Software Tools for Technology Transfer (accepted manuscript)

Monitoring Modulo Theories

Normann Decker, Martin Leucker and Daniel Thoma

Institute for Software Engineering and Programming Languages
University of Lübeck, Germany

{decker, leucker, thoma}@isp.uni-luebeck.de

Abstract. This paper considers a generic approach to runtime verification of temporal properties over first-order theories. This allows especially for the verification of multi-threaded, object-oriented systems. It presents a general framework lifting monitor synthesis procedures for propositional temporal logics to a temporal logic over structures within some first-order theory. To evaluate such specifications SMT solving and classical monitoring of propositional temporal properties are combined. The monitoring procedure was implemented for linear-time temporal logic (LTL) based on the Z3 SMT solver and evaluated regarding runtime performance.

1 Introduction

Modern software systems are increasingly complex and exhibit a highly dynamic behaviour, for example due to an object-oriented design and multi-threaded execution. The analysis of such systems is therefore an important task that is difficult to achieve statically. Important information on a program's execution is often only available at runtime, for example when it depends heavily on its execution environment or when third party code is used.

Runtime Verification is therefore concerned with techniques for efficient and precise monitoring of concrete system executions. Rather than considering the whole possible behaviour of the system under scrutiny the aim is to check whether a particular run satisfies or violates a given correctness property. Runtime verification can be considered as a a lightweight verification technique complementing others such as model checking [CGP01] and testing [BJK+05].

The major challenge is to provide intuitive and expressive specification formalisms that can yet be translated into an operational model, called a *monitor*, allowing for efficient execution in order to verify the property on a given input. Monitoring can be performed based on log-files or on-the-fly and in the latter case an additional issue is the observation of relevant system events with minimal interference.

Various concepts and implementations have been proposed that approach these goals with different strategies. Monitors can be specified directly in a rather programmatic fashion in terms of automata [CPS09, BFH+12] thereby omitting complex synthesis procedures. Most frameworks [CR05, BRH07, BLS11, BKM10] implement monitor synthesis procedures for high-level, declarative specification formalisms based on temporal logic or grammars. Regarding synthesis, there is an inherent trade-off between expressiveness of the specification formalism and efficiency of the evaluation. In particular for very expressive formalisms generation and optimisation of monitors is conceptually challenging and represents a central research question in the field of runtime verification. For a survey on different monitoring approaches see [LS09].

The integration of monitors for on-line monitoring is typically accomplished using instrumentation frameworks [CR05, DLT13b]. However, monitor execution can also be performed exterior to the system by independent components [DKT14] or even on dedicated hardware [BHW+13], which makes observation of the system less intrusive.

Monitoring and Data. In recent years, various monitor synthesis algorithms have been developed that differ in the expressiveness of the underlying specification formalism and the resulting monitoring approach. Within the setting of multiple, in general arbitrarily many instances of program parts, for example in terms of threads or objects, a software engineer is naturally interested in verifying that the interaction of individual instances follows general rules. The ability of taking into account the dynamics of process IDs, individual objects and data

structures and values in general is a desirable feature for specification and verification approaches. As such, the expressiveness of plain propositional temporal logics such as LTL does not suffice, as they do not allow for specifying complex properties on the data processed in a system.

On the other hand, substantial theoretical and technical advances have been achieved in automatic reasoning in first-order logic. Today's *SMT solvers* are highly optimised tools that can check the satisfiability of formulae over a variety of first-order theories such as arithmetic, arrays, lists and uninterpreted functions. They allow for reasoning on a large class of data structures used in modern software systems. These advances have already led to significant improvements in verification technology [dMB11].

Outline. In this article we enhance traditional runtime verification techniques for propositional temporal logics by first-order theories for reasoning about data, based on SMT solvers. The proposed framework provides a powerful tool for verifying complex properties at runtime which exceeds the expressiveness and generality of previous approaches.

Regarding specification, a combination of temporal logic and first-order logic is used that is formally defined in Section 3. Both parts are generic and the monitor construction for the combination is reduced to that of the temporal logic alone. This way the user can choose a suitable temporal logic and reuse any corresponding monitor construction. Independently of the temporal logic, data is handled by the first-order logic part and any first-order theory can be chosen for which an SMT solver is available. The approach hence can benefit from independent advances in both fields. A selection of example properties shows the specific strength of the framework in terms of expressiveness.

The monitoring algorithm takes observations from the system as input and executes a monitor incrementally. The details are presented in Section 4 where we also reason about correctness based on the formal semantics of the logic. The monitor output provides the information on whether the specification is violated on the observed execution trace. An evaluation procedure refines this information. It is presented in Section 5 where we also prove that the computed verdict coincides with the semantics of the specification. Figure 1 shows a schematic view on the monitoring approach.

To provide evidence that the framework is suitable for practical applications Section 6 reports on its implementation in our runtime verification tool jUnit$^{\text{RV}}$ [DLT13b]. Benchmarks for monitoring Java programs show that specifications can be monitored efficiently. For an extensive case study in the context of medical devices we also refer the reader to [DKT14].

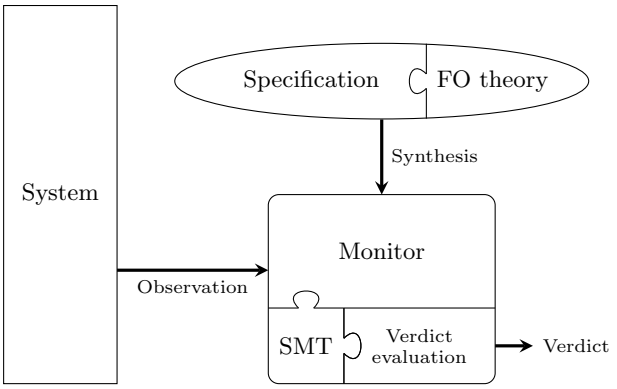


Fig. 1. Schematic overview of the monitoring approach.

1.1 Combining Monitoring and SMT

In the following we outline the idea of our approach by means of an example. Consider a mutual exclusion property where a resource must not be accessed while it is locked, stated in LTL as $\mathrm{G}(\mathsf{lock} \rightarrow \neg \mathsf{access}\,\mathrm{U}\,\mathsf{unlock})$. If there are several resource objects available at runtime, this is too restrictive and one might specifically limit foreign access to locked resources. Using variables r and p, p', intended to represent resources and processes, respectively, and suitable predicates, the property can be stated as

$$\forall_p \forall_r \;\; \mathrm{G}(\mathsf{lock}(p,r) \rightarrow (\forall p' \neq p : \neg\mathsf{access}(p',r))\,\mathrm{U}\,\mathsf{unlock}(p,r)). \tag{1}$$

The variables r and p are quantified globally whereas the quantification of p' is local meaning that temporal behaviour is not relevant. Formally, all variables range over some universe that is fixed by the application. In our example, this could be the set of object identifiers in a particular Java program.

Data theories. To define a formal semantics for expressions as those above we note that we essentially use an LTL formula and exchanged propositions by first-order formulae. In LTL, propositions are evaluated at a position in some word, i.e. a letter. To evaluate first-order formulae, such a letter must now be a first-order structure describing a system state. In the example we fixed the universe to object IDs and used a binary predicate "=" indicating equality. This is covered by the first-order theory of natural numbers with equality and can be handled by essentially all SMT solvers. It is possible to use more powerful theories, e.g., with linear order or arithmetics. Section 3.4 provides more examples.

What remains are the predicates that are not part of the theory. These specifically characterise the current system state or, more accurately, are interpreted by the current system state in terms of the current observation. We call them *observation predicates*, in the example we use binary predicates lock, unlock and access. Inspecting

the program states, we obtain, at any time, an observation g that interprets all observation predicates in terms of relations on the universe.

The first-order formulae reason about the data structures under a specific observation. We therefore refer to this logic as *data logic*. Data logic formulae may have free variables, such as p and r in the example. Summing up, we can define the semantics of a data logic formula in terms of (1) an observation interpreting the observation predicates (and possibly observation functions), (2) a theory that fixes the interpretation of all other predicates and functions and (3) a valuation that assigns a value from the universe to each free variable.

Temporal data logic. In the example, the data logic replaces the propositional part that LTL is based on. To the logic expressing the temporal aspect, we generically refer to as the *temporal logic*. Our assumptions on the temporal logic must be that it is linear (defined on words) and that it only uses atomic propositions to "access" the word. For example, the semantics of some temporal operator must not depend on the current letter directly but only on the semantics of some proposition. We formally define that requirement in Section 3 but for now only remark that typical temporal logics like LTL, the linear μ-calculus or the temporal logic of calls and returns (CaRet) [AEM04] fit into that schema.

Given a suitable temporal logic and data logic we can define the formalism we aim at. Taking the temporal logic and replacing the atomic propositions by data formulae, we obtain what we call a *temporal data logic*. The theory and universe is fixed by the data logic and the semantics of temporal data logic formulae can thus be defined over a *sequence of observations*. If the global variables are bound universally the formula is evaluated over the observation sequence for all possible valuations. The semantics of the formula is then the conjunction (more precisely, the infimum—or meet) of these results. In general, alternating global quantifiers correspond to nested meet and join operations over the valuation space.

Monitor construction. Assuming a monitor construction for the (propositional) temporal logic, we can evaluate a sequence of observations on-the-fly. The idea is to construct a *symbolic* monitor that deals atomically with data formulae. In the example formula (Formula 1) we treat $\mathsf{lock}(p, r)$, $\forall p' \neq p : \neg\mathsf{access}(p', r)$ and $\mathsf{unlock}(p, r)$ as three atomic propositions, say χ_1, χ_2 and χ_3. We obtain a temporal logic formula $\mathrm{G}(\chi_1 \rightarrow \chi_2 \,\mathrm{U}\, \chi_3)$ over a set of atomic propositions $AP = \{\chi_1, \chi_2, \chi_3\}$. A monitor can then be constructed that reads words over the finite symbolic alphabet $\Sigma := 2^{AP}$.

The free variables in these formulae are p and r and range over the universe of natural numbers $\mathbb{N}$. Given a valuation $\theta : \{p, r\} \rightarrow \mathbb{N}$ for those, mapping, e.g., p to $\theta(p) = 1$ and r to $\theta(r) = 2$, we can map an observation g to the letter $a \in \Sigma$ that contains all formulae that are satisfied by g. For example, say g interprets the observation predicates as $\mathsf{lock}_g = \{(1, 2), (10, 7)\}$ and $\mathsf{access}_g = \mathsf{unlock}_g = \emptyset$ (because objects 1 and 10 happen to lock the resources 2 and 7, respectively, and otherwise nothing happened in the current execution step of the program). Then, under θ, g is mapped to $a = \{\chi_1, \chi_2\} \in \Sigma$ since χ_1 and χ_2 hold but χ_3 does not. The observation g might be mapped to some other symbolic letter for another valuation θ'. If, for example, $\theta'(p) = 2$ then χ_1 does not hold and g is projected to $a' = \{\chi_2\} \in \Sigma$.

We present a monitoring algorithm that maintains a copy of the symbolic monitor for each valuation. For a new observation, the algorithm simulates the individual transition for each copy by projecting the observation under the specific valuation. As the universe is in general infinite, the number of monitor instances is infinite as well but the algorithm uses a data structure to finitely represent the state of all monitor instances.

Global quantification over all monitor instances can be evaluated using this finite representation. In the case of only universal quantifiers, this amounts to computing the meet over a finite set of representatives for all monitor instances. For the general case, a refinement step is needed for each quantifier alternation.

1.2 Related work

In runtime verification, handling data values to reason about the computation of a system more precisely has always been a concern. Most approaches extend some specification formalism towards data values. Mainly, two classes of specification formalisms can be distinguished: declarative formalisms like temporal logics and more operational models like automata.

Data values for runtime verification were already considered in [AAC+05]. They use a simple expression language based on the instrumentation framework AspectJ that allows for free variables that are bound to values occurring in the observed trace.

One of the first works extending LTL by parameters is by Stolz and Bodden [SB06]. Binding of parameters of propositions takes place in a Prolog-style fashion and the resulting approach is reasonable for the intended applications. However, no precise denotational semantics is given.

The works on Eagle [BGHS04] and RuleR [BRH07] consider first-order safety properties. The corresponding systems come with a rewriting-based semantics and are well-suited for specifying properties of especially finite, yet perhaps expanding traces. A comparable approach is implemented in the tool LogFire [Hav15] using the Rete algorithm [For82] known from artificial intelligence to evaluate rule systems.

In [DLS08] a runtime verification approach for the temporal evaluation of integer-modulo-constraints was presented. The underlying logic has a decidable satisfiability problem and the overall approach is anticipatory.

However, only limited computations can be followed. To reason about the temporal evolution of data values along some computation, some form of bounded unrolling like in bounded model checking [BCRZ99] can be used. For runtime verification, however, such an approach is not suitable, as the observed trace cannot be bounded.

An extension of LTL with a special binding operator has been studied in [Sto10]. It allows for binding a variable to a data value appearing at the current position in a trace.

Basin et al. [BKM10] consider a safety fragment of metric first-order temporal logic where temporal real-time properties can be expressed over first-order constraints. Quantification is, however, restricted and temporal operators are bounded.

Closely related to our work is that of Chen and Rosu [CR09, MJG+12]. It considers the setting of sequences of actions which are parameterised by identifiers (IDs). The main idea is to divide the sequence of a program into sub-sequences, called slices, containing only a single ID. The slices are then monitored independently. Hence, in contrast to our approach, no interdependencies between the different slices can be checked. Moreover, our monitoring approach is not limited to plain IDs but allows the user to reason more generally over data in terms of arbitrary (decidable) first-order theories. The work considers a dedicated temporal logic (LTL) together with the dedicated notion of parameters, whereas in our framework an arbitrary linear temporal logic can be extended by a first-order theory.

Quantified Event Automata (QEA) [BFH+12] are similar to the concept of slicing but are more expressive since a single monitor instance can be associated with multiple IDs and thus observe actions from multiple slices. Quantification is limited to finite domains and relies on automata being used as specification formalism. QEA can be encoded into our formalism using the theory of IDs and a sufficiently expressive temporal formalism such as linear μ-calculus or regular extensions of LTL. A Scala DSL based on similar ideas as has been presented in [Hav14].

Recently, Bauer et al. [BKV13] presented an approach combining LTL with a variant of first-order logic for runtime verification. However, their approach restricts quantification to finite sets always determined in advance by the system observation. This allows for finitely instantiating quantifiers during monitor execution, but also profoundly limits the expressiveness of first-order logic. Basically, it is only possible to evaluate first-order formulae over finite system observations, and not to express properties in a declarative manner.

In summary, the ability of using arbitrary first-order theories to reason on data in combination with rich temporal logics makes our framework exceed the expressiveness of previous approaches. Clearly, this needs to come at the cost of efficiency but the framework's flexibility allows the user to freely choose a suitable trade-off. Whilst we do not consider real-time constraints as first-class citizens here such properties can be expressed using a suitable first-order theory.

The present article is an extension of [DLT14] where global quantification was restricted to be only universal.

2 Preliminaries

First-order logic. A *signature* $S = (P, F, ar)$ consists of finite sets P, F of predicate and function symbols, respectively, each of some arity defined by $ar : P \cup F \to \mathbb{N}$. An *extension* of S is a signature $T = (P', F', ar')$ such that $P \subseteq P'$, $F \subseteq F'$ and $ar \subseteq ar'$.

The *syntax* of first-order formulae over the signature S is defined in the usual way (see, for example, [EFT94]) using operators $\vee$ (or), $\wedge$ (and), $\neg$ (negation), variables $x_0, x_1, \ldots$, predicate and function symbols $p \in P$, $f \in F$, quantifiers $\forall$ (universal), $\exists$ (existential). *Free* variables are not in the scope of some quantifier and are assumed to come from some set $\mathcal{V}$. The set of all first-order formulae over a signature S is denoted $\mathrm{FO}[S]$. We consider constants as function symbols f with $ar(f) = 0$. A *sentence* is a formula without free variables.

An *S-structure* is a tuple $s = (\mathcal{U}, \mathfrak{s})$ comprising a non-empty *universe* $\mathcal{U}$ and a function $\mathfrak{s}$ mapping each predicate symbol $p \in P$ to a relation $p_s \subseteq \mathcal{U}^n$ of arity $n = ar(p)$ and each function symbol $f \in F$ to a function $f_s : \mathcal{U}^m \to \mathcal{U}$ of arity $m = ar(f)$. A T-structure $t = (\mathcal{T}, \mathfrak{t})$ is an *extension* of s if T is an extension of S, $\mathcal{T} = \mathcal{U}$ and $\mathfrak{s}(r) = \mathfrak{t}(r)$ for all symbols $r \in P \cup F$.

A *valuation* is a mapping $\theta : \mathcal{V} \to \mathcal{U}$ of free variables to values. The set of all such mappings may be denoted $\mathcal{U}^{\mathcal{V}}$. The semantics of first-order formulae is defined as usual. We write $(s, \theta) \models \chi$ if a formula χ is satisfied for some structure s and valuation θ. For sentences, we refer to a sole satisfying structure as a *model*, omitting a valuation. The *theory* $\mathfrak{T}$ of an S-structure s is the set of all sentences χ such that s is a model for χ.

Temporal specifications. We use AP to denote a finite set of *atomic propositions* and $\Sigma := 2^{AP}$ for the finite alphabet over AP. For arbitrary, possibly infinite alphabets we mostly use Γ. A *word* over some alphabet Γ is a sequence of letters from Γ and Γ^*, Γ^ω denote the sets of finite and infinite words over Γ, respectively.

Monitor. A *monitor* $\mathcal{M} = (Q, \Gamma, \delta, q_0, \lambda, \Lambda)$ for a temporal property is a state machine with output where Q is a possibly infinite set of states, Γ is a possibly infinite input alphabet, $\delta : Q \times \Gamma \to Q$ is a deterministic transition function and $\lambda : Q \to \Lambda$ is a labelling function mapping states to labels from the set Λ. The operational semantics is defined in the usual way in the fashion of Moore machines. For transition functions $\delta : Q \times \Gamma \to Q$ we define the common extension $\delta^* : Q \times \Gamma^* \to Q$ to words by $\delta^*(q, \epsilon) := q$ and $\delta^*(q, \gamma g) := \delta(\delta^*(q, \gamma), g)$ for $\gamma \in \Gamma^*, g \in \Gamma$.

Trees. A *binary tree* is a tuple (V, S_1, S_2) consisting of a set V of vertices, or nodes, and two anti-symmetric, irreflexive successor relations $S_1, S_2 \subseteq V \times V$ such that for all nodes $v \in V$ there is at most one S_1-successor and one S_2-successor and there is a unique root node $r \in V$ that does not have a predecessor. A binary tree is full if every node has either both, an S_1-successor and an S_2-successor or none. We depict and refer to S_1 and S_2 as *left* and *right* side, respectively.

3 Temporal Data Logic

The aim of the framework is to enable the user to specify and check complex properties of execution traces. As described above we consider two aspects, time and data. Note that we refer to *discrete* time, as opposed to continuous notions like in timed automata. In this section we therefore formalise how and under which assumptions two logics considering time (temporal logic) and data (data logic) can be combined to a specification formalism (temporal data logic) that can express the timely behaviour of a system with respect to the data it processes. The clear separation of the aspects will give rise to a monitoring procedure.

3.1 Temporal Logic

The notion of a temporal logic (*TL*) that we consider for our monitoring framework is inspired by the intuition for LTL which is widely used for behavioural specifications, in particular in runtime verification. However, our monitoring approach does only rely on some specific properties that are shared by other, also more expressive logics. In the following we identify the required features of a suitable temporal logic for our framework.

We require the desired temporal behaviour to be specified in a finitary, linear logic, that is, the semantics is defined on *finite words* over some alphabet Γ. The truth values of the semantics need to come from a complete lattice $(\mathbb{S}, \sqcap, \sqcup)$ since we will handle multiple monitor instances and combine individual verdicts.

Second, there must be a *monitor construction* for the logic in question since our framework is intended to generically lift such a construction for handling data. We assume that such a construction turns a *TL* formula φ into a Moore machine $\mathcal{M}_\varphi$ with output $\mathcal{M}_\varphi(w) = [\![\varphi]\!](w)$ for $w \in \Gamma^*$. The restriction to Moore machines is not essential, our constructions are applicable to similar models, including Mealy machines and we do not rely on a finite state space.

As we aim at replacing atomic propositions, we require that the semantics of the temporal logic can only distinguish letters by means of the semantics of such propositions. This allows for lifting the semantics from a propositional to a complex alphabet where letters have more internal structure.

Proposition semantics. We formalise the distinction of positional and temporal aspects of a temporal logic formula using a *proposition semantics* $ps : AP \to 2^\Gamma$ mapping propositions $p \in AP$ to the set of letters $ps(p) \subseteq \Gamma$ that satisfy the proposition. Given, that the semantics of some propositional temporal logic can be defined by only referring to letters using a proposition semantics, it can be substituted without influencing the temporal aspect.

We refer to the canonical semantics for $\Gamma = \Sigma = 2^{AP}$ as $\mathsf{ps}_{AP} : AP \to 2^\Sigma$, with $\mathsf{ps}_{AP}(p) := \{a \subseteq AP \mid p \in a\}$. It is the "sharpest" in the sense that it distinguishes maximally many letters by means of combinations of propositions.

Symbolic abstraction. For an alphabet Γ, atomic propositions AP and a proposition semantics $ps : AP \to 2^\Gamma$, let $\pi_{ps} : \Gamma \to \Sigma$ be a projection with $\pi_{ps}(g) := \{p \in AP \mid g \in ps(p)\}$, mapping a letter $g \in \Gamma$ to the set of propositions that hold for it. For convenience, we lift the projection to words $g_1 \dots g_n$ ($g_i \in \Gamma$) by $\pi_{ps}(g_1 \dots g_n) := \pi_{ps}(g_1) \dots \pi_{ps}(g_n)$. Using π_{ps}, we consider the letters from Σ as symbolic abstractions of Γ with respect to AP and ps in the sense that π_{ps} maintains all the structure of Γ that is relevant for evaluating (Boolean combinations of) propositions from AP.

As argued above, for the purpose of lifting a temporal logic over atomic propositions to propositions carrying data, i.e., structure, it is essential that the semantics of propositions can be encapsulated and exchanged without influencing the temporal aspect. We can formalise this requirement on a temporal logic *TL* using the symbolic abstraction. We assume the semantics of a *TL* formula φ to be a mapping that takes linear sequences from Γ^* and assigns a truth value from the complete lattice $\mathbb{S}$. If the semantics satisfies our criterion we can make the proposition semantics $ps : AP \to 2^\Gamma$ an explicit parameter and assume the semantics of a formula φ is given by a mapping $[\![\varphi]\!](ps) : \Gamma^* \to \mathbb{S}$, or, generally, $[\![\varphi]\!] : (AP \to 2^\Gamma) \to (\Gamma^* \to \mathbb{S})$. Moreover, projecting the input word to a symbolic word and evaluating $[\![\varphi]\!](\mathsf{ps}_{AP})$ on it must not change the result.

Definition 1 (Propositional semantics). Let AP be a set of atomic propositions and Γ an alphabet. A semantics $[\![\varphi]\!] : (AP \to 2^\Gamma) \to (\Gamma^* \to \mathbb{S})$ is *propositional* iff for all proposition semantics $ps : AP \to 2^\Gamma$ and all words $\gamma \in \Gamma^*$

$$[\![\varphi]\!](ps)(\gamma) = [\![\varphi]\!](\mathsf{ps}_{AP})(\pi_{ps}(\gamma)).$$

Based on that notion of propositional semantics we can summarise the formal criteria for a temporal logic to be suitable for our monitoring framework.

Definition 2 (Temporal logic). A *temporal logic* is a specification formalism *TL* over a set of atomic propositions AP that enjoys the following properties.

1. The semantics of formulae φ is given for finite words over an input alphabet Γ by a mapping

$$[\![\varphi]\!]_{TL} : (AP \to 2^\Gamma) \to (\Gamma^* \to \mathbb{S})$$

where $(\mathbb{S}, \sqcap)$ is a complete lattice.
2. The semantics is propositional.
3. A monitor construction is available that turns a formula φ into a deterministic monitor $\mathcal{M}_\varphi$ with output $\mathcal{M}_\varphi(w) = [\![\varphi]\!]_{TL}(\mathsf{ps}_{AP})(w)$ for $w \in \Sigma^*$.

3.2 Data logic

To reason about data values our framework can use a so called *data logic DL* based on any first-order theory for which satisfiability is decidable. We assume the theory is represented by some structure which can be extended by additional predicate and function symbols that will represent observations from the system that shall be monitored.

Definition 3 (Data logic). Let $T = (P, F, \mathsf{ar})$ be a signature, $t = (\mathcal{D}, \mathfrak{a})$ some T-structure and P', F' be additional predicate and function symbols with arity defined by $\mathsf{ar}' : P' \cup F' \to \mathbb{N}$, called *observation symbols*.

A *data logic DL* is a tuple $(t, G, \mathcal{V}, \mathcal{D})$ such that $G = (P \cup P', F \cup F', \mathsf{ar} \cup \mathsf{ar}')$ is an extension of T and $\mathcal{V}$ is a finite set of first-order variables.

A *DL formula* is a first-order formula over the signature G and possibly free variables from $\mathcal{V}$. A *DL* formula is called *observation-independent*, if it does not contain observation symbols. An *observation* is a G-structure $g = (\mathcal{D}, \mathfrak{g})$ that is an extension of t. The set of all observations is denoted Γ.

The *semantics* of a *DL* formula is defined over tuples $(g, \theta) \in \Gamma \times \mathcal{D}^\mathcal{V}$ consisting of an observation and a valuation $\theta : \mathcal{V} \to \mathcal{D}$ of free variables in the usual way.

For an instance of the monitoring framework the structure t representing the theory is fixed. An observation-independent *DL* formula φ with free variables from the set $\mathcal{V}$ can be evaluated just with respect to t, without considering an observation. A decision procedure for the theory of t can thus be applied directly. Further, φ can be interpreted as a constraint on the domain of variable valuations $\mathcal{D}^\mathcal{V}$ by considering the set $\Theta_\varphi := \{\theta \in \mathcal{D}^\mathcal{V} \mid (t, \theta) \models \varphi\}$.

3.3 Temporal data logic

Given a temporal and a data logic as described above, we can now define their combination, the temporal data logic *TDL*. In *TDL* formulae we use brackets $\langle$ and $\rangle$ to clarify which parts come from the data logic.

Syntax. Let *TL* be a temporal logic, $DL = (t, G, \mathcal{V}, \mathcal{D})$ be a data logic and AP be a finite set $\{\langle\chi_1\rangle, \ldots, \langle\chi_n\rangle\}$ where $\chi_1, \ldots, \chi_n$ are *DL* formulae possibly with free variables from $\mathcal{V}$. A *TDL* formula φ consists of a *core formula* ψ which is a *TL* formula over AP and a number of preceding *global first-order quantifiers* binding free variables in ψ. Formally, the syntax of *TDL* formulae φ is defined according to the grammar

$$\varphi ::= \exists_x \varphi \mid \forall_x \varphi \mid \psi$$

where $x \in \mathcal{V}$ is a variable and ψ is a *TL* formula over AP. If there are no free variables in φ, the formula is called a *sentence*.

Semantics. A *structured word* is a finite sequence $\gamma \in \Gamma^*$ of *DL* observations. For a valuation $\theta \in \mathcal{D}^\mathcal{V}$ let the proposition semantics $\mathsf{ps}_\theta : AP \to 2^\Gamma$ be defined by

$$\mathsf{ps}_\theta(\langle\chi\rangle) := \{g \in \Gamma \mid (g, \theta) \models \chi\}$$

where $\langle\chi\rangle \in AP$. The semantics of a *TDL* formula φ is defined with respect to a valuation $\theta : \mathcal{V} \to \mathcal{D}$. It is a mapping $[\![\varphi]\!]^\theta_{TDL} : \Gamma^* \to \mathbb{S}$ defined for $\gamma \in \Gamma^*$ as

$$[\![\exists_x \varphi']\!]^\theta_{TDL}(\gamma) := \bigsqcup_{d \in \mathcal{D}} [\![\varphi']\!]^{\theta[x \mapsto d]}_{TDL}(\gamma),$$

$$[\![\forall_x \varphi']\!]^\theta_{TDL}(\gamma) := \bigsqcap_{d \in \mathcal{D}} [\![\varphi']\!]^{\theta[x \mapsto d]}_{TDL}(\gamma),$$

$$[\![\psi]\!]^\theta_{TDL}(\gamma) := [\![\psi]\!]_{TL}(\mathsf{ps}_\theta)(\gamma)$$

where ψ is a *TL* formula. If φ is a *sentence*, that is, it does not contain any free variable, we omit to annotate a specific valuation θ and write $[\![\varphi]\!]_{TDL}$ for its semantics. This is well-defined since valuations of variables that do not occur freely in φ do not affect its semantics.

Recall that for ps_θ we obtain a projection $\pi_{\mathsf{ps}_\theta} : \Gamma^* \to \Sigma^*$ from structured to symbolic words. In the following we abbreviate π_{ps_θ} by π_θ. From the semantics of *TDL* formulae without quantifiers it immediately follows that we can evaluate the semantics *symbolically* by projecting the observation to propositional letters and then evaluating the temporal formula according to the propositional semantics. This is an essential step in lifting a monitor construction for *TL* to the data setting.

Proposition 1. *Let ψ be a TL formula over $AP = \{\langle\chi_1\rangle, \ldots, \langle\chi_n\rangle\}$ where $\chi_1, \ldots, \chi_n$ are DL formulae with free variables from $\mathcal{V}$. For a valuation $\theta \in \mathcal{D}^\mathcal{V}$ and a sequence of observations $\gamma \in \Gamma^*$*

$$[\![\psi]\!]^\theta_{TDL}(\gamma) = [\![\psi]\!]_{TL}(\mathsf{ps}_{AP})(\pi_\theta(\gamma)).$$

Hence, the semantics of quantified formulae can be characterised similarly by meet and join operations over the symbolic semantics.

3.4 LTL and CaRet with Data

We now exemplify the instantiation of our framework by means of LTL. More precisely, we show that the finitary, three-valued LTL_3 semantics $[\![\varphi]\!]_3 : \Sigma^* \to \mathbb{B}_3$ can be formulated to comply with Definition 2.

The syntax of *linear-time temporal logic (LTL)* is defined in the usual way over atomic propositions AP

mutex No foreign access while locked
$\forall_f \forall_t \, \mathrm{G}(\langle \mathsf{lock}(f,t) \rangle \rightarrow \langle \forall t' \neq t : \neg \mathsf{access}(f,t') \rangle \, \mathrm{U} \langle \mathsf{unlock}(f,t) \rangle)$
access Access only open resources
$\forall_x (\langle \mathsf{open}(x) \rangle \, \mathrm{R} \, \neg \langle \mathsf{access}(x) \rangle) \wedge \mathrm{G}(\langle \mathsf{close}(x) \rangle \rightarrow \mathrm{G} \, \neg \langle \mathsf{access}(x) \rangle)$
iterator Check iterator for next element
$\forall_i \, \mathrm{G}((\langle \mathsf{iterator}(i) \rangle \vee \langle \mathsf{next}(i) \rangle) \rightarrow \mathrm{X}(\langle \mathsf{hasNext}(i, \mathrm{true}) \rangle \, \mathrm{R} \, \neg \langle \mathsf{next}(i) \rangle))$
modified Do not reuse old iterator after modifying collection
$\forall_c \forall_i \, \mathrm{G}(\langle \mathsf{iterator}(c,i) \rangle \rightarrow \mathrm{G}(\langle \mathsf{add}(c) \rangle \rightarrow (\neg \langle \mathsf{next}(i) \rangle \, \mathrm{U} \langle \mathsf{finalize}(i) \rangle)))$
server Each request is served by somebody
$\forall_t \forall_x \, \mathrm{G}(\langle \mathsf{request}(t,x) \rangle \rightarrow \mathrm{F} \langle \exists t' : \mathsf{response}(t', x, t) \rangle)$
response Response within time limit
$\forall_t \forall_x \, \mathrm{G}(\langle \mathsf{request}(t) \rangle \wedge \langle x = \mathsf{time} \rangle \rightarrow (\langle \mathsf{time} < x + 100 \rangle \, \mathrm{U} \langle \mathsf{response}(t) \rangle))$
counter c is a counter
$\forall_x \, \mathrm{G}(\langle \mathsf{c} = x \rangle \rightarrow \mathrm{X} \langle \mathsf{c} = x + 1 \rangle)$
velocity Average speed since last observation does not exceed limit
$\forall_x \forall_y \, \mathrm{G}(\langle \mathsf{s} = x \wedge \mathsf{t} = y \rangle \rightarrow \mathrm{X} \langle \mathsf{s} - x < \mathsf{vmax} \cdot (\mathsf{t} - y) \rangle)$
matching Matching open and closing tags with same name
$\forall_x \, \mathrm{G}((\langle \mathsf{call} \rangle \wedge \langle \mathsf{printOpen}(x) \rangle) \rightarrow \mathrm{X}^{\mathrm{a}} \langle \mathsf{printClose}(x) \rangle)$
bound Value of each opening node is upper bound on nesting depth
$\forall_x \forall_y \, \mathrm{G}(\langle \mathsf{open}(x) \rangle \rightarrow \mathrm{X}(\neg \langle \mathsf{ret} \rangle \rightarrow \mathrm{G}^{\mathrm{a}}(\langle \mathsf{open}(y) \rangle \rightarrow \langle x > y \rangle)))$
depth Value of each opening node is exact nesting depth of one direct child node
$\forall_x \, \mathrm{G}(\langle \mathsf{open}(x) \rangle \rightarrow \mathrm{X}((\neg \langle \mathsf{ret} \rangle \wedge \mathrm{F}^{\mathrm{a}} \langle \mathsf{open}(x-1) \rangle) \vee (\langle \mathsf{ret} \rangle \wedge \langle x = 0 \rangle)))$
controller All processes communicate only with a central controller process
$\exists_c \forall_p \, \mathrm{G} \langle \forall_{p'} \mathsf{request}(p, p') \rightarrow c = p' \rangle$
observer All process are observed by some other process, i.e. they send messages regularly to that process
$\forall_p \exists_o \, \mathrm{G} \, \mathrm{F} \langle \mathsf{request}(p, o) \rangle$
mediator For every process there is a mediator that relays every request r to the controller
$\exists_c \forall_p \exists_m \forall_r \, \mathrm{G} \langle \mathsf{request}(p, r, m) \rangle \rightarrow \mathrm{F} \langle \mathsf{request}(m, r, c) \rangle$

Table 1. Example properties using LTL and CaRet with data.

using negation, Boolean connectives and temporal operators X (next), U (until), R (release), G (globally) and F (eventually). We refer to the standard LTL semantics over infinite words $w \in \Sigma^\omega$ as LTL_ω given for an LTL formula φ by a mapping $[\![\varphi]\!]_\omega : \Sigma^\omega \rightarrow \mathbb{B}$ where $\mathbb{B} = \{\top, \bot\}$ denotes the two-valued Boolean lattice. The finitary three-valued LTL semantics LTL_3 [BLS06], is given for φ by a mapping $[\![\varphi]\!]_3 : \Sigma^* \rightarrow \mathbb{B}_3$ where $\mathbb{B}_3 = \{\top, ?, \bot\}$ denotes the three-valued Boolean lattice ordered $\top > ? > \bot$. It is defined for $w \in \Sigma^*$ as $[\![\varphi]\!]_3(w) := \top$ if $\forall u \in \Sigma^\omega : [\![\varphi]\!]_\omega(wu) = \top$, $[\![\varphi]\!]_3(w) := \bot$ if $\forall u \in \Sigma^\omega : [\![\varphi]\!]_\omega(wu) = \bot$ and $[\![\varphi]\!]_3(w) := ?$ otherwise.

The three-valued LTL_3 semantics $[\![\varphi]\!]_3 : \Sigma^* \rightarrow \mathbb{B}_3$ is defined over $\Sigma = 2^{AP}$ based on the infinitary LTL_ω semantics. The inductive definition of LTL_ω only refers to letters for atomic propositions. This can be easily reformulated in terms of an arbitrary proposition semantics $ps : AP \rightarrow 2^\Gamma$ over an arbitrary alphabet Γ. Instead of defining $[\![p]\!]_\omega(w) = \top$ iff $p \in w_0$, we let $[\![p]\!]_\omega(ps)(\gamma) := \top$ if $\gamma_0 \in ps(p)$ and $[\![p]\!]_\omega(ps)(\gamma) := \bot$ otherwise, for $\gamma \in \Gamma^\omega$. The rest of the definition remains untouched. The definition of the three-valued semantics $[\![\varphi]\!]_3$ does not at all refer to letters directly but only to LTL_ω. With these simple modifications LTL_3 fits to the notion of temporal logic in the sense of Definition 2. The corresponding monitor construction proposed in [BLS06, BLS11] can be applied.

Proposition 2. *LTL_3 is a suitable temporal logic in the sense of Definition 2.*

The mutual exclusion property presented earlier is one example for a specification based on LTL and the theory of IDs. Other common examples of temporal properties are the correct use of iterators or global request/response properties. In the propositional versions of such properties the objects in question, iterators, resources or requests, are assumed to be unique. Adding data in terms of IDs, for example, allows for a much more realistic formulation. Table 1 lists formulations of these properties and also others that cannot be expressed without distinguishing at least identities. The property *modified* requires that an iterator must not be used after the collection it corresponds to has been changed. Further, counting (*counter*) or arithmetic constraints (*response*, *velocity*), also on real numbers, are valuable features for a realistic specification. Examples *controller*, *observer* and *mediator* show properties that require alternating quantifiers.

RLTL and CaRet: Regular and nesting properties. Regular LTL [LS07] is an extension of LTL based on regular expressions. CaRet [AEM04] is a temporal logic with calls and returns expressing non-regular properties. In addition to the LTL operators, CaRet allows for abstract temporal operators such as X^{a} and G^{a}, moving forward by jumping on a word from a calling position to match-

ing return position, reflecting the intuition of procedure calls. For RLTL and CaRet monitor constructions have been proposed [DLS08, DLT13a]. Despite both are more complex the same arguments as for LTL apply. Example properties are listed in Table 1 and express matching call- and return values and nesting-depth bounds.

4 Monitoring

In this section we present our monitoring procedure for *TDL* formulae. It relies on the observation made in Proposition 1, namely that the *TDL* semantics for an input word $\gamma \in \Gamma^*$ is characterised by the *TL* semantics for projections of γ.

Any *TDL* formula can be interpreted as *TL* formula when stripping first-order quantifiers and considering all occurring data logic formulae as individual symbols. With this interpretation we can employ the monitor construction for *TL* to obtain a monitor over a finite alphabet constructed from these symbols.

Definition 4 (Symbolic monitor). Let φ be a *TDL* formula with core formula ψ. Let $\chi_1, \ldots, \chi_n$ be the data logic formulae occurring in φ and $AP = \{\langle\chi_1\rangle, \ldots, \langle\chi_n\rangle\}$. The *symbolic alphabet* for φ is the finite set $\Sigma := 2^{AP}$. The *symbolic monitor* for φ is the monitor $\mathcal{M}_\Sigma$ constructed from its core formula ψ interpreted as *TL* formula over AP.

The symbolic monitor $\mathcal{M}_\Sigma$ for a *TDL* formula φ incrementally computes the semantics $[\![\psi]\!]_{TL}(\mathsf{ps}_{AP}) : \Sigma^* \to \mathbb{S}$. Following Proposition 1, what remains is to maintain a monitor for each valuation $\theta \in \mathcal{D}^{\mathcal{V}}$ and to individually compute the corresponding projection π_θ on the input. The individual verdicts then need to be combined according to the quantifiers preceding ψ in φ.

Within this section we present an algorithm for efficiently maintaining these, in general infinitely many, monitor instances. It uses a data structure, called *constraint tree*, that represents finitely many equivalence classes of symbolic monitors. The constraint tree also allows for alternating the computation of infima and suprema over sets of outputs of monitor instances according to the universal and existential quantification of free variables, respectively. This yields the semantics of the property on the input trace read so far.

4.1 Representing and Evaluating Observations

While observations are formally defined as first-order structures, we want to use them algorithmically and must therefore choose a representation. An actual implementation of an SMT solver already fixes how to represent all objects essential for handling a certain theory, such as first-order formulae, predicates and function symbols. We have defined observations to be extensions of a structure representing the theory and want to handle them practically using an SMT solver. Consequently, we assume them to be extensions of the structure that the tool uses to represent and handle a theory. For the purpose of the implementation, it is a reasonable assumption that the semantics of observation symbols be expressible or, more precisely, expressed within the considered data theory.

Formally, for $DL = (t, G, \mathcal{V}, \mathcal{D})$ where t is a T-structure, we assume that any observation $g \in \Gamma$ induces a mapping $\hat{g} : FO[G] \to FO[T]$ s.t. for all DL formulae χ and all valuations $\theta \in \mathcal{D}^{\mathcal{V}}$ we have

$$(g, \theta) \models \chi \quad \Leftrightarrow \quad (t, \theta) \models \hat{g}(\chi).$$

Note that this can be realised by substituting observation predicates by some observation-independent formula that characterises its semantics with respect to g. Function symbols f can be replaced using existential substitution replacing expressions of the form $e(f(e'))$ by $\exists z : e(z) \wedge \xi_f(e', z)$ where an observation-free DL formula ξ_f characterises the semantics of f with respect to g.

As noted earlier, we can also employ observation-free formulae ρ to describe sets of valuations $\Theta_\rho \subseteq \mathcal{D}^{\mathcal{V}}$. While this does not allow for representing any arbitrary set of valuations, the expressiveness of the data theory suffices to express any relevant set.

Proposition 3. *Let χ be a DL formula, $g \in \Gamma$ an observation and ρ be an observation-free DL formula such that for all $\theta_1, \theta_2 \in \Theta_\rho$ we have*

$$(t, \theta_1) \models \hat{g}(\chi) \quad \Leftrightarrow \quad (t, \theta_2) \models \hat{g}(\chi).$$

Then for all $\theta \in \Theta_\rho$

$$(g, \theta) \models \chi \quad \Leftrightarrow \quad \hat{g}(\chi) \wedge \rho \text{ is satisfiable.}$$

Intuitively, if either every valuation $\theta \in \Theta_\rho$ or none of them satisfies a formula $\hat{g}(\chi)$ then ρ represents an equivalence class of valuations with respect to $\hat{g}(\chi)$. Hence, for all these valuations, $(t, \theta) \models \hat{g}(\chi)$ if and only if $(t, \theta') \models \hat{g}(\chi) \wedge \rho$ for *any* valuation $\theta' \in \mathcal{D}^{\mathcal{V}}$ since ρ already restricts to the set Θ_ρ. Note that $\hat{g}(\chi) \wedge \rho$ is an observation-free DL formula and that checking it for satisfiability is exactly what we assume an SMT solver be able to do.

4.2 Constraint Trees

We next introduce constraint trees, a data structure that is capable of storing the configurations of a set of instances of some symbolic monitor. It maintains sets of valuations $\Theta \subseteq \mathcal{D}^{\mathcal{V}}$ represented by constraints and stores for each such set a monitor state. The desired property regarding the use in our monitoring algorithm is that the sets of constraints induce a partition of the valuation space.

Definition 5 (Constraint tree). Let M be a set of labels and DL a data logic. A *constraint tree* over M is a tuple $\tau = (I, L, S_1, S_2, C, \lambda_I, M, \lambda_L)$ such that

– $(I \cup L, S_1, S_2)$ is a finite, non-empty and full binary tree with *internal nodes* I and *leaf nodes* L,
– C is a set of observation-independent DL formulae called *constraints*,
– $\lambda_I : I \to C$ labels internal nodes by constraints and
– $\lambda_L : L \to M$ labels leaf nodes by elements from M.

Let $v_0 \in I \cup L$ be the root of τ, i.e., the unique node without predecessor. For a node $v \in I \cup L$ in τ let $\rho(v)$ be the DL formula defined inductively by $\rho(v_0) = \text{true}$ and for $v \neq v_0$

$$\rho(v) = \begin{cases} \rho(v') \wedge \lambda_I(v') & \text{if } (v', v) \in S_1 \\ \rho(v') \wedge \neg\lambda_I(v') & \text{if } (v', v) \in S_2. \end{cases}$$

For a leaf $l \in L$, the formula $\rho(l)$ is called a *path constraint* in τ. The set of all constraint trees over M is denoted by $\mathcal{T}_M$.

Note that path constraints are well-defined because each node v in a tree that is not the root has a unique predecessor v' such that either $(v', v) \in S_1$ or $(v', v) \in S_2$. In a constraint tree τ, each inner node represents a constraint that is used to separate the valuation space $\mathcal{D}^{\mathcal{V}}$. S_1-branches represent the parts where the particular constraint holds while in the S_2-branches it does not. Where convenient, we may identify a path constraint ρ with the set Θ_ρ of valuations satisfying it and write, e.g., $\theta \in \rho$ if some valuation $\theta \in \Theta_\rho$ satisfies ρ in the given theory.

Proposition 4. *The set of path constraints in a constraint tree* $\tau = (I, L, S_1, S_2, C, \lambda_I, M, \lambda_L)$ *induces a* partition $\{\Theta_{\rho(l)} \subseteq \mathcal{D}^{\mathcal{V}} \mid l \in L\}$ *of the valuation space* $\mathcal{D}^{\mathcal{V}}$.

Each internal node $v \in I$ splits the set $\Theta_{\rho(v)}$ into those valuations satisfying $\lambda_I(v)$ and those that do not. Every particular valuation $\theta \in \mathcal{D}^{\mathcal{V}}$ hence satisfies exactly one path constraint $\rho(l_\theta)$ in τ for a particular leaf node $l_\theta \in L$. A constraint tree $\tau \in \mathcal{T}_M$ therefore represents a well-defined, total function $\tau : \mathcal{D}^{\mathcal{V}} \to M$ with $\tau(\theta) = \lambda_L(l_\theta)$.

We will use constraint trees $\tau \in \mathcal{T}_Q$ for maintaining instances of a symbolic monitor $\mathcal{M}_\Sigma$ with states Q. Figure 2 shows an example of a constraint tree for the valuation space of two integer variables using constraints from a first order theory with order and equality.

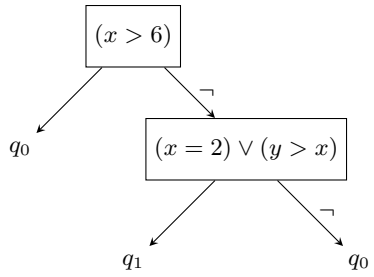


Fig. 2. Constraint tree partitioning the set of valuations $\theta : \{x, y\} \to \mathbb{Z}$ by integer constraints. To each partition one of the states q_0 and q_1 is associated.

4.3 Symbolic Monitor Execution

In the following we present an algorithm incrementally processing a sequence of observations in order to compute the semantics of some *TDL* formula φ. It maintains a constraint tree as a finite representation of a mapping of valuations to states of the symbolic monitor $\mathcal{M}_\Sigma = (Q, \Sigma, \delta, q_0, \lambda, \mathbb{S})$ for φ.

The algorithm starts on the trivial constraint tree consisting only of one leaf node labelled by the initial state q_0. This means that the monitor instances for all valuations are in state q_0. Intuitively, for an input word $\gamma \in \Gamma$ the algorithm executes one monitor instance for each valuation $\theta \in \mathcal{D}^{\mathcal{V}}$ on the respective projection $\pi_\theta(\gamma)$. For the empty word $\gamma = \epsilon$, all projections are equal and all instances are in the same state. When reading a new observation $g \in \Gamma$ which is, for all valuations, projected to the same symbolic letter $a \in \Sigma$, all monitor instances read the same projection and their state changes equally to $\delta(q_0, a)$. Otherwise, if g is mapped to different symbolic letters for different valuations, the so far uniformly handled valuation space is *split*.

Consider two valuations $\theta, \theta' \in \mathcal{D}^{\mathcal{V}}$ and an input symbol $g \in \Gamma$ such that their projections $a = \pi_\theta(g) \neq b = \pi_{\theta'}(g)$ are different. Then there is some proposition $\langle\chi\rangle \in AP$ that distinguishes a and b, e.g., let $\langle\chi\rangle \in a$ and $\langle\chi\rangle \notin b$. Note that this is because $(g, \theta) \models \chi$ and $(g, \theta') \not\models \chi$. In general, the behaviour of all monitor instances reading a letter including $\langle\chi\rangle$ may diverge from those reading a letter not including $\langle\chi\rangle$. Therefore, the algorithm records this fact by splitting the valuation space in two parts, one for which χ holds under observation g and another for which it does not. A new node is added to the tree, labelled by the constraint $\hat{g}(\chi)$ precisely distinguishing the two parts. A part may be split up further in the same way in case other propositions again distinguish valuations from it. Additional nodes are created in the constraint tree accordingly and so the path constraint ρ on the path to a leaf node $v \in L$ characterises exactly the set of valuations Θ_ρ for which the projection of the observation g is equal and is thus the state of all corresponding monitor instances. This process is continued when reading further observations. For each part Θ_ρ represented in the constraint tree, a new observation $h \in \Gamma$ is processed by checking for each proposition $\langle\chi\rangle \in AP$ if there are valuations in Θ_ρ that observe a projection including $\langle\chi\rangle$ by checking satisfiability of $\rho \wedge \hat{h}(\chi)$ and if there are others observing a projection not including $\langle\chi\rangle$ by checking satisfiability of $\rho \wedge \neg\hat{h}(\chi)$. If one of the formulae is unsatisfiable, meaning that one of the hypothetical new parts $\Theta_{\rho \wedge \hat{h}(\chi)} = \Theta_\rho \cap \Theta_{\hat{h}(\chi)}$ and $\Theta_{\rho \wedge \neg\hat{h}(\chi)} = \Theta_\rho \cap \overline{\Theta_{\hat{h}(\chi)}}$ is empty, the new observation h is projected equally with respect to $\langle\chi\rangle$ for all valuations

in the part which is thus not split by $\hat{h}(\chi)$. Only if both new parts are non-empty, the part is split by adding a new node to the constraint tree labelled by $\hat{h}(\chi)$. Once all necessary splits are performed for an observation, all propositions are evaluated yielding the projections for each (possibly new) part. According to those, the leaf nodes are updated using the transition function of the symbolic monitor.

The procedure described above is listed explicitly as Algorithm 1. There, for the set $\mathcal{T}_Q$ of all constraint trees over states Q, we use constructors $\texttt{InnerCTree} : \mathrm{FO}[S] \times \mathcal{T}_Q \times \mathcal{T}_Q \to \mathcal{T}_Q$ and $\texttt{LeafCTree} : Q \to \mathcal{T}_Q$ for sub-trees and leafs, respectively, where $\mathrm{FO}[S]$ is the set of observation-independent *DL* formulae. A (sub-)tree $\tau = \texttt{LeafCTree}(q)$ represents a single node $v \in L$ that is labelled by $\lambda_L(v) = q$. In case $\tau = \texttt{InnerCTree}(\xi, \tau_1, \tau_2)$ for some formula ξ, the represented (sub-)tree τ has at least three nodes v, v_1, v_2 such that v is the root of τ labelled by $\lambda_I(v) = \xi$, v_1, v_2 are the roots of τ_1 and τ_2, respectively, and $(v, v_1) \in S_1$ and $(v, v_2) \in S_2$.

Notice that the presentation of Algorithm 1 is not deterministic. We can however easily determinise the algorithm by choosing an arbitrary precedence for the elements of AP and therefore assume the procedure to be deterministic. Then, we denote the usual extension of $\texttt{step}$ to sequences of input letters by $\texttt{step}^*(\tau, \gamma g) = \texttt{step}(\texttt{step}^*(\tau, \gamma), g)$ for $\gamma \in \Gamma^*$, $g \in \Gamma$.

Proposition 5 (Termination of step). *On a constraint tree $\tau \in \mathcal{T}_Q$, the function* step *in Algorithm 1 terminates and has a running time in $\mathcal{O}(|\tau| \cdot |\Sigma|)$ where $|\tau|$ is the number of nodes in τ and $|\Sigma| = 2^{|AP|}$ is the number of abstract symbols.*

Proof. The algorithm does not contain a loop and to see that recursion stops consider the following well-ordering argument. Whenever the procedure is called with a tree τ and a set P as arguments and recursively calls itself with parameters τ' and P' then τ' is a sub-tree of τ and $P' \subseteq P$. Moreover, either τ' is a *proper* sub-tree or P' is a *proper* subset. In the former case, the procedure calls itself once for each subtree. In the latter case, the procedure calls itself at most twice for each proposition. The number of recursive calls on a tree τ is hence bound by $|\tau| \cdot 2^{|AP|}$. □

Based on constraint trees as data structure and the algorithm for modifying constraint trees regarding a new observation we can now define the data monitor for a *TDL* formula, where, as before, the data logic *DL* is defined over observations Γ and the temporal logic *TL* uses truth values $\mathbb{S}$.

Definition 6 (Data monitor). Let φ be a *TDL* formula with core formula ψ, $\Sigma = 2^{AP}$ the symbolic alphabet and $\mathcal{M}_\Sigma = (Q, \Sigma, \delta, q_0, \lambda_Q, \mathbb{S})$ the symbolic monitor for φ. The *data monitor* for φ is a Moore machine $\mathcal{M}_\Gamma = (\mathcal{T}_Q, \Gamma, \texttt{step}, \tau_0, \lambda_\mathcal{T})$ where the transition function $\texttt{step} : \mathcal{T}_Q \times \Gamma \to \mathcal{T}_Q$ is given by Algorithm 1. The initial state $\tau_0 \in \mathcal{T}_Q$ is the constraint tree consisting of a single leaf node labelled with the initial state q_0 of $\mathcal{M}_\Sigma$.

For a state $\tau = (I, L, S_1, S_2, C, \lambda_I, Q, \lambda_L) \in \mathcal{T}_Q$, the *output* of $\mathcal{M}_\Gamma$ is defined by the function $\lambda_\mathcal{T} : \mathcal{T}_Q \to \mathcal{T}_\mathbb{S}$ with $\lambda_\mathcal{T}(\tau) := (I, L, S_1, S_2, C, \lambda_I, \mathbb{S}, \lambda_Q \circ \lambda_L)$ where $(\lambda_Q \circ \lambda_L)(l) = \lambda_Q(\lambda_L(l))$ for $l \in L$. The output of $\mathcal{M}_\Gamma$ on a word $\gamma \in \Gamma^*$ is defined as

$$\mathcal{M}_\Gamma(\gamma) := \lambda_\mathcal{T}(\texttt{step}^*(\tau_0, \gamma)).$$

The output of the data monitor is not the typical single, explicit verdict but a constraint tree that represents a function from from variable valuations to verdicts. As we will show, it precisely characterises the semantics of the core formula of φ.

That way the monitor output provides all information needed to compute the actual semantics of φ on the observed input. Moreover, it allows us to separate the actual monitoring procedure and the process of evaluating the verdict. In fact, the information provided by the monitor allows for computing the semantics for any quantification of free variables in the core formula.

Evaluating the verdict for *TDL* formulae where global quantification is either only universal or only existential amounts to simply compute the meet or the join, respectively, over the leafs of the output tree. The general case of arbitrary alternation is more involved and, most importantly, computationally harder. Due to the separation of monitoring and evaluation, however, a more expensive verdict evaluation can be implemented and even executed independently of the monitoring procedure, e.g., by delegating it to dedicated time frames or hardware.

4.4 Correctness

In this section we settle the correctness of the monitoring procedure presented above by showing that the output of the data monitor $\mathcal{M}_\Gamma$ for a *TDL* formula φ is a constraint tree representing the semantics of the core formula of φ with respect to all input words and all variable valuations.

For the following we fix the data logic *DL* and write π_θ for π_{ps_θ}. Recall that by Proposition 1 the *TDL* semantics of ψ can be represented in terms of its *TL* semantics over the symbolic abstraction $\pi_\theta(\gamma)$ of γ by

$$[\![\psi]\!]^\theta_{TDL}(\gamma) = [\![\psi]\!]_{TL}(\mathsf{ps}_{AP})(\pi_\theta(\gamma)).$$

The semantics of ψ depends on valuations from an infinite domain. However, given a finite word $\gamma \in \Gamma^*$, the valuation space $\mathcal{D}^\mathcal{V}$ can be partitioned into finitely many equivalence classes $\Theta_1, \ldots, \Theta_n \subseteq \mathcal{D}^\mathcal{V}$ such that the projection of γ is unique for each class Θ_i, i.e.,

$$\forall_{\theta, \theta' \in \Theta_i} : \pi_\theta(\gamma) = \pi_{\theta'}(\gamma).$$

Thus, also the semantics of ψ is equal for all valuations from the same class. This partition exists and is in fact represented by the constraint tree computed by the monitoring algorithm.

Algorithm 1 Split constraints and simulate monitor steps

```
1  function split: 2^AP × FO[T] × Σ × T_Q × Γ → T_Q

3  function split =
4    // recursively process subtrees and accumulate constraints
5    case (P, ρ, a, InnerCTree(φ, τ1, τ2), g) then
6      InnerCTree(φ, split(P, ρ ∧ φ, a, τ1, g), split(P, ρ ∧ ¬φ, a, τ2, g))

8    // evaluate propositions, split partition if necessary
9    case ({⟨χ⟩} ∪ P, ρ, a, LeafCTree(s), g) then
10     τ1 := if SAT(ρ ∧ ĝ(χ)) then split(P, ρ ∧ ĝ(χ), a ∪ {⟨χ⟩}, LeafCTree(s), g)
11           else Empty

13     τ2 := if SAT(ρ ∧ ¬ĝ(χ)) then split(P, ρ ∧ ¬ĝ(χ), a, LeafCTree(s), g)
14           else Empty

16     if (τ1 = Empty) then τ2
17     else if (τ2 = Empty) then τ1
18     else InnerCTree(ĝ(χ), τ1, τ2)

20   // store new state
21   case (∅, ρ, a, LeafCTree(s), g) then LeafCTree(δ(s, a))

23 function step(τ ∈ T_Q, g ∈ Γ): T_Q = split(AP, true, ∅, τ, g)
```

Lemma 1. *Let $\mathcal{M}_\Gamma = (\mathcal{T}_Q, \Gamma, \delta_\Gamma, \tau_0, \lambda_\Gamma)$ and $\mathcal{M}_\Sigma = (Q, \Sigma, \delta_\Sigma, q_0, \lambda_\Sigma)$ be the data monitor and the symbolic monitor, respectively, for some TDL formula φ. For an input word $\gamma \in \Gamma^*$ let $\tau_\gamma := \mathtt{step}^*(\tau_0, \gamma)$. Then, for all valuations $\theta \in \mathcal{D}^\mathcal{V}$,*

$$\tau_\gamma(\theta) = \delta^*_\Sigma(q_0, \pi_\theta(\gamma)).$$

Proof. We prove the lemma by induction on the length of γ. For $\gamma = \epsilon$, $\theta \in \mathcal{D}^\mathcal{V}$ and $\tau_\epsilon = step^*(\tau_0, \epsilon)$

$$\tau_\epsilon(\theta) = \tau_0(\theta) = q_0 = \delta^*_\Sigma(q_0, \epsilon) = \delta^*_\Sigma(q_0, \pi_\theta(\epsilon)).$$

Now, consider a word $\gamma g \in \Gamma^+$ for $g \in \Gamma$, of length at least 1. The monitor proceeds letter-wise, so $\tau_{\gamma g} = \delta^*_\Gamma(\tau_0, \gamma g) = \delta_\Gamma(\delta^*_\Gamma(\tau_0, \gamma), g)$.

Let $\tau_\gamma = \mathtt{step}^*(\tau_0, \gamma) = (I, L, S_1, S_2, C, \lambda_I, Q, \lambda_L)$ and $q := \tau_\gamma(\theta)$. Then there is a leaf $l \in L$ with label $\lambda_L(l) = q$ and $(t, \theta) \models \rho(l)$. Let $a := \pi_\theta(g)$ be the projection of g under θ. By definition of π_θ we have also $(t, \theta) \models \hat{a}$ for

$$\hat{a} := \Big(\bigwedge_{\langle\chi\rangle \in a} \hat{g}(\chi) \Big) \wedge \bigwedge_{\langle\chi\rangle \in AP \setminus a} \neg \hat{g}(\chi)$$

and hence $(t, \theta) \models \rho(l) \wedge \hat{a}$. This means in particular that $\rho(l) \wedge \hat{a}$ is satisfiable. It follows that in $\tau_{\gamma g} = \mathtt{step}(\tau_\gamma, g)$ there is a leaf l' with $\rho(l') \equiv \rho(l) \wedge \hat{g}(a)$ because the algorithm substitutes the leaf l in τ_γ by a tree that has a branch for each combination of formulae from AP that is satisfiable in combination with $\rho(l)$.

The algorithm moreover explicitly computes a when constructing the corresponding branch in the substitution for l and stores $\delta_\Sigma(q, a)$ as the label of the new leaf l'. By induction $q = \tau_\gamma(\theta) = \delta^*_\Sigma(q_0, \pi_\theta(\gamma))$ and since $(t, \theta) \models \rho(l')$ the valuation θ is mapped to

$$\begin{aligned} \tau_{\gamma g}(\theta) &= \delta_\Sigma(q, a) = \delta_\Sigma(\delta^*_\Sigma(q_0, \pi_\theta(\gamma)), \pi_\theta(g)) \\ &= \delta^*_\Sigma(q_0, \pi_\theta(\gamma g)). \end{aligned}$$

□

Given, that the monitor indeed maintains all instances of the symbolic monitor it immediately follows that the output of $\mathcal{M}_\Gamma$ represents the semantics of the core formula ψ.

Theorem 1 (Monitoring correctness). *Let φ be a TDL formula with core formula ψ and $\mathcal{M}_\Gamma$ the data monitor for φ. For all $\gamma \in \Gamma^*$ and all $\theta \in \mathcal{D}^\mathcal{V}$ we have*

$$\mathcal{M}_\Gamma(\gamma)(\theta) = [\![\psi]\!]^\theta_{TDL}(\gamma).$$

Proof. For $\tau_\gamma = \mathtt{step}^*(\tau_0, \gamma)$ we have

$$\begin{aligned} \mathcal{M}_\Gamma(\gamma)(\theta) &= \lambda_\Gamma(\tau_\gamma)(\theta) = \lambda_\Sigma(\tau_\gamma(\theta)) \\ &\overset{\text{Lem. 1}}{=} \lambda_\Sigma(\delta^*_\Sigma(g_0, \pi_\theta(\gamma))) = \mathcal{M}_\Sigma(\pi_\theta(\gamma)) \\ &\overset{\text{Def. 4}}{=} [\![\psi]\!]_{TL}(\mathsf{ps}_{AP})(\pi_\theta(\gamma)) \\ &\overset{\text{Prop. 1}}{=} [\![\psi]\!]^\theta_{TDL}(\gamma) \end{aligned}$$

□

5 Verdict Evaluation

The monitoring approach presented above computes the semantics of core formulae in terms of a constraint tree. To obtain the actual verdict for a *TDL* property on a

sequence of observations this tree needs to be evaluated according to the global quantification.

To provide an intuition for the evaluation scheme we propose, this section starts with an example before explaining the algorithmic details. After these considerations we investigate the algorithm more formally in order to prove its correctness.

Consider the *TDL* formula

$$\varphi = \forall_{x_2} \exists_{x_1} \mathrm{G}\langle P(x_2)\rangle \rightarrow \mathrm{F}\langle Q(x_1, x_2)\rangle.$$

The semantics of φ is defined over the semantics of its core formula $\psi = \mathrm{G}\langle P(x_2)\rangle \rightarrow \mathrm{F}\langle Q(x_1, x_2)\rangle$ by

$$[\![\varphi]\!]_{TDL} = \bigsqcap_{d_2 \in \mathcal{D}} \bigsqcup_{d_1 \in \mathcal{D}} [\![\psi]\!]_{TDL}^{\{x_1 \mapsto d_1, x_2 \mapsto d_2\}}.$$

For $\mathrm{G}\langle P(x_2)\rangle \rightarrow \mathrm{F}\langle Q(x_1, x_2)\rangle$ we employ a monitor construction for the underlying temporal logic, LTL in this case, to construct a symbolic monitor $\mathcal{M}_\Sigma$ where $\Sigma = 2^{\{\langle P(x_2)\rangle, \langle Q(x_1, x_2)\rangle\}}$. From our considerations in the previous section we know that using the data monitor $\mathcal{M}_\Gamma$ for φ we can maintain the information how the symbolic monitor behaves with respect to all valuations of the free variables x_1 and x_2. This effectively computes the semantics of that core formula in terms of a constraint tree so that

$$[\![\varphi]\!]_{TDL}(\gamma) = \bigsqcap_{d_2 \in \mathcal{D}} \bigsqcup_{d_1 \in \mathcal{D}} \mathcal{M}_\Gamma(\gamma)(\{x_1 \mapsto d_1, x_2 \mapsto d_2\})$$

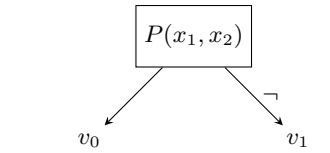
In this section we develop a procedure that allows for evaluating the above meet and join operations over the infinite domain $\mathcal{D}$ on an arbitrary constraint tree τ, i.e., to compute $\bigsqcap_{d_2 \in \mathcal{D}} \bigsqcup_{d_1 \in \mathcal{D}} \tau(\{x_1 \mapsto d_1, x_2 \mapsto d_2\})$ and thereby the semantics of φ. More generally, we propose an algorithm that takes a sequence of quantifiers $u = Q_n x_n \dots Q_1 x_1 \in (\{\forall, \exists\} \times \mathcal{V})^*$ and a constraint tree $\tau \in \mathcal{T}_{\mathbb{S}}$ representing a function $\tau : \mathcal{D}^{\mathcal{V}} \rightarrow \mathbb{S}$ for $\mathcal{V} = \{x_1, \dots, x_n\}$ and computes the verdict

$$\texttt{eval}(u, \tau) = \bigotimes_{d_n} \dots \bigotimes_{d_1} \tau(\{x_1 \mapsto d_1, \dots, x_n \mapsto d_n\})$$

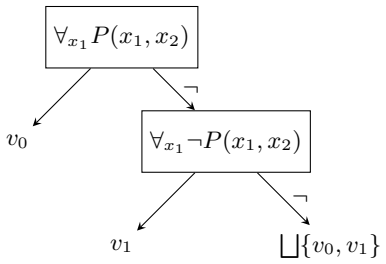
where $\otimes_i = \bigsqcap$ if $Q_i = \forall$ and $\otimes_i = \bigsqcup$ if $Q_i = \exists$.

5.1 Uniform Quantification

In the case where $n = 1$, that is, if there is only a single quantifier, the evaluation amounts to computing the meet or join, respectively, over the leafs of τ. This is because there are finitely many equivalence classes of valuations and all valuations in the same class impose the same behaviour of the monitor and thus yield the same verdict. The meet and join operations are associative and idempotent and thus it suffices to consider one representative for each class. For example we can characterise the meet over an infinite domain $\mathcal{D}$ in terms of the finitely many



(a) A simple constraint tree.



(b) Reduced constraint tree.

Fig. 3. A constraint tree over $\mathcal{V} = \{x_1, x_2\}$ and its reduced version where x_1 is eliminated, i.e., does not occur freely anymore.

representative verdicts that occur in some constraint tree τ over a single variable x by

$$\bigsqcap_{d \in \mathcal{D}} \tau(\theta_d) = \bigsqcap \{v \in \mathbb{S} \mid \exists d \in \mathcal{D} : \tau(\{x \mapsto d\}) = v\}.$$

Note that this set of verdicts is simply the set of leafs in τ. This does not change when considering several universally quantified variables $\mathcal{V}$. The constraint tree characterises a finite partition of valuations $\mathcal{D}^{\mathcal{V}}$ and all of them are combined by the associative meet operation. The analogue applies for the join, i.e., existential quantification.

5.2 Alternating Quantification

To apply a similar scheme in presence of *alternating* quantification we need, however, a more fine-grained partition of the valuation space. We use a refinement procedure based on the following idea.

Consider variables $\mathcal{V} = \{x_1, x_2\}$ and quantification of the form $\forall_{x_2} \exists_{x_1}$ that shall be evaluated over the constraint tree presented in Figure 3(a). Following the intuition of a lazy evaluation of the predicate P we have three classes $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \subseteq \mathcal{D}$ of values being assigned to x_2 by a valuation $\theta \in \mathcal{D}^{\mathcal{V}}$:

- Assigning x_2 a value $\theta(x_2) \in \mathcal{D}_1$ already determines that $P(d, \theta(x_2))$ holds, independently of the choice of $d \in \mathcal{D}$,
- for a value $\theta(x_2) \in \mathcal{D}_2$ it is already determined that $P(d, \theta(x_2))$ does not hold for any $d \in \mathcal{D}$ and
- for $\theta(x_2) \in \mathcal{D}_3$ the evaluation of $P(d, \theta(x_2))$ strictly depends on d, i.e., there are $d, d' \in \mathcal{D}$ such that $P(d, \theta(x_2))$ and $\neg P(d', \theta(x_2))$ holds.

The sets $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ form a partition of $\mathcal{D}$ and the class of the value $\theta(x_2)$ determines the set $\{v \in \mathbb{S} \mid \exists_{d \in \mathcal{D}} :$

$\tau(\theta[x_1 \mapsto d]) = v\}$ of possible values of τ when fixing $\theta(x_2)$. The constraint tree maps by definition all valuations θ with $\theta(x_2) \in \mathcal{D}_1$ or $\theta(x_2) = \mathcal{D}_2$ to v_0 or v_1, respectively. On the other hand, among the valuations assigning x_2 to some $d \in \mathcal{D}_3$, there is always one that is mapped to v_0 and one mapped to v_1 by τ.

Considering the task of computing, e.g.,

$$\bigsqcap_{d_2 \in \mathcal{D}} \bigsqcup_{d_1 \in \mathcal{D}} \tau(\{x_1 \mapsto d_1, x_2 \mapsto d_2\})$$

we conclude that the value of the inner operator only depends on the class of d_2. We can therefore construct from τ the constraint tree $\hat{\tau}$ depicted in Figure 3(b) that determines, for $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x_1\}}$, the class of $\theta(x_2)$ and maps θ to

$$\hat{\tau}(\theta) = \bigsqcup_{d \in \mathcal{D}} \tau(\theta[x_1 \mapsto d]) = \begin{cases} \bigsqcup\{v_0\} & \text{for } \theta(x_2) \in \mathcal{D}_1 \\ \bigsqcup\{v_1\} & \text{for } \theta(x_2) \in \mathcal{D}_2 \\ \bigsqcup\{v_0, v_1\} & \text{otherwise.} \end{cases}$$

We then have

$$\bigsqcap_{d_2 \in \mathcal{D}} \bigsqcup_{d_1 \in \mathcal{D}} \tau(\{x_1 \mapsto d_1, x_2 \mapsto d_2\}) = \bigsqcap_{d_2 \in \mathcal{D}} \hat{\tau}(\{x_2 \mapsto d_2\})$$

and thus arrive at the case involving only a single operator which can be evaluated over the set of leafs of $\hat{\tau}$ as above.

This idea can be generalised to arbitrary constraint trees. Given a constraint tree $\tau \in \mathcal{T}_{\mathbb{S}}$ over constraints with free variables $\mathcal{V} = \{x_1, \ldots, x_n\}$, an operator $\otimes \in \{\bigsqcap, \bigsqcup\}$ on subsets of $\mathbb{S}$ and a free variable $x_1 \in \mathcal{V}$ to be eliminated we construct a constraint tree $\hat{\tau}$ representing a mapping $\hat{\tau} : \mathcal{D}^{\mathcal{V}\setminus\{x_1\}} \to \mathbb{S}$ with

$$\hat{\tau}(\theta) = \bigotimes_{d_1 \in \mathcal{D}} \tau(\theta[x_1 \mapsto d_1])$$

By iterating this reduction $n = |\mathcal{V}|$ times to construct from τ a reduced tree $\hat{\tau}_1$ and by reducing it further $\hat{\tau}_2, \ldots, \hat{\tau}_n$ we can compute the value of a term

$$\begin{aligned} &\bigotimes_{d_n}{}_n \ldots \bigotimes_{d_1}{}_1 \tau(\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}) \\ &= \bigotimes_{d_n}{}_n \ldots \bigotimes_{d_2}{}_2 \hat{\tau}_1(\{x_2 \mapsto d_2, \ldots, x_n \mapsto d_n\}) \\ &= \quad \vdots \\ &= \bigotimes_{d_n}{}_n \hat{\tau}_{n-1}(\{x_n \mapsto d_n\}) = \hat{\tau}_n \end{aligned}$$

where $\hat{\tau}_n$ is a constant. The procedure is presented as Algorithm 2 using a method `reduce` to perform the reduction. This method is discussed in detail in the following section.

5.3 Reducing Constraint Trees

The algorithmic idea of how to eliminate a free variable $x \in \mathcal{V}$ from a constraint tree $\tau : \mathcal{D}^{\mathcal{V}} \to \mathcal{T}_{\mathbb{S}}$ is to compute for each valuation $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ the set of leafs in τ that can be reached by extensions of θ.

Algorithm 2 Evaluates a constraint tree with respect to variable quantification

```
1 function eval: ({∀,∃} × V)* × T_S → T_S ∪ S
2 function eval =
3   case (ε, LeafCTree(s)) then s
4   case (ε, InnerCTree(φ, τ0, τ1)) then
5     InnerCTree(φ, τ0, τ1)
6   case (u::(∀,x), τ) then
7     eval(u, reduce(⨅, x, τ))
8   case (u::(∃,x), τ) then
9     eval(u, reduce(⨆, x, τ))
```

More precisely, for values $d, d' \in \mathcal{D}$, extensions $\theta[x \mapsto d], \theta[x \mapsto d'] \in \mathcal{D}^{\mathcal{V}}$ of θ may satisfy different path constraints $\rho(l)$, $\rho(l')$ leading to leafs $l, l' \in L$ in τ labelled by verdicts v, v'. Thus, for the constraint tree

$$\tau = (I, L, S_1, S_2, C, \lambda_I, \mathbb{S}, \lambda_L)$$

over $\mathcal{D}^{\mathcal{V}}$ a valuation $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ induces a finite set of verdicts

$$\begin{aligned} \{v \in \mathbb{S} \mid \exists_{d \in \mathcal{D}} \exists_{l \in L} : (t, \theta[x \mapsto d]) \models \rho(l) \lambda_L(l) = v\} \\ = \{\tau(\theta[x \mapsto d]) \mid d \in \mathcal{D}\}. \end{aligned}$$

Having computed this set allows for applying the intended operator $\otimes \in \{\bigsqcap, \bigsqcup\}$ to it in order to evaluate

$$\bigotimes_{d \in \mathcal{D}} \tau(\theta[x \mapsto d]) = \otimes\{\tau(\theta[x \mapsto d]) \mid d \in \mathcal{D}\}. \quad (2)$$

The idea of the algorithm is thus to identify that particular finite set for a given valuation θ and evaluate it using the operator.

Notice that two valuations that induce the same subset of leafs in τ necessarily evaluate to the same verdict in the equation above. While there are infinitely many valuations, there are only finitely many subsets of leafs in τ and thus there are only finitely many equivalence classes of valuations that need to be distinguished. Bluntly speaking, it is therefore only a matter of choosing and arranging constraints appropriately in order to construct a constraint tree that expresses this distinction.

For a particular valuation θ, there are three distinct cases for the set of leafs induced by θ: Either all leafs are from the left (S_1) subtree τ_1 of τ, all leafs are from the right (S_2) one τ_2 or there are leafs from both. This generalises the three cases $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$ from the example above. The constraint of the root in τ is used to partitions the valuations space $\mathcal{D}^{\mathcal{V}\setminus\{x\}}$. In the example we encountered the special case of $|\mathcal{V}| = 2$.

After having used the constraint of the root of τ to split the valuation space we can descend to the subtrees and recursively continue splitting. For the former two cases, only a single subtree is relevant whereas in the third case the leafs come from both subtrees of a node. Therefore the recursive algorithm passes on a *working set* of subtrees which are then processed to obtain a set of leafs. This procedure, `reduce`, is presented as Algorithm 3.

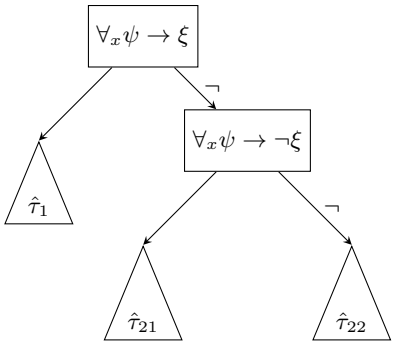


Fig. 4. The constraint tree $\hat{\tau}$ constructed by `reduce`.

Every subtree in the working set carries an additional guard ψ in terms of a formula. This guard characterises the valuations that are relevant for the corresponding subtree. Initially the guard is simply the formula *true* since all valuations are relevant. Given that the input tree is of the form

$$\tau = \texttt{InnerCTree}(\xi,\ \tau_1,\ \tau_2),$$

`reduce` constructs a tree $\hat{\tau}$ as depicted in Figure 4 where $\psi = \text{true}$. The three subtrees $\hat{\tau}_1$, $\hat{\tau}_{21}$ and $\hat{\tau}_{22}$ are constructed recursively. Notice that the new tree is constructed using constraints from τ but the variable x is always bound by some first-order quantifier and hence does not occur freely anymore. The result is therefore a constraint tree over $\mathcal{V} \setminus \{x\}$ as intended.

The algorithm maintains information about its current position in the constructed tree $\hat{\tau}$ in terms of the path constraint ρ. As soon as this path constraint becomes unsatisfiable, recursion stops simply meaning that the corresponding branch in $\hat{\tau}$ is pruned. This is, however, not essential and could also be performed after the tree is constructed or simply omitted. Branches in a constraint tree where the path constraint is not satisfiable are irrelevant to its semantics in terms of a function.

Initially the algorithm is started on the working set containing only τ but in later recursions the working set may contain several (guarded) subtrees of τ. The general scheme to construct the new left subtree $\hat{\tau}_1$ is that the algorithm replaces τ by τ_1 in the current working set and recursively calls `reduce` on that set. The subtree $\hat{\tau}_{21}$ is constructed in the same way replacing τ by τ_2. The guards of τ_1 and τ_2 in the new respective working sets are the same formula ψ that guards τ (and are thus true on the first level of recursion).

The set of leafs of the subtree $\hat{\tau}_{22}$ contains leafs from both of the subtrees τ_1 and τ_2. Therefore, $\hat{\tau}_{22}$ is obtained by removing τ from the current working set and in turn adding both, τ_1 and τ_2. Then the recursion continues on this set. Here, the guards of τ_1 and τ_2 are strengthened. The constraint ξ that is attached to the root of τ distinguishes the valuations concerning τ_1 from those concerning τ_2. To maintain the information that τ_1 corresponds to (only) the case where ξ holds the guard ξ is added to the guard ψ of τ in the working set. Conversely, the constraint $\neg\xi$ is added to ψ to obtain the guard of τ_2. This restricts the valuations that are relevant for τ_2 to those violating ξ. This information is necessary to compute the correct set of leafs from the subtree in later phases.

The parameters of the recursive procedure `reduce` as it is presented in Algorithm 3 are therefore the following.

- $\otimes \in \{\bigsqcup, \bigsqcap\}$: The operator used for quantification
- $x \in \mathcal{V}$: The variable that is to be eliminated
- $W \subseteq FO[T] \times \mathcal{T}_{\mathbb{S}}$: A set of (sub-)trees that still need to be processed guarded by constraints that characterise the valuations relevant for the particular subtree
- ρ: The conjunction of the constraints in the new tree from the root to the currently constructed nodes

The recursion of the algorithm stops when all trees in the working set W have been entirely processed, meaning that every tree in W consists of a single leaf. At this point, intuitively, enough decisions were made such that the valuations $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ satisfying the corresponding constraints can not be distinguished any further by the input tree τ. The working set W is therefore precisely the set of leafs in τ that are relevant to extensions of θ.

The intended value of $\hat{\tau}(\theta)$ for any such θ is hence obtained by applying the operator $\otimes$ to the set of labels of these leafs. The algorithm consequently creates a leaf node with that label.

5.4 Correctness

In the following we provide the necessary arguments for the correctness of the evaluation procedure. The key component in this procedure is the elimination of some free variable x from a constraint tree $\tau : \mathcal{D}^{\mathcal{V}} \to \mathbb{S}$. We show that this elimination is sound, i.e., it computes a constraint tree that yields the exact outcome of quantifying over x in τ. This is formally stated in the following theorem which to prove is the main concern in this section.

Theorem 2 (Reduction). *Let $\tau \in \mathcal{T}_{\mathbb{S}}$ be a constraint tree with $\tau : \mathcal{D}^{\mathcal{V}} \to \mathbb{S}$, $x \in \mathcal{V}$ and $\otimes \in \{\bigsqcap, \bigsqcup\}$. The reduced constraint tree $\hat{\tau} = \texttt{reduce}(\otimes, x, \tau) \in \mathcal{T}_{\mathbb{S}}$ induces the mapping $\hat{\tau} : \mathcal{D}^{\mathcal{V}\setminus\{x\}} \to \mathbb{S}$ with*

$$\hat{\tau}(\theta) = \bigotimes_{d \in \mathcal{D}} \tau(\theta[x \mapsto d]).$$

For the following considerations we arbitrarily fix an operator $\otimes \in \{\bigsqcap, \bigsqcup\}$ and a variable $x \in \mathcal{V}$. Also we fix the T-structure t representing the data theory and omit it in writing $\theta \models \varphi$ instead of $(t, \theta) \models \varphi$ meaning that some *DL* formula φ is satisfied by a valuation θ in the theory of t. Given a valuation $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ and a value $d \in \mathcal{D}$ we define $\theta_d \in \mathcal{D}^{\mathcal{V}}$ to be the mapping $\theta[x \mapsto d]$ that coincides with θ and maps x to d.

Rather than on a single constraint tree, the algorithm works on a working set $W \subseteq DL \times \mathcal{T}_{\mathbb{S}}$ of constraint trees

Algorithm 3 Eliminates a free variable from a constraint tree

```
1  function reduce: {⨆,⨅} × 𝒱 × 2^{FO[T]×𝒯_𝕊} × FO[T] → 𝒯
2  function reduce =
3    // base case; working set contains only leafs
4    case (⊗, x, W ⊆ FO[T]×LeafCTree, ρ) then LeafCTree(⊗{l | (ψ,LeafCTree(l)) ∈ W})

6    case (⊗, x, {(ψ, InnerCTree(ξ,τ1,τ2))} ∪̇ M, ρ) then
7      // new (left) subtree for the universally positive class (tree is empty if class is empty)
8      τ̂1 := if SAT(ρ ∧ ∀x ψ → ξ) then
9                reduce(⊗, x, M ∪ {(ψ,τ1)}, ρ ∧ ∀x ψ → ξ)
10           else Empty
11     // new (right−left) subtree for the universally negative class (tree is empty if class is empty)
12     τ̂21 := if SAT(ρ ∧ ∀x ψ → ¬ξ) then
13               reduce(⊗, x, M ∪ {(ψ,τ2)}, ρ ∧ ∀x ψ → ¬ξ)
14          else Empty
15     // new (right−right) subtree for the existential class (tree is empty if class is empty)}
16     τ̂22 := if SAT(ρ ∧ (∃x ψ ∧ ξ) ∧ (∃x ψ ∧ ¬ξ)) then
17               reduce(⊗, x, M ∪ {(ψ ∧ ξ,τ1),(ψ ∧ ¬ξ,τ2)}, ρ ∧ (∃x ψ ∧ ξ) ∧ (∃x ψ ∧ ¬ξ))
18          else Empty

20     // At least one of τ̂1, τ̂21 and τ̂22 is non−empty.
21     τ̂2 := if (τ̂21 = Empty) then τ̂22
22           else if (τ̂22 = Empty) then τ̂21
23           else InnerCTree(∀x ψ → ¬ξ, τ̂21, τ̂22)

25     if (τ̂1 = Empty) then τ̂2
26     else if (τ̂2 = Empty) then τ̂1
27     else InnerCTree(∀x ψ → ξ, τ̂1, τ̂2)

29 function reduce: {⨅,⨆} × 𝒱 × 𝒯_𝕊 → 𝒯_𝕊
30 function reduce(⊗, x, τ) = reduce(⊗, x, {(true,τ)}, true)
```

that carry an additional guard. During execution, the guarding *DL* formulae provide the context of subtrees in the global construction. More precisely, they characterise relevant implications of the (partial) path constraint in $\hat{\tau}$ to the node that is currently processed. Let

$$\rho(W) := \bigwedge_{(\psi,\tau)\in W} \exists_x \psi.$$

When processing a subtree of τ, only those valuations are relevant to it that satisfy the constraints along the path to it. The formula $\rho(W)$ is satisfied by those valuations $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ for which each tree in W is relevant, meaning that there is some value for the variable x such that the corresponding guard is satisfied.

The intention is to start with the formula true as initial guard, marking all valuations relevant. When descending, the algorithm processes certain subtrees of the original input tree and passes on the necessary information about the path to the subtree in terms of the guards.

As mentioned earlier, a valuation induces a set of possible verdicts of a tree τ. For proving correctness of the evaluation algorithm we define this set also more generally for working sets. Let, for a set $W \subseteq FO[T] \times \mathcal{T}_{\mathbb{S}}$ and a valuation $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$, denote

$$W(\theta) := \{\tau(\theta_d) \mid d \in \mathcal{D}, (\psi,\tau) \in W, \theta_d \models \psi\}$$

the set of possible verdicts when applying the relevant trees in W to extensions of θ. A tree in W is relevant for θ if it is admitted by the corresponding formula ψ. Notice that for $W = W' \cup W''$ we have $W(\theta) = W'(\theta) \cup W''(\theta)$ for all $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$.

To prove Theorem 2 we need to consider all the information that is passed on by the recursive algorithm. We rely on the following generalisation of Equation 2 above.

Lemma 2. *Let $W \subseteq FO[T] \times \mathcal{T}_{\mathbb{S}}$ be set of constraint trees over valuations $\mathcal{D}^{\mathcal{V}}$ guarded by DL formulae over free variables $\mathcal{V}$. Let $\hat{\tau} = \mathsf{reduce}(\otimes, x, W, \rho)$ for some DL formula ρ with $\rho \Rightarrow \rho(W)$. Then, for all valuations $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ with $\theta \models \rho$,*

$$\hat{\tau}(\theta) = \otimes W(\theta).$$

Proof. We use (well-founded) induction on the structure of W. For example, we can well-order such sets using the number and heights of the contained trees.

For the proof we ignore the satisfiability checks. It is easy to see that the propagated formula ρ is always at least as restrictive as the partial path constraint to the node that is currently processed. That is, if ρ is not satisfiable, the path constraint is not either and the semantics of the constraint tree will ignore this branch of the tree. In fact, starting the procedure with $\rho =$

true causes the algorithm to pass on precisely the path constraints.

Second, we assume ρ to be as weak as it can be which means $\rho \equiv \rho(W)$. That way the result can only become stronger since weakening ρ makes the lemma apply for even more valuations. For better readability, we then omit ρ and also the fixed operator $\otimes$ and variable x in the signature of `reduce`.

Base case. Assume that W contains only constraint trees consisting of a single leaf and let $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ be a valuation with $\theta \models \rho(W)$. Then

$$\begin{aligned}\otimes W(\theta) &= \otimes\{v \mid d \in \mathcal{D}, (\psi, \texttt{LeafCTree}(v)) \in W, \theta_d \models \psi\}\\ &= \otimes\{v \mid (\psi, \texttt{LeafCTree}(v)) \in W\}\end{aligned}$$

since for each guard ψ there is some value d for x such that $\theta_d \models \psi$ because $\theta \models \rho(W)$ implies that $\theta \models \exists_x \psi$. This is, moreover, precisely what reduce computes on W.

Induction. Now assume $W = W' \cup M$ where $W' = \{(\psi, \tau)\}$ contains a proper tree $\tau = \texttt{InnerCTree}(\xi, \tau_1, \tau_2)$. The algorithm selects one proper tree from W and w.l.o.g. we assume this be (ψ, τ).

The procedure reduce(W) uses the sets

$$\begin{aligned}W_1 &= \{(\psi, \tau_1)\} \cup M,\\ W_2 &= \{(\psi, \tau_2)\} \cup M, \text{ and}\\ W_3 &= \{(\psi \wedge \xi, \tau_1), (\psi \wedge \neg\xi, \tau_2)\} \cup M.\end{aligned}$$

to construct a tree $\hat{\tau}$ as depicted in Figure 4 with

$$\begin{aligned}\hat{\tau}_1 &= \mathsf{reduce}(W_1),\\ \hat{\tau}_{21} &= \mathsf{reduce}(W_2) \text{ and}\\ \hat{\tau}_{22} &= \mathsf{reduce}(W_3).\end{aligned}$$

Let $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$ with $\theta \models \rho(W)$. We show that $\otimes W(\theta) = \mathsf{reduce}(W)(\theta)$ by analysing the three (exhaustive) cases for θ distinguished by $\hat{\tau}$.

Assume $\theta \models \forall_x \psi \to \xi$ and $W_1' := \{(\psi, \tau_1)\}$.We have for all θ_d satisfying ψ that $\tau(\theta_d) = \tau_1(\theta_d)$ and hence

$$\begin{aligned}W(\theta) &= W'(\theta) \cup M(\theta)\\ &= \{\tau(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi\} \cup M(\theta)\\ &= \{\tau_1(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi\} \cup M(\theta)\\ &= W_1'(\theta) \cup M(\theta) = W_1(\theta)\end{aligned}$$

Also, $\theta \models \rho(W_1)$ since $\rho(W) = \rho(W_1)$ and thus by induction

$$\hat{\tau}(\theta) = \hat{\tau}_1(\theta) = \otimes W_1(\theta) = \otimes W(\theta).$$

The case for W_2 where we assume $\theta \models \forall_x \psi \to \neg\xi$ follows analogously.

For the remaining case assume now that

$$\theta \models (\exists_x \psi \wedge \xi) \wedge (\exists_x \psi \wedge \neg\xi)$$

and let $W_3' := \{(\psi \wedge \xi, \tau_1)\}$ and $W_3'' := \{(\psi \wedge \neg\xi, \tau_2)\}$. For $d \in \mathcal{D}$ we have

$$\begin{aligned}\tau(\theta_d) &= \tau_1(\theta_d) \quad \text{if } \theta_d \models \xi,\\ \tau(\theta_d) &= \tau_2(\theta_d) \quad \text{if } \theta_d \models \neg\xi.\end{aligned}$$

Therefore, similar to the case above,

$$\begin{aligned}W(\theta) &= W'(\theta) \cup M(\theta)\\ &= \{\tau(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi\} \cup M(\theta)\\ &= \{\tau(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi \wedge c\}\\ &\quad \cup \{\tau(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi \wedge \neg c\} \cup M(\theta)\\ &= \{\tau_1(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi \wedge c\}\\ &\quad \cup \{\tau_2(\theta_d) \mid d \in \mathcal{D}, \theta_d \models \psi \wedge \neg c\} \cup M(\theta)\\ &= W_3'(\theta) \cup W_3''(\theta) \cup M(\theta) = W_3(\theta)\end{aligned}$$

Also, $\theta \models (\exists_x \psi \wedge \xi) \wedge (\exists_x \psi \wedge \neg\xi)$ and $\theta \models \rho(W) = \rho(W') \wedge \rho(M)$ implies that

$$\theta \models (\exists_x \psi \wedge \xi) \wedge (\exists_x \psi \wedge \neg\xi) \wedge \rho(M) = \rho(W_3).$$

Using the induction hypothesis we can hence conclude that

$$\hat{\tau}(\theta) = \hat{\tau}_{22}(\theta) = \otimes W_3(\theta) = \otimes W(\theta)$$

which completes the proof. □

We obtain Theorem 2 directly by applying Lemma 2 to the call of reduce. Applying the algorithm to a constraint tree τ yields, for $W_0 := \{(\text{true}, \tau)\}$ and $\theta \in \mathcal{D}^{\mathcal{V}\setminus\{x\}}$,

$$\begin{aligned}\texttt{reduce}(\otimes, x, \tau)(\theta) &= \texttt{reduce}(\otimes, x, W_0, \text{true})(\theta)\\ &= \otimes W_0(\theta) = \otimes\{\tau(\theta_d) \mid d \in \mathcal{D}\}.\end{aligned}$$

As argued before, an immediate consequence of Theorem 2 is that iterating the reduction computes the value of a term

$$\underset{d_n}{\otimes_n} \ldots \underset{d_1}{\otimes_1} \tau(\theta_{(d_1,\ldots,d_n)}).$$

Taking τ to be the output of a data monitor $\mathcal{M}_\varphi$ for some *TDL* formula φ on an observation sequence $\gamma \in \Gamma^*$ and $\otimes_n, \ldots, \otimes_n$ to be the operators corresponding to the global quantifiers in φ we obtain by Theorem 1 the *TDL* semantics of φ.

Theorem 3 (Evaluation correctness). *For a TDL sentence $\varphi = Q_n x_n \ldots Q_1 x_1 \psi$ with core formula ψ let $\mathcal{M}_\varphi$ be the data monitor for φ and $\gamma \in \Gamma^*$ an observation sequence. Then*

$$\texttt{eval}(Q_n x_n \ldots Q_1 x_1, \mathcal{M}_\varphi(\gamma)) = [\![\varphi]\!]_{TDL}(\gamma).$$

5.5 Remarks and Optimizations

To improve runtime performance in common special cases, several optimisations to the algorithms described above can be considered. In order to make our approach viable in practice, unnecessary information has to be eliminated to keep the size of the constraint tree small. To this end an inner node $\texttt{InnerCTree}(\varphi, \tau_1, \tau_2)$ can be replaced by τ_2 if $\forall_{\theta \in D^V}(\theta \models \varphi \Rightarrow \tau_1(\theta) = \tau_2(\theta))$ or conversely by τ_1 if $\forall_{\theta \in D^V}(\theta \models \neg\varphi \Rightarrow \tau_1(\theta) = \tau_2(\theta))$. In our experiments presented in Section 6 it proved to be sufficient to enforce this property only for the special case where at least one subtree, τ_1 or τ_2, is a leaf node.

Impartiality and anticipation. An impartial semantics distinguishes between preliminary and final verdicts. A final verdict for some word indicates that it will not change for any continuation. Impartiality is desirable as monitoring can be stopped as a soon as a final verdict is encountered (cf. [BLS07, DLS08]). In the context of our framework this gains even more importance. When the underlying monitor is impartial, a branch already yielding a final verdict can be pruned. This immensely improves runtime performance. Another desired property is anticipation, i.e., evaluating to a final verdict as early as possible. While in general not transferred from the symbolic to the data monitor, this may still lead to better performance.

Dedicated theories as first-class citizens. The monitoring framework is also flexible in the sense that one can trade efficiency for generality. When the properties intended to monitor are simple enough it is reasonable to extend the algorithm to directly evaluate constraints. As we show in the experiments this works well, in particular for properties concerning only object IDs.

Satisfiability checks for monitor evaluation. The computation of the monitor output becomes much more complex when alternated universal and existential quantifiers are used. Algorithm 3 then generates new constraints containing universal quantifiers. Deciding satisfiability of first-order formulae containing universal quantifiers is usually much more expensive and even renders some common first-order theories undecidable. Fortunately, these constraints do not have to be checked in the special case, where a constraint does only contain a single variable. Consider the constraints $\forall_x \psi \to \xi$ and $\forall_x \psi \to \neg\xi$ generated by the algorithm. Assuming that in the constraint tree unsatisfiable branches are pruned, we already know that $\psi \to \xi$ and $\psi \to \neg\xi$ are satisfiable since ψ describes a path and ξ is the constraint of the node reached by that path. If ξ does not contain x, $\forall_x \psi \to \xi$ and $\forall_x \psi \to \neg\xi$ have to be satisfiable. If ξ does contain only x as free variable, $\forall_x \psi \to \xi$ and $\forall_x \psi \to \neg\xi$ can not be satisfiable at the same time. Thus, the satisfiability checks in this case can be completely eliminated. This optimisation can be generalised even further. When a constraint ξ contains multiple variables, the constraint $\forall_x \psi \to \xi$ (and $\forall_x \psi \to \neg\xi$, respectively) can be simplified by removing those constraints from ψ that do not influence ξ, i.e. they do not contain a variable contained in ξ or some other constraint influencing ξ.

Delayed monitor evaluation. It can be observed that the execution of a monitor is completely independent from the computation of its output. Therefore, this computation does not have to be carried out in every step. It can, e.g., be moved to a separate thread or carried out on demand or in larger time intervals. This is especially interesting for log-file analysis where it can be sufficient to compute a result after analysing the complete log-file.

Dynamic monitor evaluation. When the monitor progresses the modifications to the constraint tree representing its state are usually minor and often limited to the leaf nodes. Algorithm 3 can be implemented in a more dynamic fashion where the previous result tree is kept. Only subtrees depending on modified parts of the input tree have to be recomputed.

6 Experimental Results

We implemented the presented framework as part of jUnit^{RV} [DLT13b][1]. This tool allows for monitoring applications running on the Java Virtual Machine with respect to temporal properties. Our implementation is limited to the fragment where all global quantifiers, i.e. the quantifiers in front of the temporal formula, are universal. The previous version of jUnit^{RV} supported classical LTL specifications referring to, e.g., the invocation of a method of some class. With the approach proposed here it is now possible, for example, to specify properties that relate to individual objects and their evolution in time. The implementation is based on a generic interface to an SMT solver. We present benchmarks using the SMT solver Z3 [dMB08]. We relied on the Java Native Access API (JNA) to integrate Z3. It provides a convenient way to invoke APIs of native libraries without writing any native code. For comparison, we additionally implemented a dedicated solver for the theory of IDs (i.e., conjunctions of equality constraints on natural numbers).

For the benchmarks[2], we have chosen representative properties from Table 1. The property *mutex* is a typical example for interaction patterns in object-oriented systems. It was evaluated on a program with resource objects and user objects randomly accessing them. The *iterator* example was evaluated on a simple program using randomly one of two iterator objects for traversing a list. Third, we evaluated a typical client-server

[1] Project page: `www.isp.uni-luebeck.de/junitrv`

[2] The benchmarks and corresponding implementation are available at `www.isp.uni-luebeck.de/~thoma/junitrv-sttt14.zip`

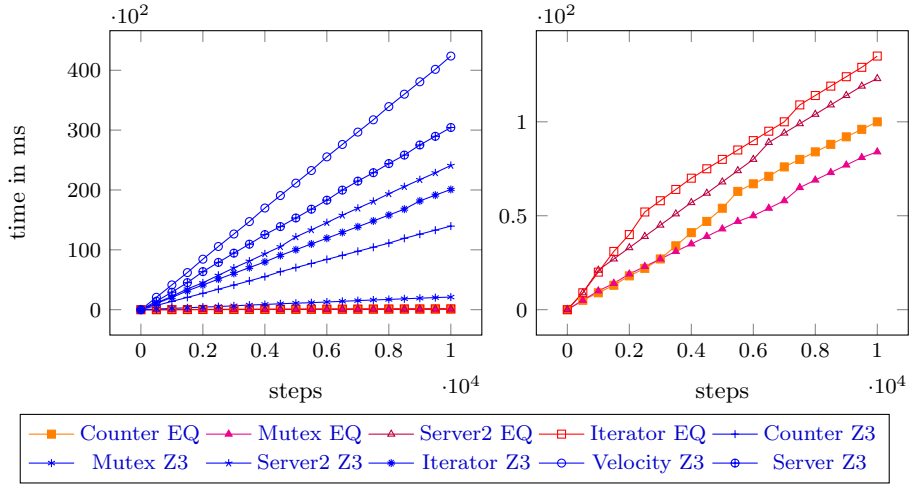


Fig. 5. Experimental results

response pattern (*server*) on a program simulating a number of server threads that receive requests and responses. For handling existential quantification, we rely on Z3. For comparison, we also evaluate the property $\forall_t \forall_x \, \mathrm{G}(\langle \mathsf{request}(t,x) \rangle \rightarrow F \langle \mathsf{response}(x,t) \rangle)$ (*server2*) as a variation without existential quantification that can therefore be handled by our simple solver. The *counter* property covers the counting of natural numbers which is a very elementary aspect in computer programs and uses an unbounded number of different data values. A property involving a rather complex theory is *velocity*. The free variables refer to real numbers as data values and the constraints that have to be checked are multi-dimensional.

In our experiments we measured the execution time of a program with an integrated monitor over the number of monitoring steps. The measurements were taken up to 10^4 steps. Very simple programs were used, since the measured runtime is thereby essentially the runtime of the monitoring algorithm. The linear graphs obtained for every example show that the execution time for a monitoring step is constant. The most complex properties, *velocity* and *server* induce the most overhead due to a higher computational cost by the SMT solver. However, even the performance for *velocity* of 4.2 ms/step is acceptable for many applications. Thus, employing an SMT solver is viable whenever performance is not a main concern, for instance in case a monitoring step is not expected to happen frequently with respect to the overall computation steps. Our dedicated implementation is much faster (by factor 100) and hence can only be distinguished in the right-hand diagram. These results demonstrate, that performance can be improved for specific settings and the approach can still be employed when performance is more critical. The gain of performance is due to several reasons. Firstly, the limitation to only very simple constraints over IDs allows us to use a very fast decision procedure relying on hash sets. Secondly, our monitoring procedure only modifies constraints slightly in each step. Our solver allows to reuse previous computation results to check these constraints for satisfiability. While common SMT solvers as Z3 offer similar functionality their API is too limited to avoid recomputations in our setting. Thirdly, our solver being written in Java avoids any overhead that might come from using JNA for executing native calls.

As mentioned before, the number of calls to the SMT solver is linear in the size of the constraint tree. Hence, the overhead may increase up to linearly in the number of runtime objects that need to be tracked. In our examples the maximal size of the constraint tree was six. All experiments were carried out on an Intel i5 (750) CPU.

7 Conclusion

The presented framework addresses a central issue in runtime monitoring: convenient specification and efficient verification of system executions. The combination of propositional temporal logics and first-order theories allows for a precise, yet high-level and universal formulation of behavioural properties. The ability to formulate declarative specifications at a higher level of abstraction than that of an actual implementation helps the user to avoid modelling errors.

The specification formalism exceeds the expressiveness of previous approaches which clearly comes at the cost of runtime overhead. However, the framework's flexibility allows the user to freely choose a suitable trade-off in terms of a theory to reason about data and a temporal logic for expressing temporal behaviour. Common special cases such as reasoning only on data in terms of IDs can be handled by dedicated decision routines directly integrated into the monitoring system.

Additionally, the formal basis provides a conceptual presentation that does not depend on a particular implementation, making it easier to develop extensions and

optimisations. For example, while real-time constraints can in principle be expressed using an appropriate theory, this is not taken into account in the monitor synthesis procedure. Integrating timing constraint as first-class citizens in the temporal logic could allow for applying corresponding optimisations [BLS11] during monitor synthesis. Apart from such dedicated efforts, the monitoring approach can take advantage from independent improvements in SMT solving and monitor synthesis.

The evaluation of concrete verdicts can become computationally expensive for properties relying on extensive alternation of global quantifiers. Here the separation of the monitoring and evaluation procedures allows for their independent execution as there is no conceptual need to perform it continuously, synchronously or even within the same execution environment.

Although we believe that the great flexibility is valuable to deal with special situations the universal fragment appears to be the most relevant for online monitoring. Global existential quantification is useful to express properties like *controller*, *observer* and *mediator* presented in Section 3.4, Table 1. Consider, for example, the *controller* property. In case of a real system a controller thread would typically be marked with a special attribute and thus the property can be reformulated using only universal quantification $\forall_p \mathrm{G}\langle\forall_{p'} \mathsf{request}(p, p') \rightarrow \mathsf{controller}(p')\rangle$. Existential quantification can, however, be useful when monitoring or log-file analysis is used in a more investigative manner. For example, it might be unknown whether the system is organised using a controller thread and how the controller thread can be recognised. Then existential quantification still allows the user to formulate and check the property as a hypothesis.

Our implementation and the experimental evaluation show that the approach is applicable in the setting of object-oriented systems and that the runtime overhead is reasonably small. Note that this is although the properties expressible in our framework are hard to analyse. The satisfiability problem, for example, is already undecidable for the combination of LTL and the very basic theory of identities.

References

AAC+05. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364. ACM, 2005. 3

AEM04. Rajeev Alur, Kousha Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004. 3, 7

BCRZ99. Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifiying safety properties of a power PC microprocessor using symbolic model checking without bdds. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1999. 4

BFH+12. Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012. 1, 4

BGHS04. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004. 3

BHW+13. Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime verification for multicore soc with high-quality trace data. *ACM Trans. Design Autom. Electr. Syst.*, 18(2):18, 2013. 1

BJK+05. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005. 1

BKM10. David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010. 1, 4

BKV13. Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture*

Notes in Computer Science, pages 59–75. Springer, 2013. 4

BLS06. Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006. 7

BLS07. Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007. 17

BLS11. Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14, 2011. 1, 7, 19

BRH07. Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From eagleto ruler. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007. 1, 3

CGP01. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. 1

CPS09. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — safer monitoring of real-time java programs (tool paper). In Dang Van Hung and Padmanabhan Krishnan, editors, *Seventh IEEE International Conference on Software Engineering and Formal Methods, SEFM 2009, Hanoi, Vietnam, 23-27 November 2009*, pages 33–37. IEEE Computer Society, 2009. 1

CR05. Feng Chen and Grigore Rosu. Java-mop: A monitoring oriented programming environment for java. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 2005. 1

CR09. Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2009. 4

DKT14. Normann Decker, Franziska Kühn, and Daniel Thoma. Runtime verification of web services for interconnected medical devices. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 235–244. IEEE, 2014. 1, 2

DLS08. Wei Dong, Martin Leucker, and Christian Schallhart. Impartial anticipation in runtime-verification. In Sung Deok Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan, editors, *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*, volume 5311 of *Lecture Notes in Computer Science*, pages 386–396. Springer, 2008. 3, 8, 17

DLT13a. Normann Decker, Martin Leucker, and Daniel Thoma. Impartiality and anticipation for monitoring of visibly context-free properties. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2013. 8

DLT13b. Normann Decker, Martin Leucker, and Daniel Thoma. junit$^{\mathrm{rv}}$-adding runtime verification to junit. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, volume 7871 of *Lecture Notes in Computer Science*, pages 459–464. Springer, 2013. 1, 2, 17

DLT14. Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2014. 4

dMB08. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. 17

dMB11. Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011. 2

EFT94. Heinz-Dieter Ebbinghaus, Jörg Flum, and Wolfgang Thomas. *Mathematical logic (2. ed.)*. Undergraduate texts in mathematics. Springer, 1994. 4

For82. Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artif. Intell.*, 19(1):17–37, 1982. 3

Hav14. Klaus Havelund. Monitoring with data automata. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II*, volume 8803 of *Lecture Notes in Computer Science*, pages 254–273. Springer, 2014. 4

Hav15. Klaus Havelund. Rule-based runtime verification revisited. *STTT*, 17(2):143–170, 2015. 3

LS07. Martin Leucker and César Sánchez. Regular linear temporal logic. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, editors, *Theoretical Aspects of Computing - ICTAC 2007, 4th International Colloquium, Macau, China, September 26-28, 2007, Proceedings*, volume 4711 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2007. 7

LS09. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009. 1

MJG+12. Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012. 4

SB06. Volker Stolz and Eric Bodden. Temporal assertions using aspectj. *Electr. Notes Theor. Comput. Sci.*, 144(4):109–124, 2006. 3

Sto10. Volker Stolz. Temporal assertions with parametrized propositions. *J. Log. Comput.*, 20(3):743–757, 2010. 4