Verified Evaluation of Recursive Expressions in Metric First-Order Dynamic Logic

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Verified Evaluation of Recursive Expressions in Metric First-Order Dynamic Logic

Master's Thesis by Sheila Zingg

Supervising Professor
Prof. Dr. David Basin

Thesis Supervisors
Prof. Dr. Dmitriy Traytel
Joshua Schneider

Submission date: 14.7.2021
Computer Science Department
Study Regulations of 2009

Abstract

Monitors are algorithms whether verify that a system works as intended by checking if event sequences, also called traces, violate a given policy. VeriMon is a monitor that can evaluate fragments of metric first-order dynamic logic (MFODL). Additionally to that, VeriMon is a verified monitor as its correctness has been proven using Isabelle/HOL. VeriMon's current specification language only allows for limited recursion by means of regular expressions, where the main limitation comes from the fact that the variable binding is fixed for the entire recursion. For some recursive formulas, such as the transitive closure, however, the variable binding need needs to change from one step to the next. There exist policies, such as the Linux kernel deadline inheritance protocol, that could easily be monitored using recursion, but would be difficult to handle without. We propose two recursive operators that can be used for different tasks and extend VeriMon with those operators. The first operator allows for recursion if the recursive predicate occurs strictly in a past context. The second one does not have this restriction. Both operators need to restrict the recursive formulas, as, for instance, future operators containing the recursive predicate may create an infinite loop. The necessary restrictions are different for both operators as their strength is incomparable. There exist other monitors for different specification languages that can evaluate recursive formulas, but many of them require a strict temporal guard, which is not needed for the second operator. Furthermore, we allow future operators as long as they do not contain the recursive predicate, which is not always allowed in other monitors. Lastly, we update VeriMon with the operators, which entails both updating and adding relevant definitions, functions, lemmas, and proofs such that VeriMon's correctness lemma holds for all operators including the two new recursive ones. The difference in restrictions led to two distinct implementations for the two operators.

Acknowledgements

I deeply thank my supervisors Prof. Dr. Dmitriy Traytel and Joshua Schneider for their extensive support throughout this thesis. All ideas presented in this thesis as well as all Isabelle/HOL proofs were devised in close collaboration with my supervisors, who were always able to guide me in the right direction with their expertise, their willingness to help, and their openness to new ideas.

I, furthermore, thank Prof. Dr. David Basin for allowing me to write my thesis in the Information Security group and for helping me find such an interesting topic.

I also thank Dr. Srdjan Krstic for providing material to get to know both MonPoly and VeriMon as well as Isabelle/HOL and for introducing me to the monitoring group within the Information Security group.

Contents

1 Introduction

As systems become more complex and legal regulations become more stringent, it is important to verify that the systems work as intended. One way to achieve this is to define the intended system behavior as a set of policies and continuously monitor the system for possible policy violations. For this method to succeed the language used to formalize the policies must be versatile enough to express the multitude of different policies that may be used for systems. This is not trivial, as even simple policies such as: "If one day prior, a customer has suspended their credit card for X days, then that card may not be accepted anywhere for the next X days unless the customer revoked their suspension", "No customer may have a total debt that exceeds X" or "A customer should not be able to authenticate if they already attempted and failed three times in a row" require many different operators such as past ("one day prior"), future ("the next X days"), negation ("may not be accepted"), aggregation ("total debt"), quantifiers ("no customer") and regular expressions ("failed three times"). Metric first-order dynamic logic (MFODL) is, thus, a fitting specification language, as it provides all those operators. VeriMon [13] was created as a formally verified monitor for fragments of MFOTL, which is MFODL without regular expressions, and was later upgraded [4] to a formally verified monitor for fragments of MFODL. VeriMons correctness has been proven using Isabelle/HOL, which increases its trustworthiness.

The usefulness of a monitor is linked to the expressiveness of its specification language. Currently, VeriMon only supports recursion by means of regular expressions, for which the variable binding is fixed for the whole recursion. For some recursive formulas, such as the transitive closure, however, the variable binding needs to change from one step to the next, and thus, they cannot be expressed with regular expressions alone. We consider the non-recursive let operator (Let $p\ \phi\ \psi$) that VeriMon provides for the basis of the new recursive operator. The let operator is composed of a predicate p, a formula ϕ, and a formula ψ and evaluates any occurrence of p in ψ with the evaluation of ϕ. If ϕ contains the predicate p, then p will be interpreted as coming from the context surrounding the let, instead of from the context inside of the let, which makes the operator non-recursive. It is, thus, possible to make the let operator recursive by changing the context of any p in ϕ.

In this thesis, we define and implement two recursive let operators, which each serve different purposes. Creating such operators is not trivial as, in a naive implementation, infinite recursion may occur, which cannot be supported by monitors. The recursive let operators we propose are LetPrev and LetRec, both of which can be evaluated without needing infinite recursion. LetPrev enforces that any occurrence of p that is contained in ϕ is strictly in the past. Both operators need to restrict ϕ as, for instance, if p were contained in a future operator, an infinite loop could occur. The evaluation result of a recursive formula is always a so-called fixpoint, which is a set of results for which a subsequent recursive call yields the same set of results. If such a fixpoint exists and is computable in finitely many steps then no infinite recursion can occur. We limit the operators in the ϕ of LetPrev and LetRec such that the ϕ always has a fixpoint that can be computed in finitely many steps. The restrictions of LetPrev and LetRec, and thus, neither operator is strictly stronger than the other but both serve different purposes. This in turn leads to fundamentally different implementations for LetPrev and LetRec. After defining the two operators, we implement them in VeriMon by adding and adapting relevant lemmas, definitions, and proofs.

In the following, we give a short overview of the contents of this thesis. In Section 2, we discuss some related work. Section 3 introduces all concepts and tools necessary to understand this thesis. We cover the non-recursive let operator, how the Isabelle/HOL proof assistant works, what can be expressed using MFODL, how VeriMon is constructed, and what notation will be used. Section 4 explores the recursive let operator from a more theoretical viewpoint. We motivate our operators by examining some use cases for a monitor with recursive formulas. After that, we discuss the semantics of the recursive let and why infinite recursion may occur. We describe the two operators LetPrev and LetRec that we propose in this thesis and explain how they work. Lastly, we analyze the differences between the operators. Section 5 explains how we integrated the new LetPrev operator in VeriMon by showing the definitions and lemmas that we adapted and created. Similarly, Section 6 explains how we integrated the new LetRec operator in VeriMon. Finally, Section 7 summarizes the findings of this thesis and discusses some interesting future work.

In this thesis, we focus on the most interesting and insightful properties of the implementation. Furthermore, we will omit all proofs in this thesis, as they would be hard to understand on paper, due to the powerful automation tools provided by Isabelle/HOL. We refer to the code, which can be found under the URL `https://bitbucket.org/jshs/monpoly/src/master/` in the branches letprev_sz and

letrec, for the complete set of lemmas and all proofs.

Contribution In this thesis, we investigate the recursive let operator and integrate the operator in VeriMon. The recursive let operator is of the form "Let p ϕ ψ" and signifies that any occurrence of the predicate p in the formula ψ will be substituted with the formula ϕ. The formula ϕ must be evaluated recursively as ϕ might contain p.
In the following we summarize the contributions:

- We investigate the recursive let operator (Let p ϕ ψ where ϕ may contain p) and find that infinite recursion poses a problem. Such an infinite recursion cannot be monitored, which means that we must enforce that the recursive let operator only uses a finite recursion. We propose two recursive let operators LetPrev and LetRec, that do not suffer from infinite recursion. In the first version (LetPrev), we enforce that any occurrence of p in ϕ occurs strictly in the past. In the second version (LetRec), we identify operations that might make the recursion infinite and exclude them specifically. The operations that we found to be problematic are negations anywhere in front of p, future operations anywhere in front of p, and any equality operations.
- We integrate these two new operators in VeriMon by defining the operator behavior, adapting all functions, and adding new functions as necessary.
- We prove that the monitor correctness still holds by adapting all relevant proofs and adding new lemmas as necessary.

2 Related Work

Runtime verification (RV) is used to gain precise information about the runtime behavior of a system. RV can be used for many purposes including testing, verification, and debugging. Bartocci et al. [3] give an introduction to the field of RV.

Basin et al. [7, 6, 8] introduced MonPoly as a monitor that uses metric first-order temporal logic (MFOTL) as the specification language. Using MFOTL as the specification language creates a powerful monitor that can both formalize relations between events, including formalizations that require a universal or an existential quantifier, and quantify time constraints between events. Furthermore, MonPoly supports an aggregation operator that is needed for many policies. However, MonPoly does not support regular expressions and is not formally verified.

In an independent line of work, Basin et al. [5] proposed Aerial, a monitor that uses metric dynamic logic (MDL) as the specification language. With MDL, it is possible to both express the temporal relationship between events as well as support all regular expressions, with which event sequences can be expressed. Furthermore, Aerial is almost event-rate independent, which means that the monitor's space usage only logarithmically depends on the number of events that occur in a fixed time unit. This property was shown to matter in practice and makes Aerial a more efficient monitor. However, Aerial is not first-order, and thus, does not support existential and universal quantifiers and it is also not formally verified.

Building on MonPoly, Schneider et al. [13] created VeriMon, a formally verified monitor that uses MFOTL as the specification language. Using a formally verified monitor is desirable as the behavior of the monitor is clearly stated and proven. The first version of VeriMon was inefficient and lacked support for regular expressions. Basin et al. [4] improved VeriMon by increasing its efficiency, adding support for regular expressions, and adding an aggregation operator. These improvements made VeriMon a more efficient, formally verified monitor, that uses fragments of MFODL as specification language, which is more powerful. In this thesis, we enhance VeriMon with recursive formulas.

An example of a monitor that already supports recursive formulas is DejaVu [10]. The specification language of DejaVu is first-order, however, it is incomparable to MFOTL. A recent version of DejaVu supports recursive formulas, where the recursive predicate is strictly in the past. This is similar to one of the recursive operators that we propose for VeriMon, but the difference in specification language changes the scope of the recursive operators. For instance, DejaVu does not support future operators, which we allow in the recursive formula as long as they do not contain the recursive predicate.

Rule-based monitors such as EAGLE [2] often allow for recursive rules. EAGLE specifically allows recursive rules, as long as they are guarded by a strict temporal operator. The structural difference of the monitor and specification language means that the scope of the recursive operator is different than the one in VeriMon. Furthermore, we propose one recursive let operator that does not need any strict temporal guard.

Recursive formulas are a well-studied area. Abiteboul et al. [1] discuss recursion for the evaluation of queries on databases. The relevant concept of a fixpoint, a set of query results, for which an additional recursion step would not add any new results, was reviewed in this book. In a detailed discussion, the issues that may arise when combining the negation operator with recursion and the cases in which the negation operator would pose no problem were explored. Ebbinghaus et al. [9] examine the concept of a fixpoint for finite model theory, which is the part of the semantics of formalized languages that describes the intersection of the syntactic structure of the axiom structure and the properties of its finite models. In this thesis, we will also use fixpoints to evaluate the recursive operators. Furthermore, we will use the knowledge gained from the discussion of recursive formulas with negation to evaluate which MFODL formulas can be recursively monitored and which cannot.

3 Background

In this section, we explain the background necessary to understand the thesis. We start by introducing the non-recursive let operator, which is the basis for the recursive let operators. Then, we explain the Isabelle/HOL proof assistant, which was used to prove the correctness of the monitor. After this, we introduce metric first-order dynamic logic, which is the specification language of VeriMon. Then, we provide the relevant details about VeriMon, which is the verified monitor that we are extending. Lastly, we elaborate on the notation that will be used later.

3.1 The Non-Recursive Let Operator

The non-recursive let operator has the following form: "Let p ϕ ψ", where p is a predicate name, and ϕ and ψ are formulas. This operator is often written as "let $p(x, y, ...) \equiv \phi$ in ψ", however, we use a different way of writing this operator here, as it is more in line with the definition used in VeriMon. Intuitively, this operator indicates that every occurrence of p in ψ is substituted with ϕ. When evaluating a let formula, it is possible to first evaluate ϕ and then use the evaluation of ϕ for p in the evaluation of ψ instead of substituting every p in ψ with ϕ and then evaluating ψ. As this is a non-recursive let operator, each instance of p in ϕ is interpreted as if it occurred in the context surrounding the let. This corresponds to the following formal definition of the let operator:

Definition 1 (Non-Recursive Let [14])**.** *The semantics are a relation $(\sigma, S, i, v) \models \phi$, where σ and S are temporal structures, i is the evaluation time-point, and v is a set of valuations. S is a relation of the form $S(p, i, a)$ that is true iff the predicate symbol p at time-point i contains the tuple a. With this, we get the following semantics*

$$(\sigma, S, i, v) \models (\mathit{Let}\ p\ \phi\ \psi) \iff (\sigma, S', i, v) \models \psi$$

$$\mathit{where:}\ S'(q, j, a) \iff \begin{cases} (\sigma, S, j, a) \models \phi & \mathit{if}\ q = p \\ S(q, j, a) & \mathit{otherwise} \end{cases}$$

This Semantic definition is non-recursive as the evaluation of S' uses S instead of S'.

3.2 The Isabelle/HOL Proof Assistant

Isabelle/HOL [12] is a proof assistant for higher-order logic (HOL). A proof assistant allows users to specify properties and proofs for those properties. The proof assistant will verify if the proof submitted is sufficient to show the stated property and reject the proof if it is not. If the correctness of a program, or in our case a monitor, needs to be proven, the properties relevant to the correctness need to be defined such that they can be used in proofs. Isabelle/HOL provides many useful tools, such as recursive functions and data types that can be used for specifying and proving properties. An important aspect of the proof assistant is that it only verifies the correctness of the proofs submitted by the user. If users do not define the properties correctly or do not consider all properties that are necessary for the program to run as intended, the proof assistant cannot help. It, thus, remains very important to thoroughly consider all properties necessary for the program to run as intended.

3.2.1 Structures

For a proof assistant to be able to verify the correctness of proofs, all elements used in the proof must be written in a specific, well-defined way. Next, we examine all elements that can be used in Isabelle/HOL. We will use the example of a binary tree to illustrate how each of the elements is used. A binary tree consists of nodes, that have a value and are connected to a left subtree and a right subtree, and leaves that just have a value.

Data Types Before any functions can be defined, we need to define the types used in those functions. This can be done using the *datatype* keyword. Types defined with *datatype* can be formed using multiple cases and can be recursive, in which case they must contain one or more base cases and one or more recursive steps. It is possible to give each part of the definition a name. As an example, we show how one could define the *binary_tree* type:

Example 2 (binary_tree in Isabelle/HOL)**.** *We define the binary tree recursively using two cases. The first case is the recursive step, which we call Node. The node consists of a natural number that is its value and a binary_tree as left subtree and another one as right subtree. The second case is the base case, which we call Leaf. The Leaf has no subtrees, and thus, only consists of a natural number, which is its value.*

datatype *binary_tree* =
Node *nat binary_tree binary_tree*
| Leaf *nat*

In Example 2 we used *nat*, which is a data type that we did not define, as it is a *predefined data type*. Isabelle/HOL provides a few of the commonly used data types as *predefined data type*, the most important of which are *nat*, *bool*, *char*, *string* and *list*. We omit their definitions here, as they are defined as one would assume.

Lastly, it is possible to define synonyms for types using the *type_synonym* keyword. This can make definitions more concise by exchanging complex types with a synonym or can make definitions more clear by having different synonyms for different variables. As an example we show how one would define a named binary tree, which is a pair of string and binary_tree:

Example 3 (named_binary_tree in Isabelle/HOL)**.** *A named binary tree is a pair of string and binary_tree, which can be written using the expression (string* × *binary_tree). We here define a synonym called named_binary_tree for this type:*

type_synonym *named_binary_tree* = *(string* × *binary_tree)*

Functions We can use the data types, that we created to define functions. Functions take some input and produce some output. The input and output types can be specified manually, or Isabelle/HOL can infer them from the function's definition. Inference usually yields the correct types, however, sometimes types must be specified manually. In all examples in this section, we will specify the types. In Isabelle/HOL there are three ways to define a function, namely using *pattern matching*, *inductively* or as *definitions*. We will show how to write a function using each of these three ways. For all examples, we will use the data type binary_tree that we showed before.

We start by explaining functions based on pattern matching, which use the *fun* or *function* keyword in Isabelle/HOL. For such functions, we define what happens in each of the cases defined by the data type, but cases may be combined using pattern matching. Furthermore, the cases may contain recursions. The difference between *fun* and *function* is that *fun* automatically tries to prove that no patterns overlap, as well as exhaustiveness, and termination, whereas in *function* this needs to be done manually. We will use *function* only if the automatic proofs fail for *fun*. Example 4 illustrates a function that uses pattern matching.

Example 4 (plus_n_tree in Isabelle/HOL)**.** *We define a function that takes a binary tree* tree *and a number* n *and outputs a binary tree that is equivalent to* tree *but has* n *added to each node and leaf value.*

fun *plus_n_tree :: "binary_tree* ⇒ *nat* ⇒ *binary_tree where"*
"plus_n_tree (Node *a l r*) *n* = Node (*a*+*n*) (*plus_n_tree l n*) (*plus_n_tree r n*)*"*
| *"plus_n_tree* (Leaf *a*) *n* = Leaf (*a*+*n*)*"*

Next, we explain how functions can be defined inductively using the *inductive* keyword. Inductive functions always output a value of the type *bool*. First, some preconditions are specified and then a postcondition is specified. If all preconditions are true, then the specified postcondition also evaluates to true. There can be multiple cases and the cases can be evaluated recursively. Example 5 illustrates an inductive function.

Example 5 (in_order_tree in Isabelle/HOL)**.** *We define a function that takes a binary tree* tree *and evaluates if* tree *is in in-order. A tree is in-order if for all nodes the value n of that node is larger than the value of the first node of the left subtree and smaller than the value of the first node of the*

right subtree.

inductive *in_order_tree :: "binary_tree ⇒ bool"*
"in_order_tree (Leaf *a)"*
|"a ≥ b ⟹ a ≤ c
⟹ in_order_tree (Node *a* (Leaf *b*) (Leaf *c))"*
|"a ≥ b ⟹ a ≤ c ⟹ in_order_tree (Node *c l r*)
⟹ in_order_tree (Node *a* (Leaf *b*) (Node *c l r*))"
|"a ≥ b ⟹ a ≤ c ⟹ in_order_tree (Node *b l r*)
⟹ in_order_tree (Node *a* (Node *b l r*) (Leaf *c*))"
|"a ≥ b ⟹ a ≤ c ⟹ in_order_tree (Node *b l r*) *⟹ in_order_tree* (Node *c l r*)
⟹ in_order_tree (Node *a* (Node *b l r*) (Node *c l r*))"

The final function type is the definition, which uses the *definition* keyword. Definitions may neither have multiple cases nor recursion, but can otherwise be complex and require computation. Unlike the other function types, it is possible to avoid using definitions by writing the full version every time, however, using definitions can make things easier and keep proofs concise. Example 6 illustrates a definition:

Example 6 (is_in_order_tree in Isabelle/HOL)**.** *We define a function that takes a binary tree* tree *as input and outputs* 1 *if* tree *is in-order and* 0 *otherwise*

definition *is_in_order_tree :: "binary_tree ⇒ nat" where*
"is_in_order_tree tree = (if (in_order_tree tree) then 1 else 0)"

As mentioned earlier, Isabelle/HOL has some predefined data types. Each of those data types comes with its predefined functions. We will mention some of the important ones here, for most, we will omit any explanation and for all, we will omit a formal definition, as they should be as expected. For *nat* all arithmetic operations (e.g. $+$, $*$, $<$) are predefined. Furthermore, there is a successor function *Suc*, which returns the number after the input number. For *bool* all common Boolean operations (e.g. $\vee$, $\wedge$, $\rightarrow$) are predefined. For *list* the most important predefined functions are *[]*, which is the empty list, *length*, which returns the length of the list, # that prepends a value to a list, and @ that appends a list to another list.

Lemmas After creating some functions, we can start proving properties of those functions. In this section, we will show how lemmas are defined, however, we are going to omit the proof, as we will in all of this thesis. Isabelle/HOL provides powerful tools that automate some of the proving steps, such as tools that can automatically find relevant lemmas and apply them (e.g. auto, force, blast, etc) and tools that can find proofs for certain statements (sledgehammer). If automation is not possible, it is often necessary to apply lemmas individually, to make the statement simpler, step by step.

Lemmas are defined using the *lemma* keyword. Lemmas may be given a name. This is not compulsory but highly recommended if it needs to be used in future proofs. After that, the property, that should be shown is stated. One can also define some preconditions which would be written before the statement using the *assumes* keyword. The statement would then be written after a *shows* keyword. Lastly, the proof for that property needs to be provided. Example 7 illustrates a lemma:

Example 7 (lemma_in_order_tree in Isabelle/HOL)**.** *We define a lemma that shows that if a tree is in-order it is still in-order after plus_n_tree was applied to it using any number*

lemma *plus_n_in_order: "in_order_tree t ⟹ in_order_tree* (*plus_n_tree t n*)*"*
\\proof omitted

3.2.2 Some Important Functions

In this section, we will go through some predefined functions that we used in this thesis and that are a bit more involved.

Options Isabelle/HOL provides an option data type for variables where it is not clear if they have a value or not. A variable x of the data type option can either be None or Some *val*. The function the applied on an option x returns either the value *val* is option x = Some *val* or some default value if option x = None.

Definition 8 (Option Data Type Isabelle/HOL)**.** *The option data type in Isabelle/HOL:*

datatype *'a option* =
None
| *Some (the: 'a)*

Note: 'a is a placeholder that can be initialized with any arbitrary data type.

While While loops are commonly used in programming for actions that need to be performed repeatedly and where it is not clear beforehand, how often the action should repeat. We will use such a while loop in this thesis. In an Isabelle/HOL the while loop starts with a continuation condition which takes the initial value or the loop body result as input and evaluates if the loop should continue. Following this is the while body that computes the desired function taking as input either the initial value or the previous result from the while body. Lastly, the initial value is defined. If the continuation condition is false, then the last while body result is returned as result. Example 9 shows such a while loop:

Example 9 (while_loop in Isabelle/HOL)**.** *A while loop in Isabelle/HOL looks as follows:*

$$\text{while } (\lambda x.\ x > 0)\ (\lambda x.\ \mathit{Suc}\ x)\ (1)$$

The loop shown in Example 9 does not terminate, as the loop would only terminate if x were no longer larger than 0, which is not possible with this loop. This is an issue with while loops, as their termination is not obvious and can at times not be given at all. It is not possible to prove anything about the output of a loop that does not terminate, thus, for most lemmas, one would have to first prove the loop termination, which can be difficult. A way to avoid this issue in Isabelle/HOL is to use while_option instead of while. The while_option function returns the loop result if it terminates and the value None if it does not. This allows us to handle the case in which the loop does not terminate specially. For Example 9, we may, for instance, want to define a function, that just returns 0 if the loop does not terminate, which would look like this:

Example 10 (while_option_loop in Isabelle/HOL)**.** *We show a definition that takes returns the result of a while loop if it terminates and* 0 *otherwise:*

definition *"loop_res =*
(case while_option $(\lambda x.\ x > 0)$ $(\lambda x.\ \mathit{Suc}\ x)$ (1) *of* None $\Rightarrow 0$ | Some $x \Rightarrow x$)

list_all2 The function list_all2 takes three inputs. The first input is a function that takes two inputs and the second and third inputs are lists. If and only if the length of xs and ys are equal and for all indexes i in the length of xs the value of xs at index i and the value of ys at index i satisfy the function P, then ist_all2 P xs ys evaluates to true. This can be formalized as follows:

Definition 11 (list_all2 in Isabelle/HOL)**.** *We show the definition of list_all2:*

$$\mathit{list\_all2}\ P\ xs\ ys \longleftrightarrow \mathit{length}\ xs = \mathit{length}\ ys \wedge (\forall (x, y) \in \mathit{set}\ (\mathit{zip}\ xs\ ys).\ P\ x\ y)$$

Note:
The function set takes a list and converts it to a set.
The function zip takes two lists of equal length and creates a list of tuples where one element comes from the first list and the other comes from the second list.

3.3 Metric first-order dynamic logic (MFODL)

Metric first-order dynamic logic (MFODL) is a language that enhances metric first-order temporal logic (MFOTL) with regular expressions. MFOTL encompasses predicates, past and future temporal operators, quantifiers, binary operators, aggregation, and negation. The scope of a language is best shown using examples showing what they should be able to express. We use a bank as an example. A policy that the bank has may be "If one day prior, a customer has suspended their credit card for X days, then that card may not be accepted anywhere for the next X days unless the customer revoked their suspension", which is not an unrealistic policy for a bank, but shows the high level of expressiveness that is required from the language. To formalize this statement predicates are needed to encode all events (e.g., suspending a credit card) and actors (e.g., the customer), past operators are needed to specify the "one day prior" part, future operators are needed to formalize that the card may not be used "in the next X days", binary operators are needed to specify the "unless" property, and negation is needed to describe that the card "may not be accepted". This simple example already makes use of many operations that MFOTL has to offer, however, some operations are not used.

Another policy that the bank in our example has is that "No customer may have a total debt at the bank that exceeds X", which is a simple property, that uses other aspects of the language. To formalize this statement predicates are again needed to encode all events, quantifiers are needed to specify that the property needs to hold "for all customers", negation is needed to flip the "for all customers" to "no customer" and aggregation is needed to get the "total debt" of each customer. Between the two examples, we have now used all operations that MFOTL offers, which indicates that languages weaker than MFOTL would limit the usability significantly.

Lastly, for the e-banking platform, the bank in our example has the policy "A customer should not be able to authenticate if they already attempted and failed three times in a row". This property sounds easy enough but needs regular expressions to be formalized. Policies like this one are often used, but a monitor based on MFOTL would not be able to monitor it, as it lacks regular expressions. Thus, MFODL, which has all properties of MFOTL, but additionally supports regular expressions, seems suited the best to express the various policies that need to be monitored.

3.4 VeriMon: A Verified Monitor for MFODL

VeriMon was first proposed by Schneider et al. [13] as a formally verified online monitor for a restricted fragment of MFOTL. It was later strengthened by Basin et al. [4] to a formally verified online monitor for a much more permissive fragment of MFODL. In this section, we explain how VeriMon functioned, before the changes of this thesis were added. We will base ourselves on the two papers mentioned before but will only explain the parts that are important to this thesis. We note that VeriMon had already been improved between the publishing of Basin et al. [4] and the writing of this thesis. Most importantly, a non-recursive let operator had been added. We will go over some of those changes as well.

A system's desired functionality can be specified using a set of properties that should hold. An online monitor continuously runs while the underlying system is running and checks if the negation of any of the stated properties occurs at any time. If a negated property occurs, the monitor will notify the user, as this means that the system violates the specified property. In VeriMon all properties must be specified using MFODL. To monitor the underlying system VeriMon continuously takes pairs of sets of events and timestamps and processes them. We call one such pair of sets of events and timestamps or time-points a step. VeriMon allows for empty steps, where the set of events is empty and there may be multiple consecutive steps with the same timestamp. VeriMon assumes that all timestamps are in order, thus, if the following step has an earlier timestamp, the monitor output is not well-defined. In the following, we discuss the details of VeriMon that are relevant to this thesis. We split the discussion into four parts Formula, Monitor, Progress, and Correctness.

3.4.1 Formula

We start by describing the data types that are used in VeriMon. Each event is a pair of a string (its name) and a list of event_data (its content), where event_data can be an integer, a double, or a string. Multiple events that occur at the same time-point can be grouped into a database. The timestamps (ts) are modeled as natural numbers. The infinite stream of events and their corresponding timestamps is called the trace. The timestamps in the trace are ordered increasingly. The data type env describes a list of event_data. Furthermore, we define a table data type that is a tuple set of a given

type. Lastly, there is the formula data type that specified all formula operators that are supported by VeriMon. Each formula can contain free and bound variables. In VeriMon free variables are each given an index to identify them. The function fv takes a formula and returns the set of free variables in that formula. The function nfv takes a formula and returns the largest free variable index in that formula plus one. This can be different from the number of free variables if some indices are skipped. However, for simplicity, we assume that no indices are skipped and nfv returns the number of free variables in a function.

Definition 12 (VeriMon Data Types)**.** *The type_synonyms that VeriMon uses and that will be used in this thesis.*

datatype *even_data* = EInt *integer* | EFloat *double* | EString *string*
type_synonym *name* = *string*
type_synonym *event* = (*name* × *event_data list*)
type_synonym *database* = (*name, event_data list set list*) mapping
type_synonym *ts* = *nat*
type_synonym *trace* = (*name* × *event_data list*) *trace*
where $'a$ *trace creates an ordered stream of type* ($'a$ × *nat*)
type_synonym *env* = *event_data list*
type_synonym *'a table* = *'a tuple set*

Note: $'a$ *is a placeholder that can be initialized with any arbitrary data type.*

Definition 13 (formula)**.** *The definition of the formula data type in VeriMon. This describes all operators present before the new operators of this thesis were added:*

datatype *formula* = Pred *name "trm list"* | Let *name formula formula*
| Eq *trm trm* | Less *trm trm* | LessEq *trm trm* | Neg *formula*
| Or *formula formula* | And *formula formula* | Ands *"formula list"*
| Exists *formula* | Agg *nat agg_op nat trm formula*
| Prev *I formula* | Next *I formula*
| Since *formula I formula* | Until *formula I formula*
| MatchF *I "formula regex"* | MatchP *I "formula regex"*

Notes:
trm are variables constants and operators that can be performed on them.
agg_op are aggregation operators.
I is a non-empty interval.
formula regex is a regular expression consisting of atoms (atms) that use the type formula
We omit a full definition of these terms here as they are not important.

The property that should be monitored is given to VeriMon as a formula of type *formula*. VeriMon recursively computes all satisfying valuations, i.e., assignments of data to free variables, at each time-point for the given formula. There may be an infinite number of satisfying valuations. For instance, the formula $\neg p(x)$ has an infinite number of satisfying valuations as there is an infinite number of possible values for x and only for a finite number of those $p(x)$ can be true. VeriMon only operates on fragments of MFODL for which the number of satisfying valuations is finite. The function safe_formula describes for each operation what must hold so that the number of valuations stays finite. Some formulas, such as Next require information about future events to be evaluated. However, a formula that needs information that is infinitely far in the future to be evaluated, cannot be monitored. For this reason, there is a function called future_bounded that checks for a formula if all information needed is finitely far in the future. The function sat evaluates if for a given trace, time-point, and formula a given valuation is satisfying. Additionally sat takes an input that defines specific assignments for certain values.

Example 14 (Infinite Satisfying Valuations)**.** *We show an example of a formula that has an infinite number of satisfying valuations*

$$\neg p(x)$$

There is an infinite number of values that could be used for x and at any point, for only finitely many of those x, the predicate $p(x)$ can be true. Thus, the formula $\neg p(x)$ will always have an infinite number of satisfying valuations.

Definition 15 (safe_formula)**.** *Function header for safe_formula, where the detailed implementation is omitted. The function safe_formula takes a formula(ϕ) as input and outputs true if this formula only has a finite number of satisfying valuations and false otherwise. As an example, the function is shown for the let formula.*

fun *safe_formula :: "formula ⇒ bool" where*
...
"safe_formula (Let $p\ \phi\ \psi$) =
$\{0.. < nfv\ \phi \subseteq fv\ \phi\} \wedge$ *safe_formula* $\phi \wedge$ *safe_formula* ψ*"*
...

Definition 16 (future_bounded)**.** *Function header for future_bounded, where the detailed implementation is omitted. Future_bounded takes a formula(ϕ) as input and outputs true if all information needed to evaluate this formula is finitely far in the future. As an example, the function is shown for the let formula.*

...
fun *future_bounded :: "formula ⇒ bool" where*
"future_bounded (Let $p\ \phi\ \psi$) = *future_bounded* $\phi \wedge$ *future_bounded* ψ*"*
...

Definition 17 (sat)**.** *Function header for sat, where the detailed implementation is omitted. Sat takes five inputs; the trace(σ), a mapping of specific predicate names and time-points to data values (V), a valuation (v), a time-point (i), and a formula (ϕ) and outputs true if v was a satisfying valuation and false otherwise. As an example, the function is shown for the let formula.*

fun *sat :: "trace ⇒ (name ⇀ nat ⇒ event_data list set) ⇒ env ⇒ nat ⇒ formula ⇒ bool" where*
...
"sat $\sigma\ V\ v\ i$ (Let $p\ \phi\ \psi$) = *sat* σ ($V(p \mapsto \lambda\ i.\ \{v.\ length\ v = nfv\ \phi \wedge sat\ \sigma\ V\ v\ i\ \phi\})$) $v\ i\ \psi$*"*
...

Notes:
The function nfv ϕ denotes the number of free variables in ϕ.

3.4.2 Monitor

To explain the important aspects of the monitoring part, we first introduce the mformula data type. The monitor evaluates the given formula for each step and notifies if a problematic property ever occurs. It is, however, not always possible to evaluate the entire formula by just looking at one time-point. For instance, to evaluate if the formula "Next $p(x)$" holds at time-point i it is necessary to observe an event at a time-point greater or equal to $i+1$. Similarly, to evaluate the formula "Previous $p(x)$" at time-point i it is necessary to remember what the evaluation of $p(x)$ was at time-point $i-1$. Thus, it is important to have a data type that allows storing such important information easily, which is exactly what mformula achieves. The mformula data type holds the same operations as formula but gives each of them helper variables that are necessary for the step-by-step evaluation of formulas.

Example 18 (Formulas that need the Information of Multiple Time-Points)**.** *We show an example of two formula that need the information of multiple time-points to be evaluated accurately:*

Next $p(x)$

*To evaluate if "*Next $p(x)$*" is true at a given time-point i, it is necessary to observe events at a time-point greater or equal then $i+1$. If $p(x)$ occurs at time-point $i+1$, then "*Next $p(x)$*" evaluates to true, if it does not occur at that time-point or if there is no event at that time-point then "*Next $p(x)$*" evaluates to false.*

Previous $p(x)$

*To evaluate if "*Previous $p(x)$*" is true at a given time-point i, it is necessary to remember what the evaluation of $p(x)$ was at time-point $i-1$. If $p(x)$ occurred at time-point $i-1$ then "*Previous $p(x)$*" evaluates to true, if it did not occur at that time-point or there was no event at that time-point "*Previous $p(x)$*" evaluates to false.*

These two examples show that depending on the formula it is necessary to wait for more events before outputting anything respectively to keep past information for future use.

Definition 19 (mformula Datatype)**.** *The data type mformula is adjacent to formula. It implements the same operations as formula but gives helper variables that allow for easier step-by-step evaluation of formulas. The detailed description of mformula is omitted. We show the definition of MNext as an example. MNext has two helper variables, namely a Boolean that indicates if the first event has happened yet, or not and a "ts list" that saves the ts for which a verdict still needs to be made.*

```
datatype mformula =                                    ...
| MNext I mformula bool "ts list"
...
```

The properties that should be monitored are parsed into the formula data type by VeriMon. This formula must then be translated into the data type mformula, which is done by the function minit0 using recursive evaluation and by initializing all helper variables with their respective default value based on the number of free variables in the given formula.

Definition 20 (minit0)**.** *Function header for minit0, where the detailed implementation is omitted. The function minit0 takes the number of free variables in a formula (n) and that formula (ϕ) as inputs. It then recursively translates the formula into an mformula, initializing all helper variables with their default values based on the number of free variables. As an example, the function is shown for the let formula.*

```
function minit0 :: "nat ⇒ formula ⇒ mformula" where
  ...
  "minit0 n (Let p φ ψ) =
   MLet p (nfv φ) (minit0 (nfv φ) φ) (minit0 n ψ)"
  ...
```

After initializing the mformula, the monitor can evaluate it step-by-step on the input events and determine if any of the given inputs satisfy the mformula. This is done by the meval function, which evaluates an mformula using a database of input events and a list of ts. A feature of meval is that it can process multiple time-points at once, thus, it takes a list of ts as input instead of just the current ts. Furthermore, meval takes the number of free variables in the full mformula as input to make evaluations faster, as otherwise, the full mformula would have to be reconstructed and traversed every time this number was needed. The output of meval is a pair. The first element of that pair is a list of event_data tables, which represent the satisfying valuations of the given formula at all ts in the input ts list. The second element in the pair is an updated mformula, that will be used in subsequent steps. As described above, mformula is designed to save important information in its helper variables and that information needs to be updated in each step.

Definition 21 (meval)**.** *Function header for meval, where the detailed implementation is omitted. The function meval takes the number of free variables in the full mformula (n), a list of all time-stamps relevant for evaluating the current mformula (ts), a database of events on which to evaluate the current mformula (db) and the current mformula (ϕ) as inputs and outputs a pair of type event_data table list $\times$ mformula. The first element of that pair represents the evaluation results of the given formula at all given ts. The second element in the pair is an updated mformula, that will be used in*

subsequent steps, which is necessary as mformula stores important information in its helper variables. As an example, the function is shown for the let formula.

function *meval ::*
"nat ⇒ ts list ⇒ database ⇒ mformula ⇒ event_data table list × mformula" where
...
"meval n ts db (MLet *p m ϕ ψ*) =
(let (xs, ϕ) = meval j m ts db ϕ;
(ys, ψ) = meval j n ts (Mapping.update p (map (image (map the)) xs) db) ψ
in (ys, (MLet *p m ϕ ψ*))
...

3.4.3 Progress

As described above, for some formulas information about future events is needed before a verdict can be made. When proving properties of the meval function, it is necessary to keep track up until what time-point verdicts can be made. For this reason, the monitor has a function named progress, that tracks the latest time-point for which a verdict can be made. If no future operators are contained in a formula, the progress would coincide with the current time-point, however, if future operators are present, this is not the case. Progress evaluates a formula regarding the trace and the current time-point. As seen in the meval definition for MLet, the let formula is evaluated by first evaluating ϕ and then using the result of ϕ for p in ψ. Thus, in ψ the progress of p should coincide with the progress of p in ϕ, which is achieved using the P, which maps predicates to progress values. The output of progress is a natural number denoting the latest time-point for which a verdict can be made.

Definition 22 (progress)**.** *Function header for progress, where the detailed implementation is omitted. Progress takes a formula (ϕ), the trace (σ), and the current time-point (j) as input. For let formulas, the progress of the predicate p is computed using ϕ and then used in ψ. This can be achieved using the P that stores progress values for certain predicates. The output is a natural number that denotes the latest time-point for which a verdict for the given formula can be made. As an example, the function is shown for the let formula.*

fun *progress:: "trace ⇒ (name ⇀ nat) ⇒ formula ⇒ nat ⇒ nat" where*
...
"progress σ P (Let *p ϕ ψ*) *j* =
progress σ (*P*(*p* ↦ *progress σ P ϕ j*)) *ψ j"*
...

For a monitor to be useful, it must eventually output a verdict for every time-point. This is equivalent to stating that every progress value must be reached eventually. There are, however, formulas for which the monitor may not make any progress. For instance "Until I $p(x)$ $q(x)$", where I is an infinite interval, may run forever, if $p(x)$ never occurs and $q(x)$ always occurs. For this reason, we only allow formulas that are future_bounded to be monitored as those formulas only use finite intervals, which means that the monitor will always make progress.

Example 23 (Progressless Function)**.** *We show an example of a formula for which the monitor may never make any progress:*

$$\text{Until } I\ p(x)\ q(x)$$

If I is an infinite interval, $p(x)$ never occurs, and $q(x)$ always occurs, the monitor will never make any progress on this formula.

Definition 24 (progress_ge_gen)**.** *Lemma statement for progress_ge_gen, where the proof is omitted. This lemma states that for every formula ϕ that is future bounded, the monitor will eventually make*

progress.

lemma *progress_ge_gen: future_bounded* ϕ
$\Longrightarrow \exists\ P\ j.\ dom\ P = S \wedge range\ mapping\ i\ j\ P \wedge i \leq progress\ \sigma\ P\ \phi\ j$"
proof omitted

Notes:
"range_mapping i j P" describes that the value for every p in P is between i and j.
dom P returns a set of all p that occur in P.

3.4.4 Correctness

The main correctness proof of VeriMon is a lemma called meval, which shows that each evaluation step that VeriMon performs returns the intended results. This lemma requires that the various involved environments are well-founded (wf_envs) and that the invariant describing the monitor state (wf_mformula) is true.

Definition 25 (wf_mformula)**.** *Function header for wf_mformula, where the detailed implementation is omitted. The function wf_mformula inductively evaluates if the given inputs evaluate to true. The inputs are the trace, the current time-point, a mapping of predicate names to the latest time-point at which a verdict can be output for them, a mapping of a pair of predicate name and ts to the satisfying valuations of that predicate at that time-point, the number of free variables contained in the original formula, a mformula, and its corresponding formula.*

inductive *wf_mformula:: "trace* $\Rightarrow$ *nat* $\Rightarrow$ *(name* $\rightharpoonup$ *nat)* $\Rightarrow$ *(name* $\rightharpoonup$ *nat* $\Rightarrow$ *event_data list set)*
$\Rightarrow$ *nat* $\Rightarrow$ *mformula* $\Rightarrow$ *formula* $\Rightarrow$ *bool" for* σ *j where*
\\ *omitted*

Definition 26 (wf_envs)**.** *Definition of wf_envs.*

The function wf_envs takes the current trace (sigma), the current time-point (j), the time elapsed until the following time-point (δ), a mapping of predicate names to the latest time-point at which a verdict can be output for them for the current time-point (P), a mapping of predicate names to the latest time-point at which a verdict can be output for them for the next time-point (P′), a mapping of a pair of predicate name and ts to the satisfying valuations of that predicate at that time-point (V), and a database as inputs. The output is a Boolean that denotes if the given inputs create a well-founded environment.

definition *wf_envs::*
"trace $\Rightarrow$ *nat* $\Rightarrow$ *nat (name* $\rightharpoonup$ *nat)* $\Rightarrow$ *nat (name* $\rightharpoonup$ *nat)* $\Rightarrow$ *database* $\Rightarrow$ *bool" where*
"wf_envs σ *j* δ *P P′ V db =*
(dom V = dom P $\wedge$
Mapping.keys db $\supseteq$ *dom P* $\cup$ $(\bigcup k \in \{j..< j + \delta\}.\ \{p.\ p \in fst\ `\ \Gamma\ \sigma\ k\ \}) \wedge$
rel_mapping P P' ∧
pred_mapping $(\lambda i.\ i \leq j)$ *P* $\wedge$
pred_mapping $(\lambda i.\ i \leq j + \delta)$ *P' ∧*
$(\forall p \in$ *Mapping.keys db - dom P. the (Mapping.lookup db p)*
= map $(\lambda k.\ \{ts.\ (p,\ ts) \in \Gamma\ \sigma\ k\})\ [j..< j + \delta]) \wedge$
$(\forall p \in$ *dom P.*
list_all2 $(\lambda\ i\ X.\ X =$ *the (V p) i)[the (P p)..< the (P' p)](the (Mapping.lookup db p))))*

Notes:
The function rel_mapping P P′ denotes that the same predicate names are contained in P and P' and that the respective value for a given predicate is always smaller or equal in P.
The function pred_mapping a P denotes that for every predicate name in P its respective value satisfies the function a.
The $\Gamma\ \sigma\ k$ *function returns the k-th event that occurs in the trace* σ*.*
The dom P function returns a set of all p that occur in P.

VeriMon's main correctness proof is an invariant that must hold for any evaluation step. It states that the result of meval using the next time-point, all ts between the current time-point and the next, any mformula, and any database is a pair that has two properties. The first element of the pair is a wf_mformula for the new time-point. The second element in the pair consists of the satisfying valuations for the time-points between the progress at the current time-point and the progress at the next time-point. Thus, for any valuation in the second element and their corresponding progress time-point, the sat function must evaluate to true. The preconditions we need to make this lemma work is that the formula on which we perform meval is a wf_mformula at the current time-point and that we have a wf_envs for the current time-point and the input database.

Definition 27 (lemma_meval)**.** *Lemma statement for meval, the main correctness proof of VeriMon, where the proof is omitted. This lemma describes that if there is a wf_envs at the current time-point and ϕ is a wf_mformula at the current time point then after calling meval on ϕ, ϕ will be a wf_mformula for the new time-point and all results of meval will be satisfying valuations.*

lemma *meval:*
assumes *"wf_mformula σ j P V n ϕ ϕ'" "wf_envs σ j δ P P' V db"*
shows *"case meval $(j+\delta)$ n (map $(\tau\ \sigma\ )[j..<j+\delta\ ])$ db ϕ of $(xs,\ \phi_n)$*
$\Rightarrow$ wf_mformula σ $(j+\delta)$ P' V n ϕ_n ϕ' $\wedge$
list_all2 $(\lambda\ i.\ qtable\ n\ (fv\ \phi')\ (\lambda\ v.\ sat\ \sigma\ V\ (map\ the\ v)\ i\ \phi'))$
[progress σ P ϕ' $j..<$ progress σ P' ϕ' $j+\delta$]"
\\proof omitted

Notes:
The τ σ k function returns the timestamp of the k-th element in the trace σ.
the function qtable n A P X describes that n A X is a table and that any x for which P x is true, the length of x is equal to n, and all indices for which x has no value do not occur in A is an element of the list X.

3.5 Notation

In this section, we explain the notation that will be used in this thesis. Table 1 shows all operators that will be used for the examples in this thesis, with their respective explanations and their signature in both VeriMon style and a more commonly used style. In this thesis we use both styles interchangeably, choosing the one that makes the formulas clearer or shorter. We note that the eventually operator does not exist in VeriMon, but it can be written using "Until $\phi\ I\ \top$". Furthermore, we note that not all operators supported by VeriMon are shown in the list, but only those that will be used in examples.

Unless otherwise noted, we will drop all intervals in examples and assume those intervals to be $[0; \infty]$. Pred p $trms$ is a predicate with name p, however, we may shorten this by simply saying the predicate p. We say that a predicate p is contained in/below a formula ϕ if either ϕ is that predicate (i.e., ϕ = Pred p $trms$) or if ϕ has a subformula in which p is contained (e.g., ϕ = AND ψ (Pred p $trms$)). Instead of saying that p is contained in ϕ, we can also say that ϕ is above p.

The time-points we use are 0-indexed, which means that the first time-point is time-point 0, many variables in VeriMon denote the number of time-points lapsed instead of the time-point. We will call the number of time-points elapsed, the input length. The current input length is always one larger then the current time-point, and thus, the input length 0 translates to the time-point -1.

VeriMon Style	Explanation	Alternative Style
Pred p $trms$	A predicate with name p that uses the variables in the $trms$ list	$p(trms)$
Eq $t1$ $t2$	$t1$ is equal to $t2$	$t1 = t2$
Less $t1$ $t2$	$t1$ is less than $t2$	$t1 < t2$
LessEq $t1$ $t2$	$t1$ is less or equal to $t2$	$t1 \leq t2$
Let p ϕ ψ	Every occurrence of p in ψ will be exchanged with ϕ	let $p = \phi$ in ψ
Neg ϕ	Negation of the formula ϕ	$\neg\ \phi$
Or ϕ ψ	Logical or between the formulas ϕ and ψ	$\phi \vee \psi$
And ϕ ψ	Logical and between the formulas ϕ and ψ	$\phi \wedge \psi$
Exists ϕ	There exists a x such that ϕ holds	$\exists\ x.\ \phi$
Prev I ϕ	There was another event in the time interval I for the previous event ϕ holds	-
Next I ϕ	The next event will happen within the time interval I and for the next event ϕ holds	-
Since ϕ I ψ	ϕ was true somewhere in the interval I and for all following time-points in the interval I ψ is true	-
Until ϕ I ψ	ϕ is true somewhere in the interval I and for all previous time-points in the interval I ψ is true	-
Eventually I ϕ	ϕ is true somewhere in the interval I	-
Let p ϕ ψ	Every occurrence of p in ψ is replaced with ϕ	let $p(x, y, ...) = \phi$ in ψ

Table 1: Notation of formulas that will be used.

4 Recursive Let

In this section, we give an overview of the recursive let operator. We start by giving an idea for what such an operator could be used. Then, we define the operator and how it is supposed to function. After that, we discuss the issue of infinite recursion and under what circumstances it can occur. Then, we propose two possible recursive let operators that do not suffer from the described issue. Lastly, we compare the two operators to illustrate each of their strengths and weaknesses. We will not discuss anything regarding the monitoring algorithm and its correctness proofs here, but leave this for later sections. However, understanding the theoretical foundation will help to understand the design choices we made in the implementation.

4.1 Use Cases

Before getting into the details of the recursive let operator, we take a step back to explore some policies that can only easily be described using a recursive operator. We first consider a very simple example that uses straightforward recursion 28. We assume a network of nodes, which each have one of three safety labels; safe, unsafe, and neutral. The underlying system continuously links and unlinks nodes. We say that two nodes are connected via a path, if they are either linked together or if one node is linked to an intermediary node, that is in turn connected via a path to the other node. The entire system is in an unsafe state as soon as a safe node is connected via a path to an unsafe node, which is a critical problem and should never happen. A monitor that checks if the system enters an unsafe state at any time would receive a list of all currently active links as input from the system and would evaluate the formula that describes an unsafe state for the system. The formula would need to contain two parts. The first part entails the computation of all paths using the received links, which needs some form of recursion. The second part checks for every found path if one of the end nodes was safe and the other was unsafe. This example shows us that there are properties that rely on recursion to be easily defined and that may be useful to monitor.

Example 28 (Monitoring Paths Between Linked Nodes)**.** *We assume a network of nodes, which can each be safe, unsafe, or neutral. At each time-point, some pairs of nodes are connected via links. There is a path between two nodes a and b, if there is either a link between a and b or there is a link between a and an intermediary node c and a path between c and b. Paths between safe and unsafe nodes are prohibited. A monitor that checks for violations of this property would need to evaluate some formula akin to the following one:*

$$\begin{aligned}&\text{Let } \textit{"path"}\\ &\quad \textit{"path}(a,\ b) \vee (\exists\ c.\ \textit{link}(a,c) \wedge \textit{path}(c,\ b)\\ &\quad \textit{"path}(a,\ b) \wedge (\textit{safe}(a) \wedge \textit{unsafe}(b) \vee \textit{unsafe}(a) \wedge \textit{safe}(b))\textit{"}\end{aligned}$$

All paths between nodes are computed recursively, after which it can be determined if any path connects a safe node to an unsafe node.

The second example we consider is a bit more involved but the property used in the example is taken directly from the Linux kernel 29. A Linux [11] system continuously holds tasks that it must complete. However, it may not be possible for all tasks to be handled at the same time, thus, Linux must schedule which task gets processed when. To facilitate this, each task has a priority based on their deadline, the time at which the task must be completed at the latest. Some system resources may only be accessed by one task at a time. Such resources are located in a critical section, which only one task can enter at a time. If a task has entered the critical section any other task that wants to access the resource must wait at the entrance of the critical section until the other task leaves the critical section regardless of their priority. This may lead to issues, for instance, when a low priority task is in the critical section and a high priority task is waiting at the entrance of the critical section. Medium priority tasks would be scheduled before the low priority task, which would indirectly also stop the high priority task, even though the high priority task is more important. To solve this issue, tasks in a critical section inherit the priority of another task waiting at the entrance of that critical section if that task has a higher priority. If a task inherits a priority, it is put into a boosted state. Tasks in a boosted state can be deboosted, at which point they return to their original priority unless they get boosted again. Priorities may change every time-point, however, the priority of the current time-point must be calculated using the priority information of the previous time-point. A monitor

that checks if a task t has the correct priority d needs to evaluate three properties. Is the priority of t boosted to the priority of another task t' and did t have the priority d in the last time-point? Was t deboosted and not boosted again in the meantime and was the priority of t before being boosted d? Was the priority of t in the previous time-point d and was t neither boosted nor deboosted? If one of these three questions, two of which use recursion to be evaluated, can be answered with yes, then t has the correct priority d. This example shows two things. Firstly, it is a real-world example of a property that would be useful to monitor but needs recursion to be defined easily, which indicates the usefulness of a recursive operator for monitors. Secondly, the predicate that is being evaluated recursively is always strictly in the past in this formula, as to evaluate the current priority, only priorities strictly in the past are necessary. This justifies the creation of LetPrev, the past-only recursive operator.

Example 29 (Monitoring Linux Kernel Priority (Deadline) Inheritance [11])**.** *Certain resources may only be accessed by one task at a time, such resources are located in a so-called critical section. If a task is in the critical section, no other task may enter the critical section until the other task exits the critical section. Each task has a priority, where higher priority tasks are scheduled before lower priority tasks. However, when a high priority task is waiting for a low priority task to leave the critical section, then it can happen that the high priority task is indirectly blocked by a medium priority task because the low priority task that is in the critical section is scheduled after the medium priority tasks. This is undesirable, as the high priority task may be stopped indefinitely by tasks with a lower priority. Priority (deadline) inheritance is a way to circumvent this issue. A task in a critical section inherits the priority of a task waiting at the entrance of the critical section if that task has a higher priority and enters a boosted state. A task can also get deboosted from a boosted state. A monitor that checks for a given task if it has the correct priority would need to evaluate something akin to the following property:*

Let *"priority"*
"$(\exists\ t'.\ pi\_boosted(t,t') \wedge (Prev\ priority(t',d))) \vee$
$(pi\_deboosted(t) \wedge$
$(((\nexists\ t'.\ pi\_boosted(t,t'))\ SINCE\ ((\exists\ t'.\ pi\_boosted(t,t')) \wedge (Prev\ priority\ (t,d))))))) \vee$
$((Prev\ priority(t,d)) \wedge \neg\ (\exists\ t'.\ pi\_boosted(t,t')) \wedge \neg\ pi\_deboosted(t)) \vee$
$arrive(t,d)$*"*
"$priority(t,d)$*"*

The priority d of a task t is computed using four cases. We omit a discussion of the arrive predicate as it is not important for the recursive evaluation. If t is boosted to the priority of task t' and t' had the priority d at the previous time-point, then t currently has the priority d. If task t was deboosted and was not boosted to the priority of any other task in the meantime then t currently has the priority that t had before being boosted. If t was not boosted to the priority of any other task and was not deboosted, then t has the same priority as it had at the previous time-point.

4.2 Definition

The definition of the recursive let operator is very similar to the non-recursive let operator described in Section 3.1. However, there are some subtle differences. The operator looks as follows "Let $p\ \phi\ \psi$", where p is a predicate name, and ϕ and ψ are formulas. Intuitively, this operator indicates that every occurrence of p in ψ is substituted with ϕ. In the recursive version of the let operator any p in ϕ comes from the context within the let operator. This means, that they need to be evaluated recursively. In the following we show the formal definition of the recursive let operator:

Definition 30 (Recursive Let [14])**.** *The semantics are a relation $(\sigma, S, i, v) \models \phi$, where σ and S are temporal structures, i is the evaluation time-point, and v is a set of valuations. S is a relation of the form $S(p, i, a)$ that id true iff the predicate symbol p at time-point i contains the tupel a. With this we get the following semantics:*

$$(\sigma, S, i, v) \models (Let\ p\ \phi\ \psi) \iff (\sigma, S', i, v) \models \psi$$
$$where:\ S'(q, j, a) \iff \begin{cases} (\sigma, S', j, a) \models \phi & if\ q = p \\ S(q, j, a) & otherwise \end{cases}$$

The only difference between this definition and Definition 1, is that the evaluation of S' again uses S'. This means that the evaluation of ϕ needs to be done recursively. This is no problem for the mathematical definition, however, an evaluation algorithm for the formula may run into issues. The recursive evaluation of ϕ would end as soon as one recursion step does not change the result set. Unfortunately, such a recursive evaluation may lead to an infinite recursion depending on the other operators contained in ϕ. Monitors cannot monitor a formula that needs an infinite recursion to be evaluated, as this operation would not terminate. Thus, it is important to define the operation in a way that such recursions cannot occur.

4.3 Infinite Recursion

In this section, we go over all operations that might lead to an infinite recursion. As stated above, the recursive let (Let $p\ \phi\ \psi$) is evaluated by first evaluating ϕ recursively and then evaluating ψ using the result of ϕ. Thus, only ϕ determines if the recursion ends or not, which is why we ignore ψ here. Furthermore, we keep in mind that the recursive evaluation of ϕ ends, when the result set of one recursive call is equal to the result set of the previous recursive call. This means that we search for operators that may change the result set in every recursive step.

The following four groups of operations may lead to an infinite recursion if they are contained in the ϕ of a recursive let:

1. Future operators (Next, Until, MatchF) that occur above p (See 3.5)
2. Negation operators (Neg) that occur above p
3. Equality operators (Eq, LessEq, Less)
4. Aggregation operators (Agg)

To illustrate the issue with future operators we use the example where $\phi = \text{Next } p(x)$. In the first recursion step $p(x)$ is evaluated using "Next $p(x)$". In the second recursion step $p(x)$ is evaluated using "Next (Next $p(x)$)". Every further step follows the same pattern, which means that throughout the whole recursion, in every step one next operation is added to the evaluation. As there is no final future point, this recursion continues indefinitely.

Example 31 (Infinite Recursion with Future Operators)**.** *We use the following formula to illustrate how future operators can lead to an infinite recursion:*

$$\phi = Next\ p(x)$$

The recursive evaluation of ϕ progresses as follows:

Step 0:
$$p(x) = p(x)$$
Step 1:
$$p(x) = \text{Next } p(x)$$
Step 2:
$$p(x) = \text{Next (Next } p(x))$$
Step 3:
$$p(x) = \text{Next (Next (Next } p(x)))$$
$$\dots$$

As there is no final future point, this evaluation will never end.

For the second issue, we use the example where $\phi = \text{Neg } p(x)$. In the first recursion step $p(x)$ is evaluated using "Neg $p(x)$". In the second recursion step $p(x)$ is evaluated using "Neg (Neg $p(x)$)" which is equivalent to $p(x)$. This continues throughout the recursion, where the result of $p(x)$ flips between the evaluation of "$p(x)$" and "Neg $p(x)$" in every step. Since there is no clear termination point, it is impossible to decide which of the two should be used.

Example 32 (Infinite Recursion with Negation)**.** *We use the following formula to illustrate how negation can lead to an infinite recursion:*

$$\phi = Next\ p(x)$$

The recursive evaluation of ϕ progresses as follows:

Step 0:
$$p(x) = p(x)$$
Step 1:
$$p(x) = \text{Neg } p(x)$$
Step 2:
$$p(x) = \text{Neg } (\text{Neg } p(x)) = p(x)$$
Step 3:
$$p(x) = \text{Neg } (\text{Neg } (\text{Neg } p(x))) = \text{Neg } p(x)$$
$$\dots$$

The evaluation keeps flipping between $p(x)$ and Neg $p(x)$*.*

The third issue must be illustrate a bit differently than the other two. To illustrate this example we use $\phi = (x = 0) \vee p(x - 1)$. At first, the result set of p is empty. In the first recursion step, the result set would only contain $\{p(0)\}$ as for that term the left side of the or holds. In the next step, we would get a result set containing $\{p(0), p(1)\}$. $p(0)$ remains there because of the left side of the or. $p(1)$ now comes from the right side of the or and the fact that $p(0)$ holds and can be used to evaluate this term. In each further recursion step, the result set will increase by one element, as there will always be exactly one new x for which $p(x-1)$ holds. Thus, this recursion would construct the set of all natural numbers one number at a time and since there are infinitely many natural numbers, this recursion is infinite.

Example 33 (Infinite Recursion with Equality)**.** *We use the following formula to illustrate how equality operators can lead to an infinite recursion:*

$$\phi = \ (x = 0) \vee p(x - 1)$$

The result set of the recursive evaluation of ϕ progresses as follows:

Step 0:
$$\{\}$$
Step 1:
$$\{p(0)\}$$
Step 2:
$$\{p(0),\ p(1)\}$$
Step 3:
$$\{p(0),\ p(1),\ p(2)\}$$
$$\dots$$

The evaluation keeps adding $p(x)$, where $x - 1$ is the largest index for a p currently in the list, to the list. This constructs the list of p for all indexes in the natural numbers, which are infinitely large.

The last issue can also be illustrated using the result set. We use the example $\phi =$ Sum $(p(x) \vee q(y))$ for this case and assume that the event $q(1)$ was observed. At first, the result set of p is empty. In the first recursion step, the result set would contain $\{p(1)\}$ because the sum of $q(1)$ is 1. In the next step the result set would contain $\{p(2)\}$ because the sum of $q(1)$ and $p(1)$ is 2. In each further recursion step, the result in the result set will shift by one, as one is always added to the current x. Thus, this recursion would visit all natural numbers one number at a time and since there are infinitely many natural numbers, this recursion is infinite.

Example 34 (Infinite Recursion with Aggregation)**.** *We use the following formula to illustrate how aggregation operators can lead to an infinite recursion:*

$$\phi = \ \phi = \mathit{Sum}\ (p(x) \vee q(y))$$

The result set of the recursive evaluation of ϕ progresses as follows:

Step 0:
$\{\}$
Step 1:
$\{p(1)\}$
Step 2:
$\{p(2)\}$
Step 3:
$\{p(3)\}$
$\dots$

The evaluation keeps shifting the $p(x)$ by one. This sequentially visits all natural numbers, of which there are infinitely many.

After discussing all operators that might lead to problems and showing why they can lead to problems, we take a moment to show that these operators may be harmless when combined with other operators. A simple way to illustrate this would be by using $\phi =$ Prev (Next $p(x)$). In this formula there is a next operator above the p, however, there would be no need for an infinite recursion to evaluate this term, as it is equivalent to $\phi = p(x)$. Furthermore, even though a class of operators may lead to problems, not all operations using this class are necessarily problematic. For instance, equality operations that do not contain variables, cannot lead to infinite recursion, even though equality with variables can. This shows us that it may be possible to evade some of the infinite recursion issues, by crafting the formulas in a specific way. In fact, one of our recursive let operators, LetPrev, will make use of the fact that the last three issues no longer apply, if p always occurs strictly in the past in ϕ.

4.4 LetPrev: Enforcing Strict Pastness

The first recursive let operator, LetPrev, is based on the fact that infinite recursion cannot occur if any p contained in ϕ is strictly in the past. The reason for this is that there is a specific first time-point, namely 0. If p occurs strictly in the past, every recursion step would bring the evaluation of p closer to the evaluation of p at time-point 0 and the semantics of past operators at time-point 0 would stop the recursion. This will always happen in a finite number of steps.

Thus, our task shifts to enforcing that p always occurs strictly in the past in ϕ. To this end, we assume an implicit previous operator around any p that occurs in ϕ. This means that a formula written as "LetPrev p $p(x)$ ψ" actually translates to "LetPrev p (Prev $p(x)$) ψ". This alone is, unfortunately, not enough, as future operators might destroy the strict pastness of p. For instance in "LetPrev p (Next $p(x)$) ψ" p is no longer strictly in the past. It is, thus, necessary to additionally exclude any future operators above p.

LetPrev is similar to the recursive let operator found in DejaVu [10]. However, there are some differences such as that our operator allows for future operators in the recursive formula as long as they do not contain the recursive predicate.

4.5 LetRec: Exclude all Problematic Operators

The second recursive let operator, LetRec, is more closely related to the points mentioned in Section 4.3. LetRec does not implicitly change to form of the formula, but simply disallows any of the four groups of operations mentioned in Section 4.3. As these are the only operations that can lead to infinite recursion, excluding them is sufficient. However, this means that we do not necessarily move towards the past in each recursion step, and thus, we need a different termination condition for the recursion. The semantics of the operator are defined using a counter variable that gets decreased in every recursion step and ends at 0. Using such a construction trivially makes the recursion terminate,

however, it is not clear what number this counter should initially be and if it is too small, then the recursion will end, before the actual result set was found, which is problematic. Thus, we take the final result set as the union of all result sets of all possible starting counters. This method will yield the correct result set if the recursion is finite, however, computing infinitely many sets and then taking their union takes infinitely much time. In the monitor, we, thus, use a different termination condition, namely that a fixpoint has been reached. A fixpoint is a set of results, for which the previous recursion step already had the same set of results, and thus, any further recursion step will again generate the same set of results. If there exists a fixpoint then the two methods yield the same result set but proving this is not trivial.

LetRec is based on the recursive evaluation of database queries [1]. However, there are many differences as database queries do not use temporal operators.

4.6 LetPrev vs. LetRec

At a first glance, it may seem like LetRec is strictly stronger than LetPrev, as it does not assume a previous operator around every occurrence of p in ϕ. However, this is not true. Each of these operators can be used to evaluate formulas that the other cannot. To show a formula that LetPrev can evaluate but LetRec cannot we use the example of $\phi = \text{Neg (Prev } p(x))$, which is a formula that can be evaluated without needing infinite recursion due to the previous operator. LetRec cannot evaluate this formula as it specifically disallows any negation operators that occur above p. On the other hand, LetPrev can evaluate this formula. It can be written as "LetPrev p (Neg $p(x)$) ψ", where the previous operator is added implicitly and which does not violate any rules of LetPrev. For an example of a formula that LetRec can evaluate and LetPrev cannot, we can use $\phi = \text{And } p(x)$ $q(x)$, which can be evaluated without needing infinite recursion. This formula cannot be evaluated by LetPrev, as there is no previous operator in front of the $p(x)$. It can, however, be evaluated by LetRec using "LetRec p (And $p(x)$ $q(x)$) ψ", which does not violate any rules of LetRec.

This shows us that there can be multiple different recursive let operators that are equally strong but serve different purposes. It may seem like it would be better to merge these two operators into one, that performs both tasks, but this would be difficult as they use two different algorithms. Thus, leaving both versions intact and using the appropriate one for any given task is the better option.

5 Recursive Let with Implicit Previous Operator (LetPrev) in VeriMon

In this section, we show how we formalized the LetPrev operator in Isabelle. We will describe the data types, functions, and proofs we updated and will introduce any new lemmas created for the proofs. This section will follow the structure of section 3.4. Wherever possible, we will omit anything we did not update. We will use the notation "LetPrev p ϕ ψ" for the LetPrev operator. Any mention of ϕ, ψ, and p refers to this notation unless otherwise noted.

5.1 Formula

The LetPrev formula is implemented by enforcing that the recursive predicate is strictly in the past as described in Section 4.4. LetPrev takes three arguments p, ϕ and ψ. ϕ may contain p, but any instance of p in ϕ is implicitly surrounded by a Prev operator. Any instance of p in ψ needs to be replaced by the result of evaluating ϕ recursively for p.

Definition 35 (LetPrev formula)**.** *The definition of the formula data type for LetPrev. (See Definition 13 for other cases.)*

datatype *formula* = ...
| LetPrev *name formula formula*
...

Due to the definition of LetPrev, any p contained in LetPrev is different from any p outside of the LetPrev context. To illustrate this we use the following two formulas (Also shown in Example 36): "ϕ_1 = Prev $p(x)$" and "ϕ_2 = LetPrev p $p(x)$ $p(x)$". It may seem at first glance that these two formulas are equivalent. The $p(x)$, which constitutes the ϕ of the LetPrev formula, is implicitly surrounded by a Prev operator. Furthermore, substituting the p in $p(x)$, which constitutes the ψ in the LetPrev formula, with ϕ would result in "Prev $p(x)$", which is equivalent to ϕ_1. However, ϕ must be evaluated recursively **before** it is used in ψ. To illustrate how this makes a difference, we assume that we get the event "$p(1)$" at time-point 0. Thus, at time-point 1 ϕ_1 would be satisfied for the variable x set to 1. However, evaluating ϕ_2 at time-point 1 returns a different result. First, "$p(x)$", which is actually "Prev $p(x)$" is evaluated recursively for p. After one recursion step, the formula equates to "Prev (Prev $p(x)$)", which is the value of "$p(x)$" at the time-point -1, which in turn is defined as $\{\}$. Thus, for our example ϕ_2 is not satisfied for any assignment of x at time-point 1. In fact, ϕ_2 will never be satisfied, as the ϕ of that LetPrev will always evaluate to $\{\}$.

Example 36 (Context of p in LetPrev)**.** *This example shows how any p contained in* LetPrev *is different from any p outside of the* LetPrev *context. We use the following two formulas to illustrate this:*

$$\phi_1 = Prev\ p(x)$$
$$\phi_2 = LetPrev\ p\ p(x)\ p(x)$$

We assume the event $p(1)$ at time-point 0. At time-point 1 ϕ_1 is satisfied for the variable x set to 1. However, evaluating the ϕ of LetPrev *of ϕ_2 recursively, would result in "Prev (Prev $p(x)$)" after one recursion step, which is the value of $p(x)$ at time-point -1, which in turn is defined as $\{\}$. Thus, at time-point 1 ϕ_2 is not satisfied for any assignment of the variable x, which means that ϕ_1 and ϕ_2 are not equivalent.*

Not all formulas created using the above-mentioned formula types are monitorable, thus the limitations must be defined. The safe_formula function defines one of those limitations and currently ensures that there are only a finite number of satisfying valuations for any given formula. However, when adding recursion, it is also necessary to ensure that every satisfying valuation can be found in a finite amount of time. For LetPrev, the following condition ensures that the recursion is finite: there is no future operation above the predicate p in ϕ. This definition is, as described in Section 4.3, an overestimation of the cases that may lead to an infinite recursion, but still provides good flexibility while not being overly complicated. To specify this property, we must first find a way to evaluate if a given operator occurs above some given predicate, which is equivalent to evaluating if a given

predicate occurs below a given operator. The function contains_pred shown in Definition 37 evaluates if the predicate p occurs below the function ϕ. Using contains_pred, we can define the function safe_letprev 38, which checks that no predicate with name p is contained below a future operator in ϕ, and thus, ensures that the recursive evaluation of LetPrev is finite. The safety condition for LetPrev shown in Definition 39 is a combination of the finite recursion property, the safety of the sub-formulas, and that no variable indexes are missing in ϕ.

For both contains_pred and safe_letprev the Let and LetPrev cases are complicated and slightly different, which is why we investigate contains_pred in detail (safe_letprev functions analogously). In LetPrev the first conjunct states that "$p \neq e$". If e and p share a name, all occurrences of p in ϕ will be interpreted as coming from the context within LetPrev, which is different than the predicate p which comes from the context outside of LetPrev, and any occurrence of p in ψ will be substituted, thus, LetPrev would not contain the predicate p. This is not the case for Let as all occurrences of p in ϕ are interpreted as coming from the context outside of the Let. The second conjunct is the same for both Let and LetPrev. If the predicate p appears in ϕ then any occurrence of e in ψ is substituted with a formula containing p, thus, if ψ contains e then the resulting formula contains the predicate p. Lastly, ψ may directly contain the predicate p. For Let, it is necessary to state that $p \neq e$ as otherwise the p would be overwritten with ϕ. This is not needed for LetPrev, as the first conjunct already states that assumption.

Definition 37 (contains_pred)**.** *Function definition for contains_pred. the contains_pred function takes a name (p) and a formula (ϕ) as input and outputs true if ϕ contains a predicate with the name p and false otherwise.*

fun *contains_pred :: "name ⇒ formula ⇒ bool" where*
"contains_pred p (Eq *t1 t2*) = *False"*
| *"contains_pred p* (Less *t1 t2*) = *False"*
| *"contains_pred p* (LessEq *t1 t2*) = *False"*
| *"contains_pred p* (Pred *e ts*) = (*e*=*p*)*"*
| *"contains_pred p* (Let *e* ϕ ψ) =
(*contains_pred p* ϕ ∧ *contains_pred e* ψ) ∨ ($p \neq e$ ∧ *contains_pred p* ψ))*"*
| *"contains_pred p* (LetPrev *e* ϕ ψ) =
($p \neq e$ ∧ ((*contains_pred p* ϕ ∧ *contains_pred e* ψ) ∨ *contains_pred p* ψ))*"*
| *"contains_pred p* (Neg ϕ) = *contains_pred p* ϕ*"*
| *"contains_pred p* (Or ϕ ψ) = (*contains_pred p* ϕ ∨ *contains_pred p* ψ)*"*
| *"contains_pred p* (And ϕ ψ) = (*contains_pred p* ϕ ∨ *contains_pred p* ψ)*"*
| *"contains_pred p* (Ands *l*) = (∃ ϕ ∈ *set l*. *contains_pred p* ϕ)*"*
| *"contains_pred p* (Exists ϕ) = *contains_pred p* ϕ*"*
| *"contains_pred p* (Agg *y* ω b' *f* ϕ) = *contains_pred p* ϕ*"*
| *"contains_pred p* (Prev *I* ϕ) = *contains_pred p* ϕ*"*
| *"contains_pred p* (Next *I* ϕ) = *contains_pred p* ϕ*"*
| *"contains_pred p* (Since ϕ *I* ψ) = (*contains_pred p* ϕ ∨ *contains_pred p* ψ)*"*
| *"contains_pred p* (Until ϕ *I* ψ) = (*contains_pred p* ϕ ∨ *contains_pred p* ψ)*"*
| *"contains_pred p* (MatchP *I r*) = (∃ ϕ ∈ *atms r*. *contains_pred p* ϕ)*"*
| *"contains_pred p* (MatchF *I r*) = (∃ ϕ ∈ *atms r*. *contains_pred p* ϕ)*"*

Definition 38 (safe_letprev)**.** *Function header for safe_letprev. The safe_letprev function takes a name (p) and a formula (ϕ) as input and outputs true only if evaluating p recursively using phi cannot lead to an infinite recursion assuming that every p occurring in ϕ is surrounded by a Prev*

operator. As discussed in Section 4.3 this is an over approximation.

fun *safe_letprev :: "name ⇒ formula ⇒ bool" where*
"safe_letprev p (Eq *t1 t2*) = *True"*
| *"safe_letprev p* (Less *t1 t2*) = *True"*
| *"safe_letprev p* (LessEq *t1 t2*) = *True"*
| *"safe_letprev p* (Pred *e ts*) = *True"*
| *"safe_letprevp* (Let *e φ ψ*) = (*safe_letprev p φ* ∧
(¬ *contains_pred p φ* ∨ *safe_letprev e ψ*) ∧ (*p* = *e* ∨ *safe_letprev p ψ*))"
| *"safe_letprev p* (LetPrev *e φ ψ*) = (*safe_letprev e φ* ∧
p = *e* ∨ *safe_letprev p φ* ∧ (¬ *contains_pred p φ* ∨ *safe_letprev e ψ*) ∧ *safe_letprev p ψ*)"
| *"safe_letprev p* (Neg *φ*) = *safe_letprev p φ"*
| *"safe_letprev p* (Or *φ ψ*) = (*safe_letprev p φ* ∧ *safe_letprev p ψ*)"
| *"safe_letprev p* (And *φ ψ*) = (*safe_letprev p φ* ∧ *safe_letprev p ψ*)"
| *"safe_letprev p* (Ands *l*) = (∀ *φ* ∈ *set l*. *safe_letprev p φ*)"
| *"safe_letprev p* (Exists *φ*) = *safe_letprev p φ"*
| *"safe_letprev p* (Agg *y ω b′ f φ*) = *safe_letprev p φ"*
| *"safe_letprev p* (Prev *I φ*) = *safe_letprev p φ"*
| *"safe_letprev p* (Next *I φ*) = (¬ *contains_pred p φ* ∧ *safe_letprev p φ*)"
| *"safe_letprev p* (Since *φ I ψ*) = (*safe_letprev p φ* ∧ *safe_letprev p ψ*)"
| *"safe_letprev p* (Until *φ I ψ*) =
(¬ *contains_pred p φ* ∧ *safe_letprev p φ* ∧ ¬ *contains_pred p ψ* ∧ *safe_letprev p ψ*)"
| *"safe_letprev p* (MatchP *I r*) = (∀ *φ* ∈ *atms r*. *safe_letprev p φ*)"
| *"safe_letprev p* (MatchF *I r*) = (∀ *φ* ∈ *atms r*. ¬ *contains_pred p φ* ∧ *safe_letprev p φ*)"

Definition 39 (safe_formula for LetPrev)**.** *Safe_formula definition for LetPrev. (See Definition 15 for an explanation of this function.)*

fun *safe_formula where*
...
"safe_formula (LetPrev *p φ ψ*) =
safe_letprev p φ ∧ {*0.. < nfv φ* ⊆ *fv φ*} ∧ *safe_formula φ* ∧ *safe_formula ψ"*
...

Notes:
The function nfv φ denotes the number of free variables in φ.
The function fv φ are all free variables in φ.
The formula {*0.. < nfv φ* ⊆ *fv φ*} *ensures that there are no free variable indexes missing in φ. This is technically possible as free variable indexes can be skipped, but for simplicity sake, we do not allow it in the φ of LetPrev.*

Another limitation is defined using the function future_bounded. Until LetPrev was added, the future_bounded function only evaluated if all intervals contained in a formula were finite. However, in lemmas where future_bounded is used, the property needed is an intersection of finite recursion and finite intervals. As we wanted to preserve the structure of the existing lemmas as much as possible, we added safe_letprev to the definition of future_bounded for LetPrev (Definition 40) instead of modifying all lemmas that use future_bounded. This makes the naming of future_bounded a bit strange and it may seem convoluted to add safe_letprev to both safe_formula and future_bounded, but it is not a big sacrifice.

Definition 40 (future_bounded for LetPrev)**.** *Future_bounded definition for LetPrev. (See Defini-*

tion 16 for an explanation of this function.)

fun *future_bounded where*

...

"future_bounded (LetPrev p ϕ ψ) = *safe_letprev* p ϕ $\wedge$ *future_bounded* ϕ $\wedge$ *future_bounded* ψ*"*

...

Note: Until LetPrev was added future_bounded only ensured that all intervals used were finite. However, in lemmas where this property was used, we needed an intersection of finite intervals and finite recursion, which is why safe_letprev was added here.

The satisfying valuations for LetPrev can be found by first determining the satisfying valuations for ϕ recursively and then using those valuations for any predicate with name p in ψ as shown in Definition 42. To this end, we first evaluate ϕ and then save the satisfying valuations of ϕ under the name p in the V that will be used to evaluate ψ. In sat we also increment the input length for which we evaluate ϕ recursively, because of the implicit Prev operator around p. The input length is always one larger than the time-point, which means that in the recursive evaluation function letprev_sat, shown in Definition 41, the case for 0 corresponds to the time-point -1 instead of 0, and thus, can automatically be evaluated as $\{\}$. For all other cases, the length of the list representing the satisfying valuations must be equal to the number of free variables in ϕ, which comes from the fact that a valuation must assign all free variables to some data. Furthermore, the satisfying valuation will be determined by substituting p with the valuations from a function call to itself.

Definition 41 (letprev_sat)**.** *Definition of the letprev_sat function. letprev_sat takes the number of free variables in the full formula (n), a function with which we determine the satisfying valuations (sat) and a natural number for input length at which p is being evaluated (i) as input. It is used to recursively compute all satisfying valuations for the ϕ of LetPrev p ϕ ψ.*

fun *letprev_sat:: "nat* $\Rightarrow$ *((nat* $\Rightarrow$ *'b list set)* $\Rightarrow$ *'b list* $\Rightarrow$ *nat* $\Rightarrow$ *bool)* $\Rightarrow$ *nat* $\Rightarrow$ *'b list set" where*

"letprev_sat n sat 0 = $\{\}$*"*

"letprev_sat n sat (Suc i) =

$\{v.$ *length* $v = n \wedge$ *sat* $(\lambda\ j.$ *(if* $j \leq i$ *then letprev_sat n sat j else* $\{\}))$ v $i\}$*"*

Note: The if $j \leq i$ comes from the fact that no predicate should be mapped to anything beyond the current input length in V.

Definition 42 (sat for LetPrev)**.** *Definition of the sat function for LetPrev. (See Definition 17 for an explanation of this function.) We can find all valuations for LetPrev by evaluating ψ using the precomputed valuations of ϕ for any p in ψ. Thus, ϕ is first evaluated recursively using letprev_sat and then p is mapped to these valuations in the V for ψ.*

fun *sat where*

"sat σ V v i (LetPrev p ϕ ψ) =

sat σ $(V(p \mapsto$ *letprev_sat* (*nfv* ϕ) $(\lambda\ X\ v\ i.$ *sat* σ $(V(p \mapsto X))$ v i $\phi) \circ$ *Suc*)) v i ψ*"*

Following the definition of sat for LetPrev, we show a few useful lemmas for letprev_sat and sat. The first lemma is not_contains_pred_sat shown in Definition 43, which states that for any formula ϕ that does not contain the predicate p, the mapping of p in V can be ignored when determining the satisfying valuations of ϕ. This originates from the fact that the mapping of p in V is only used to determine the valuation of any p contained in ϕ. This lemma will allow us to easily evaluate sat for LetPrev in the cases where p is not contained in ψ or ϕ.

Definition 43 (not_contains_pred_sat)**.** *Lemma statement for not_contains_pred_sat, where the proof is omitted. This lemma states that the mapping of a predicate p that does not appear in the formula ϕ is not important when computing the satisfying valuations of that formula.*

lemma *not_contains_pred_sat:*

"$\neg$ *contains_pred p* $\phi \Longrightarrow$ *sat* σ $(V(p \mapsto x))$ v i ϕ = *sat* σ V v i ϕ*"*

\\proof omitted

The second lemma is sat_subst_letprev_sat 44, which states that the function used for sat in letprev_sat can be substituted with another function, as long as for any possible input, in which the time-point is smaller or equal to the current time-point, both functions evaluate to the same result. This comes from the fact that sat is only used in the recursion to find satisfying valuations at time-points smaller or equal to the one given as initial input.

Definition 44 (sat_subst_letprev_sat)**.** *Lemma statement for sat_subst_letprev_sat, where the proof is omitted. This lemma formalizes the congruence rule for letprev_sat.*

lemma *sat_subst_letprev_sat:*

$$\text{"}(\bigwedge X\ v\ j.\ j \leq i \Longrightarrow f\ X\ v\ j = g\ X\ v\ j) \Longrightarrow \textit{letprev\_sat}\ n\ f\ i = \textit{letprev\_sat}\ n\ g\ i\text{"}$$

proof omitted

The last lemma is V_subst_letprev 45, which states that the mapping of p in V can be exchanged for a different mapping when evaluating the satisfying valuations of ϕ if ϕ is a safe_letprev formula and for all j smaller than the current input length both mappings hold the same value. This comes from the fact that sat is only concerned with valuations before the current input length for any p for which ϕ is a safe_letprev formula, as safe_letprev disallows future operators above p.

Definition 45 (V_subst_letprev)**.** *Lemma statement for V_subst_letprev, where the proof is omitted. This lemma states that for a safe_letprev formula ϕ the mapping of p in V can change in the sat function as long as the result of the new mapping is equal to the result of the old mapping for all time-points smaller than the one that is being evaluated.*

lemma *V_subst_letprev:*

$$\text{"}\textit{safe\_letprev}\ p\ \phi \Longrightarrow (\bigwedge j.\ j \leq i \Longrightarrow f\ j = g\ j)$$
$$\Longrightarrow \textit{sat}\ \sigma\ (V(p \mapsto f))\ v\ i\ \phi = \textit{sat}\ \sigma\ (V(p \mapsto g))\ v\ i\ \phi\text{"}$$

proof omitted

5.2 Monitor

We start by introducing the mformula definition of LetPrev. MLetPrev 46 takes six arguments, three arguments more then LetPrev. The first argument is equivalent to the p of LetPrev. The second argument is the first helper variable, denoting the number of free variables in ϕ. The third and fourth arguments are equivalent to the ϕ and ψ of LetPrev. The fifth argument is the second helper variable, denoting the input length until which ϕ has been evaluated recursively, initially its value is 0. The sixth argument is the third helper variable, denoting the valuations for which ϕ has not yet been evaluated recursively, initially its value is an empty table. MLetPrev is initialized using minit0 47, by initializing ϕ and ψ recursively and initializing all other arguments with their initial values. The helper values allow us to keep track of the evaluation of p for the previous input length so that this can be used for future computations.

Definition 46 (LetPrev mformula)**.** *The definition of the mformula data type for LetPrev. (See Definition 19 for more details.) MLetPrev uses three helper variables. The second variable denotes the number of free variables in ϕ. The second to last variable denotes until what time-point the recursive evaluation of ϕ has concluded. The last variable saves all valuations for which the recursive phi evaluation has not yet concluded.*

datatype *mformula =...*

| MLetPrev *name nat mformula mformula nat "event_data table list"*

...

Definition 47 (minit0 for LetPrev)**.** *Definition of the function minit0 for LetPrev. (See Definition 20 for an explanation of this function.) The function minit0 recursively initializes the mformula. We use nfv ϕ instead of n in the recursive call for ϕ because ϕ is evaluated separately from the rest of the*

formula.

function *minit0 where*

```
...
"minit0 n (LetPrev p φ ψ) =
 MLetPrev p (nfv φ) (minit0 (nfv φ) φ) (minit0 n ψ) 0 [empty_table]"
...
```

The evaluation of MLetPrev for some input event can be performed by first evaluating ϕ recursively and then using the result of that recursive evaluation in the evaluation of ψ as shown in Definition 51. The recursive evaluation of ϕ, which uses the function letprev_meval and is shown in Definition 50, is not straightforward, as ϕ may contain future operations as long as they are not above any predicate with name p. We illustrate the difficulty letprev_meval poses using two examples, one in which ϕ does not contain any future operators and the other in which ϕ does contain future operators.

The first example we consider is the formula "LetPrev p $(p(x) \vee q(x))$ ψ". Furthermore, we assume that at time-point 0 the event $q(1)$ happens and at time-point 1 no event happens. The ϕ of LetPrev, which is "$(p(x) \vee q(x))$" in our case and implicitly means "((Prev $p(x)$) $\vee$ $q(x)$)", will be evaluated using letprev_meval. Evaluating the ϕ at time-point 0 results in the satisfying valuation 1, as $q(1)$ is true. With this, everything up to the current time-point has been evaluated and we cannot continue evaluation, as the monitor is only allowed to make verdicts for the future. If we, however, do not save this resulting valuation in some way, we lose that information, which is important considering the implicit previous operator around the predicate p. For instance, at time-point 1, the set containing 1 again is a satisfying valuation because 1 was a satisfying valuation for p at time-point 0. It would be possible to compute the satisfying valuations for all time-points between 0 and the current time-point every time, but this would be slow. Instead, we use a buffer to save the valuations, that need to be used for p when letprev_meval is called the next time.

Example 48 (letprev_meval without Future Operators)**.** *This example illustrates the necessary components when recursively evaluating a formula ϕ that does not contain any future operators. We use the following formula, where we write the implicit Prev operator explicitly:*

$$\phi = ((\text{Prev } p(x)) \vee q(x))$$

Furthermore, we assume the event $q(1)$ at time-point 0 and no event at time-point 1. Evaluating ϕ at time-point 0 returns 1 as a satisfying assignment of x. After that, the monitor cannot proceed, as it cannot evaluate formulas past the current time-point. To evaluate ϕ at time-point 1 it is necessary to first find the valuations of ϕ at time-point 0. Instead of recomputing all previous results, it is beneficial to save the satisfying valuations that need to be used for the next recursion step.

The second example we consider is the formula "LetPrev p $(p(x) \vee (\text{Next } q(x)))$ ψ". Furthermore, we assume that at time-point 0 no event happens and at time-point 1 the event $q(1)$ happens. The ϕ of LetPrev, which is "$(p(x) \vee (\text{Next } q(x)))$" in our case and implicitly means "((Prev $p(x)$) $\vee$ (Next $q(x)$))", will be evaluated using letprev_meval. Evaluating the ϕ at time-point 0 results in no valuations. Because of the Next operator, it is necessary to wait for the arrival of the next event to finish evaluating ϕ at this time-point. Thus, if the evaluation of ϕ returns no valuations, the evaluation needs to end, as no verdict can be made yet. In our example, the event at time-point 1 renders the assignment of x to 1 a satisfying valuation at time-point 0. For this reason, it is necessary to continue evaluating these formulas when more data is available. We, thus, need to keep track of the latest time-point for which a verdict has been made for ϕ.

Example 49 (letprev_meval with Future Operators)**.** *This example illustrates the necessary components when recursively evaluating a formula ϕ that contains future operators. We use the following formula, where we write the implicit Prev operator explicitly:*

$$\phi = ((\text{Prev } p(x)) \vee (\text{Next } q(x)))$$

Furthermore, we assume no event at time-point 0 and the event $q(1)$ at time-point 1. Evaluating ϕ at time-point 0 returns an empty list, as the Next operator prevents the formula from being fully evaluated at this time-point. At time-point 1 the evaluation of time-point 0 will conclude with the satisfying assignment 1 for x. Thus, it is necessary to save, the last time-point for which a verdict can be made for the formula ϕ.

In letprev_meval, we combine the insights of the two examples shown above. The argument buf is used to save the valuations of the previous recursion step, for use in the next recursion step. The argument i keeps track of the last time-point for which a verdict can be made for Prev ϕ. If there are no future operators present in ϕ then i will always be equal to j when letprev_meval returns a value, as we can always evaluate ϕ until the current time-point and buf will contain one element for use in the next recursion step. If *phi* contains any future operators, then buf can be empty when letprev_meval returns a value, if there is some information missing for some time-points and in that case, i will always be strictly smaller than j. Some parts of the function are used for ease of proving. For instance, we write "take (j-i) buf" and "drop (j-i) buf", but buf would never contain more elements than j-i. Instead, the entire content of the initial buf is always consumed and dropped in one recursion step. Lastly, this function is only defined for *eval* functions which preserve the size of the formula, which our *eval* function meval does. We need to use a generic *eval* instead of directly using meval as letprev_meval is needed to define meval and using meval in letprev_meval would create a cyclic dependency.

Definition 50 (letprev_meval)**.** *Function definition for letprev_meval. Letprev_meval recursively computes the satisfying valuation of the ϕ of LetPrev given an input database. The termination proof is omitted.*

The first input to this function is an evaluation function (eval) that takes a list of ts, a database, and an mformula and outputs a pair of a list of satisfying valuations and an mformula. The other inputs are the current time-point (j), the time-point up to which the recursion has been evaluated (i), a list of satisfying valuations (ys), a list of valuations for which eval still needs to be called recursively (buf), the name of the predicate that is recursively evaluated (p), the ts on which the evaluation is performed (ts), the input events (db) and the mformula that is being evaluated (ϕ). The function recursively computes the new time-point up to which the recursion has been evaluated, the new list of satisfying valuations, the new list of events that still need to be evaluated, and the new mformula that will be evaluated.

```
function letprev_meval:: "(ts list ⇒ database ⇒ mformula ⇒ event_data table list × mformula)
      ⇒ nat ⇒ nat ⇒ event_data table list ⇒ event_data table list
      ⇒ name ⇒ ts list ⇒ database ⇒ mformula ⇒
      nat × event_data table list × event_data table list × mformula" where
      "letprev_meval eval j i ys buf p ts db φ=
       (let xs = take (j − i) buf;
          (ys', φ') = eval ts (Mapping.update p (map (image (map the)) xs) db) φ;
          buf' = drop (j − i) buf @ ys'
       in if size φ' ≠ φ then undefined
       else if buf' = [] ∨ i + length xs ≥ j then (i + length xs, ys @ ys', buf', φ')
       else letprev_meval
        eval j (i + length xs) (ys @ ys') buf' p [] (Mapping.map_values (λ _ _. []) db) φ')"
      \\ termination proof omitted
```

Definition 51 (meval for LetPrev)**.** *Definition of the meval function for LetPrev. (See Definition 21 for an explanation of this function.) First, ϕ is evaluated recursively using letprev_meval, then that result is used to compute the evaluation of ψ.*

```
function meval where
      ...
      "meval n ts db (MLetPrev p m φ ψ i buf) =
       (let (i, xs, buf, φ) = letprev_meval (meval j m) j i [] buf p ts db φ;
          (ys, ψ) = meval j n ts (Mapping.update p (map (image (map the)) xs) db) ψ
        in (ys, (MLetPrev p m φ ψ i buf))
      ...
```

5.3 Progress

We can define the progress of LetPrev as the progress of ψ where the progress of p is mapped to the progress of the recursive evaluation of ϕ. The progress of the recursive evaluation of ϕ is the predecessor of the i that was described for the letprev_meval function, but, as progress is defined on formulas and not mformulas, we do not have implicit access to i. When evaluating ϕ the progress of p will always be one step ahead of the progress of ϕ due to the Prev operator, unless the progress of ϕ is already at the current time-point, as no formula or predicate can progress beyond the current time-point. This idea can be formulated into the progress fixpoint for ϕ which is an i smaller or equal to the current time-point j, that is equal to the progress of ϕ with p mapped to the successor of i or j for the case where i is equal to j. As the largest time-point for which a verdict can be made is needed, the supremum of all progress fixpoints is used to define the progress for the recursive evaluation of ϕ.

Definition 52 (progress for LetPrev)**.** *Progress definition for LetPrev. (See Definition 22 for an explanation of this function.) The progress of LetPrev is split into two parts. First the maximum progress of the predicate p is determined, which is equivalent to the largest progress fixpoint for ϕ. Then that progress is used to determine the progress of ψ.*

fun *progress where*

...

"progress σ P (LetPrev p ϕ ψ) j =

progress σ $(P(p \mapsto (Sup\ \{i.\ i \leq j \wedge i = progress\ \sigma\ (P(p \mapsto min\ (Suc\ i)\ j))\ \phi\ j\}))))\ \psi\ j$"

...

Following this, we will describe some important lemmas for progress. We start by considering the lemma is not_contains_pred_progress which is shown in the Definition 53. This lemma is analogous to not_contains_pred_sat and states that the mapping of p in P can be ignored when evaluating a formula ϕ that does not contain p. This is the case, as the mapping of p in P is only used to evaluate the progress of any predicate with name p in ϕ. This lemma simplifies both the evaluation of ψ and the evaluation of the progress fixpoints in the cases where ψ respectively ϕ do not contain a predicate with name p.

Definition 53 (not_contains_pred_progress)**.** *Lemma header for not_contains_pred_progress, where the proof is omitted. This lemma states that the mapping of a predicate p that does not appear in the formula ϕ is not important for the progress of that formula.*

lemma *not_contains_pred_progress:*

$\neg$ contains_pred p ϕ $\Longrightarrow$ progress σ $(P(p \mapsto x))$ ϕ j = progress σ P ϕ j"

\\proof omitted

The next lemma is progress_fixpoint_ex 54, which states that for our definition of progress a progress fixpoint always exists. This is important many useful lemmas about the supremum require that the set is non-empty.

Definition 54 (progress_fixpoint_ex)**.** *Lemma statement for progress_fixpoint_ex, where the proof is omitted. This lemma states that for our definition of progress, a progress fixpoint always exists.*

lemma *progress_fixpoint_ex:*

assumes *"$(\bigwedge P.$ pred_mapping $(\lambda x.\ x \leq j)$ $P \Longrightarrow$ progress σ P ϕ $j \leq j)$"*

assumes *"$(\bigwedge P\ P'.$ pred_mapping $(\lambda x.\ x \leq j)$ $P \Longrightarrow$ pred_mapping $(\lambda x.\ x \leq j)$ P'*

$\Longrightarrow$ rel_mapping $(\leq)$ P $P' \Longrightarrow$ progress σ P ϕ $j \leq$ progress σ P' ϕ $j)$"

shows *"pred_mapping $(\lambda x.\ x \leq j)$ P*

$\Longrightarrow \exists i.\ i \leq j.\ i =$ progress σ $(P(p \mapsto min\ (Suc\ i)\ j))$ ϕ j"

\\proof omitted

Note: The two assumptions are needed to prove this statement for all possible ϕ and these properties are proven for our definition of progress.

This is followed by the lemma min_letprev_progress_upd shown in Definition 55. This lemma states that for a safe_letprev formula ϕ for p, the progress of ϕ where p is mapped to x is larger or equal to the minimum of x and the progress of ϕ without the mapping of p. If ϕ does not contain p, then the progress of ϕ with p mapped to x can be larger than x even for small x, as x is not important. If ϕ does contain p, the progress can neither continue beyond the progress of p nor continue beyond the progress of any other formula contained in ϕ.

Definition 55 (min_letprev_progress_upd)**.** *Lemma statement for min_letprev_progress_upd, where the proof is omitted. This lemma states that the progress of a safe_letprev formula ϕ using a mapping where p is mapped to the progress x is larger or equal to the minimum of x and the progress of ϕ where p is not mapped to x.*

lemma *min_letprev_progress_upd: "safe_letprev p ϕ $\Longrightarrow$ pred_mapping (λ x. x $\leq$ j) P*
$\Longrightarrow$ x $\leq$ j $\Longrightarrow$ progress σ (P(p $\mapsto$ x)) ϕ j $\geq$ min x (progress σ P ϕ j)"
\\proof omitted

Note: pred_mapping a P denotes that the value of every predicate p in P satisfies a.

The following lemma sup_alt (Definition 56) is a consequence of the previous lemma. It states that the largest progress fixpoint for ϕ can be written as the progress of ϕ where p is mapped to the current input length j if ϕ is a safe_letprev formula for p. The reason for this can be seen using the examples 48 and 49. If ϕ contains no future operators then the progress of ϕ will always be the current input length j, as any formula in ϕ can be evaluated until j. If ϕ, however, contains formulas with future operators, these formulas can only be evaluated up to some input length smaller or equal than j. As ϕ is a safe_letprev formula for p, the formulas in ϕ that contain future operators cannot contain any predicate with name p, which means that the mapping of p is irrelevant for the progress as long as it is larger or equal to the progress of ϕ without any mapping of p. As j is always larger or equal to the progress of ϕ without any mapping of p, there is no issue in mapping p to j.

Definition 56 (sup_alt)**.** *Lemma statement for sup_alt, where the proof is omitted. This lemma shows that for any safe_letprev formula ϕ the largest progress fixpoint is equivalent to the progress where p is mapped to j.*

lemma *sup_alt:*
assumes *"safe_letprev p ϕ"*
assumes *"pred_mapping (λ x. x $\leq$ j) P"*
shows *"Sup {i. i $\leq$ j $\wedge$ i = progress σ (P(p $\mapsto$ min (Suc i) j)) ϕ j}*
= progress σ (P(p $\mapsto$ j)) ϕ j"
\\proof omitted

The following lemma min_letprev_progress_upd2 shown in Definition 57 states that for any formula ϕ that contains the predicate p and is a safe_letprev formula for p, the progress of ϕ with p mapped to some x is equal to the minimum of some $y \geq x$ and the progress of ϕ with p mapped to the current time-point j. If the progress of ϕ with p mapped to j is smaller than x, then the mapping of p to x will not affect the progress of ϕ. If x is, however, smaller than the progress of ϕ with p mapped to j, then that mapping will affect the progress. It is not possible to simply use the minimum of x and the progress of ϕ with p mapped to j, as p may be surrounded by past operators, in which case the progress would be larger than x.

Definition 57 (min_letprev_progress_upd2)**.** *Lemma statement for min_letprev_progress_upd, where the proof is omitted. This lemma states that for any safe_letprev formula ϕ that contains the predicate p there exists a $y \geq x$ such that the progress of ϕ where p is mapped to x is equal to the minimum of y and the progress of phi where p is mapped to j.*

lemma *min_letprev_progress_upd2:*
"safe_letprev p ϕ $\Longrightarrow$ pred_mapping (λ x. x $\leq$ j) P $\Longrightarrow$ x $\leq$ j $\Longrightarrow$ contains_pred p ϕ
$\Longrightarrow$ $\exists$ y $\geq$ x. progress σ (P(p $\mapsto$ x)) ϕ j = min y (progress σ (P(p $\mapsto$ j)) ϕ j)"
\\proof omitted

Using the previous lemma it is possible to prove the following lemma called fixpoint_unique (Definition 58). This lemma states that there is only one progress fixpoint for a ϕ at a time-point j if ϕ is a safe_letprev formula for p. From this, the lemma fixpoint_sup (Definition 59), which states that if there exists a progress fixpoint for ϕ then that progress fixpoint is the largest one, directly follows. These two lemmas together with the lemma progress_fixpoint_ex 54, which states that a progress fixpoint always exists, allow for different definitions for the progress of LetPrev. Instead of the supremum, it would be possible to use the infimum, the maximum, or the minimum as there exists exactly one fixpoint which is the largest and smallest fixpoint at the same time. Furthermore, it would be possible to simply use the progress of ϕ with p mapped to j as the progress of the recursive evaluation of ϕ due to the lemma sup_alt 56. It may seem like having no fixpoint is always better, however, the definition of letprev_meval uses the current progress in ϕ, which creates a natural connection to the progress fixpoint.

Definition 58 (fixpoint_unique)**.** *Lemma statement for fixpoint_unique, where the proof is omitted. This lemma states that for any safe_letprev formula ϕ there only exists one progress fixpoint smaller than j.*

lemma *fixpoint_unique:*
assumes *"safe_letprev p ϕ"*
assumes *"pred_mapping (λ x. x $\leq$ j) P"*
shows *"i $\leq$ j $\Longrightarrow$ i' $\leq$ i $\Longrightarrow$ i = progress σ (P($p \mapsto$ min (Suc i) j)) ϕ j*
$\Longrightarrow$ i' = progress σ (P($p \mapsto$ min (Suc i') j)) ϕ j $\Longrightarrow$ i $\leq$ i'"
\\proof omitted

Definition 59 (fixpoint_sup)**.** *Lemma statement for fixpoint_sup, where the proof is omitted. This lemma states that if we find a progress fixpoint i for any safe_letprev formula ϕ then that is also the largest progress fixpoint.*

lemma *fixpoint_sup:*
assumes *"safe_letprev p ϕ"*
assumes *"pred_mapping (λ x. x $\leq$ j) P"*
shows *"i $\leq$ j $\Longrightarrow$ i = progress σ (P($p \mapsto$ min (Suc i) j)) ϕ j*
$\Longrightarrow$ i = Sup {i. i $\leq$ j $\wedge$ i = progress σ (P($p \mapsto$ min (Suc i) j)) ϕ j"
\\proof omitted

Lastly, the lemma letprev_progress_ge 60 is proven. This lemma states that for a safe_letprev formula ϕ for p, if there exists a P and a j such that some x is smaller than the progress of ϕ for P and j, then there exists a P and a j such that the same x is smaller than the largest progress fixpoint of ϕ for P and j. This lemma allows us extend the proof of the lemma progress_ge_gen 24, which ensures that the monitor will always make progress, to LetPrev formulas.

Definition 60 (letprev_progress_ge)**.** *Lemma statement for letprev_progress_ge, where the proof is omitted. This lemma states that for a safe_letprev formula ϕ for which there exists a P and a j such that x is smaller than the progress of ϕ using P an input length j, then there exists a P and a j such that x is smaller than the largest progress fixpoint of ϕ using P at input length j.*

lemma *letprev_progress_ge: "safe_letprev p ϕ $\Longrightarrow$ p $\in$ S*
$\Longrightarrow$ ($\exists$ P j. dom P = S $\wedge$ range_mapping x j P $\wedge$ x $\leq$ progress σ P ϕ j)
$\Longrightarrow$ ($\exists$ P j. dom P = S $\wedge$ range_mapping x j P $\wedge$
x $\leq$ Sup {i. i $\leq$ j $\wedge$ i = progress σ (P($p \mapsto$ min (Suc i) j)) ϕ j}"
\\proof omitted

Note: range_mapping a b P denotes that the value of every predicate p in P is larger or equal to a and smaller or equal to b.

5.4 Correctness

Before getting into the implementation details, we take a step back and remember VeriMon's main correctness theorem which was introduced in Definition 27. A high-level summary of this lemma is: if

the monitor state invariant holds and the monitoring environment is well-formed, then the evaluation of a given formula will result in the correct satisfying valuations for that formula and the monitor state invariant will still hold.

We first discuss the monitor state invariant for LetPrev because it yields the assumptions that can be used for the entire proof. The monitor state invariant depends on the formula that is being evaluated, and thus, it the invariant defines for each formula type, what properties are needed for the evaluation algorithm to succeed. As such the properties it guards are closely related to meval and specifically letprev_meval. The ϕ of LetPrev needs to be a safe_letprev formula as otherwise a recursive evaluation may not terminate. Furthermore, after one evaluation step of letprev_meval the progress of p in ϕ shifts from the previous progress fixpoint to the current progress fixpoint, thus, the monitor state invariant asserts that the progress of p in ϕ is at the current progress fixpoint. If the progress fixpoint is at the current input length then there should be an element in the buffer that describes the satisfying valuations of the current letprev_meval step and that will be used to evaluate the next letprev_meval step. If the progress fixpoint is not at the current input length then the buffer should be empty. The full monitor state invariant for LetPrev is shown in in Definition 61.

Definition 61 (wf_mformula for LetPrev)**.** *Wf_mformula definition for LetPrev. (See Definition 22 for an explanation of this function.) The input length until which a valuation for the recursive evaluation of ϕ' can be determined (i) should be the minimum of the largest progress fixpoint of ϕ' and j. The valuations in buf which have not yet been recursively evaluated need to be satisfying valuations of ϕ'.*

inductive *wf_mformula where*

$$
\begin{aligned}
&\ldots\\
&\mid \textit{LetPrev: ``wf\_mformula } \sigma\ j\ (P(p \mapsto i))\\
&\quad (V(p \mapsto \textit{letprev\_sat } m\ (\lambda\ X\ v\ i.\ \textit{sat } \sigma\ (V(p \mapsto X))\ v\ i\ \phi')))\ m\ \phi\ \phi'\\
&\implies i = \textit{min } (\textit{Suc } (\textit{Sup } \{i.\ i \leq j \wedge i = \textit{progress } \sigma\ (P(p \mapsto \textit{min } (\textit{Suc } i)\ j))\ \phi'\ j\}))\ j\\
&\implies \textit{list\_all2 } (\lambda\ i.\ \textit{qtable } m\ (\textit{fv } \phi')\\
&\quad (\lambda\ v.\ \textit{map the } v \in \textit{letprev\_sat } m\ (\lambda\ X\ v\ i.\ \textit{sat } \sigma\ (V(p \mapsto X))\ v\ i\ \phi')\ i))\\
&\quad [i..<\textit{Suc } (\textit{Sup } \{i.\ i \leq j \wedge i = \textit{progress } \sigma\ (P(p \mapsto \textit{min } (\textit{Suc } i)\ j))\ \phi'\ j\})]\ \textit{buf}\\
&\implies \textit{wf\_mformula } \sigma\ j\\
&\quad (P(p \mapsto (\textit{Sup } \{i.\ i \leq j \wedge i = \textit{progress } \sigma\ (P(p \mapsto \textit{min } (\textit{Suc } i)\ j))\ \phi'\ j\})))\\
&\quad (V(p \mapsto \textit{letprev\_sat } m\ (\lambda\ X\ v\ i.\ \textit{sat } \sigma\ (V(p \mapsto X))\ v\ i\ \phi') \circ \textit{Suc}))\ n\ \psi\ \psi'\\
&\implies \textit{safe\_letprev } p\ \phi' \implies \{0..< m\} \subseteq \textit{fv } \phi' \implies m = \textit{nfv } \phi'\\
&\implies \textit{wf\_mformula } \sigma\ j\ P\ V\ n\ (\text{MLetPrev } p\ m\ \phi\ \psi\ i\ \textit{buf})\ (\text{LetPrev } p\ \phi'\ \psi')\text{''}\\
&\ldots
\end{aligned}
$$

In the proof of the main correctness theorem, the wf_mformula assumption for LetPrev and the wf_envs 26 assumption can be used to prove for LetPrev, that the monitor state invariant still holds after one evaluation step and that the computed valuations are the correct satisfying valuations. Furthermore, as the main correctness theorem is proven using induction, the induction hypotheses for ϕ and ψ are given. One evaluation step can entail multiple iterations of letprev_meval, thus, it cannot be assumed that one letprev_meval iteration yields all properties that we desire from the final result. This issue can be solved by defining a loop postcondition that states all properties that should hold after letprev_meval concludes and a loop invariant that should hold at the beginning of every call to letprev_meval. By proving the two properties: the loop invariant holds for the first call to letprev_meval and if the loop invariant holds initially then the loop postcondition holds for the return values, the postcondition can be shown for meval.

The postcondition for letprev_meval shown in 63 incorporates all necessary information about the result of letprev_meval to prove both the monitor state invariant as well as the correctness of the satisfying valuations. We shortly summarize the properties that should hold after letprev_meval concludes. The updated mformula returned by letprev_meval needs to adhere to the monitor state invariant. As stated above, the progress of p in ϕ must reach a progress fixpoint when letprev_meval concludes. The buffer should be empty except if the progress fixpoint has reached the current input length, then it should contain the satisfying valuation needed to compute the evaluation for the next time-point. All valuations computed by letprev_meval should be the correct satisfying valuations for their respective input lengths.

The invariant for letprev_meval shown in 62 describes the information that can be gained from each call of letprev_meval, which in turn facilitates the proving of the postcondition. An important aspect of letprev_meval is that recursive calls use [] as *ts* because the list of new timestamps gets consumed in the first call. Thus, j - length *ts* is the input length without the new timestamps in the first call of letprev_meval but the input length with the new timestamps in all recursive calls. Initially, the progress of p in ϕ described by i is the previous progress fixpoint. In all recursive calls, i will be somewhere between the previous progress fixpoint and the next progress fixpoint. The formula used for in letprev_meval should always satisfy the monitor state invariant for the current i. All valuations in the buffer, as well as all valuations computed in the current call stack of letprev_meval, should be the correct satisfying valuations for their respective input lengths. The argument k is used to save the value of the progress fixpoint before calling letprev_meval as this value cannot be recomputed after the list of new timestamps is consumed.

Definition 62 (letprev_meval_invar)**.** *Definition for letprev_meval_invar. The loop invariant for the recursive letprev_meval function.*

definition *letprev_meval_invar where*

"letprev_meval_invar n V sigma P ϕ' *m* j' *i ys buf p ts db* ϕ *k=*
(let $j = j'$ *- length ts in*
wf_mformula σ *j* $(P(p \mapsto i))$
$(V(p \mapsto$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi'))))$ *m* ϕ $\phi' \wedge$
$i \leq$ *min (Suc (progress* σ $(P(p \mapsto i))$ ϕ' *j)) j* $\wedge$
list_all2 $(\lambda$ *i. qtable m (fv* $\phi')$
$(\lambda$ *v. map the v* $\in$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi')$ *i))*
[i..<Suc (Sup {i. i $\leq j \wedge i =$ *progress* σ $(P(p \mapsto$ *min (Suc i) j))* ϕ' *j})] buf* $\wedge$
list_all2 $(\lambda$ *i. qtable m (fv* $\phi')$
$(\lambda$ *v. map the v* $\in$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi')$ *i))*
[k..<Suc (Sup {i. i $\leq j \wedge i =$ *progress* σ $(P(p \mapsto$ *min (Suc i) j))* ϕ' *j})] ys* $\wedge$
$k \leq$ *Suc (progress* σ $(P(p \mapsto i))$ ϕ' *j))"*

Definition 63 (letprev_meval_post)**.** *Definition for letprev_meval_post. The loop postcondition for the recursive letprev_meval function.*

definition *letprev_meval_post where*

"letprev_meval_post n V sigma P ϕ' *m j i' xs buf' p* ϕf *k=*
(wf_mformula σ *j* $(P(p \mapsto i'))$
$(V(p \mapsto$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi')))$ *m* ϕf $\phi' \wedge$
$i =$ *min (Suc (Sup {i. i* $\leq j \wedge i =$ *progress* σ $(P(p \mapsto$ *min (Suc i) j))* ϕ' *j})) j* $\wedge$
list_all2 $(\lambda$ *i. qtable m (fv* $\phi')$
$(\lambda$ *v. map the v* $\in$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi')$ *i))*
[i'..<Suc (Sup {i. i $\leq j \wedge i =$ *progress* σ $(P(p \mapsto$ *min (Suc i) j))* ϕ' *j})] buf* $\wedge$
list_all2 $(\lambda$ *i. qtable m (fv* $\phi')$
$(\lambda$ *v. map the v* $\in$ *letprev_sat m* $(\lambda$ *X v i. sat* σ $(V(p \mapsto X))$ *v i* $\phi')$ *i))*
[k..<Suc (Sup {i. i $\leq j \wedge i =$ *progress* σ $(P(p \mapsto$ *min (Suc i) j))* ϕ' *j})] xs)"*

The loop invariant should hold for any values with which we call letprev_meval, thus, it should also hold for the values with which letprev_meval is called initially. The lemma letprev_meval_invar_init shown in Definition 64 proves that the loop invariant holds for the values with which letprev_meval is called initially.

Definition 64 (letprev_meval_invar_init)**.** *Lemma statement for letprev_meval_invar_init, where the proof is omitted. This lemma states that the loop invariant for letprev_meval holds for the initial*

values with which letprev_meval is called.

lemma *letprev_meval_invar_init:*
assumes *"pred_mapping (λ x. x ≤ j) P"*
assumes *"wf_mformula σ (j - length ts) P V n* (MLetPrev *p m φ ψ i buf*) (LetPrev *p φ′ ψ′*)*"*
shows *"letprev_meval_invar n V σ P φ′ m j i [] buf p ts db φ*
(Suc (Sup {i. i ≤ j ∧ i = progress σ (P(p ↦ min (Suc i) j)) φ′ j}))"
\\proof omitted

Note: pred_mapping a P denotes that the value of every predicate p in P satisfies a.

Using the fact that the loop invariant holds initially, proving the postcondition entails showing that the loop postcondition holds for the return values of letprev_meval if the loop invariant holds for the values with which letprev_meval was called initially. This can be split into two properties. The first property states that, if the loop invariant holds at the beginning of letprev_meval and letprev_meval calls itself recursively, then the loop invariant also holds for the new values with which letprev_meval is called. This property is proven in the lemma invar_recursion_invar shown in Definition 65. The second property states that, if the loop invariant holds at the beginning of letprev_meval and letprev_meval does not call itself recursively, then the loop postcondition holds for the return values. This property is proven in the lemma invar_recursion_post shown in Definition 66.

Definition 65 (invar_recursion_invar)**.** *Lemma statement for invar_recursion_invar, where the proof is omitted. This lemma states that if the loop invariant holds before executing the loop body and the recursion does not end after the loop body, then the loop invariant holds after the loop body as well. The meval assumption is a modified version of the induction hypotheses given in the proof of the main correctness theorem.*

lemma *invar_recursion_invar:*
assumes *"pred_mapping (λ x. x ≤ j) P"*
assumes *"pred_mapping (λ x. x ≤ (j - length ts)) P′"*
assumes *"rel_mapping (≤) P P′"*
assumes *"case meval j m ts (Mapping.update p (map ((‘) (map the)) xs) db) φ of (xs′, φn) ⇒*
wf_mformula σ j (P′(p ↦ i + length xs))
(V(p ↦ letprev_sat m (λ X v i. sat σ (V(p ↦ X)) m φn φ′ ∧
list_all2 (λ i. qtable m (fv φ′)
(λ v. sat σ (V(p ↦ λ j. if j ≤ i then
letprev_sat m (λ X v i. sat σ (V(p ↦ X)) v i φ′) j else {})) (map the v) i φ′))
[progress σ (P(p ↦ i + length xs)) φ′ (j - length ts)..< progress σ (P′(p ↦ i)) φ′ j]"
shows *"letprev_meval_invar n V σ P φ′ m j i ys buf p ts db φ k*
⟹ xs = take (j - i) buf
⟹ (ys′, φf) = meval j m ts (Mapping.update p(map ((‘) (map the)) xs) db) φ
⟹ buf′ = drop (j - i) buf @ ys′ ⟹ buf′ ≠ [] ⟹ i + length xs < j
⟹ letprev_meval_invar n V σ P′ φ′ m j (i + length xs) (ys@ys′) buf′ p []
(Mapping.map_values (λ_ _. []) db) φf k"
\\proof omitted

Notes:
rel_mapping (≤) P P′ denotes that P and P′ contain the same predicates and that for every predicate p the value to which p is mapped in P′ is larger than the value to which p is mapped to in P.

Definition 66 (invar_recursion_post)**.** *Lemma statement for invar_recursion_post, where the proof is omitted. This lemma states that if the loop invariant holds before executing the loop body and the recursion ends after the loop body, then the loop postcondition holds after the loop body. The meval assumption is a modified version of the induction hypotheses given in the proof of the main correctness*

theorem.

lemma *invar_recursion_post:*
assumes *"safe_letprev p ϕ'"*
assumes *"pred_mapping (λ x. $x \leq j$) P"*
assumes *"pred_mapping (λ x. $x \leq$ (j - length ts)) P'"*
assumes *"rel_mapping ($\leq$) P P'"*
assumes *"case meval j m ts (Mapping.update p (map ((`) (map the)) xs) db) ϕ of (xs', ϕ_n) $\Rightarrow$*
wf_mformula σ j (P'(p $\mapsto$ i + length xs))
(V(p $\mapsto$ letprev_sat m (λ X v i. sat σ (V(p $\mapsto$ X)) m ϕ_n ϕ' $\wedge$
list_all2 (λ i. qtable m (fv ϕ')
(λ v. sat σ (V(p $\mapsto$ λ j. if j $\leq$ i then
letprev_sat m (λ X v i. sat σ (V(p $\mapsto$ X)) v i ϕ') j else {})) (map the v) i ϕ'))
[progress σ (P(p $\mapsto$ i + length xs)) ϕ' (j - length ts)..< progress σ (P'(p $\mapsto$ i)) ϕ' j]"
shows *"letprev_meval_invar n V σ P ϕ' m j i ys buf p ts db ϕ k*
$\Longrightarrow$ xs = take (j - i) buf
$\Longrightarrow$ (ys', ϕf) = meval j m ts (Mapping.update p(map ((`) (map the)) xs) db) ϕ
$\Longrightarrow$ buf' = drop (j - i) buf @ ys' $\Longrightarrow$ buf' = [] $\vee$ i + length xs $\geq$ j
$\Longrightarrow$ letprev_meval_post n V σ P' ϕ' m j (i + length xs) (ys@ys') buf' p ϕf k"
\\proof omitted

The lemma letprev_meval_invar_post shown in Definition 67 internally uses the two lemmas shown before: invar_recursion_invar and invar_recursion_post to prove the postcondition of letprev_meval assuming that the loop invariant holds initially. By using letprev_meval_invar_post together with letprev_meval_invar_init it can be proven that the letprev_meval postcondition holds for every call of meval on a formula containing LetPrev.

Definition 67 (letprev_meval_invar_post)**.** *Lemma statement for letprev_meval_invar_post, where the proof is omitted. This lemma states that if the loop invariant holds before starting the recursion then the postcondition will hold after completing the recursion. The meval assumption is a modified version of the induction hypotheses given in the proof of the main correctness theorem. The use of us distinguishes between the first call of letprev_meval, in which a list of new timestamps is given, and*

subsequent calls, where they already have been consumed.

lemma *letprev_meval_invar_post:*
assumes *"safe_letprev p ϕ'"*
assumes *"pred_mapping (λ x. $x \leq j$) P"*
assumes *"pred_mapping (λ x. $x \leq$ (j - length ts)) P'"*
assumes *"rel_mapping ($\leq$) P P'"*
assumes *"m = nfv ϕ"*
assumes *"{0..< m} $\subseteq$ m"*
assumes *wf_mformula σ (j - length ts) (P($p \mapsto i$))*
(V($p \mapsto$ letprev_sat m (λ X v i. sat σ (V($p \mapsto X$)) v i ϕ'))) m ϕ ϕ'
assumes *"$\bigwedge$ xs db ϕ_m us P P' V. size ϕ_m = size ϕ*
$\Longrightarrow$ wf_mformula σ (j - length us) P V m ϕ_m ϕ'
$\Longrightarrow$ wf_envs σ (j - length us) (length us) P P' V
(Mapping.update p (map ((`) (map the)) xs) db)
$\Longrightarrow$ us = [] $\vee$ us = ts
$\Longrightarrow$ case meval j m ts (Mapping.update p (map ((`) (map the)) xs) db) ϕ_m
of (xs', ϕ_n) $\Rightarrow$
wf_mformula σ j P' V m ϕ_n ϕ' $\wedge$
list_all2 (λ i. qtable m (fv ϕ') (λ v. sat σ V (map the v) i ϕ'))
[progress σ P ϕ' (j - length us)..< progress σ P' ϕ' j]"
shows *"letprev_meval_invar n V σ P ϕ' m j i ys buf p ts db ϕ k*
$\Longrightarrow$ (i', ys', buf', ϕf) = letprev_meval m j i ys buf p ts db ϕ
$\Longrightarrow$ letprev_meval_post n V σ P' ϕ' m j i' ys' buf' p ϕf k"
\\proof omitted

The well-formedness of the environment expressed through wf_envs is a necessary condition for the main correctness proof and, in turn, also for the induction hypotheses it provides. The induction hypotheses are often used with updated P, P', and V, and thus, it is necessary to update the wf_envs. For LetPrev, two different updates of the P and P' are needed one for the cases in which the progress of p is a progress fixpoint and one for the cases where it is not. Thus, the first wf_envs lemma wf_envs_update_sup shown in Definition 68 is used for proofs outside of letprev_meval and the second wf_envs lemma wf_envs_update_sup2 shown in Definition 69 is used for proofs regarding letprev_meval. This concludes all lemmas necessary for the proof of the main correctness theorem for LetPrev.

Definition 68 (wf_envs_update_sup)**.** *Lemma header for wf_envs_update_sup, where the proof is omitted. This lemma states that if we have a wf_envs for some P, P' and V, we can get a wf_envs for an*

updated P, P' and V.

lemma *wf_envs_update_sup:*
assumes *"wf_envs σ j δ P P' V db"*
assumes *"m = nfv ϕ"*
assumes *"$\{0..< m\} \subseteq m$"*
assumes *"rel_mapping $(\leq)$ P P'"*
assumes *"list_all2 (λ i. qtable m (fv ϕ)*
(λ v. map the $v \in$ letprev_sat m (λ X v i. sat σ ($V(p \mapsto X)$) v i ϕ) i))
[Suc (Sup $\{i.\ i \leq j \wedge i$ = progress σ ($P(p \mapsto$ min (Suc i) j)) ϕ $j\}$)..<
Suc (Sup $\{i.\ i \leq (j+\delta) \wedge i$ = progress σ ($P(p \mapsto$ min (Suc i) $(j+\delta)$)) ϕ $(j+\delta)\}$)] xs'"
shows *"wf_envs σ j δ ($P(p \mapsto$ Sup $\{i.\ i \leq j \wedge i$ = progress σ ($P(p \mapsto$ min (Suc i) j)) ϕ $j\}$))*
($P'(p \mapsto$ Sup $\{i.\ i \leq (j+\delta) \wedge i$ = progress σ ($P(p \mapsto$ min (Suc i) $(j+\delta)$)) ϕ $(j+\delta)\}$))
($V(p \mapsto$ letprev_sat m (λ X v i. sat σ ($V(p \mapsto X)$) v i ϕ') $\circ$ Suc))
(Mapping.update p (map (image (map the)) xs') db)"
\\proof omitted

Definition 69 (wf_envs_update_sup2)**.** *Lemma header for wf_envs_update_sup2, where the proof is omitted. This lemma states that if we have a wf_envs for some P, P' and V, we can get a wf_envs for an updated P, P' and V.*

lemma *wf_envs_update_sup2:*
assumes *"wf_envs σ j δ P P' V db"*
assumes *"m = nfv ϕ"*
assumes *"$\{0..< m\} \subseteq m$"*
assumes *"rel_mapping $(\leq)$ P P'"*
assumes *"$i \leq i'$"*
assumes *"$i \leq j$"*
assumes *"$i' \leq j + \delta$"*
assumes *"list_all2 (λ i. qtable m (fv ϕ)*
(λ v. map the $v \in$ letprev_sat m (λ X v i. sat σ ($V(p \mapsto X)$) v i ϕ) i))
[$i..< i'$] xs"
shows *"wf_envs σ j δ ($P(p \mapsto i)$) ($P'(p \mapsto i')$)*
($V(p \mapsto$ letprev_sat m (λ X v i. sat σ ($V(p \mapsto X)$) v i ϕ')))
(Mapping.update p (map (image (map the)) xs') db)"
\\proof omitted

6 Fixpoints of Bounded, Monotone Formulas (LetRec) in VeriMon

In this section, we describe how we implemented the LetRec operator in Isabelle. We will describe the data types, functions, and proofs we updated and will introduce any new lemmas created for the proofs. This section will follow the structure of section 3.4. Wherever possible, we will omit anything we did not update. We will use the notation "LetRec p ϕ ψ" for the LetRec operator. Any mention of ϕ, ψ, and p refers to this notation unless otherwise noted. Some definitions and proofs are not finalized, however, an idea on how they could be completed will be given.

Overview of the Differences between LetRec and LetPrev Both LetPrev and LetRec are recursive let operators, and thus, many parts of their implementation are rather similar. However, there are some key differences that greatly impact the implementation. The restrictions on the recursive formula are different in LetPrev and LetRec, which means that the safety property shown in the definition of safe_formula 74 and specifically the safety property of the ϕ shown in the definition of safe_letrec 73 are different. Specifically, the fact that, unlike LetPrev, not every occurrence of the predicate p in the ϕ of LetRec is strictly in the past, changes how the functions sat 77, and its underlying recursive evaluation letrec_sat 76, and meval 84, and its underlying recursive evaluation letrec_meval 84, function. This in turn leads to a different invariant letrec_meval_invar 90 and a different postcondition letrec_meval_post 91 for the recursive letrec_meval. All lemmas concerning these properties need to be changed as well.

Overview of the Incomplete Functions and Proofs Even though some functions and proofs are incomplete, we conjecture that much of the current implementation can be used as-is. Everything in Section 6.3 can be used as-is. In Section 6.1, the proof of the finiteness of the satisfying valuations (Definition 80) is incomplete. However, we outline how this proof could be completed. In Section 6.2, the function letrec_meval (Definition 84) is incomplete, which poses the biggest problem and affects the functions and proofs in Section 6.4. Because of the incompleteness of letrec_meval, its invariant is too weak to prove the postcondition that is needed for VeriMon's main correctness theorem. It is unclear if completing letrec_meval and its invariant would necessitate the change of other functions such as wf_mformula or even require the mformula data type to be updated with more helper variables for LetRec.

6.1 Formula

The LetRec formula is implemented by excluding future operators above the recursive predicate, negation operators above the recursive predicate, equality operators, and aggregation operators in the ϕ of the recursive let as described in Section 4.5. LetRec takes three arguments p, ϕ and ψ. ϕ may contain p and no assumptions are made about any p in ϕ. Any instance of p in ψ needs to be replaced by the result of evaluating ϕ recursively for p. Just like for LetPrev, the p in the LetRec context is different from any p outside of this context. This fact is described in more detail in Section 5.1.

Definition 70 (LetRec formula)**.** *The definition of the formula data type for LetRec. (See Definition 13 for other cases.)*

datatype *formula* = ...
| LetRec *name formula formula*
...

The two functions contains_pred 37, that checks if a predicate with name p occurs below a certain formula and safe_letprev 38, that checks if a formula can be used as the ϕ in a LetPrev formula must be updated to accommodate LetRec formulas. Nested recursion is no issue for our implementation, even if LetPrev is nested in LetRec or vice versa. In part due to this fact, the functions contains_pred and safe_letprev are defined the same for LetRec as for LetPrev. The definition of the functions for LetRec is shown in Definition 71 and Definition 72.

Definition 71 (contains_pred for LetRec)**.** *The definition of the function contains_pred for LetRec. (See Definition 37 for an explanation of this function.)*

fun *contains_pred :: "name ⇒ formula ⇒ bool" where*
...
| *"contains_pred p* (LetRec *e ϕ ψ*) =
(*p* ≠ *e* ∧ ((*contains_pred p ϕ* ∧ *contains_pred e ψ*) ∨ *contains_pred p ψ*))"
...

Definition 72 (safe_letprev)**.** *The definition of the function safe_letprev for LetRec. (See Definition 38 for an explanation of this function.)*

fun *safe_letprev :: "name ⇒ formula ⇒ bool" where*
...
| *"safe_letprev p* (LetRec *e ϕ ψ*) = (*safe_letprev e ϕ* ∧
p = *e* ∨ *safe_letprev p ϕ* ∧ (¬ *contains_pred p ϕ* ∨ *safe_letprev e ψ*) ∧ *safe_letprev p ψ*)"
...

The function safe_formula ensures both that any given formula only has a finite number of satisfying valuations and that any satisfying valuation can be found in finite time. For LetRec the following conditions ensure that the recursion is finite: there is no future operation above a predicate with the name p in ϕ, there is no negation above a predicate with name p in ϕ, there are no equality operators in ϕ, and there are no aggregation operators in ϕ. This definition is, as described in Section 4.3, an overestimation of the cases that may lead to an infinite recursion, but still provides good flexibility while not being overly complicated. The function safe_letrec shown in Definition 73 checks the properties described above using the contains_pred 37 function that was defined earlier. The safety condition for LetRec, shown in Definition 74, is a combination of safe_letrec enforcing finite recursion, the safety of the sub-formulas, and that no variable indexes are missing in ϕ.

Definition 73 (safe_letrec)**.** *Function header for safe_letrec. The function safe_letrec takes a name (p) and a formula (ϕ) as input and outputs true if evaluating p recursively using ϕ cannot lead to an*

infinite recursion.

fun *safe_letrec :: "name ⇒ formula ⇒ bool" where*
"safe_letrec p (Eq *t1 t2*) = *False"*
| *"safe_letrec p* (Less *t1 t2*) = *False"*
| *"safe_letrec p* (LessEq *t1 t2*) = *False"*
| *"safe_letrec p* (Pred *e ts*) = *True"*
| *"safe_letrec p* (Let *e φ ψ*) = (*safe_letrec p φ* ∧
(¬ *contains_pred p φ* ∨ *safe_letrec e ψ*) ∧ (*p* = *e* ∨ *safe_letrec p ψ*))"
| *"safe_letrec p* (LetPrev *e φ ψ*) = (*safe_letrec e φ* ∧
p = *e* ∨ *safe_letrec p φ* ∧ (¬ *contains_pred p φ* ∨ *safe_letrec e ψ*) ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (LetRec *e φ ψ*) = (*safe_letrec e φ* ∧
p = *e* ∨ *safe_letrec p φ* ∧ (¬ *contains_pred p φ* ∨ *safe_letrec e ψ*) ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (Neg *φ*) = (¬ *contains_pred p φ* ∧ *safe_letrec p φ*)"
| *"safe_letrec p* (Or *φ ψ*) = (*safe_letrec p φ* ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (And *φ ψ*) = (*safe_letrec p φ* ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (Ands *l*) = (∀ *φ* ∈ *set l*. *safe_letrec p φ*)"
| *"safe_letrec p* (Exists *φ*) = *safe_letrec p φ"*
| *"safe_letrec p* (Agg *y ω b′ f φ*) = *False"*
| *"safe_letrec p* (Prev *I φ*) = *safe_letrec p φ"*
| *"safe_letrec p* (Next *I φ*) = (¬ *contains_pred p φ* ∧ *safe_letrec p φ*)"
| *"safe_letrec p* (Since *φ I ψ*) = (*safe_letrec p φ* ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (Until *φ I ψ*) =
(¬ *contains_pred p φ* ∧ *safe_letrec p φ* ∧ ¬ *contains_pred p ψ* ∧ *safe_letrec p ψ*)"
| *"safe_letrec p* (MatchP *I r*) = (∀ *φ* ∈ *atms r*. *safe_letrec p φ*)"
| *"safe_letrec p* (MatchF *I r*) = (∀ *φ* ∈ *atms r*. ¬ *contains_pred p φ* ∧ *safe_letrec p φ*)"

Definition 74 (safe_formula for LetRec)**.** *The definition of the function safe_formula for LetRec. (See Definition 15 for an explanation of this function.)*

fun *safe_formula where*
...
"safe_formula (LetRec *p φ ψ*) =
safe_letrec p φ ∧ {*0.. < nfv φ* ⊆ *fv φ*} ∧ *safe_formula φ* ∧ *safe_formula ψ"*
...

Notes:
The function nfv φ denotes number of free variables in φ.
The function fv φ are all free variables in φ.
The function {0.. < nfv φ ⊆ fv φ} ensures that there are no free variable indexes missing in φ. This is technically possible as free variable indexes can be skipped, but for simplicity sake, we do not allow it in the φ of LetRec.

A given formula is future_bounded iff it only contains finite interval and any recursive evaluation needed terminates in a finite number of steps, which can be guaranteed using safe_letrec. The definition of future_bounded for LetRec is shown in Definition 75.

Definition 75 (future_bounded for LetRec)**.** *Definition of the future_bounded function for LetRec.*

(See Definition 16 for an explanation of this function.)

fun *future_bounded where*

...

"future_bounded (LetRec p ϕ ψ) = *safe_letrec* p ϕ $\wedge$ *future_bounded* ϕ $\wedge$ *future_bounded* ψ*"*

...

The satisfying valuations for LetRec can be found by first determining the satisfying valuations for ϕ recursively and then using those valuations for any predicate with name p in ψ as shown in Definition 42. To this end, we first evaluate ϕ and then save the satisfying valuations of ϕ under the name p in the V that will be used to evaluate ψ. The first issue encountered when creating this definition is that the recursive evaluation of ϕ does not have a clear termination condition. To circumvent this issue, we introduce a recursion limit k that will be decreased in every recursion step. As it is not possible to know beforehand, how many recursion steps are necessary for the evaluation of ϕ to reach the fixpoint, the union of all possible k values is used for the evaluation. For any formula that is a safe_letrec formula, the union of all results for the different k values should be equal to the fixpoint.

The recursive evaluation function letrec_sat is shown in Definition 76, where *sat* is the function that evaluates ϕ and *new* holds the resulting satisfying valuations. When the last input value reaches 0, the list *new* containing all satisfying valuations is returned. In all other cases, *new* will be updated to the satisfying valuations of the current recursion step, where the valuation length is equal to the number of free variables in ϕ. It is important to note that this method allows for an infinite number of satisfying valuations under the name p in V, as k infinitely many sets are used for the union.

Definition 76 (letrec_sat)**.** *Definition of the letrec_sat function. The function letrec_sat takes the number of free variables in the formula (n), a function with which we determine the satisfying valuations (sat) and a natural number for the recursion limit (k) as input. It is used to recursively compute all satisfying valuations for ϕ.*

fun *letrec_sat:: "nat* $\Rightarrow$ *((nat* $\Rightarrow$ *'b list set)* $\Rightarrow$ *'b list* $\Rightarrow$ *nat* $\Rightarrow$ *bool)* $\Rightarrow$
(nat $\Rightarrow$ *'b list set)* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *'b list set" where*
"letrec_sat n sat new 0 = new"
"letrec_sat n sat new (Suc k) =
letrec_sat n sat (λ *i.* $\{v.$ *length* $v = n \wedge$ *sat new* v *i*$\}$) *k"*

Definition 77 (sat for LetRec)**.** *The definition of the sat function for LetRec. (See Definition 17 for an explanation of this function.) All valuations for LetRec can be found by evaluating ψ using the precomputed valuations of ϕ for any p in ψ. Thus, ϕ is first evaluated recursively using letrec_sat and then p is mapped to these valuations in the V for ψ. The recursive evaluation is performed with all possible recursion limits and valuations are computed as the union of the result sets.*

fun *sat where*

"sat σ V v i (LetRec p ϕ ψ) = *sat* σ
($V(p \mapsto (\lambda\ i.\ \bigcup k.$ *letrec_sat* (*nfv* ϕ) (λ X v $i.$ *sat* σ ($V(p \mapsto X)$) v i ϕ) (λ $i.$ $\{\}$) k i)))
v i ψ*"*

Following the definition of sat for LetRec, we show a few useful lemmas for letrec_sat and sat. The first lemma is sat_subst_letrec_sat 78, which states that the *sat* as well as the *new* used for sat in letrec_sat can be substituted under two conditions. The first states that for any possible input, in which the time-point is smaller or equal to the current time-point, both functions must evaluate to the same result even for different first arguments, so long as those first arguments evaluate to the same result for any time-point smaller or equal to j. The second states that for any possible input, in which the time-point is smaller or equal to the current time-point, the *new* values must be equal. This comes from the fact that *sat* is only used in the recursion to find satisfying valuations at time-points smaller or equal to the one given as initial input.

Definition 78 (sat_subst_letrec_sat)**.** *Lemma statement for sat_subst_letrec_sat, where the proof is omitted. This lemma states that the sat function used in letrec_sat can be substituted with another function so long as that other function returns the same result for all time-points smaller or equal to the one that is being evaluated.*

lemma *sat_subst_letrec_sat:*

$$\begin{aligned}&\text{"}(\bigwedge X\ Y\ v\ j.\ j \leq i \Longrightarrow (\bigwedge k.\ k \leq j \Longrightarrow X\ k = Y\ k) \Longrightarrow f\ X\ v\ j = g\ Y\ v\ j)\\ &\Longrightarrow (\bigwedge j.\ j \leq i \Longrightarrow a\ j = b\ j)\\ &\Longrightarrow \mathit{letrec\_sat}\ n\ f\ a\ x\ i = \mathit{letrec\_sat}\ n\ g\ b\ x\ i\text{"}\\ &\backslash\backslash \textit{proof omitted}\end{aligned}$$

The second lemma is V_subst_letrec 79, which states that the mapping of p in V can be exchanged for a different mapping when evaluating the satisfying valuations of ϕ if ϕ is a safe_letrec formula and for all j smaller than the current time-point both mappings hold the same value. This comes from the fact that sat is only concerned with valuations before the current time-point for any p for which ϕ is a safe_letrec formula, as safe_letrec disallows future operators above p.

Definition 79 (V_subst_letrec)**.** *Lemma statement for V_subst_letrec, where the proof is omitted. This lemma states that for a safe_letrec formula ϕ the mapping of p in V can change in the sat function as long as the result of the new mapping is equal to the result of the old mapping for all time-points smaller than the one that is being evaluated.*

lemma *V_subst_letrec:*

$$\begin{aligned}&\text{"}\mathit{safe\_letrec}\ p\ \phi \Longrightarrow (\bigwedge j.\ j \leq i \Longrightarrow f\ j = g\ j)\\ &\Longrightarrow \mathit{sat}\ \sigma\ (V(p \mapsto f))\ v\ i\ \phi = \mathit{sat}\ \sigma\ (V(p \mapsto g))\ v\ i\ \phi\text{"}\\ &\backslash\backslash \textit{proof omitted}\end{aligned}$$

As stated above, the number satisfying valuations mapped to p in V in the sat of LetRec 77 may be infinite as it is a union of an infinite number of sets. By definition, any formula that is future_bounded and a safe_formula has a fixpoint that is computable in finite time. The lemma safe_formula_finite shown in Definition 80 formalizes this property and states that for any ϕ that is both a safe_formula and future_bounded the number of satisfying valuations is finite. Unfortunately, this proof has yet to be completed, as the LetRec case is still unproven. It is possible to prove the LetRec case by defining a superset of the union of the letrec_sat result for all recursion limits and proving that that superset is finite. We believe this superset can be defined using the active domain, which is a set of all event_data values occurring in the trace up to the current input length (plus some bounded lookahead because of the future operators) and the formula.

Definition 80 (safe_formula_finite)**.** *Conjecture statement for safe_formula_finite, where the proof is incomplete. This lemma states that for a ϕ that is a safe_formula and future_bounded, the number of satisfying valuations is finite.*

lemma *safe_formula_finite:*

$$\begin{aligned}&\text{"}\mathit{safe\_formula}\ \phi \Longrightarrow \mathit{future\_bounded}\ \phi \Longrightarrow n \geq (\mathit{nfv}\ \phi) \Longrightarrow \forall\ i.\ \mathit{finite}\ (\Gamma\ \sigma\ i)\\ &\Longrightarrow \forall\ p \in \mathit{dom}\ V.\ \mathit{finite}\ (\mathit{the}\ (V\ p)\ i)\\ &\Longrightarrow \mathit{finite}\ \{v.\ \mathit{wf\_tuple}\ n\ (\mathit{fv}\ \phi)\ v \wedge \mathit{sat}\ \sigma\ V\ (\mathit{map}\ \mathit{the}\ v)\ i\ \phi\}\text{"}\\ &\backslash\backslash \textit{proof incomplete}\end{aligned}$$

6.2 Monitor

MLetRec shown in Definition 81 takes only one argument more than LetRec, which is the second argument and denotes the number of free variables in ϕ. The first, third and fourth arguments are equivalent to the p, ϕ and ψ respectively. MLetRec is initialized using minit0 (Definition 82), by initializing ϕ and ψ recursively and initializing all other arguments with their initial values.

Definition 81 (LetRec mformula)**.** *The definition of the mformula data type for LetRec. (See Definition 19 for more details.) MLetRec only uses the number of free variables in the formula as extra variable.*

datatype *mformula =...*
| MLetRec *name nat mformula mformula"*
...

Definition 82 (minit0 for LetRec)**.** *The definition of the minit0 function for LetRec. (See Definition 20 for an explanation of this function.) The minit0 function recursively initializes the mformula. We use nfv ϕ instead of n in the recursive call for ϕ because ϕ is evaluated separate from the rest of the formula.*

function *minit0 where*
...
"minit0 n (LetRec p ϕ ψ) =
MLetRec p (*nfv* ϕ) (*minit0* (*nfv* ϕ) ϕ) (*minit0 n* ψ)"
...

The evaluation of MLetRec for some input event proceeds by first evaluating ϕ recursively and then using the result of that recursive evaluation in the evaluation of ψ as shown in Definition 85. The recursive evaluation of ϕ uses the function letrec_meval and is shown in Definition 84. This function recursively evaluates ϕ using the *eval* functions and the satisfying valuations for p computed in the previous step, until a fixpoint, meaning a set of results for which the previous recursion step already had the same set of results, is reached. The function while_option is used instead of a simple while to catch cases for which the loop does not terminate. The values returned, in that case, are generic default values. Lastly, if the *eval* function returns a formula of a size different than the original one, then the result is undefined. In our application, *eval* will always be meval, but this cannot be used in the definition of letrec_meval as meval uses letrec_meval in its definition, and thus, this would create a cyclic dependency.

The current definition of letrec_meval is incomplete as it only computes the result of ϕ for one input length, however, there are some formulas that can return the result for multiple input lengths at once. As an example we use the formula "Eventually $p(x)$" and the inputs no event at time-point 0, no event at time-point 1 and $p(x)$ at time-point 2. At time-point 2 the formula would be true for the input lengths 1, 2, and 3, as "Eventually $p(x)$" is observed at the same time for all of them. To get the results for all possible input lengths, letrec_meval needs to be applied, until one application returns the empty list, indicating that no verdict can be made yet.

Example 83 (Result for Multiple Input Lengths at the Same Time)**.** *This example shows a formula for which it can happen that the result of multiple input lengths is output at the same time.*

$$\phi_1 = \mathit{Eventually}\ p(x)$$

Furthermore, we assume the events, no event at time-point 0*, no event at time-point* 1 *and* $p(x)$ *at time-point* 2*. At time-point* 2 *the monitor would, thus, output the result for the input lengths* 1*,* 2*, and* 3*, as all of them conclude at the same time, as they all encounter* $p(x)$ *at the same time.*

Definition 84 (letrec_meval)**.** *Function definition for letrec_meval. The function letrec_meval uses a while loop to recursively compute the satisfying valuation of ϕ for LetRec given an input database. This function is currently incomplete, as it only evaluates the result for one ts and not for all of them, which can happen for instance with an Until operator.*

The first input to this function is an evaluation function (eval) that takes a list of ts, a database, and an mformula and outputs a pair of a list of satisfying valuations and an mformula. The other inputs are the name of the predicate that is recursively evaluated (p), the ts on which the evaluation is performed (ts), the input events (db), and the mformula that is being evaluated (ϕ). The function recursively computes the list of satisfying valuations and the new mformula that will be evaluated from this point on. The state of the evaluation is described using the tuple $((xs,\ \phi),\ xs\_old,\ first)$*, where* $(xs,\ \phi)$ *are the satisfying valuations and updated mformula of the current evaluation step. The argument xs_old holds the satisfying valuations of the previous step, which is necessary for the*

termination condition. Finally, first describes if this is the first evaluation step or not. By definition xs = xs_old in the first evaluation step, but the evaluation should still be performed.

definition *letrec_meval ::*
"(ts list ⇒ database ⇒ mformula ⇒ event_data table list × mformula) ⇒
name ⇒ ts list ⇒ database ⇒ mformula ⇒ (event_data table list × mformula)" where
"letrec_meval eval p ts db φ=
(if ((∀ ts db ψ. size ψ = size φ ⟶ size (snd (eval ts db ψ)) = size φ)) then
(case while_option (λ ((xs, φ), xs_old, first). xs ≠ xs_old ∨ first)
(λ ((xs, φ2), xs_old, first).
((eval ts (Mapping.update p (map (image (map the)) xs) db) φ), xs, False))
(([{}], φ), [{}], True)
of None ⇒ ([{}], φ) | Some y ⇒ fst y)
else undefined)

Definition 85 (meval for LetRec)**.** *The definition of the meval function for LetRec. (See Definition 21 for an explanation of this function.) First, φ is evaluated recursively using letrec_meval, then that result is used to compute the evaluation of ψ.*

function *meval where*
...
"meval n ts db (MLetRec *p m φ ψ*) =
(let (xs, φ) = letrec_meval (meval j m) p ts db φ;
(ys, ψ) = meval j n ts (Mapping.update p (map (image (map the)) xs) db) ψ
in (ys, (MLetRec *p m φ ψ*))
...

6.3 Progress

The progress of LetRec is similar to the progress of LetPrev 52. LetRec, too, computes the result of p for a recursion step using its result from the previous recursion step. The only difference is that p is not surrounded by an implicit previous operator, and thus, the progress fixpoint would not need to account for that. However, in lemma sup_alt 56 we have proven that there is a more direct expression for the progress fixpoint, which also holds if there is no implicit previous operator around p, and thus, we use this expression for the progress definition of LetRec shown in Definition 86.

Definition 86 (progress for letrec)**.** *the definition of the progress function for LetRec. (See Definition 22 for an explanation of this function.)*

fun *progress where*
...
"progress σ P (LetRec *p φ ψ*) *j* =
progress σ (P(p ↦ progress σ (P(p ↦ j)) φ j)) ψ j"
...

For a formula ϕ that is a safe_letrec formula for p, the progress of ϕ where p is mapped to x is larger or equal to the minimum of x and the progress of ϕ without the mapping of p. If ϕ does not contain p, then the progress of ϕ with p mapped to x can be larger than x even for small x, as x is not important. If ϕ does contain p, the progress can neither continue beyond the progress of p nor continue beyond the progress of any other formula contained in ϕ. The lemma min_letrec_progress_upd shown in Definition 87 summarizes this observation.

Definition 87 (min_letrec_progress_upd)**.** *Lemma statement for min_letrec_progress_upd, where the proof is omitted. This lemma states that the progress of a safe_letrec formula φ using a mapping where*

p is mapped to the progress x is larger or equal to the minimum of x and the progress of ϕ where p is not mapped to x.

lemma *min_letrec_progress_upd: "safe_letrec p ϕ $\Longrightarrow$ pred_mapping (λ x. x $\leq$ j) P*
$\Longrightarrow$ x $\leq$ j $\Longrightarrow$ progress σ (P(p $\mapsto$ x)) ϕ j $\geq$ min x (progress σ P ϕ j)"
\\proof omitted

Note: pred_mapping a P denotes that the value of every predicate p in P satisfies a.

Using the previous lemmas, the lemma letrec_progress_ge (Definition 88) can be proven. This lemma states that for a formula ϕ that is a safe_letrec formula for p, if there exists a P and a j such that some x is smaller than the progress of ϕ for P and j, then there exists a P and a j such that the same x is smaller than the progress of ϕ for P and j, where the p in P is mapped to j. This lemma allows us extend the proof of the lemma progress_ge_gen 24, which ensures that the monitor will always make progress, to LetRec formulas.

Definition 88 (letrec_progress_ge)**.** *Lemma statement for letrec_progress_ge, where the proof is omitted. This lemma states that for a safe_letrec formula ϕ for which there exists a P and a j such that x is smaller than the progress of ϕ using P at time-point j, then there exists a P and a j such that x is smaller than the progress of ϕ using P, where p is mapped to j, at time-point j.*

lemma *letrec_progress_ge: "safe_letrec p ϕ $\Longrightarrow$ p $\in$ S*
$\Longrightarrow$ ($\exists$ P j. dom P = S $\wedge$ range_mapping x j P $\wedge$ x $\leq$ progress σ P ϕ j)
$\Longrightarrow$ ($\exists$ P j. dom P = S $\wedge$ range_mapping x j P $\wedge$ x $\leq$ progress σ (P(p $\mapsto$ j)) ϕ j"
\\proof omitted

Note: range_mapping a b P denotes that the value of every predicate p in P is larger or equal to a and smaller or equal to b.

6.4 Correctness

Because the letrec_meval function is incomplete, it is not possible to fully prove VeriMon's main correctness theorem. Similarly to LetPrev a loop invariant and a loop postcondition will be used to prove the necessary properties of letrec_meval. The loop postcondition that we propose is strong enough to prove the main correctness theorem, however, the current invariant is too weak to show that the postcondition holds. Except for proof of the postcondition assuming the invariant, all proofs work. We conjecture that after updating letrec_meval it is possible to define a loop invariant strong enough to prove the postcondition.

We remember the high-level summary of VeriMon's main correctness theorem (Definition 27): if the monitor state invariant holds and the monitoring environment is well-formed, then the evaluation of a given formula will result in the correct satisfying valuations for that formula and the monitor state invariant will still hold. One of the assumptions in the main correctness proof is the monitor state invariant for the given formula. The definition of this invariant for LetRec is shown in Definition 89 and mainly specifies the safety of ϕ' as otherwise infinite recursion could occur.

Definition 89 (wf_mformula for LetRec)**.** *The definition of the wf_mformula function for LetRec. (See Definition 22 for an explanation of this function.) Both the ϕ and ψ of LetRec need to be a wf_mformula regarding their respective P and V mappings. Furthermore, the safety properties for*

phi′ must hold.

inductive *wf_mformula where*

$$
\begin{aligned}
&\ldots\\
&\mid \textit{LetRec: ``wf\_mformula } \sigma\ j\ (P(p \mapsto j))\\
&\quad (V(p \mapsto (\lambda\ i.\ \bigcup k.\ \textit{letrec\_sat}\ m\ (\lambda\ X\ v\ i.\ \textit{sat}\ \sigma\ (V(p \mapsto X))\ v\ i\ \phi')\ (\lambda\ i\ \{\})\ k\ i)))\\
&\quad m\ \phi\ \phi'\\
&\Longrightarrow \textit{wf\_mformula}\ \sigma\ j\ (P(p \mapsto \textit{progress}\ \sigma\ (P(p \mapsto j))\ \phi'\ j))\\
&\quad (V(p \mapsto (\lambda\ i.\ \bigcup k.\ \textit{letrec\_sat}\ m\ (\lambda\ X\ v\ i.\ \textit{sat}\ \sigma\ (V(p \mapsto X))\ v\ i\ \phi')\ (\lambda\ i\ \{\})\ k\ i)))\\
&\quad n\ \psi\ \psi'\\
&\Longrightarrow \textit{safe\_letrec}\ p\ \phi' \Longrightarrow \{0..<m\} \subseteq \textit{fv}\ \phi' \Longrightarrow m = \textit{nfv}\ \phi'\\
&\Longrightarrow \textit{wf\_mformula}\ \sigma\ j\ P\ V\ n\ (\text{MLetRec}\ p\ m\ \phi\ \psi)\ (\text{LetRec}\ p\ \phi'\ \psi')\text{''}\\
&\ldots
\end{aligned}
$$

In the proof of the main correctness theorem, the wf_mformula assumption for LetRec and the wf_envs 26 assumption can be used to prove for LetRec, that the monitor state invariant still holds after one evaluation step and that the computed valuations are the correct satisfying valuations. Furthermore, as the main correctness theorem is proven using induction, the induction hypotheses for ϕ and ψ are given. One evaluation step can entail multiple iterations of the letrec_meval loop, thus, it cannot be assumed that all desired properties can be shown after one execution of the letrec_meval loop body. This issue can be solved by defining a loop postcondition that states all properties that should hold after letrec_meval terminates and a loop invariant that should for every loop iteration in letrec_meval. By proving the three properties: the letrec_meval loop terminates in finite time, the loop invariant holds for the first call to letrec_meval and if the loop invariant holds initially then the loop postcondition holds for the return values, the postcondition can be proven for the call to letrec_meval in meval.

The postcondition for letrec_meval shown in 91 incorporates all necessary information about the result of letrec_meval to prove both the monitor state invariant as well as the correctness of the satisfying valuations. We shortly summarize the properties that should hold after letrec_meval concludes. The updated mformula returned by letrec_meval needs to adhere to the monitor state invariant. The valuations computed by letrec_meval should be the correct satisfying valuations for their respective input lengths.

The invariant for letrec_meval shown in 90 describes the information that can be gained from each call of letrec_meval. Unlike letprev_meval, letrec_meval uses the same list of timestamps and the same mformula in every loop iteration. The loop invariant consists of a single statement guarding that the monitor state invariant was satisfied for the input length without the new timestamps and for the initial mformula with which letrec_meval is called.

Definition 90 (letrec_meval_invar)**.** *Definition for letrec_meval_invar. The loop invariant for the letrec_meval loop.*

definition *letrec_meval_invar where*

$$
\begin{aligned}
&\textit{``letrec\_meval\_invar}\ n\ V\ \textit{sigma}\ P\ \phi'\ m\ j'\ \textit{ys}\ p\ \textit{ts}\ \textit{db}\ \phi =\\
&(\textit{let}\ j = j' - \textit{length ts}\ \textit{in}\\
&\ \textit{wf\_mformula}\ \sigma\ j\ (P(p \mapsto j))\\
&\quad (V(p \mapsto (\lambda\ i.\ \bigcup k.\ \textit{letrec\_sat}\ m\ (\lambda\ X\ v\ i.\ \textit{sat}\ \sigma\ (V(p \mapsto X))\ v\ i\ \phi')\ (\lambda\ i\ \{\})\ k\ i)))\\
&\quad m\ \phi\ \phi')\text{''}
\end{aligned}
$$

Definition 91 (letrec_meval_post)**.** *Definition for letrec_meval_post. The postcondition for the le-*

trec_meval loop.

definition *letrec_meval_post where*

"letrec_meval_post n V sigma P P' ϕ' m j xs p ts db ϕf =
(wf_mformula σ j (P'(p $\mapsto$ j))
(V(p $\mapsto$ (λ i. $\bigcup$ k. letrec_sat m (λ X v i. sat σ (V(p $\mapsto$ X)) v i ϕ') (λ i {}) k i)))
m ϕf ϕ' $\wedge$
list_all2 (λ i. qtable m (fv ϕ')
(λ v. map the v $\in$
($\bigcup$ k. letrec_sat (nfv ϕ') (λ X v i. sat σ (V(p $\mapsto$ X)) v i ϕ') (λ i. {}) k i)))
[progress σ (P(p $\mapsto$ (j - length ts))) ϕ' (j - length ts)..<
progress σ (P'(p $\mapsto$ j)) ϕ' j] xs)"

The loop invariant should hold for any values with which letrec_meval is called, and thus, it should also hold for the values with which letrec_meval is called initially. The lemma letrec_invar_init shown in Definition 92 proves that the loop invariant holds for the values with which letrec_meval.

Definition 92 (letrec_invar_init)**.** *Lemma statement for letrec_meval_invar_init, where the proof is omitted. This lemma states that the loop invariant for letrec_meval holds for the initial values with which letrec_meval is called.*

lemma *letrec_meval_invar_init:*
assumes *"wf_mformula σ (j - length ts) P V n (MLetRec p m ϕ ψ) (LetRec p ϕ' ψ')"*
shows *"letrec_meval_invar n V σ P ϕ' m j [{}] p ts db ϕ*
\\proof omitted

The next step is to prove that the postcondition holds for the letrec_meval loop if the loop invariant holds initially. This is not possible with the definition of current definition letrec_meval and letrec_meval_invar as the postcondition assumes the correct satisfying valuations for all input lengths for which a verdict can be made and not just for one. We conjecture that after completing the letrec_meval function, we will be able to define a loop invariant strong enough to show the loop postcondition. Furthermore, the termination of letrec_meval for a safe_letrec formula has yet to be shown. This should be possible using the same method as we conjecture for safe_formula_finite 80. Definition 93 shows the lemma that needs to be proven.

Definition 93 (letrec_invar_post)**.** *Conjecture statement for letrec_invar_post, where the proof is incomplete. This conjecture states that if the loop invariant holds before executing letrec_meval then the*

postcondition will hold after letrec_meval terminates.

lemma *letrec_invar_post:*
assumes *"safe_letrec p ϕ'"*
assumes *"m = nfv ϕ"*
assumes *"{0..< m} ⊆ m"*
assumes *"wf_mformula σ (j - length ts) P V n* (MLetRec *p m ϕ ψ*) (LetRec *p ϕ' ψ'*)"
assumes *"wf_envs σ (j - length ts) (length ts) P P' V db"*
assumes *"$\bigwedge$ xs ϕ_m P P' V. size ϕ_m = size ϕ*
$\Longrightarrow$ wf_mformula σ (j - length ts) P V m ϕ_m ϕ'
$\Longrightarrow$ wf_envs σ (j - length ts) (length ts) P P' V
(Mapping.update p (map ((') (map the)) xs) db)
$\Longrightarrow$ case meval j m ts (Mapping.update p (map ((') (map the)) xs) db) ϕ_m
of (xs', ϕ_n) ⇒
wf_mformula σ j P' V m ϕ_n ϕ' ∧
list_all2 (λ i. qtable m (fv ϕ') (λ v. sat σ V (map the v) i ϕ'))
[progress σ P ϕ' (j - length ts)..< progress σ P' ϕ' j]"
shows *"letrec_meval_invar n V σ P ϕ' m j ys p ts db ϕ*
$\Longrightarrow$ (ys', ϕf) = letrec_meval m j p ts db ϕ
$\Longrightarrow$ letrec_meval_post n V σ P P' ϕ' m j ys' p ts db ϕf"
\\proof incomplete

The well-formedness of the environment expressed through wf_envs is a necessary condition for the main correctness proof and, in turn, also for the induction hypotheses it provides. The induction hypotheses are often used with updated P, P', and V, and thus, it is necessary to update the wf_envs. For LetRec we use one updated wf_envs where the P and P' are updated to the progress of ϕ at input lengths j and $j + \delta$ respectively. All these lemmas and conjectures combined would be sufficient to prove the main correctness proof.

Definition 94 (wf_envs_update_letrec)**.** *Lemma statement for wf_envs_update_letrec, where the proof is omitted. This lemma states that if we have a wf_envs for some P, P' and V, we can get a wf_envs for an updated P, P' and V.*

lemma *wf_envs_update_letrec:*
assumes *"wf_envs σ j δ P P' V db"*
assumes *"m = nfv ϕ"*
assumes *"{0..< m} ⊆ m"*
assumes *"list_all2 (λ i. qtable m (fv ϕ)*
(λ v. map the v ∈
($\bigcup$ j. letrec_sat (nfv ϕ') (λ X v i. sat σ (V(p ↦ X)) v i ϕ') (λ i. {}) j i)))
[progress σ (P(p ↦ j)) ϕ' j..< progress σ (P'(p ↦ (j + δ))) ϕ' (j + δ)] xs'"
shows *"wf_envs σ j δ (P(p ↦ progress σ (P(p ↦ j)) ϕ j))*
(P'(p ↦ progress σ (P'(p ↦ (j + δ))) ϕ (j + δ)))
(V(p ↦ (λ i. $\bigcup$ j. letrec_sat m (λ X v i. sat σ (V(p ↦ X)) v i ϕ') (λ i {}) j i)))
(Mapping.update p (map (image (map the)) xs') db)"
\\proof omitted

7 Conclusion

In this thesis, we explored ways in which to enhance VeriMon with a recursive let operator (Let p ϕ ψ). This operator replaces any occurrence of p in ψ with ϕ, which can be achieved by first evaluating ϕ and then evaluating ψ with p set to the evaluations of ϕ. As p might occur in ϕ, the formula ϕ needs to be evaluated recursively. This may be problematic as the recursive evaluation of ϕ may not terminate, which would render it unmonitorable. We identified four groups of operators that could cause infinite recursion, namely, future operators above p, negation operators above p, equality operators, and aggregation operators. Furthermore, the recursion would terminate if any occurrence of p in ϕ was strictly in the past. These two ideas are the foundation for the two recursive let operators, LetPrev and LetRec, that we propose. LetPrev enforces that every occurrence of p in ϕ is strictly in the past by adding an implicit previous operator in front of every p and disallowing any future operator above p. LetRec, on the other hand, disallows the four groups of operators that we found to be problematic. It may seem as if one of the operators is strictly stronger than the other, however, for each method, there are formulas that can be evaluated by that method but not by the other. Merging the two operators would also not be helpful as their different assumptions lead to different implementations.

We integrated both of these operators into VeriMon, by adding LetPrev and LetRec to the definitions, functions, and proofs that already existed as well as adding new definitions and lemmas as needed. For the recursive operators the idea of a fixpoint, a set of results for which all subsequent recursion steps generate the same result, was used frequently. For instance, the definitions of sat and meval for LetRec were based on the fixpoint. Furthermore, the progress of LetPrev was based on the idea of a fixpoint. For some lemmas, the existence and uniqueness of that fixpoint were required, which we were both able to prove. Furthermore, we introduced and proved a more direct expression for the fixpoint, which we subsequently used for LetRec. The evaluation of ϕ in both LetPrev and LetRec may require multiple recursive calls. We used loop invariants and postconditions to make statements about these functions. We were able to fully define and prove all necessary components for LetPrev, culminating in the proof of VeriMon's main correctness theorem for LetPrev. LetRec has not yet been fully proven, as some functions are incomplete. We give some ideas for how these functions could be completed.

Future Work To further enhance the results of this thesis, we propose following ideas for future work, some of which are currently already being worked on.

- **Completing Proofs** There are some proofs for LetRec that were left due to time constraints. Specifically, the sat_slice_strong and the safe_formula_finite lemmas were left unproven. Furthermore, the definition of letrec_meval is currently incomplete, which is why the postcondition of that function cannot be proven yet. Unfortunately, this means that meval currently has not been proven for LetRec. However, we believe that this should be possible.

- **Optimizing Computations** Some of the current functions use more steps than would be necessary. Optimizing these functions may make the monitor run faster, and thus, be more viable. For instance, in the recursive computation of LetRec, in each recursive step, all predicates are evaluated again, instead of only those that got added newly.

- **Enhance LetPrev / LetRec** It may be possible to tweak the definition of these operators a bit, to allow for more generality. For instance, it may seem a bit excessive to disallow all equality operators in LetRec. Instead, it may be possible to disallow only those that contain a variable. Another idea would be to change LetPrev such that the implicit Prev operator is no longer implicit, does not have to be Prev, and does not have to occur right in front of p, which would increase its scope.

- **Testing** The new operators have yet to be tested. This would show us clearly how long the evaluation of different formulas containing the new operators take, which in turn would let us know if such an evaluation is viable for a real-time monitor.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004.

[3] E. Bartocci and Y. Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[4] D. Basin, T. Dardinier, L. Heimes, S. Krstić, M. Raszyk, J. Schneider, and D. Traytel. A formally verified, optimized monitor for metric first-order dynamic logic. In *International Joint Conference on Automated Reasoning*, pages 432–453. Springer, 2020.

[5] D. A. Basin, B. N. Bhatt, S. Krstic, and D. Traytel. Almost event-rate independent monitoring. *Formal Methods Syst. Des.*, 54(3):449–478, 2019.

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

[7] D. A. Basin, F. Klaedtke, S. Müller, and E. Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[8] D. A. Basin, F. Klaedtke, and E. Zalinescu. The monpoly monitoring tool. In G. Reger and K. 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.

[9] H. Ebbinghaus and J. Flum. *Finite model theory*. Perspectives in Mathematical Logic. Springer, 1995.

[10] K. Havelund and D. Peled. An extension of LTL with rules and its application to runtime verification. In B. Finkbeiner and L. 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 239–255. Springer, 2019.

[11] J. Lelli, C. Scordino, L. Abeni, and D. Faggioli. Deadline scheduling in the linux kernel. *Softw. Pract. Exp.*, 46(6):821–839, 2016.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[13] J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In *International Conference on Runtime Verification*, pages 310–328. Springer, 2019.

[14] Schneider, Joshua. Recursive Semantics. Internal document (not to be published).

ETH

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 Evaluation of Recursive Expressions in Metric First-Order Dynamic Logic

Authored by (in block letters):

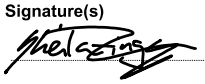
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Zingg	Sheila Rina

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)**
Geroldswil, 14.07.2021	

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