregating over. Since we are using De Bruijn indices, b and y are simply integers. Supported by VeriMon are the sum, count, average, minimum, maximum and median aggregation operators.

Figure 1.4 shows the semantics of the aggregation operator, as defined in VeriMon. The free variables of φ' which are not part of the b bound ones determine the group of satisfactions we are computing aggregations over, while v ! y contains the aggregation result.

Intuitively, the multiset M contains all possible values the term t can evaluate to under the satisfactions of φ' that belong to the group determined by v, along with how many times they appear. Specifically, the pair (x, k) is inside M if and only if there are $k \neq 0$ different possible assignments of the b bound variables, such that the assignment v' obtained by concatenating such an assignment z with v satisfies the formula φ', and evaluating the term t under v' returns x (handled by the etrm function, which takes a list of variable assignments v and a term t and evaluates t under v). The second element k is an extended natural number as the number of such assignments could possibly be infinite. In such a case, the result of the aggregation equals a default value. The formula is then satisfied if the evaluation of the aggregation operator on this multiset agrees with the value assigned to the variable stored at the index y.

The evaluation and semantics of the aggregation operator is defined by a function eval_agg_op (here omitted), that transforms the multiset into a list by replicating each element according to the number of times it appears, and then folds the list according to the aggregation operator we are computing (for instance, for the sum aggregation we add all elements of the list, for minimum we take the minimum of all elements in the list, etc.). In the case that the computed multiset M is empty, we use the convention that the formula is not satisfied unless all free variables of the subformula φ' are part of the b bound ones, in order to avoid an infinite number of satisfactions of the aggregation formula.

```
definition eval_agg :: nat ⇒ bool ⇒ nat ⇒ Formula.agg_op ⇒ nat ⇒
Formula.trm ⇒ event_data table ⇒ event_data table where
  eval_agg n g0 y ω b f rel =
    (if g0 ∧ rel = {}
    then singleton_table n y (eval_agg_op ω {})
    else (λk. let group = filter (λx. drop b x = k) rel;
      M = (λy. (y, ecard (filter (λx. meval_trm f x = y) group))) ‘
        meval_trm f ‘ group
      in k[y:=Some (eval_agg_op ω M)]) ‘ (drop b) ‘ rel)
```

Figure 1.5: Current VeriMon implementation of the aggregation operator

In the following and later examples of aggregation operators, since for clarity we use named variables instead of De Bruijn indices, we follow MonPoly's syntax and we replace b by the names of the variables which are not bound, i.e. the names of the variables used for grouping. Let us look at an example evaluation of the sat function for the formula $\omega = z \leftarrow \mathrm{Sum}\, x \cdot x; y.\ P(x, y)$ on the following database:

$$@1 \quad P(1,2)\, P(3,2)\, P(-1,2)\, P(1,1)$$

We want to check satisfaction for the valuation v where $v(z) = 11$ and $v(y) = 2$ (we denote by $v(k)$ the value of variable k under the valuation v, note that the value of $v(x)$ is irrelevant as x is bound by the aggregation operator).

The free variables of φ' (which in this case is the formula $P(x, y)$) are x and y, where x is bound by the aggregation operator and thus the group we are aggregating over is determined by the variable y, specifically we are aggregating over the group $v(y) = 2$. The satisfactions (x, y) of the subformula φ' (we represent a satisfaction v here as a pair $(v(x), v(y))$) for which $v(y) = 2$ are $(1, 2)$, $(3, 2)$ and $(-1, 2)$. We thus have three possible ways of extending our original satisfaction v to a valid satisfaction v' for φ', by setting $v'(x)$ to 1, 3 and 4 respectively. The evaluation of the term t under these 3 satisfactions is 1, 9 and 1 respectively, giving us the following multiset $M = \{(1, 2),\ (9, 1)\}$ Finally, the eval_agg_op function transforms this multiset M into the list $[1, 1, 9]$, computes the sum of all three elements and returns 11, which is indeed the value of $v(z)$. It follows that v is a valid satisfaction for φ. The only other valid satisfaction for the formula φ is the one with $v(y) = 1$ and $v(z) = 1$, as every other value of $v(y)$ leads to an empty multiset M.

We now look at how the current version of VeriMon computes the aggregation operator, shown in Figure 1.5. The computation of the satisfactions of the aggregation operator is handled by the eval_agg function. Its last argument rel is a table of satisfactions of the subformula we are aggregating over. The argument $g0$ is a Boolean flag which is set if all the free variables of the

subformula are bound by the aggregation operator, and the argument ω stores the type of aggregation we are computing, together with a default value for aggregations over empty or infinite sets. The remaining arguments are exactly the same as described earlier.

We first check if we are in the special case of no free variables and an empty set of satisfactions in the subformula we aggregate over, and if so, return a default value (the value singleton_table n y k is a tuple of length n where all options are set to None except for the y-th element, which is set to k). Otherwise we proceed in a similar manner as when evaluating the sat function, the main difference being that while in the sat function we only had to compute the aggregation over one specific group, decided by the variable assignment v, here we need to compute the aggregation over all non-empty groups at the same time: we start with all the satisfactions of the subformula we are aggregating over and determine all possible groups those satisfactions belong to. Afterwards, for each group we compute the multiset M corresponding to it in the same way as was done by the sat function (here the meval_trm function behaves exactly as the etrm function, but taking a list of options of data values, instead of a list of data values), and then evaluate the aggregation operator over that multiset. Finally, we update the value of the variable y to the computed aggregation value, and return a table with all the groups and the corresponding aggregation results so obtained.

A major problem with the current approach is that at every time-point we are iterating through the entire table rel from scratch even if most of the elements remain the same as for the previous time-point. In the next chapter we explain how we can take advantage of how satisfactions of temporal operators are updated in order to more efficiently calculate aggregations over them.

1.3 Related work

The formalization effort uses Isabelle [10], a proof assistant based on higher-order logic [11]. The original formalization [12] of the MonPoly [3] monitoring tool supported MFOTL [2], but did not provide support for aggregations, which was added in later [1]. We build upon this latest version.

Another formally verified monitor for metric temporal logic is described in [4]. It provides aggregations with a lesser scope than VeriMon does, but it does implement them using an optimized queue-based sliding window algorithm.

Algorithms for efficiently computing aggregations are well researched as they have many important applications e.g. in databases and stream processing, and multiple specialized algorithms exist depending on the properties of the operation we use to aggregate. Our algorithm provides a formally

verified implementation of the subtract on evict and the order statistics tree approaches described in [7].

The order statistic trees used in our efficient implementation of the minimum, maximum and median operations are based on the Isabelle formalization [9] of the weight-balanced trees described by Hirai and Yamamoto [6].

Chapter 2

The optimized aggregation algorithm

In the current implementation of VeriMon, the aggregation results are computed by iterating over all of the satisfactions of the subformula we aggregate over each time we obtain a table of satisfactions for the subformula at a new time-point, without reusing the results computed for previous time-points. In the case of temporal operators, we can reuse the aggregation results from previous time-points by updating them according to the newly added/removed satisfactions. In this section, we describe the general data structure used to implement the optimized algorithm to compute aggregations efficiently, and in the next section we show how to actually integrate it into the previous algorithms and their corresponding correctness proofs for evaluating Since and Until formulas.

2.1 The aggaux data structure

Our data structure needs to support the following operations:

1. Add a new set of satisfactions of the subformula we aggregate over
2. Remove a set of satisfactions of the subformula we aggregate over
3. Output the results of the aggregation over all current satisfactions of the subformula

The definition of aggaux and the signature of its main operations are shown in Figure 2.1.
The actual structure of aggaux depends on what type of aggregation we are computing, but it always follows the same structure: it consists of a mapping from groups (with one entry for each group in our current set of satisfactions) to some auxiliary data structure from which we can efficiently (in constant or logarithmic time) compute the aggregation value for that group, and which we can efficiently update after recieving a new element to be added or removed.

datatype agg_type = Agg_Cnt | Agg_Min | Agg_Max |
Agg_Sum | Agg_Avg | Agg_Med
type_synonym agg_op = *agg_type* × *event_data*

record aggargs =
aggargs_cols :: *nat set*
aggargs_n :: *nat*
aggargs_g0 :: *bool*
aggargs_y :: *nat*
aggargs_ω :: *Formula.agg_op*
aggargs_b :: *nat*
aggargs_f :: *Formula.trm*

datatype list_aux = LInt *integer treelist* | LString *string8 treelist*

type_synonym *'a agg_map = (event_data tuple, 'a) mapping*

datatype *aggaux* = CntAux *nat agg_map*
| SumAux *(nat* × *integer) agg_map*
| RankAux *(list_aux agg_map* × *type)*

fun valid_maggaux :: *aggargs* ⇒ *aggaux* ⇒ *event_data table* ⇒ *bool*
fun init_maggaux :: *aggargs* ⇒ *(bool* × *aggaux)*
fun insert_maggaux :: *aggargs* ⇒ *event_data table* ⇒ *aggaux* ⇒ *bool* × *aggaux*
fun delete_maggaux :: *aggargs* ⇒ *event_data table* ⇒ *aggaux* ⇒ *bool* × *aggaux*
fun result_maggaux :: *aggargs* ⇒ *aggaux* ⇒ *event_data table*

Figure 2.1: The aggaux data structure and its operations

The details of an aggregation operator are stored inside an aggargs record: all of the fields are the same as described in the previous chapter, with the addition of the aggargs_cols field which stores the indices of all the variables that are bound by the aggregation operator. This record is then passed as the first argument in all the functions which update the aggaux data structure, instead of storing it directly in the data structure.

The function valid_maggaux takes an aggargs record containing the aggregation paramaters, an aggaux data structure, and a set of satisfactions for the subformula we aggregate over and states the invariants relating our aggaux data structure to the satisfactions set that need to hold in order for it to be valid. In our correctness proofs we make sure that the aggaux data structure returned by init_maggaux together with an empty set satisfies those invariants, that the invariants are maintained by the insert_maggaux and delete_maggaux operations, and that the results returned by the result_mag-

gaux applied to an aggaux data structure valid for a specific set are the same as the results returned by the naive way of computing the aggregations over that set described in the previous chapter. The init_maggaux function takes the aggregation parameters and returns an aggaux data structure of the correct type initialized with an empty mapping, as at the beginning we have no satisfactions yet and thus no groups either. The insert_maggaux, delete_maggaux and result_maggaux functions handle the insertion/removal of satisfactions and the output of the aggregation results, but their behaviour depends on the type of aggaux, as follows:

CntAux: This type handles the count aggregation, in this case the mapping is from groups to an integer containing the number of elements in that group. The insert_maggaux function iterates over all elements that have to be added, computes the group they belong to and increments the value corresponding to that group in the mapping. The delete_maggaux function does the same but decrements instead, deleting the entry for that group entirely if the counter reaches 0, and finally the result_maggaux function iterates over all the key/value pairs in the mapping, updates the key by setting the value of the result variable to the result of the aggregation (in this case the value corresponding to that key in the mapping), and returns the set of tuples thus obtained.

SumAux: This type handles the sum and average aggregations over integers (bundled together since for computing averages you need to compute the sum as well). In this case the mapping is from groups to a pair of integers, the first one containing the number of elements in the group, like in the CntAux case, the second one containing the sum of elements in that group. The insert_maggaux function iterates over all elements that have to be added, computes the group they belong to, increments the count for that group and adds to the sum the value of the integer obtained by evaluating the term we are aggregating over according to the variable assignments defined by the element we are adding. The delete_maggaux function does the same but decrements the count and subtracts from the sum, deleting the entry for that group entirely if the count reaches 0, and finally the result_maggaux function behaves as in the CntAux case, setting the value of the result variable to the sum if we are computing the sum aggregation, or to the sum divided by the count if we are computing the average aggregation.

RankAux: This type handles the minimum, maximum, and median aggregations. In this case the mapping is from groups to lists, where the list simply contains all elements belonging to that group. The insert_maggaux function iterates over all elements that have to be added, computes the group they belong to and adds the evaluated term, according to the variable assignments from the element we are adding, to the corre-

sponding list. The delete_maggaux function does the same but removes the element instead, deleting the entry for that group entirely if the list becomes empty, and finally the result_maggaux function behaves as in the previous cases, obtaining the result of the aggregation by sorting the list stored in the mapping and returning the first, last, or average of the middle two elements for the minimum, maximum and median aggregation, respectively. Note that using lists is just as slow as the naive way of computing the aggregations. The lists will later be refined to a more efficient representation (balanced binary search trees), as explained in Section 2.4.

2.2 Handling invalid data

The invariants our aggaux data structure has to satisfy imply some constraints that are not necessarily satisfied by the existing monitoring algorithm, and thus we cannot guarantee in our proofs that every sequence of possible inputs we receive will result in a valid aggaux data structure that we can use to obtain aggregation results efficiently. To handle this, the init_maggaux, insert_maggaux and delete_maggaux functions return an additional Boolean flag indicating if the aggaux data structure that was returned satisfies the invariants defined in the valid_maggaux function. If the boolean flag is set to false it means the input it received was invalid and we should revert to compute the aggregations in the naive way. The following cases are invalid and cause the boolean flag to be set to false:

Infinite sets: The semantics of MFOTL with aggregations does not force the set of satisfactions to be finite. As a consequence, our aggaux data structure could receive an infinite set of elements to be added, which is not allowed by the invariants. Any time a new set of elements to be added is received, we thus first have to check if that set is finite, and if it is not, we set the Boolean flag to false. This case can never happen in the exported code as it would abort immediately. As such, this flag will never actually be set at runtime, but is still needed for our correctness proofs. While we could prove that all sets we add are always finite for the safe formulas that VeriMon is restricted to, which would allow us to drop the flag in this case, this has not yet been done in the current implementation.

Invalid types: As the order in which elements are processed in the aggaux data structure may differ from the naive way of computing the aggregations, we need some additional type restrictions on the elements inserted/removed to ensure that the results are the same: specifically, for the sum aggregation we require that the term we are aggregating over always evaluates to an integer (as the sum operation on floats is

not associative, and the sum operations on strings returns a undefined value), and for the RankAux aggregations we do not allow floats (since the comparison operations on floats do not form a linear order, our data structure cannot efficiently return the result defined by the semantics of the monitoring algorithm, as it depends on the order the elements are processed in). If we try to insert/remove an element of the invalid type, the flag is again set to false. Additionally, for the RankAux case, even though we allow both integers and strings, in the current implementation we do not allow mixing of the two. That is why we add the additional *type* flag indicating if all the elements are supposed to be strings or integers. If we try to add a string when the *type* flag is set to integer or inserting an integer when the flag is set to strings, we again set the valid flag to false.

Note that trying to delete an element that was not previously inserted does not trigger the invalid flag, as we can actually prove that every element that is removed was inserted previously from the monitoring algorithm.

2.3 A simple example

Let us look at a small example of our aggaux data structure applied to the formula $\omega = z \leftarrow (\text{Avg}, 0.0)\, x; x.\, \varphi$, where φ is a formula with free variables x and y (Note that while the aggregation result as well as the default value is a float, we require all terms we aggregate over to be integers, as described earlier).

Figure 2.2 shows some function calls for the above formula and the computed aggaux. The mapping is represented as a set of (key, value) pairs. For brevity, we omit some of the arguments from the function calls (the aggregation parameters and the aggaux data structure obtained from the previous function call). We first initialize our aggaux data structure using the aggargs record corresponding to the above aggregation. Since the aggregation type is average, we use the SumAux constructor. We then insert three new satisfactions of the form (x, y) of the formula φ. As we are aggregating over the y variable, the first two tuples will update the group corresponding to $y = 2$, making the value corresponding to that group be $(2, 7)$, since we have two entries and their corresponding x values sum to 7. The final tuple updates the group entry corresponding to $y = 1$.

The delete call then updates the mapping in an analogous way, deleting the entry corresponding to group $y = 1$ entirely, as the first integer (the count) has reached 0 and thus there are no more elements in that group left. Finally, when we want to obtain the result, we iterate over all (key, value) pairs in the mapping and compute the average for each one, and return the resulting (key, average) pairs.

function call	returned value
init_maggaux	SumAux $\{\}$
insert_maggaux $\{(3,2),(4,2),(2,1)\}$	SumAux $\{(2,(2,7)),(1,(1,2))\}$
delete_maggaux $\{(3,2),(2,1)\}$	SumAux $\{(2,(1,4))\}$
insert_maggaux $\{(5,2),(4,1)\}$	SumAux $\{(2,(2,9)),(1,(1,4))\}$
result_maggaux	$\{(2,4.5),(1,4.0)\}$

Figure 2.2: An example of using the aggaux data structure

2.4 Data refinement from list to trees

As mentioned in Section 2.1, the implementation of RankAux using lists is still very inefficient, as we are sorting and folding the entire list every time we need to compute the aggregations. For the exported code, we want to replace these lists by balanced binary search trees which allow us to perform the insertion, deletion and lookup operations all in logarithmic time. We denote the i-th element in a sorted list as the element of rank i (that is, the minimum element of the list is the element of rank 1, the maximum is the element of rank l, where l is the length of the list). If we simply use a standard balanced binary tree, we have logarithmic time lookup of the minimum and maximum, which is not enough, as we also need to be able to access the elements of rank $\lceil \frac{l}{2} \rceil$ and $\lfloor \frac{l}{2} + 1 \rfloor$ for the median, which we can only do in linear time.

We instead use an order statistic tree, a balanced binary tree with the additional invariant that every node also stores the size of the subtree rooted at that node. By comparing the rank of the element we need to look up and the size of the subtree rooted in the left child, we can determine if the element we need resides in the left or right subtree (or in the node we currently are inspecting), and then proceed recursively, allowing us to lookup elements of arbitrary rank in logarithmic time. One way to implement order statistic trees is by simply augmenting a traditional balanced tree data structure (for instance, red-black trees) to also hold the subtree size at each node, and change the insertion/deletion/balancing operations accordingly to maintain the invariant. This has the drawback that the bookkeeping information we need to store at each node is increased, as not only are we storing information for balancing the tree but also the subtree size. The other option is to reuse the subtree size information for balancing as well: by comparing, with an appropriate heuristic, the sizes of the left and right subtrees at a node after an insertion/deletion, we can maintain logarithmic runtime for all operations. We follow the latter approach, specifically the balancing strategy described by Hirai and Yamamoto [6]. As a starting point for our implementation we use the work by Tobias Nipkow and Stefan Dirix [9], with some adjustments to allow for insertion of multiple equal elements.

Our correctness proofs require the insert_maggaux and delete_maggaux

typedef *'a treelist* = UNIV :: *'a list set*

lift_definition insert_treelist :: *'a::linorder ⇒ 'a treelist ⇒ 'a treelist* is insort
lift_definition remove_treelist :: *'a::linorder ⇒ 'a treelist ⇒ 'a treelist* is del_list
lift_definition get_treelist :: *'a::linorder treelist ⇒ nat ⇒ 'a* is (*λ l n*. (sort *l*) ! *n*)

lift_definition Collapse :: *'a::linorder wf_wbt ⇒ 'a treelist* is inorder
code_datatype Collapse

lemma insert_treelist_code[code]:
  insert_treelist *x* (Collapse *y*) = Collapse (tree_insert *x y*)
lemma remove_treelist_code[code]:
  remove_treelist *x* (Collapse *y*) = Collapse (tree_remove *x y*)
lemma get_treelist_code[code]:
  get_treelist (Collapse *y*) *n* = tree_select *y n*

Figure 2.3: Data refinement from list to trees

functions to be commutative, which does not hold if we use binary trees directly, as insertion and deletion on the above mentioned binary trees is not necessarily commutative (because the resulting trees can have different shapes even though they contain the same elements, as different insertion order will lead to balancing at different times). We instead use a data-refinement approach as described in [5], following the outline shown in Figure 2.3.

We first define a new type which is a copy of the list type, as we do not actually want every single list occurrence to be replaced by trees, but only those used in the RankAux implementation. We then lift all required list operations to our new treelist datatype, and provide a function (called Collapse) from the type we want the lists to be refined to (trees) to our new list type, in our case this is simply the standard binary tree inorder traversal. By defining this new function as a code datatype, in the exported code the new treelist type will be represented internally by the tree datatype, while still allowing our original Isabelle code to use exclusively the treelist type, simplifying correctness proofs and allowing us to use the commutativity of list operations. Finally, we need to provide code equations that relate the previous list operations to equivalent ones on trees, so that the exported code uses the tree operations: the insertion and deletion of an element from the list will be replaced by insertion or deletion on the underlying tree, while the lookup of the element of rank i will be replaced by the corresponding tree lookup, as described earlier.

Chapter 3

Integration into the temporal operators' algorithms

The VeriMon monitoring algorithm computes, in an efficient way, satisfactions for formulas of the type $\alpha\ S\ \beta$ or $\alpha\ U\ \beta$. In the case of an aggregation over such formulas, it simply computes the aggregation result after processing each time-point by iterating over the whole set of satisfactions, in time linear in the number of satisfactions of the temporal formula. To optimize the computation of an aggregation over a temporal formula, we augment these algorithms with the aggaux data structure described in the previous section so that the aggregations results are updated incrementally the same way as the set of satisfactions is, such that after each time-point the aggregation results can be obtained in time linear in the number of groups. If the amount of groups is roughly equal to the amount of elements (that is, every group only contains 1-2 elements), the runtime of this new algorithm is quadratic just as before. If instead we only have a constant amount of large groups, the new algorithm runs in linear time. We first describe the general approach of augmenting the previously existing data structures with an auxiliary data structure (in this case an aggaux data structure), which is used for both the Since and the Until case, and then describe the actual implementation for both in more detail.

3.1 Augmenting existing data structures and algorithms

As the behaviour of the previously existing temporal operators algorithm does not change and we simply need to build on top of it, we first describe the general approach used to augment the existing data structure in order to reuse previous correctness proofs.

We first describe the notation used in the Isabelle formalization: the abstract interfaces which describe the behaviour of the Since and Until algorithms are

type_synonym maux = ...

fun valid_maux :: ... ⇒ *maux* ⇒ *bool*
fun update_maux :: ... ⇒ *maux* ⇒ *maux*
fun result_maux :: ... ⇒ *maux* ⇒ *event_data table*

Figure 3.1: Generalized mmaux instance

called msaux and muaux respectively. The instances of those interfaces are called mmsaux and mmuaux, while our enhanced data structures which support efficient aggregation computations are called mmasaux and mmauaux. Function names follow the same pattern.

We denote by mmaux a placeholder for either the mmsaux or mmuaux instances used to compute the satisfactions for temporal formulas. It is shown together with generalized validity and update functions in Figure 3.1. (In the actual formalization, the update functions would be replaced by the update functions specific to the respective temporal algorithm.) The type mmaux holds all necessary state in order to compute satisfactions for temporal formulas in an efficient manner, the valid_mmaux predicate states the invariants needed in order for the state to be valid, the update_mmaux function updates the state when processing the satisfactions at a new time-point, and the result_mmaux function returns the set of satisfactions of the formula we are monitoring using the provided mmaux state.

We augment the type mmaux to $mmaux' = (mmaux \times new\_state)$ which contains both a copy of the existing state, and some additional state new_state which provides additional functionality to $mmaux$. The updated valid_maux' function reuses the valid_maux function to ensure the invariants on the old state are still kept, states the invariants the new state has to satisfy, and finally states the invariants which relate the additional state to the existing one. The update_maux' function updates the existing state in exactly the same way as the update_maux function, and additionally updates the new state in the appropriate way in order to maintain the valid_maux' invariants. Since the old state is updated in the same way as was previously done, we can reuse the old correctness proofs and only need to prove that the new state is updated in a correct way. Finally, the new result_maux' function should output the same as the previous version, potentially in a faster way using the additional state.

In our case, we want to add the aggaux data structure as the new state. As not every temporal formula we monitor will include aggregations and in order to handle cases where the assumptions needed by the aggaux data structure are not met, we augment mmaux with an aggaux option, so that the updated data structure becomes $mmaaux = (mmaux \times aggaux\ option)$. If there is no aggregation or the input or an update on the aggaux data structure

type_synonym mmasaux = (mmsaux, aggaux option)

fun valid_mmasaux :: *args* ⇒ ... ⇒ *mmasaux* ⇒ *bool*
fun add_new_table_mmasaux :: ... ⇒ *mmasaux* ⇒ *mmasaux*
fun join_mmasaux :: ... ⇒ *mmasaux* ⇒ *mmasaux*
fun shift_end_mmasaux :: ... ⇒ *mmasaux* ⇒ *mmasaux*
fun add_new_ts_mmasaux :: ... ⇒ *mmasaux* ⇒ *mmasaux*
fun result_mmasaux :: *args* ⇒ *mmasaux* ⇒ *event_data table*

Figure 3.2: Optimized mmasaux instance

returns the invalid flag, the *aggaux option* is set to None, and from there on the augmented data structure behaves the exact same way as the existing one.

In the result_maaux function, if we have an aggregation and the *aggaux option* is not set to None, we can use the additional agguax state to compute the aggregations in an efficient manner. Otherwise, we obtain the satisfactions of the temporal subformula that were produced by the old algorithm using the old state, and then aggregate over these using the old method, described in Section 1.2.

3.2 Integration into Since

We now describe how the new aggaux state is updated in the case of the Since temporal operator. Figure 3.2 shows a simplified view of the optimized Since aggregation instance and its main operations. We omit the actual details of how the algorithm to compute satisfactions of the Since temporal operator works, as a detailed description of it can be found in [1].

The mmsaux data structure holds all the necessary state to compute satisfactions of the Since temporal operator in an efficient manner. The only part of the state that is relevant to the optimized aggregations algorithm is the tuple_in mapping, whose keys are exactly the satisfactions of the Since temporal operator. The remaining parts of the mmsaux state, here omitted, contain helper information to efficiently update the tuple_in mapping. The new mmasaux data structure additionally holds an aggaux option in its state, which is set to Some in the case we are computing an aggregation, to allow us to compute the aggregation results efficiently. The valid_mmasaux function not only checks the validity of the mmsaux state as before, but additionally makes sure that the aggaux option, if set, is a valid for the keys of the tuple_in mapping, using the valid_maggaux function described in Section 2.1. The various state updating functions are exactly the same ones as in the mmsaux instance, but additionally need to update the aggaux state, if the option is set,

in order to preserve the validity invariant. To this end, whenever the tuple_in mapping is updated, we update the aggaux state accordingly: every time we add new keys to the mapping, we call the insert_maggaux function to insert those new keys into the aggaux state, and every time we remove keys from the mapping, we call the delete_maggaux function to remove them from the mapping. If any of the insertion or deletion operations on the aggaux data structure return the invalid flag, the aggaux option is set to None.

Updates to the tuple_in mapping happen in two ways: either by updating all the keys inside some set X to a new value, or by filtering the mapping according to some condition. In both cases, we want to avoid having to iterate over all the keys of the tuple_in mapping. In the first case, we can update the aggaux data-structure efficiently by iterating over the set X, checking if the element is not inside the mapping, and if so add it to our aggaux data structure. For the second case, depending on the type of Since formula we are monitoring, we can again replace the filtering on the mapping with an iteration over a set X. But if the formula we are aggregating over has the form $\alpha\ S_I\ \beta$ and α and β and the free variables of α and β do not coincide, we cannot avoid filtering through the whole mapping. Thus, for such formulas, our monitoring algorithm will run in quadratic time.

Finally, in the new result_mmasaux function, instead of calling the naive aggregation function on the keys of the tuple_in mapping, we can instead directly call the result_maggaux function on our aggaux data structure to retrieve the aggregation results in an efficient manner.

3.3 The Until algorithm

Before describing the integration of the aggaux data structure into the existing Until algorithm, we first give a detailed explanation of how the algorithm works.

The main difference between the Since and the Until temporal operators is that in the Since case, each time we process a new time-point we can immediately output the satisfactions of the Since operator for that time-point, since they only depend on previous time-points which we already have processed. In the Until case on the other hand, the satisfactions of the Until operator at a specific time-point depend on future time-points. This means that after processing a specific time-point we cannot yet output the satisfactions of the Until operator for that time-point, and instead we need to wait until we have read all future time-points whose timestamps are inside the interval. This also means that after processing a certain time-point, we might be able to output satisfactions for multiple previous time-points, if we have processed all time-points within the interval with regard to the timestamp at those time-points (unlike the Since case, where after processing

fun valid_mmuaux :: *args* ⇒ ... ⇒ *'a mmuaux* ⇒ *bool*
fun eval_step_mmuaux :: *args* ⇒ *event_data mmuaux* ⇒ *event_data mmuaux*
fun shift_mmuaux :: *args* ⇒ *ts* ⇒ *event_data mmuaux* ⇒ *event_data mmuaux*
fun add_new_mmuaux :: *args* ⇒ *event_data table* ⇒ *event_data table* ⇒ *ts* ⇒ *event_data mmuaux* ⇒ *event_data mmuaux*
fun eval_mmuaux :: *args* ⇒ *ts* ⇒ *event_data mmuaux* ⇒ *event_data table list* × *event_data mmuaux*

Figure 3.3: The mmuaux interface

each time-point we obtained satisfactions for only one time-point, the one we just had processed).

Consider for instance evaluating the formula $\alpha\, U_{[0,5]}\, \beta$, and we process in order time-points with timestamps 2, 3 and 10. The satisfaction of the formula at the time-point with timestamp 2 depends on all future time-points with timestamps up to timestamp 7, and so after processing just the first two time-points we cannot output anything yet, as we don't know if future time-points will have timestamps 3 up to 7. Once we process the time-point with timestamp 10, we know that no more time-points with timestamps that are inside the interval with respect to timestamps 2 and 3 will appear (as their upper limit is 7 and 8 respectively), and thus we can output satisfactions for both at the same time.

Figure 3.3 shows the function used to update the mmuaux state used in the Until algorithm. The most relevant parts of the full state (here omitted) are the tss and tables queues, the done list and the a1_map and a2_map mappings. The tss and done part of the state are used to handle the difficulty described above: tss is a queue of integers which holds all the timestamps of time-points that we have encountered but have not been fully processed yet (because we might still encounter time-points with timestamps that are inside their interval), while done is a list which holds the satisfactions of all time-points we have been able to compute (we need to store them in a list since we are able to complete multiple after processing only one time-point, as seen above).

The core of the algorithm are the a1_map and a2_map mappings. We consider the generalized Until temporal formula $\omega = \alpha\, U_{[a,b]}\, \beta$, and explain how these two mappings are used to compute satisfactions of ω in an efficient way. The mapping a1_map is from variable assignments to time-points, and its exact behaviour depends on the type of formula we are monitoring, specifically if the formula has the form $\alpha\, U_{[a,b]}\, \beta$ or $\neg\alpha\, U_{[a,b]}\, \beta$. In the first case, where the subformula α is not negated, the key v is in the mapping with value tp if and only if the subformula α with the variable assignment v holds for all time-points starting from tp up to the current one, and if this time-point tp is the most recent one possible (that means, the subformula α does not hold at

time-point $tp - 1$ for that assignment). Thus the a1_map stores all possible "starting points" for the formula $\alpha\, U_{[a,b]}\, \beta$. If any of the assignments in the keys of a1_map with lookup value tp also satisfies β at the current time-point, it follows that the formula ω holds for all time-points between tp and the current time-point (provided they are also inside of the interval). In the second case, where the subformula α is negated, the keys of a1_map are all possible assignments that satisfied α at time-points up to the current one, and the lookup value corresponding to each key is the latest time-point where that assignment satisfied α. Thus, if the key v is in a1_map with lookup value tp, it means that for the variable assignment v and all time-points following tp (from $tp + 1$ to the current time-point) the subformula α does not hold, and thus those time-points are a "starting point" for the formula $\neg\alpha U_{[a,b]}\beta$. Note that all assignments which are not keys of a1_map never satisfy α, and thus for those assignments any time-point whose timestamp respects the interval is a starting point.

Using the a1_map mapping, whenever we receive a new set of satisfactions of the subformula β we can easily compute all possible time-points and assignments for which the formula ω holds with β at the current time-point being the witness for ω satisfaction. Storing these satisfactions for the Until operator is handled by the a2_map mapping. The keys of the a2_map mapping are all the time-points that have not been completely processed yet, while the lookup value for a key tp holds all possible variable assignments such that the formula ω holds at time-point tp, together with the timestamp (or time-point, if 0 is part of the interval) of the latest β (represented as a mapping from variable assignments to timestamps or time-points). For instance, if the key/value pair (v, tp') is in the mapping corresponding to time-point tp, then for the variable assignment v the subformula α holds for all time-points from tp up to tp', the subformula β holds at tp' and the difference between the timestamps at tp' and tp is inside the interval. Moreover, there is no $tp' < tp'' \leq curr$, where $curr$ is the current time-point, such that all previous conditions hold. Finally, for each satisfaction of ω, we only store it at the earliest "starting point" inside the a2_map mapping. If for instance the satisfaction v is part of the keys of the mapping stored at time-point tp, it cannot be part of the keys of mappings stored at time-points tp' where $tp' < tp$).

With this definition of the a2_map mapping, the lookup value corresponding to the lowest (oldest) time-point of the a2_map will contain all the satisfactions for that time-point. Thus when the timestamp for that time-point falls outside the interval (which means all the time-points that can influence the set of satisfactions at that time-point have been fully processed), we can output the satisfactions for that time-point by simply taking the keys of the corresponding mapping. After that, before we can delete the entry for that time-point from the mapping, we need to add all assignments that are still

valid (meaning that the timestamp difference is still at least the lower bound of the interval) to the entry for the next time-point.
Suppose for instance we are monitoring with an interval of $[5, 10]$, we just evaluated the set of satisfactions at a time-point with timestamp 2, the mapping corresponding to it holds the key/value pairs $(v, 8)$ and $(v', 7)$, and the next time-point has timestamp 3. After adding the set of satisfactions to the done array, we need to move the pairs $(v, 8)$ and $(v', 7)$ to the mapping for the next time-point. Note however that the pair $(v', 7)$ is not valid anymore, as it has fallen out of the interval: $7 - 3 = 4 \notin [5, 10]$. Thus, before merging the mapping with the one from the next time-point, we need to remove all assignments which become invalid by falling out of the interval. One approach to find those assignments, which is used by the current VeriMon algorithm, is to iterate over all key/value pairs of the mapping and filter out those that would fall outside of the interval, but this becomes highly inefficient when the mappings become large. What we do instead in our optimized version is storing in our state all the variable assignments that have been added to a2_map at each timestamp, and then take all assignments corresponding to timestamps that have fallen out of the interval and filter over those instead. This is exactly what the tables part of our state does, implemented as a queue of (timestamp, variable assignments) pairs.

We can now look at the various functions of the mmuaux interface: the eval_step_mmuaux function handles processing a single time-point tp for which its corresponding timestamp ts that has fallen out of the interval: we add the results for that time-point to the done array, we take all the variable assignments that have fallen out of the interval from the tables queue and use those to remove assignments that are no longer valid from the mapping corresponding to tp before merging it with the mapping corresponding to $tp + 1$, and finally delete the entry for tp in a2_map. The shift_mmuaux function takes a new timestamp nt, takes all unprocessed time-points from the queue tss whose timestamps have now fallen out from the interval, and calls the eval_step_mmuaux function for each one of them. The add_new_-mmuaux functions handles adding a new time-point for which we obtained the set of satisfactions by recursively evaluating the subformulas: it receives the timestamp nt of the new time-point, and two list of satisfactions of the subformulas α and β at that time-point respectively. It firsts updates the state to be valid up to the new timestamp nt using the shift_mmuaux function, uses the two tables of satisfactions to update the a1_map and a2_map mappings to maintain the invariants described before, and finally adds the new variable assignments that were added (or for which the lookup value was changed) to the a2_map together with the new timestamp to the tables queue, and the new timestamp nt to the tss queue. Finally, the result_mmuaux function simply returns the done array, which stores the results for all time-points for which we processed all time-points in the interval with regard to those

type_synonym mmauaux = (mmuaux, aggaux option)

fun valid_mmauaux :: *args* ⇒ ... ⇒ *mmauaux* ⇒ *bool*
fun eval_step_mmauaux :: *args* ⇒ *mmauaux* ⇒ *mmauaux*
fun shift_mmauaux :: *args* ⇒ *ts* ⇒ *mmauaux* ⇒ *mmauaux*
fun add_new_mmauaux :: *args* ⇒ *event_data table* ⇒ *event_data table* ⇒ *ts* ⇒ *mmauaux* ⇒ *mmauaux*
fun eval_mmauaux :: *args* ⇒ *ts* ⇒ *mmauaux* ⇒ *event_data table list* × *mmauaux*

Figure 3.4: Optimized mmauaux instance

time-points.

3.4 Integration into Until

The integration of the aggaux data structure into the Until algorithm proceeds exactly the same as the Since case. Again, we augment the previous state of the mmuaux instance with an *aggaux option* to obtain the mmauaux state. The valid_mmauaux function again checks for the validity of the previous mmuaux state, and additionally makes sure that the aggaux part of the state is valid for the keys of the mapping corresponding to the oldest time-point in the a2_map (recall that the keys of mappings corresponding to specific time-points in the a2_map are satisfactions for the Until formula starting at that time-point, and the keys of the mapping for the oldest time-point are the ones that will be outputted next in the done array, and thus the ones for which we need to compute aggregations). In the eval_step_mmauaux function, if the *aggaux option* is set, instead of computing the aggregations the naive way over the keys of the mapping, we simply add the result of the result_maggaux call to the done array. Afterwards, we remove from the aggaux state, using the delete_maggaux function, the same satisfactions we filter out from the mapping (constructed by filtering the satisfactions taken from the tables queue), and using the insert_maggaux function, add in the keys of the mapping corresponding to the next time-point, (to account for the merging of the two mappings). In the add_new_mmmauaux function, when updating the a2_map, we also add to the aggaux state the satisfactions that are added to the oldest time-point. Finally, the eval_mmauaux function is identical to the old eval_mmuaux function, as the actual computation of the results is done by the eval_step_mmauaux function, and the eval_mmauaux function simply returns the done array.

Chapter 4

Empirical evaluation

We now evaluate the performance of our improved aggregation algorithm [8], and compare it to the previous VeriMon implementation and the unverified MonPoly implementation.

4.1 Methodology

In all our tests, we monitor a formula of the form

$$c \leftarrow \omega\ x; y\ (\alpha\ S_I\ P(x, y))$$

or

$$c \leftarrow \omega\ x; y\ (\alpha\ U_I\ P(x, y))$$

where ω is any of the supported aggregation operators, S and U are the Since and the Until temporal operator respectively, and I is the interval.

As the monitoring algorithm behaves differently when monitoring Since or Until temporal operators, we construct our tests differently in each case. In both cases, we always consider formulas where the left-hand side α is negated: if α is not negated, then updating the temporal operator's state takes time linear in the size of its resulting table and thus we cannot expect more than a constant-factor improvement.

For monitoring aggregations over the Since temporal operator, the performance of the temporal operator's algorithm (and thus ours as well) depends on the relationship between the free variables of the subformulas α and $P(x, y)$, and we divide the formulas into three types accordingly:

1. In the first type α is set to True (which is an abbreviation for $\neg$ False), and the formula becomes

$$c \leftarrow \omega\ x; y\ (\blacklozenge_I\ P(x, y))$$

which is exactly the type of formula for which MonPoly features optimized aggregations. We thus expect both MonPoly and our optimized VeriMon to be efficient, with updates taking time linear in the size of the set of elements we are adding/deleting (with an additional logarithmic factor in the size of the group for VeriMon, due to the use of binary trees).

2. In the second type α is set to $\neg Q(x, y)$. In this case the left and right subformula have the same free variables and thus, as described earlier, we can efficiently perform the filtering operations on the tuple_in mapping. We thus expect VeriMon's performance on this case to be similar to the previous one, while MonPoly's should be worse.

3. The final type is when α is set to $\neg Q(y)$. In this case the left and right subformula do not have the same free variables and we cannot efficiently compute the filtering operations on the tuple_in mapping. We thus expect VeriMon's performance on this case to be worse than the previous ones (updating the temporal operator's state at a time-point takes time linear in the size of the temporal operator's resulting table and thus we cannot expect more than a constant-time improvement when evaluating the aggregation operator). Note that the inefficiency in this case is directly related to the monitoring algorithm not being optimzied, while the inefficiency in monitoring the non-negated formula was due to the runtime overhead involved in generating enough satisfactions for the left-hand side being comparable to the one caused by computing the aggregations.

When evaluating aggregations over Until, the monitoring algorithm behaves independently of the relationship between the free variables of the two formulas. As a consequence, we monitor formulas of the first form, i.e.

$$c \leftarrow \omega\ x; y\ (True\, U_I\, P(x, y))$$

as we need no events for satisfying the left-hand side of the temporal formula, and thus can keep log size (and the overhead involved in parsing it) at a minimum.

Our log generator provides random logs for benchmarking by fixing various parameters, namely: the range of the x and y variables in the $P(x, y)$, $Q(x, y)$ and $Q(y)$ events, the range of the timestamps (with one event happening at each timestamp), the left and right interval boundaries, the type of temporal operator (Since or Until), and finally the formula type (from the three described earlier). Since we group by the y variables, generally we keep the range of y small in order to have few large groups.

For all our experiments, we generate the logs with a range of 10000 for the x variable and 20 for the y variable (so there are 20 groups), and we vary

the range of timestamps while keeping one event for each timestamp. The left and right interval boundaries are equal to $\frac{1}{5}$ and $\frac{4}{5}$ of the total log length respectively, so that when increasing the total number of timestamps, we also at the same time increase the maximum set size that we aggregate over by proportionally the same amount.

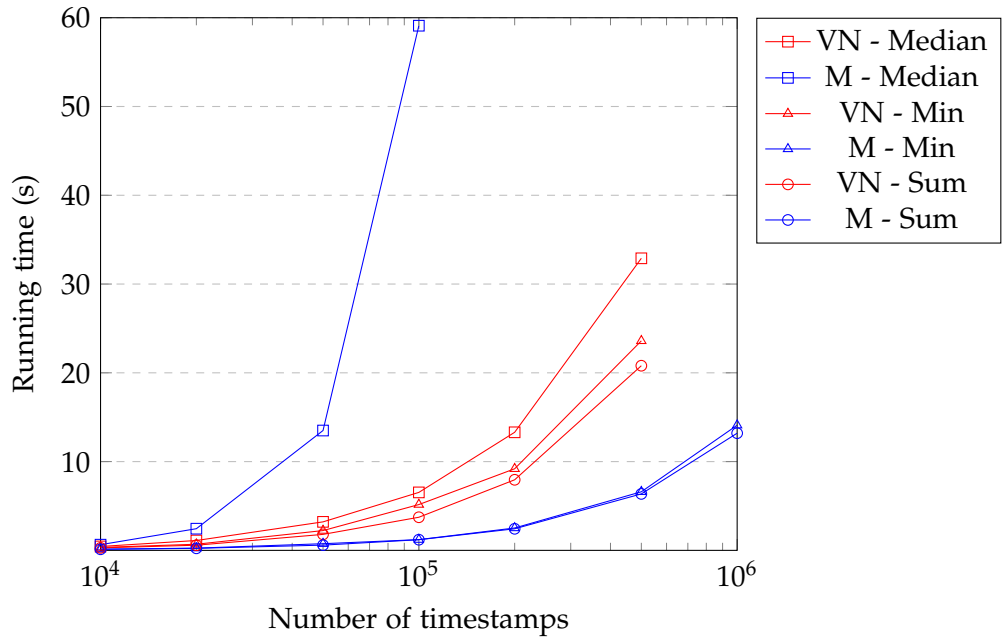
The log is generated according to the formula type: for the first type, at each time-point we generate a random $P(x, y)$ event; for formulas of the second type, with probability $\frac{1}{3}$ we instead generate a $Q(x, y)$ event to invalidate a previously appearing $P(x, y)$ event, if there are any (meaning that on average, about half of the $P(x, y)$ will have been invalidated by the end of the log); and for formulas of the third type with probability $\frac{ys}{\text{timestamps}}$ instead of a $P(x, y)$ event we generate a random $Q(y)$ event to invalidate all $P(x, y)$ events for that specific y, where ys is the range of the y variable (and thus the number of groups). The probability here is chosen to be much lower than in the previous case, as the range of y is small and a single $Q(y)$ invalidates a lot of $P(x, y)$ events. On average, the number of groups that will be invalidated is equal to the total number of groups, but it is very likely that some groups will never get invalidated until the end of the log (and thus become large).

4.2 Results

All the experiments in this section were run on a 2019 15-inch MacBook Pro running macOS Big Sur, with a 2,4 GhZ 8-Core Intel i9 and 32 GBs of 2400 MHz DDR4. The running times shown are the average of 10 executions.

We first compare the running time between our optimized VeriMon and MonPoly when monitoring the Since temporal operator on formulas of the first type, using different aggregation types. We omit the maximum aggregation as it is symmetric to the minimum one, and the average and count aggregations as they are analogous to the sum aggregation (average involves one additional division per group when outputting results, while count requires one less addition per group when updating). For this test we omit old VeriMon, as it always took more than 60 seconds. The results are summarised in Figure 4.1. Note that all plots are linear-log, i.e. the x-axis (number of timestamps) is shown using a logarithmic scale while the y-axis (seconds) has a linear scale.

For the median aggregation, MonPoly follows the list approach without the subsequent refinement to trees, leading to much slower runtimes than our implementation. On the minimum and maximum aggregations, MonPoly's algorithm is faster asymptotically than ours as it takes advantage of the FIFO nature of the updates of the keys of the mapping, with which it can, for each group, perform updates in amortized constant time. Our tree approach instead takes logarithmic time for each update. Note that the

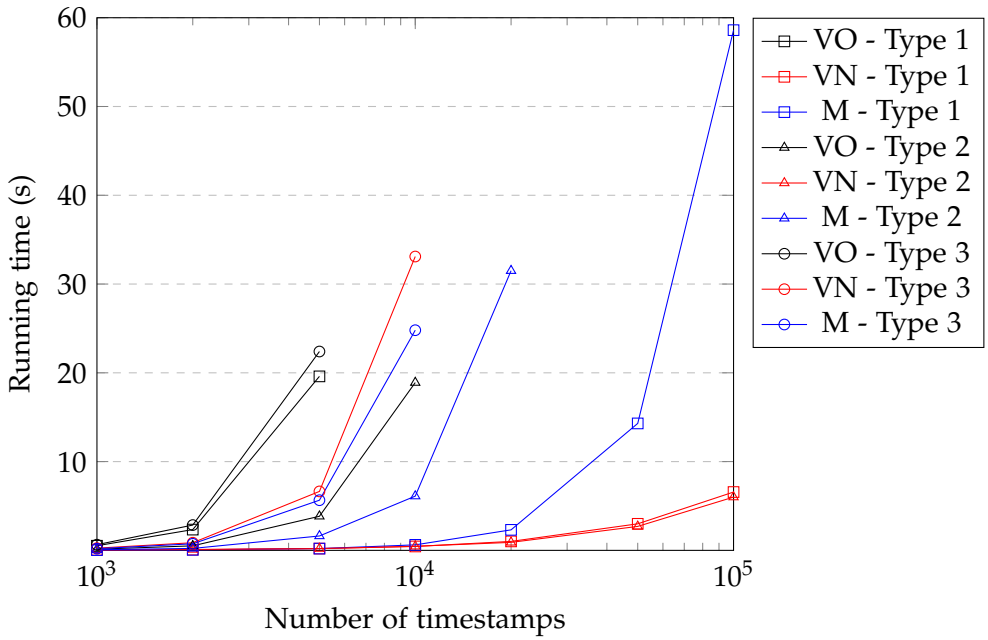


	Median		Min		Sum	
Timestamps	VN	M	VN	M	VN	M
10000	0.44	0.64	0.33	0.13	0.28	0.12
20000	1.11	2.44	0.69	0.25	0.58	0.23
50000	3.21	13.5	2.23	0.75	1.8	0.59
100000	6.53	59.1	5.16	1.21	3.73	1.20
200000	13.3	TO	9.19	2.52	7.95	2.43
500000	32.9	TO	23.6	6.60	20.8	6.36
1000000	SO	TO	SO	14.1	SO	13.2

Figure 4.1: Comparison of different aggregations on formulas of type 1 (VN = new VeriMon, M = MonPoly, TO = timeout (60 seconds), SO = stackoverflow)

minimum/maximum aggregations are still faster in our version than the median aggregation as outputting the results takes only one logarithmic call rather than two for the median. On the sum/count/average aggregations, VeriMon is very slightly faster than on the minimum/maximum ones, due to not having binary trees involved in updating and outputting the results anymore. MonPoly again outperforms VeriMon, possibly due to the increased runtime overhead VeriMon has due to the formally verified code.

Next, we compare the runtime of MonPoly and old and new VeriMon when monitoring the median aggregation operator over the Since temporal operator, for each of the three types described earlier. We monitor the median aggregation as it is the slowest one due to the logarithmic factor involved in updating the groups and retrieving results. The results are summarised



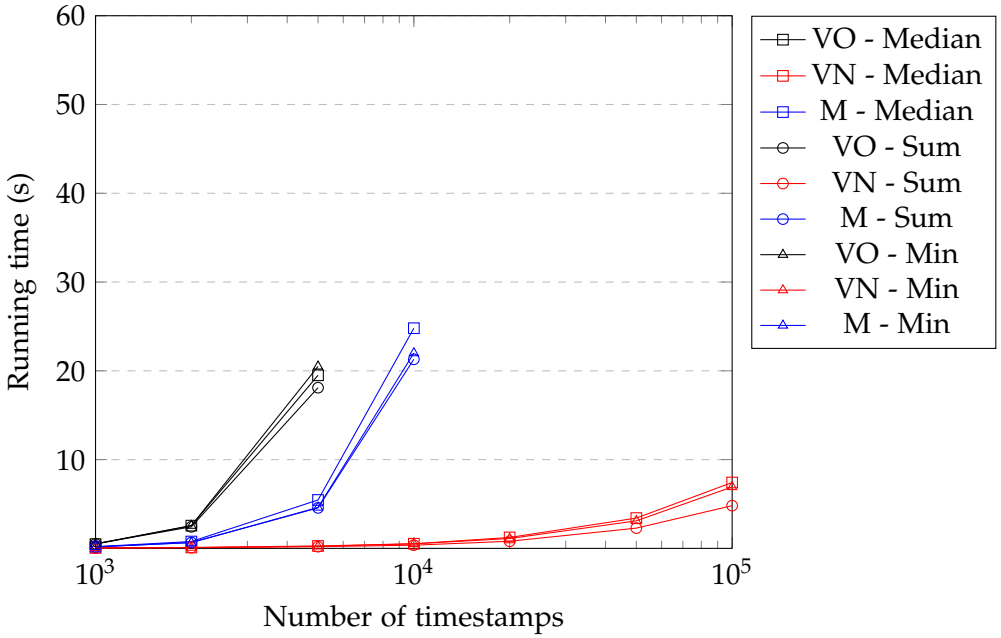
	True $S_I\ P(x,y)$			$\neg Q(x,y)\ S_I\ P(x,y)$			$\neg Q(y)\ S_I\ P(x,y)$		
Timestamps	VO	VN	M	VO	VN	M	VO	VN	M
1000	0.52	0.06	0.03	0.13	0.05	0.07	0.64	0.25	0.19
2000	2.34	0.08	0.06	0.51	0.10	0.22	2.85	0.86	0.74
5000	19.6	0.20	0.20	3.84	0.20	1.62	22.4	6.64	5.65
10000	TO	0.44	0.62	18.9	0.46	6.11	TO	33.1	24.8
20000	TO	1.01	2.31	TO	0.90	31.5	TO	TO	TO
50000	TO	2.99	14.3	TO	2.7	TO	TO	TO	TO
100000	TO	6.57	58.6	TO	6.0	TO	TO	TO	TO

Figure 4.2: Evaluation of the median operator over Since (VO = old VeriMon, VN = new VeriMon, M = MonPoly, TO = timeout (60 seconds))

in Figure 4.2. Again, as MonPoly does not provide any optimizations for the median algorithm, our improved algorithm vastly outperforms both the old VeriMon one and the MonPoly one on formulas of the first two types. Specifically for formulas of the second type, our improved algorithm provides both the optimized algorithm over incrementally updated sets (which MonPoly only provided for the Once operator), and the optimized median calculation using the refinement to trees.

For formulas of the third type, as described earlier, our algorithm is not optimized. While still being faster than the old VeriMon implementation, it is again slightly outperformed by MonPoly.

Finally, we compare the runtime of the three tools when monitoring formulas of the first type over the Until temporal operator. The results are summarised



	Median			Min			Sum		
Timestamps	VO	VN	M	VO	VN	M	VO	VN	M
1000	0.48	0.05	0.19	0.52	0.05	0.16	0.51	0.04	0.17
2000	2.56	0.10	0.77	2.56	0.09	0.65	2.43	0.08	0.66
5000	19.5	0.27	5.45	20.5	0.23	4.61	18.1	0.19	4.56
10000	TO	0.52	24.8	TO	0.51	22.0	TO	0.36	21.3
20000	TO	1.23	TO	TO	1.11	TO	TO	0.80	TO
50000	TO	3.41	TO	TO	3.10	TO	TO	2.28	TO
100000	TO	7.42	TO	TO	6.92	TO	TO	4.81	TO

Figure 4.3: Evaluation over Until operator (VO = old verimon, VN = new verimon, M = monpoly, TO = timeout (60 seconds))

in figure 4.3. In this case, MonPoly does not do any optimizations and our new algorithm greatly outperforms both the old VeriMon and MonPol. Again, aggregations which do not involve order statistic trees to represent the groups' results are slightly faster.

Chapter 5

Conclusion

We have designed, implemented and formally verified in Isabelle/HOL an efficient algorithm for computing aggregations over incrementally updated sets, using standard approaches for computing aggregations over sliding windows. We then integrated these optimizations into the VeriMon monitoring tool, and by doing so not only have we brought it up to par with the previous optimizations for the Once temporal operator present in the MonPoly monitoring tool, but also generalized them to all forms of the more general Since and Until operator, providing in most cases better or comparable performance to MonPoly. In this way, VeriMon not only provides formally verified correctness of its algorithm compared to the unverified MonPoly, but also significantly better performance for specific formulas, making it viable for monitoring aggregations over large sets. Moreover, while integrating these optimizations into the existing tool, we implemented some slight improvements to the current algorithm for evaluating the Until temporal operator.

Several improvements to the aggregation algorithm remain possible, such as directly integrating the assumptions of correct types and finite sets needed by our algorithm into the monitoring algorithm, in order to drop the validity checks in our data structure. Another improvement could be implementing MonPoly's algorithm for computing minimum/maximum aggregations for the specific cases where the satisfactions of the subformula are added and removed in FIFO order, which would allow us to replace the order statistic trees with queues, and thus improve the performance for updating/outputting the result for a group from logarithmic in the size of the group to amortized constant.

Bibliography

[1] David A. Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, 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, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 432–453. Springer, 2020.

[2] 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.

[3] David A. Basin, Felix Klaedtke, and Eugen Zălinescu. The MonPoly monitoring tool. In Giles Reger and Klaus Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.

[4] Agnishom Chattopadhyay and Konstantinos Mamouras. A verified online monitor for metric temporal logic with quantitative semantics. 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 383–403. Springer, 2020.

[5] Florian Haftmann, Alexander Krauss, Ondrej Kuncar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26,*

2013. Proceedings, volume 7998 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2013.

[6] Yoichi Hirai and Kazuhiko Yamamoto. Balancing weight-balanced trees. *J. Funct. Program.*, 21(3):287–307, 2011.

[7] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. Sliding-window aggregation algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017*, pages 11–14. ACM, 2017.

[8] Emanuele Marsicano. Verimon with optimized aggregations. https://github.com/EmanueleFWM/verimon-optimized-aggregations, August 2021.

[9] Tobias Nipkow and Stefan Dirix. Weight-balanced trees. *Arch. Formal Proofs*, 2018, 2018.

[10] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[11] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[12] 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.

[13] The Coq Development Team. The Coq proof assistant, January 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Verified Incremental Evaluation of Aggregation Operators in Metric First-Order Temporal Logic

Authored by (in block letters):

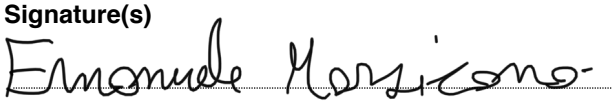
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Marsicano	Emanuele

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
September 11, Zurich	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.