Technical Report

Nr. TUD-CS-2012-0014
January 17th, 2012

Identifying meaningless parameterized linear-temporal-logic formulas

Authors
Eric Bodden (EC SPRIDE, CASED)

Identifying meaningless parameterized linear-temporal-logic formulas*

Eric Bodden
bodden@acm.org

Secure Software Engineering Group
European Center for Security and Privacy by Design
Technische Universität Darmstadt

Abstract. Parameterized runtime monitoring formalisms allow predicates to bind free variables to values during the program's execution. Some runtime monitoring tools, like *J-LO*, increase the formalism's expressiveness by allowing predicates to query variables already during the matching process. This is problematic because, if no special care is taken, the predicate's evaluation may need to query a variable that has not yet been bound, rendering the entire formula meaningless.
In this paper we present a syntactic checking algorithm that recognizes meaningless formulas in future-time linear temporal logic. The algorithm assures that a predicate accesses a potentially unbound variable only when the truth value of this predicate cannot possibly impact the truth value of the entire formula at the time the predicate is being evaluated. Our approach allows users to specify a wide range of meaningful parameterized logic formulas, while at the same time forbidding such formulas that would otherwise have an unclear semantics due to insufficient bindings.
We have implemented the checking algorithm in the *J-LO* runtime verification tool.

1 Introduction

Many current runtime verification formalisms are parametric [1], i.e., they allow formulas to contain parameters, free variables that can bind to concrete values during evaluation over a program trace. Parametric formalisms are more expressive than their non-parametric counterparts. Consider the parameterless formula "$G(\texttt{close} \rightarrow XG(\neg\texttt{write}))$" in Linear Temporal Logic [2] (LTL), which is supposed to induce a runtime monitor alerting the user whenever writing to a closed connection.[1] Such a parameterless formula can easily lead to unintended answers when events involve multiple different program values, e.g., method calls on different objects. For instance, consider a program that generates the call sequence "c_1`.close()` c_2`.write(..)`" when executing. Monitoring this program execution with the pattern "`close write`" will cause a warning to be issued although the `close` and `write` method calls occur on different connection objects (c_1 vs. c_2), and hence the program never writes to a closed connection.

Parametric formalisms solve this problem by allowing for free variables. For instance, we could write the regular expression "$G(\texttt{close(c)} \rightarrow XG(\neg\texttt{write(c)}))$", which is parametric in `c`. The usual semantics of such parametric formalisms is to evaluate formulas against fragments of the trace that share a consistent variable binding [1, 3], i.e., bind variables such as `c` to the same value (or object) at all

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

[1] Here X stands for "next" and G for "globally".

events that bind `c` within that trace fragment. For instance, for the above regular expression and the trace "`close(c=`c_1`) write(c=`c_2`) close(c=`c_2`) write(c=`c_1`)`", the prominent runtime monitoring tool JavaMOP [4] would issue a warning only for c_1 (after reading the last event), but not for c_2. Such a semantics has proven quite intuitive for monitoring object-oriented programs. Another advantage of parameterized formalisms is that some runtime monitoring tools, like tracematches [5] for instance, allow users to access the values of bound variables once a match has been completed. That way, a monitor can operate on the very objects that actually caused the observed property violation, for instance c_2 on the example above.

Some runtime monitoring tools even go one step further. The *J-LO* tool [6, 7] allows predicates in its LTL formulas to access the values of bound variables already during the matching process. For instance, assume that we want to warn users when using an iterator `i` over a collection `c` if `c` has been cleared after `i` had been created. (This is a specialized version of the *FailSafeIter* property [3].) The following formula expresses this condition in Linear Temporal Logic:

$$G(\texttt{createIter(c,i)} \rightarrow XG(\texttt{clear(c)} \rightarrow XG(\neg\texttt{next(i)})))$$

In this formula, the event `createIter` is parametric in `c` and `i` (we also say that it *binds* `c` and `i`), `clear` is parametric in `c` and `next` in `i`. The above formula has the problem that it is over-specified: it is ok for the user to clear the collection (and then still use the iterator) if the collection at that point is empty already. In *J-LO*, users could hence refine the formula as follows:

$$G(\texttt{createIter(c,i)} \rightarrow XG((\texttt{clear(c)} \wedge \neg\underline{\texttt{isEmpty(c)}}) \rightarrow XG(\neg\texttt{next(i)})))$$

In this formula, and in the remainder of this paper, we underline that do not represent program events but rather represent additional conditions that must hold when other events, such as `clear` occur. Such underlined "query predicates" query variables rather than establishing a variable binding. As we can see, the query predicate in the example enhances the formula's expressiveness by filtering out, at the time of the `clear` event, such traces on which the cleared collection is empty already.

Unfortunately, if no special care is taken, query predicates allow programmers to write quite meaningless formulas for which it is unclear which truth value they should evaluate to. For instance, consider the formula $\varphi = p(x) \vee X\underline{q(x)}$. Here $p(x)$ binds x to a value and $\underline{q(x)}$ queries that variable binding. This formula is unproblematic on traces where p does hold initially: in this case the runtime monitor can just evaluate $\underline{q(x)}$ using the variable binding that $p(x)$ produced earlier. Crucially, however, the formula has no semantics if p does not hold at the beginning of the trace. In this case the truth value of φ only depends on the truth value of $\underline{q(x)}$ at the second observed event. But how can we evaluate $\underline{q(x)}$ in a meaningful way if $p(x)$ does not hold, and hence no variable binding for x was ever recorded? The semantics of such a formula is unclear. In this paper we propose a checking algorithm that will flag such a formula as meaningless ahead of time.

Telling apart meaningful from meaningless formulas is non-trivial. For instance, consider the very similar formula $\varphi' = \neg p(x) \vee X\underline{q(x)}$, where p binds x under negation. In this case, if p does not hold, because φ' is a disjunction, the truth value of φ' is *true* irrespective of the truth value of $\underline{q(x)}$. This formula hence always has a truth value, and, opposed to φ, should be allowed by our checking algorithm.

In this paper we present a checking algorithm that operates on the syntactic structure on an LTL formula to determine whether the formula's truth value could be undetermined due to query predicates accessing potentially unbound values. The algorithm is designed to allow programmers to define a large set of meaningful parametric LTL formulas, while at the same time rejecting such formulas that are

clearly meaningless. We have implemented the checking algorithm in the *J-LO* runtime verification tool.

To summarize, this paper contains the following original contributions:

- An static, syntactic checking algorithm for identifying and rejecting LTL formulas that are meaningless because predicates may access potentially unbound variables.
- An implementation of this algorithm in the *J-LO* runtime verification tool.

The remainder of this paper is organized as follows. In Section 2 we give a finite-path semantics for LTL formulas. In Section 3 we then show that any such LTL formula φ can be split into two formulas, $now(\varphi)$ and $next(\varphi)$, which are evaluated over the current and next state respectively. In Section 4 we explain the intuition behind our checking algorithm in terms of those *now* and *next* formulas. Section 5 finally presents our checking algorithm. We present related work in Section 7 and conclude in Section 8.

2 A finite-path semantics for future-time LTL

Linear-time temporal logic (*LTL*) [2] is a subset of the Computation Tree Logic CTL* and extends propositional logic with operators which describe events along a computation path. The operators of LTL have the following meaning:

- "Next" ($\mathbf{X}\ \varphi$): The property φ holds in the next step
- "Finally" ($\mathbf{F}\ \varphi$): φ will hold at some state in the future
- "Globally" ($\mathbf{G}\ \varphi$): At every state on the path φ holds
- "Until" ($\varphi\ \mathbf{U}\ \psi$): φ has to hold until finally ψ holds.
- "Release" ($\varphi\ \mathbf{R}\ \psi$): Dual of **U**; expresses that the second property holds along the path up to and including the first state where the first property holds, although the first property is not required to hold eventually.

Usually LTL is defined over infinite paths. For runtime verification, we assume that the verification process is stopped at some point in time and correspondingly the LTL formulas have to be evaluated over a *finite* path. In particular, all proof obligations must be fulfilled before the finite teace ends. Thus we declare the semantics as follows.

Let P be a set of atomic propositions and $w = w[1]...w[n] \in (2^P)^n$ a finite path. For each path position $w[j]$ $(1 \leq j \leq n)$ and proposition $p \in \{p_1, ..., p_m\}$ and formulas φ and ψ:

$$
\begin{array}{lll}
w[j] \models \mathbf{true} & & \\
w[j] \not\models \mathbf{false} & & \\
w[j] \models p & \text{iff} & p \in w[j] \\
\models \mathbf{X}\ \varphi & \text{iff} & j < n \text{ and } w[j+1] \models \varphi \\
\models \mathbf{F}\ \varphi & \text{iff} & \exists k\ (j \leq k \leq n) \text{ such that } w[k] \models \varphi \\
\models \mathbf{G}\ \varphi & \text{iff} & \forall k\ (j \leq k \leq n) \rightarrow w[k] \models \varphi \\
\models \varphi\ \mathbf{U}\ \psi & \text{iff} & \exists k\ (j \leq k \leq n) \text{ such that } w[k] \models \psi \\
 & & \quad \wedge\ \forall l\ (j \leq l < k) \rightarrow w[l] \models \varphi \\
\models \varphi\ \mathbf{R}\ \psi & \text{iff} & \forall k\ (j \leq k \leq n) \rightarrow w[k] \models \psi \\
 & & \quad \vee\ \exists l\ (j \leq l < k) \text{ such that } w[l] \models \varphi
\end{array}
$$

We write $w \models \varphi$ if $w[1] \models \varphi$. Furthermore we consider formulas in *negation normal form* where negations are pushed down to the propositions using basic equivalences. We call the set of all LTL formulas in this form LTL_{NN}. The negation normal form can be computed using a function *nnf* as follows:

$$
\begin{aligned}
nnf : LTL &\to LTL_{NN} \\
\neg \mathbf{true} &\mapsto \mathbf{false} \\
\neg \mathbf{false} &\mapsto \mathbf{true} \\
\neg p &\mapsto \neg p \\
\neg\neg\phi &\mapsto nnf(\phi) \\
\neg(\phi \wedge \psi) &\mapsto nnf(\neg\phi) \vee nnf(\neg\psi) \\
\neg(\phi \vee \psi) &\mapsto nnf(\neg\phi) \wedge nnf(\neg\psi) \\
\neg \mathbf{X}\ \phi &\mapsto \mathbf{X}\ \neg nnf(\phi) \\
\neg(\phi\ \mathbf{R}\ \psi) &\mapsto (nnf(\neg\phi)\ \mathbf{U}\ nnf(\neg\psi)) \\
\neg(\phi\ \mathbf{U}\ \psi) &\mapsto (nnf(\neg\phi)\ \mathbf{R}\ nnf(\neg\psi))
\end{aligned}
$$

Here we assume that the operators **F** and **G** have already been reduced to **U** and **R** using the following equivalences:

$$
\begin{aligned}
\mathbf{F}\ \phi &\equiv \mathbf{true}\ \mathbf{U}\ \phi \\
\mathbf{G}\ \phi &\equiv \mathbf{false}\ \mathbf{R}\ \phi
\end{aligned}
$$

3 Dividing formulas into *now* and *next*

In the following we will show that any LTL formula φ can be partitioned, with respect to the current state $w[i]$, into two formulae $now(\varphi)$ and $next(\varphi)$ in such a way that $w[i] \models \varphi$ iff $w[i] \models now(\varphi)$ and $w[i] \models next(\varphi)$. We explain this splitting as it will be important to understand the intuition behind our static checking algorithm.

In the following we assume a path $w = w[1], \ldots, w[n]$ with $n \geq 1$.

Definition 1 (Function *now*). *The function $now : LTL \to LTL$ is recursively defined as:*

$$
\begin{aligned}
now(p) &:= p \\
now(\neg p) &:= \neg now(p) \\
now(\mathbf{X}\ \varphi) &:= \mathbf{true} \\
now(\varphi \wedge \psi) &:= now(\varphi) \wedge now(\psi) \\
now(\varphi \vee \psi) &:= now(\varphi) \vee now(\psi) \\
now(\varphi\ \mathbf{U}\ \psi) &:= now(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
&= now(\psi) \vee now(\varphi) \\
now(\varphi\ \mathbf{R}\ \psi) &:= now(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
&= (\ now(\psi) \wedge now(\varphi)\ ) \vee now(\psi) \\
&= now(\psi)
\end{aligned}
$$

Note that for any φ the result of $now(p)$ is a Boolean combination of propositions. The function $now(\varphi)$ reflects that part of φ that can be fully evaluated in state $w[i]$, under the assumption that φ holds on the subsequent path.

Definition 2 (Function *next*). *For $1 \leq i \leq n$, the function $next : LTL \to LTL$ is recursively defined by $next(\varphi) := \mathbf{X}\ next'(\varphi)$ with $next'(\varphi)$ defined as:*

If $i < n$ then:

$$
\begin{aligned}
next'(p) &:= \begin{cases} \textbf{true} & \text{if } w[i] \models p, \\ \textbf{false} & \text{otherwise} \end{cases} \\
next'(\neg p) &:= \neg\ next'(p) \\
next'(\mathbf{X}\ \varphi) &:= \varphi \\
next'(\varphi \wedge \psi) &:= next'(\varphi) \wedge next'(\psi) \\
next'(\varphi \vee \psi) &:= next'(\varphi) \vee next'(\psi) \\
next'(\varphi\ \mathbf{U}\ \psi) &:= next'(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
next'(\varphi\ \mathbf{R}\ \psi) &:= next'(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi)))
\end{aligned}
$$

Else ($i = n$):

$$next'(\varphi) := \textbf{false}$$

Note that the definition of next depends on the state $w[i]$. Also note that whenever $next(p) \in \{\textbf{true}, \textbf{false}\}$, then we have the opportunity to apply early fault detection: The formula is fully determined. One can report satisfaction respectively failure at once.

Example 1 (Functions now and next). Given the formula $\varphi = p\ \mathbf{U}\ q$ and the path $w = \{p\}\{q\}$, such that $w \models \varphi$ holds.
The calculation of *now* leads to:
$now(\varphi) = now(p\ \mathbf{U}\ q) = q \vee p$
The calculation of *next* leads to:
$next(\varphi)$
$= \mathbf{X}\ next'(p\ \mathbf{U}\ q)$
$= \ldots$
$= \mathbf{X}(\ \textbf{false} \vee (\ next'(p) \wedge next'(\mathbf{X}(p\ \mathbf{U}\ q))\ )\ )$
$= \mathbf{X}(\ \textbf{false} \vee (\ \textbf{true} \wedge (p\ \mathbf{U}\ q)\ )\ )$
$= \mathbf{X}(\ p\ \mathbf{U}\ q\ )$

Now it holds that $w = \{p\}\{q\} \models now(\varphi) = q \vee p$ and $w = \{p\}\{q\} \models next(\varphi) = \mathbf{X}(\ p\ \mathbf{U}\ q\ )$.

Theorem 1 (Correctness of *now* and *next*). *For all $\varphi \in LTL$ and all $w \in S^+$ it holds that:*

$$w \models \varphi \iff w \models now(\varphi) \wedge next(\varphi)$$

We prove this theorem correct in [7]. □

4 Propagation of variable bindings

In this section we will use some simple examples to explain the basic intuition behind our checking algorithm. We will show that certain variable bindings propagate through a conjunction, i.e., across the $\wedge$ operator, while others propagate through disjunctions, i.e., across the $\vee$ operator. Section 5 will explain our checking algorithm in a formal manner.

Example 2 (Propagation across "$\wedge$"). Let us consider the following formula:

$$\varphi(x) := p(x) \wedge \mathbf{X}\ \mathbf{F}\ \underline{q(x)}$$

Like any LTL formula, this formula can be split into two sub-formulas, one that requires evaluation in the current state, and another formula that requires evaluation in the next state. The formula has an obvious splitting to the sub-formulas $now(\varphi(x)) = p(x)$, and $next(\varphi(x))$, which depends on the current state $w[i]$.

Here, two cases can occur:

1. $w[i] \models p(x)$, say with a binding $x = 1$. This binding is available for the rest of the path and in particular for the evaluation of $next_{\varphi(x,y)}(\varphi(1)) = \mathbf{F}\ \underline{q(1)}$ on subsequent states.
2. $p(x) \not\models p(x)$. In this case, we have no binding for x at the current state. However, this binding would not be needed anyway, since the formula $now(\varphi(x, y))$ evaluates to **false** already, and hence determines the truth value of the entire formula.

Hence, informally one can say that *bindings defined by propositions propagate over the $\wedge$-operator*: A binding that is defined by a proposition in one branch of an $\wedge$-term is also available in the other branch.

A case which is a bit harder to identify is the following.

Example 3 (Propagation across "$\vee$").

$$\varphi(x) := p(x) \rightarrow \mathbf{X}\ \mathbf{F}\ \underline{q(x)}$$

which in negational normal form amounts to:

$$\neg p(x) \vee \mathbf{X}\ (\ \mathbf{true}\ \mathbf{U}\ \underline{q(x)}\ )$$

At a first glance it seems unclear how a binding should ever be available for the evaluation of $\mathbf{F}\ \underline{q(x)}$, given that $p(x)$ occurs in negated form.

However, again it helps to consider all the possible cases:

1. $p(x)$ does not hold at the current state s. In this case, we have no binding for x at the current state. However, again this does not hurt, since both formulas $now(\varphi(x)) = \neg p(x)$ and $next(\varphi(x))$ evaluate to **true**.
2. $p(x)$ holds at the current state s, say with a binding $x = 1$. Again, this binding is available for the rest of the path and in particular for the evaluation of $next(\varphi(1)) = \mathbf{F}\ \underline{q(1)}$ on subsequent states.

Informally, one can conclude that *bindings defined by negated propositions propagate over the $\vee$-operator.*

5 Checking algorithm

In this section we present the main contribution of this paper, our structural checking algorithms for parameterized LTL formulas. The checking consists of three steps. First, we define a function $def : LTL_{NN} \rightarrow 2^{\mathcal{V}}$ which computes for any LTL function in negational normal form the set of variables that this formula definitely binds. Then, conversely, we define a function *use* that computes variable uses in query predicates. The checking algorithm is then implemented as a function *valid* that uses both *def* and *use*.

Definition 3 (Function *def*). *Let $\varphi \in LTL_{NN}$. For any predicate p, let $\boldsymbol{v}_{\chi_p}$ be the set of variables that p binds. Then we define $def : LTL_{NN} \rightarrow 2^{\mathcal{V}}$ as:*

$$
\begin{aligned}
def(\varphi) &:= def_+(\varphi) \cup def_-(\varphi) \\
&\textit{where} \\
def_+(p) &:= \boldsymbol{v}_{\chi_p} \\
def_+(\neg p) &:= \emptyset \\
def_+(\mathbf{X}\ \varphi) &:= \emptyset \\
def_+(\varphi \wedge \psi) &:= def_+(\varphi) \cup def_+(\psi) \\
def_+(\varphi \vee \psi) &:= def_+(\varphi) \cap def_+(\psi) \\
def_+(\varphi\ \mathbf{U}\ \psi) &:= def_+(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
&= def_+(\varphi) \cap def_+(\psi) \\
def_+(\varphi\ \mathbf{R}\ \psi) &:= def_+(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
&= def_+(\psi) \\
&\textit{and} \\
def_-(p) &:= \emptyset \\
def_-(\neg p) &:= \boldsymbol{v}_{\chi_p} \\
def_-(\mathbf{X}\ \varphi) &:= \emptyset \\
def_-(\varphi \wedge \psi) &:= def_-(\varphi) \cap def_-(\psi) \\
def_-(\varphi \vee \psi) &:= def_-(\varphi) \cup def_-(\psi) \\
def_-(\varphi\ \mathbf{U}\ \psi) &:= def_-(\psi \vee (\varphi \wedge \mathbf{X}(\varphi\ \mathbf{U}\ \psi))) \\
&= def_-(\psi) \\
def_-(\varphi\ \mathbf{R}\ \psi) &:= def_-(\psi \wedge (\varphi \vee \mathbf{X}(\varphi\ \mathbf{R}\ \psi))) \\
&= def_-(\varphi) \cap def_-(\psi)
\end{aligned}
$$

Here $def_+(\varphi)$ provides the variables which are bound by propositions contained in *non-negated* form at the current point in time. Note that, as we described in our previous section, such "positive bindings" propagate over the $\wedge$ operator. This can be seen by the fact that for this operator def_+ computes the union of the positive bindings of both subformulas.

Conversely $def_-(\varphi)$ provides those for propositions which occur *under negation*. Here, the computed "negative bindings" propagate over the $\vee$ operator.

Note that the definitions of def_+ and def_- are entirely symmetric. Also note that both functions disregard bindings produced by the *next* portion of the formula (nested inside X operators), as those bindings will only be supplied at a later point in time.

Next, we define the logical counterpart, the function *use* which represents the variables of $\mathcal{P}_\varphi$ which are *used* by any of the propositions at some point in time but not *defined* by the same proposition.

Definition 4 (Function *use*). *Let $\varphi \in LTL_{NN}$. For any predicate p, let $\underline{\boldsymbol{v}}_{\chi_p}$ be the set of variables that p uses and let $\odot_2 \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\}$. Then we define use : $LTL_{NN} \to 2^{\mathcal{V}}$ as:*

$$
\begin{aligned}
use(p) &:= \underline{\boldsymbol{v}}_{\chi_p} \\
use(\neg p) &:= use(p) \\
use(\mathbf{X}\ \varphi) &:= \emptyset \\
use(\varphi \odot_2 \psi) &:= use(\varphi) \cup use(\psi)
\end{aligned}
$$

We are now ready to define the function $valid(\varphi)$, which is **true** only if φ defines any free variable before it is used.

Definition 5 (Function *valid*). *Let $\varphi \in LTL_{NN}$. Let $p \in \mathcal{P}$. Let $\odot_1 \in \{\neg, \mathbf{X}\}, \odot_2 \in \{\wedge, \vee, \mathbf{U}, \mathbf{R}\}$. Then we define $valid : LTL_{NN} \to \mathbb{B}$ as:*

$$\begin{aligned}
valid(\varphi) &:= valid_{def(\varphi)}(\varphi) \\
&\textit{where for } \mathcal{D} \subseteq \mathcal{V}: \\
valid_{\mathcal{D}}(p) &:= use(p) \subseteq \mathcal{D} \\
valid_{\mathcal{D}}(\odot_1 \varphi) &:= valid_{\mathcal{D} \cup def(\odot_1 \varphi)}(\varphi) \\
valid_{\mathcal{D}}(\varphi \odot_2 \psi) &:= valid_{\mathcal{D} \cup def(\varphi \odot_2 \psi)}(\varphi) \ \wedge \ valid_{\mathcal{D} \cup def(\varphi \odot_2 \psi)}(\psi)
\end{aligned}$$

For our implementation in *J-LO*, we inline the definitions and so derive reductions as the following:

$$\begin{aligned}
valid_{\mathcal{D}}(\mathbf{X}\ \varphi) &:= valid_{\mathcal{D} \cup def(\mathbf{X}\ \varphi)}(\varphi) \\
&= valid_{\mathcal{D}} \\
valid_{\mathcal{D}}(\neg p) &:= valid_{\mathcal{D} \cup def(\neg p)}(\varphi) \\
&= valid_{\mathcal{D} \cup \{\boldsymbol{v}_{\chi_p}\}}
\end{aligned}$$

... and so forth. This improves the efficiency of the checking algorithm.

Example 4 (Checking of invalid formula). Let us consider the following formula, in which q queries parameter x:

$$\varphi(x) := p(x) \vee \underline{q(x)}$$

Then we obtain:

$$\begin{aligned}
valid(\varphi) &= valid_{def(\varphi)}(\varphi) = valid_{\emptyset}(\varphi) \\
&= valid_{\emptyset}(p(x) \vee \underline{q(x)}) \\
&= use(\varphi) \subseteq \emptyset \ \wedge \ valid_{\emptyset \cup def(\varphi)}(p(x)) \ \wedge \ valid_{\emptyset \cup def(\varphi)}(\underline{q(x)}) \\
&= \{\underline{x}\} \subseteq \emptyset \ \wedge \ valid_{\emptyset}(p(x)) \ \wedge \ valid_{\emptyset}(\underline{q(x)}) \\
&= \mathbf{false}
\end{aligned}$$

Theorem 2 (Correctness of function *valid*). *For any formula $\varphi \in LTL_{NN}$ it holds that:*

$$valid(\varphi) \Rightarrow \textit{any variable in } \varphi \textit{ is defined before it is used}$$

Proof (Correctness of function valid).

Let $\varphi \in LTL_{NN}$. Assume $valid(\varphi) = valid_{\mathcal{D}}(\varphi) = \mathbf{true}$ with $\mathcal{D} = def(\varphi)$. We perform a proof by structural induction and distinguish the following cases:

1. $\varphi = p$ for some $p \in \mathcal{P}$. Since $valid(p) = \mathbf{true}$, we know that $\underline{\boldsymbol{v}}_{\chi_p} \subseteq \mathcal{D}$. Also we know by the definition of *def* that $\mathcal{D}$ contains only variables which are defined on this or previous temporal layers, because later temporal layers (which are explicitly or implicitly guarded by $\mathbf{X}$) do not contribute to the function *def*. Hence, any variable in p is defined before it is used.

2. $\varphi = \odot_1 \varphi'$ for some $\varphi' \in LTL_{NN}$. Since $valid(\varphi) = \mathbf{true}$, it must also hold that $valid_{\mathcal{D} \cup def(\varphi)}(\varphi') = \mathbf{true}$. So by induction hypothesis φ' defines all variables before they are used. Since the move from φ' to φ introduces no new variables, the same holds for φ.
3. $\varphi = \varphi' \odot_2 \psi'$ for some $\varphi', \psi' \in LTL_{NN}$. This case can be handled as above.

We wish to note that the inverse of Theorem 2 does not necessarily hold:

$$\text{any variable in } \varphi \text{ is defined before it is used} \not\Rightarrow valid(\varphi)$$

In other words, our checking algorithm is conservative; the algorithm may reject formulas that are actually meaningful. However, this incompleteness is restricted to formulas that use the X operator in an unusual way, such as the following:

$$X(p(x)) \wedge X(\underline{q(x)})$$

This formula would not pass our *valid*-check because the two predicates occur in different environments (induced by the two X operators). Here, the X operator was pushed down into both branches of the conjunction. We argue that programmers would not usually write such formulas but would instead write the equivalent formula

$$X(p(x) \wedge \underline{q(x)})$$

which passes checking. Another solution would be to re-write the input formula before the checking, according to the rule above.

6 Implementation in *J-LO*

We have implemented the proposed static checks in our *J-LO* runtime verification tool.[2] *J-LO* first parses the input formula and then converts it into negational normal form. The formula is then checked using the function *valid*, as previously described.

Without the checking algorithm, our implementation could potentially yield odd runtime errors or even invalid results, as we will now demonstrate using a concrete example. In previous work [6] we have already reported that we used *J-LO* to check for violations of the so-called lock order reversal pattern: we would like to assert through an LTL formula that if two locks are taken in a specific order (with no unlocking in between), the system should warn the user if he also uses these locks in swapped order because in concurrent programs this would mean that two threads could deadlock when their execution is scheduled in an unfortunate order. The following formula expresses this pattern in LTL:

$$\begin{aligned} &\neg lock(t_1, l_2)\ \mathbf{U}\ (\underline{l_1 \neq l_2} \wedge lock(t_1, l_1) \wedge (\neg unlock(t_1, l_1)\ \mathbf{U}\ lock(t_1, l_2))) \\ \rightarrow\ &\mathbf{G} \neg (\underline{t_1 \neq t_2} \wedge \neg lock(t_2, l_1)\ \mathbf{U}\ (lock(t_2, l_2) \wedge (\neg unlock(t_2, l_2)\ \mathbf{U}\ lock(t_2, l_1)))) \end{aligned}$$

Note that, in this formula we use two query predicates (underlined) to make sure that the locks and threads are indeed distinct. *J-LO*'s formulas are based on pointcut expressions taken from the aspect-oriented programming language AspectJ [8]. Hence, in *J-LO*'s concrete syntax, the user would express an inequality $\underline{l_1 \neq l_2}$ through an AspectJ pointcut `if(l1!=l2)`. To defer evaluation of such expressions, *J-LO* parses the given LTL formula, extracting all `if`-pointcuts into closures. The pointcut above would be extracted into the following closure function:

[2] *J-LO* website: `http://www.sable.mcgill.ca/~ebodde/rv/JLO/`

```
boolean eval(Map<String,Object> binding) {
        Object l1= binding.get("l1");
        Object l2= binding.get("l2");
        return l1!=l2;
}
```

Here it can quite obviously be seen that the evaluation of this function would yield invalid results if either `l1` or `l2` were undefined, as in those cases the `get`-method would return `null` for the respective binding. Our checking algorithm ensures that the `get`-method can never return `null` for any closure. We have tested the algorithm on several realistic examples, and it typically completes checking of the formula in just a few milliseconds.

7 Related Work

7.1 HAWK and EAGLE

HAWK [9] is a programming-oriented extension of the rule-based logic EAGLE [10–12] that is allows for specifications in various temporal logics of different kinds. EAGLE computes the truth values of temporal formulae by calculating a minimal respectively maximal fixed point to the recursive definitions of temporal operators such as given by $\mathbf{F}\phi \equiv \phi \vee \mathbf{X}\ \mathbf{F}\phi$. As such EAGLE is very generic and can be used for virtually any kind of specification logic and programming language of the base program. From a specification EAGLE generates an observer that implements its semantics and is notifies whenever events of interest occur.

HAWK is a logic and tool for runtime verification of *Java* and is built on top of EAGLE. Specifications written in HAWK are ultimately being translated into EAGLE monitors. As in *J-LO*, predicates in HAWK can refer to value previously bound by other predicates. HAWK should hence suffer from the same semantic problems when referring to potentially unbound values. To the best of our knowledge, HAWK does not contain any static checks to prevent programmers from writing such formulas.

7.2 JavaMOP

JavaMOP [4] is a Java instantiation of the "Monitor-Oriented Programming" approach promoted by Roşu and others. The JavaMOP tool accepts formulas in different input formalisms including regular expressions, context free grammars, and past-time and future-time linear temporal logic. All formalisms can use predicates similar to AspectJ pointcuts, binding free variables to concrete program values, typically objects. Crucially, though, JavaMOP does not allow predicates to directly refer to variable bindings. (JavaMOP allows programmers to associate imperative Java-code actions with any event, however, those actions are not part of the actual monitoring formalism.) On the one hand that makes the formalism less expressive than it could be otherwise, but on the other hand, this prevents the problems described in this paper: a predicate can never query an undefined variable binding, as it can query no binding at all.

7.3 Tracematches

Similar to JavaMOP, tracematches [5] allow predicates in the form of AspectJ pointcuts, which can bind free variables to concrete values. Pointcuts in a tracematch cannot refer to variable bindings already established, hence also circumventing the

problems described in this paper. Users *can* query variable bindings in the tracematch body. However, this body is only executed once a match is completed. The tracematch implementation uses a data-flow analysis over the tracematch's state machine to assure that all variables will have been bound to a value once a match is completed. In contrast, our check is syntactic and requires no conversion to a finite-state machine.

8 Conclusion

In this work we have described the problem of obtaining an unclear semantics if predicates in temporal formulas query variable bindings that are potentially unbound. We have further proposed a static checking algorithm to detect and reject formulas where such accesses may impact the formula's truth value and hence lead to an unclear semantics. We have described our implementation in the *J-LO* runtime verification tool, showing a concrete example problem that the checking algorithm helps to avoid.

References

1. Chen, F., Meredith, P., Jin, D., Roşu, G.: Efficient formalism-independent monitoring of parametric properties. In: ASE. (2009) 383–394
2. Pnueli, A.: The temporal logic of programs. In: IEEE Symposium on the Foundations of Computer Science (FOCS), IEEE Computer Society (October 1977) 46–57
3. Bodden, E., Hendren, L.J., Lhoták, O.: A staged static program analysis to improve the performance of runtime monitoring. In: ECOOP. Volume 4609 of LNCS., Springer (2007) 525–549
4. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA. (October 2007) 569–588
5. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA. (October 2005) 345–364
6. Stolz, V., Bodden, E.: Temporal Assertions using AspectJ. In: 5th Workshop on Runtime Verification. Volume 144 of Electronic Notes in Theoretical Computer Science., Elsevier (July 2005) 109–124
7. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University (November 2005)
8. AspectJ team: The AspectJ home page, `http://eclipse.org/aspectj/` (2003)
9. d'Amorim, M., Havelund, K.: Event-based runtime verification of java programs. In: WODA '05: Proceedings of the third international workshop on Dynamic analysis, New York, NY, USA, ACM Press (2005) 1–7
10. H. Barringer, A. Goldberg, K.H., Sen, K.: EAGLE does space efficient LTL monitoring. Pre-Print CSPP-25, Department of Computer Science, University of Manchester (October 2003)
11. H. Barringer, A. Goldberg, K.H., Sen, K.: Eagle monitors by collecting facts and generating obligations. Pre-Print CSPP-26, Department of Computer Science, University of Manchester (October 2003)
12. H. Barringer, A. Goldberg, K.H., Sen, K.: Program monitoring with ltl in eagle. In: 18th International Parallel and Distributed Processing Symposium, Parallel and Distributed Systems: Testing and Debugging - PADTAD'04, IEEE Computer Society Press (April 2004) ISBN 0769521320.