University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2007-2908

Synthesizing Monitors for Safety Properties – This Time With Calls and Returns –

Grigore Roşu[1] and Feng Chen[1] and Thomas Ball[2]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
[2] Microsoft Research, Redmond

Abstract. We present an extension of past time LTL with call/return atoms, called ptCaRet, together with a monitor synthesis algorithm for it. ptCaRet includes abstract variants of past temporal operators, which can express properties over traces in which terminated function or procedure executions are abstracted away into a call and a corresponding return. This way, ptCaRet can express safety properties about procedural programs which cannot be expressed using conventional linear temporal logics. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula has. The stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula has: push on begins, pop on ends of procedure executions. An optimized implementation is also discussed and is available to download.

1 Introduction

Havelund and Roşu proposed a monitor synthesis algorithm for past time linear temporal logic (ptLTL) formulae φ [9]. The generated monitors implement the recursive semantics of ptLTL using a dynamic programming technique, and need $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Roşu proposed an improved monitor synthesis algorithm for ptLTL in [12] which, using a divide-and-conquer strategy, generates monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ.

Alur *et al.* gave an extension of linear temporal logic (LTL) with calls and returns (of functions or procedures) [2], called CaRet. Unlike LTL, CaRet allows for matching call/return states in linear traces, thus allowing to express properties of execution traces of programs that are not expressible using plain LTL. In particular, one can express properties on the execution stack of a program, such as "function g is always called from within function f", or properties that are allowed to be temporarily violated, such as "user u never directly accesses the passwords file (but may access it through system procedures)".

We define a past time variant of CaRet, called ptCaRet, show by examples its usefulness in expressing a series of safety properties involving calls of functions/procedures, and then propose a monitor synthesis algorithm for properties expressed as ptCaRet formulae. Motivated by practical reasons, ptCaRet distinguishes call/return states from begin/end states: the former take place in the caller's context, while the latter take place in the callee's. This simple and standard distinction allows more flexibility and elegance in expressing properties, but

requires an additional constraint on execution traces: calls always immediately precede begins, while ends always immediately precede returns.

ptCaRet conservatively extends ptLTL by adding abstract variants of temporal operators, namely "abstract previously" and "abstract since". The semantics of these operators is that of their corresponding core ptLTL operators "previously" and "since", but on the *abstract* trace obtained by collapsing executed functions or procedures into only two states, namely the caller's state at the call of the invoked function or procedure and the caller's state at its corresponding return. In other words, from the point of view of the abstract temporal operators, the intermediate states generated during function executions are invisible. Of course, the standard temporal operators continue to "see" the whole trace.

The monitors generated from ptCaRet formulae using the proposed algorithm have both a monitor state and a monitor stack, so they can be regarded as push-down automata; however, both the monitor states and the data pushed onto stacks are calculated online, on a by-need basis. The monitor state is encoded on as many bits as standard past time operators in the original formula, while the monitor stack pushes/pops as many bits of data as abstract temporal operators in the original formula. If no abstract temporal operators are used in a ptCaRet formula, that is, if the ptCaRet formula is a ptLTL formula, then its generated monitor is identical to that obtained using the technique in [12]. In other words, not only is ptCaRet a conservative extension of ptLTL, but the proposed monitor synthesis algorithm conservatively extends the best known, provably optimal monitor synthesis algorithm for ptLTL.

The proposed ptCaRet monitor synthesis algorithm has been implemented and is available to download and experiment with via a web interface at [3]. The rest of the paper is structured as follows: Section 2 discusses ptCaRet as an extension of ptLTL; Section 3 introduces useful derived operators and shows some examples of ptCaRet specifications. Section 4.2 discusses our monitor synthesis algorithm, including its implementation. Section 5 concludes the paper.

2 ptLTL and ptCaRet

We here recall past time linear temporal logic (ptLTL) and define its extension ptCaRet. For simplicity, we assume only two types of past operators, namely "previously" and "since". Other common or less common temporal operators can be added as derived operators. ptLTL contains only the usual, standard variants of temporal operators, while ptCaRet contains both standard and abstract variants. We follow the usual recursive semantics of past time LTL and adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [9], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring and can easily be accommodated.

Definition 1. *Syntactically,* ptLTL *consists of formulae over the grammar*

$$\varphi ::= \mathit{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \circ\varphi \mid \varphi \mathcal{S} \varphi,$$

where a ranges over a set A of state predicates. *Other common syntactic constructs can be defined as derived operators in a standard way: false is $\neg true$, $\diamond\varphi$ ("eventually in the past") is $true \mathcal{S} \varphi$, $\boxdot\varphi$ ("always in the past") is $\neg(\diamond\neg\varphi)$), etc.*

LTL's models, even for its safety fragment, traditionally are *infinite traces* (see, e.g., [11]), where a trace is a sequence of *states*, where a state is commonly abstracted as a set of atomic predicates in A. According to Lamport [10], a *safety property* is a set of such infinite traces (properties are commonly identified with the sets of traces satisfying them) such that once an execution "violates" it then it can never satisfy it again later. Formally, a set of infinite traces Q is a safety property if and only if for any infinite trace u, if $u \notin Q$ then there is some finite prefix w of u such that $wv \notin Q$ for all infinite traces v. It can be shown that there are as many safety properties as real numbers [12]. Unfortunately, any logical formalism can define syntactically only as many formulae as natural numbers. Thus, any logical formalism can only express a small portion of safety properties. In LTL, a common way to specify safety properties is as "always past" formulae, that is, as formulae of the form $\Box\varphi$ ($\Box$ is "always in the future"), where φ is a formula in ptLTL. There are two problems with identifying the problem of monitoring a ptLTL specification φ with checking the running system against the LTL safety formula $\Box\varphi$: on the one hand, LTL has an infinite trace semantics, while during monitoring we only have a finite number of past states available, and, on the other hand, once the LTL formula $\Box\varphi$ is violated then it can never be satisfied in the future. However, a major use of monitoring is in the context of recoverable systems, in the sense that the monitor can trigger recovery code when φ is violated, in the hope that φ will be satisfied from here on. For these reasons, we adopt a slightly modified semantics of past time LTL, namely the one on finite traces borrowed from [9]:

Definition 2. *A (program)* state *is a set of atomic predicates in A; let s, s', etc., denote states, and let ProgState denote the set of all states. A* trace *is a finite sequence of states in ProgState*; let w, w', etc., denote traces, and ϵ denote the empty trace. If $w \neq \epsilon$, that is, if $w = w's$ for some trace w' and some state s, then we let prefix(w) denote the trace w' and call it the* (concrete) prefix *of w, and let last(w) denote the state s. The satisfaction relation $w \models \varphi$ between a trace w and a* ptLTL *formula φ is defined recursively as follows:*

$w \models true$ *is always true,*
$w \models a$ *iff* $w \neq \epsilon$ *and* $a \in last(w)$,
$w \models \neg\psi$ *iff* $w \not\models \psi$,
$w \models \psi \wedge \psi'$ *iff* $w \models \psi$ *and* $w \models \psi'$,
$w \models \circledcirc\psi$ *iff* $w \neq \epsilon$ *and* $prefix(w) \models \psi$,
$w \models \psi \,\mathcal{S}\, \psi'$ *iff* $w \neq \epsilon$ *and* ($w \models \psi'$ *or* $w \models \psi$ *and* $prefix(w) \models \psi \,\mathcal{S}\, \psi'$).

We next introduce ptCaRet as an extension of ptLTL. Syntactically, it only adds abstract versions of the two temporal operators "previously" and "since" to ptLTL; semantically, some special atomic predicates corresponding to calls, returns, begins and ends of functions/procedures need to be assumed, as well as some natural and practically reasonable restrictions on traces.

Definition 3. ptCaRet *syntactically extends* ptLTL *as follows:*

$$\varphi ::= \cdots \mid \overline{\circledcirc}\varphi \mid \varphi\, \overline{\mathcal{S}}\, \varphi.$$

The former is called "abstract previously" and the latter "abstract since".

The semantics of abstract previously and since are defined exactly as the semantics of their concrete counterparts, but on an abstract version of the trace from which all the intermediate states of the terminated function or procedure executions are erased. In order for this erasure, or abstraction, process to work, we need to impose some constraints on traces that are always satisfied in practice.

Definition 4. *In* ptCaRet*, the set of atomic predicates A contains four special predicates:* call*,* begin*,* end*, and* return*. A state contains at most one of these and is called call, begin, end, or return state if it contains the corresponding predicate.* ptCaRet *traces are constrained to the following restrictions:*

(1) any call state, except when the last one, must be immediately followed by a begin state, and any begin state must be immediately preceded by a call state;

(2) any end state, except when the last one, must be immediately followed by a return state, and any return state must be immediately preceded by an end.

For a trace w as above, we let $\overline{w}$ denote its abstraction*, which is obtained by iteratively erasing contiguous subtraces $s_b w' s_e$ of w in which s_b is a begin state, s_e is an end state which is not the last one in w, and w' contains no begin or end states. One more restriction is imposed on* ptCaRet *traces:*

(3) the abstractions of ptCaRet *traces contain no return states which are not immediately preceded by call states.*

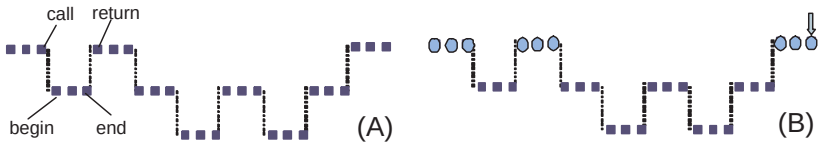

Fig. 1. ptCaRet trace (A) and abstraction (B). ⇓: end of w, ■: w state, ○: $\overline{w}$ state.

Call and return states occur in the caller's context. Thus, call/return states can contain other predicates which may not be possible to evaluate in the callee's context during runtime monitoring. The begin/end states are generated in the callee's context, at the beginning and at the end of the execution of the invoked function, respectively. Similarly, for some common programming languages, begin/end states may contain other predicates that cannot be evaluated in the caller's context. The original CaRet logic [2] did not distinguish between call and begin states or between end and return states. We included all four of them in ptCaRet for the reasons above and also because most trace monitoring systems (e.g., Tracematches [1, 4] and MOP [6, 7]) make a clear distinction between these four types of states.

Fig. 1 (A) shows a ptCaRet trace. To better reflect the call-return structure of the ptCaRet trace, states are placed on different levels: states on the higher level are generated in the caller's context while those on the lower level are generated in the callee's. The vertical dotted lines connect the corresponding call-begin and end-return pairs. Fig. 1 (B) shows the abstraction of that trace: if w ends with the state pointed by ⇓, $\overline{w}$ contains only the circled states.

Restrictions *(1)* and *(2)* on ptCaRet traces are very natural. One source of doubt though can be the sub-requirements that any return state must be preceded by an end state, and that any begin state must be preceded by a call

state. While a return or a begin can indeed happen in any programming language only after a corresponding end or call state, respectively, one may argue that monitoring of a property should be allowed to start at any moment, in particular in between call and begin, or in between end and return states. While our synthesized monitors from ptCaRet formulae (see Section 4.2) can be easily adapted to start monitoring at any moment in the trace, for the sake of a smoother and simpler development of the theoretical foundations of ptCaRet, we assume that any ptCaRet trace starts from the beginning of the program execution and thus satisfies the above-mentioned restrictions. Restriction *(3)* ensures that a trace does not contain return states that do not have corresponding matching call states, also a natural restriction on complete traces.

Our definition of trace abstraction above is admittedly operational, but we think that it captures the desired end/begin matching concept both compactly and intuitively. Alternatively, we could have followed the CaRet style in [2] and define the matching begin state of an end state as the latest begin state containing a balanced number of begin/end states in between.

Definition 5. *For a non-empty* ptCaRet *trace w, let $\overline{prefix}(w)$, called the* abstract prefix *of w (not to be confused with the abstraction of the prefix of w, $\overline{prefix(w)}$), be either $prefix(w)$ if $last(w)$ is not a return state, or otherwise the prefix of w up to and including the corresponding matching call state of $last(w)$ if it is a return state; formally, if $last(w)$ is a return state then $\overline{prefix}(w)$ is the trace $w's_c$, where $w = w'w''$ for some w'' with $\overline{w''} = s_c s_r$, where s_c and s_r are call and return states, respectively.*

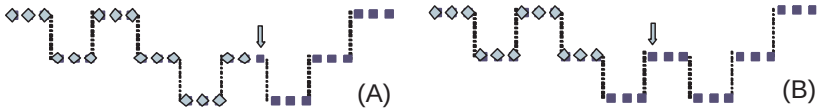

Fig. 2. $\overline{prefix}(w)$ on two traces, (A) and (B). $\Downarrow$: the end of w, $\diamond$: state in $\overline{prefix}(w)$.

Fig. 2 illustrates $\overline{prefix}(w)$ on two traces, with the down arrow pointing to the ends of the traces. In Fig. 2 (A) we assume that w ends with a state that is not a return (the arrow points to a call state) and in Fig. 2 (B) w ends with a return state (the states of the corresponding $\overline{prefix}(w)$ are marked with diamonds).

Definition 6. *The satisfaction relation between a* ptCaRet *trace w and a* ptCaRet *formula φ is defined recursively exactly like in* ptLTL *for the* ptLTL *operators, and as follows for the two abstract temporal operators:*

$w \models \overline{\circ}\psi$ *iff* $w \neq \epsilon$ *and* $\overline{prefix}(w) \models \psi$,

$w \models \psi \overline{\mathcal{S}}\, \psi'$ *iff* $w \neq \epsilon$ *and* $(w \models \psi'$ *or* $w \models \psi$ *and* $\overline{prefix}(w) \models \psi \overline{\mathcal{S}}\, \psi')$.

Therefore, a formula $\overline{\circ}\psi$ is satisfied in a return state iff ψ was satisfied at the corresponding matching call state. It is satisfied in a non-return state, including an end state, iff $\circ\psi$ is satisfied in that state (that is, if and only if ψ was satisfied in the concrete (non-abstract) previous state).

Fig. 3 compares the $\circ$ and $\overline{\circ}$ operators. The arrows point, for each state, where the formula ψ in $\circ\psi$ (A) and in $\overline{\circ}\psi$ (B) holds. For most states, their abstract

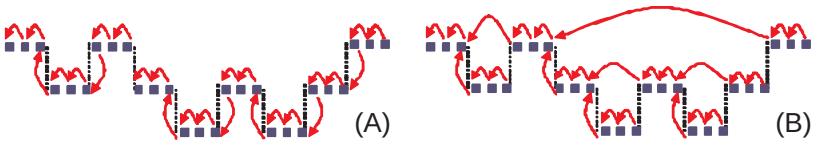

Fig. 3. Concrete (A) and abstract (B) "previous" states for $\circ$ and $\overline{\circ}$.

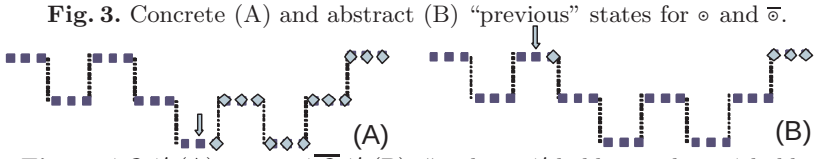

Fig. 4. $\psi \mathcal{S} \psi'$ (A) versus $\psi \overline{\mathcal{S}} \psi'$ (B). $\Downarrow$: where ψ' holds, $\diamond$: where ψ holds.

previous state is the concrete previous one; the only difference is on return states, because the abstract previous state of a return state is its call state.

Figure 4 compares $\psi \mathcal{S} \psi'$ and $\psi \overline{\mathcal{S}} \psi'$. Notice that the various call/return levels play no role in the satisfaction of $\psi \mathcal{S} \psi'$, but that they play a crucial role in the satisfaction of $\psi \overline{\mathcal{S}} \psi'$: for the latter, ψ' must hold on the same level or a higher level as the level of the current state. One can show the following expected property of abstract since:

Proposition 1. *$\varphi_1 \overline{\mathcal{S}} \varphi_2$ is semantically equivalent to $\varphi_2 \vee \varphi_1 \wedge \overline{\circ}(\varphi_1 \overline{\mathcal{S}} \varphi_2)$.*

One should not get tricked and assume that $w \models \overline{\circ}\varphi$ if and only if $\overline{w} \models \circ\varphi$, or that $w \models \varphi_1 \overline{\mathcal{S}} \varphi_2$ if and only if $\overline{w} \models \varphi_1 \mathcal{S} \varphi_2$! The reason is that subformulae φ, φ_1 or φ_2 may contain concrete temporal operators whose semantics still involve the entire execution trace, not only the abstract one. Some examples in this category are shown in Section 3. Nevertheless, the following holds:

Proposition 2. *For a* ptCaRet *trace w and formula φ containing no concrete temporal operators $\circ$ and $\mathcal{S}$, $w \models \varphi$ iff $\overline{w} \models \hat{\varphi}$, where $\hat{\varphi}$ is the* ptLTL *formula replacing each abstract temporal operator in φ by its concrete variant.*

3 ptCaRet Derived Operators and Examples

Besides the usual derived Boolean operators and past time temporal operators "eventually in the past", "always in the past", as well as "start", "stop", and "interval" operators like in [9], which can all be also defined abstract variants, we can define several other interesting, ptCaRet-specific derived operators. In the rest of the paper we use the standard notation for the derived Boolean operators, e.g., "$\rightarrow$", "$\vee$", etc., with their usual precedences, and assume that "$\circ$" binds as tight as "$\neg$" while "$\mathcal{S}$" binds tighter than the binary Boolean operators.

At beginning. Suppose that one would like a particular property, say ψ, to hold at the beginning of the execution of the current function. We can define the derived temporal operator $@_b$, say "at beginning", as follows:

$$@_b\psi \stackrel{\text{def}}{=} (\mathsf{begin} \rightarrow \psi) \wedge (\neg\mathsf{begin} \rightarrow \circ(\mathsf{begin} \rightarrow \psi)\overline{\mathcal{S}}\,\mathsf{begin}).$$

Note that the concrete "previously" operator is used inside the argument of the "abstract since" operator. The above is correct because the last begin state seen by the "abstract since" is indeed the beginning of the current function or procedure. One should not get tricked and try to define the above as:

$$@_b\psi \stackrel{\text{def}}{=} (\mathsf{begin} \rightarrow \psi) \wedge (\neg\mathsf{begin} \rightarrow (\mathsf{begin} \rightarrow \psi)\overline{\mathcal{S}}\,\mathsf{call}).$$

Fig. 5. Derived operators. ⇓: current state, ⋄: states for $@_b$, $\mathcal{S}_b$; ○: states for $@_c, \mathcal{S}_c$

That is because the current function may have called and returned from several other functions, and the "abstract since" can still see all the call/return states. The above would vacuously hold in such a case.

At call. Suppose now that one wants ψ to hold at the state when the current function was called. For the same reason as above, one cannot simply replace begin by call in the definition of $@_b$ above. However, one can define the derived temporal operator $@_c$, say "at call", in terms of "at beginning" simply as follows:

$$@_c\psi \stackrel{\text{def}}{=} @_b \circ \psi.$$

In Fig. 5 (A), supposing that the current state is the one pointed to by the arrow, ψ should hold in the diamond state for $@_b\psi$ and in the circle state for $@_c\psi$.

Stack since on beginnings. The "abstract since" can be used to write properties in which the terminated function executions are irrelevant. There may be cases in which one wants to write properties referring exclusively to the execution stack of a program, ignoring any other states. For example, one may want to say that ψ held on the stack since property ψ' held. As usual, one may be interested in properties ψ and ψ' to hold either at call time, or at execution beginning time. Let us first define a "stack since on beginnings" derived operator:

$$\psi \overline{\mathcal{S}}_b \psi' \stackrel{\text{def}}{=} (\mathsf{begin} \rightarrow \psi) \overline{\mathcal{S}} (\mathsf{begin} \wedge \psi').$$

Stack since on calls. To define a "stack since on calls" one cannot simply replace begin by call in the above. Instead, one can define it as follows:

$$\varphi_1 \overline{\mathcal{S}}_c \varphi_2 \stackrel{\text{def}}{=} (\mathsf{call} \rightarrow \varphi_1) \overline{\mathcal{S}} (\mathsf{begin} \wedge \circ\varphi_2).$$

In Fig. 5 (B), if the current state is the one pointed by the arrow, the begin stack consists of the diamonds and the call stack consists of the circles.

With the stack since derived temporal operators above, one can further define other derived operators, such as "stack eventually in the past on calls" (say $\overline{\diamond_c}$), "stack always in the past on beginnings" (say $\overline{\Box_b}$), etc.

Let us next further illustrate the strength of ptCaRet by specifying some concrete properties that would be hard or impossible to specify in ptLTL.

Suppose that in a particular context, function f must be called only directly by function g. Assuming *call_f* and *call_g* are predicates that hold when f and g are called, respectively, we can specify this property in ptCaRet as follows:

$$\mathit{call\_f} \rightarrow @_c \mathit{call\_g}.$$

Suppose now that f can be called only directly or indirectly by g: a call to g must be on the stack whenever f is called. We can specify that as follows:

$$\mathit{call\_f} \rightarrow \overline{\diamond_c} \mathit{call\_g}.$$

A common safety property in many systems is that resources acquired during a function execution must be released before the function ends. Assuming that *acquire* and *release* are predicates that hold when the resource of interest is acquired or released, respectively, we can specify this property as follows:

$$\mathsf{end} \rightarrow (\neg \mathit{acquire} \overline{\mathcal{S}} \mathsf{begin} \vee \neg(\neg \mathit{release} \overline{\mathcal{S}} \mathit{acquire})).$$

A more complex example is discussed in Section 4.3.

4 A Monitor Synthesis Algorithm for ptCaRet

As discussed in [12] for ptLTL, thanks to the recursive nature of the satisfaction relation on the standard ptLTL temporal operators (see Definition 2), the monitor generated from a ptCaRet formula needs only one global bit per standard (non-abstract) temporal operator. This bit maintains the satisfaction status of the subformula corresponding to that standard temporal operator; when a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 2 and the bit is updated. In order for this to work, one needs to have already updated or have an easy way to calculate the status of the subformulae.

The situation is more complex for the abstract temporal operators, as one needs to store enough information about the past so that one is able to update the status of abstract operators' satisfaction regardless of how the future evolves. The main complication comes from the fact that one needs to "freeze" the satisfaction status of the subformulae corresponding to abstract temporal operators whenever a begin state is observed, and then "unfreeze" it when the corresponding end state is observed, thus recovering the information that was available right when the function call took place. Fortunately, that can be obtained by using a stack to push/pop the satisfaction status of the abstract temporal subformulae.

More precisely, a stack bit is needed per abstract temporal operator in the ptCaRet formula, maintaining the satisfaction status of the subformula corresponding to that abstract operator. When a new state is observed, the satisfaction status of that subformula is recalculated according to the recursive semantics in Definition 5 and the stack bit updated; if the newly observed state is a begin, then the status of the stack bits is pushed on the stack *before* the actual monitor state update; if the newly observed state is an end, then the status of the stack bits is popped from the stack *after* the monitor state update.

4.1 The Target Language

To state and prove the correctness of any program generation algorithm, one needs to have a formal semantics of the target language. This section gives a formal syntax and semantics to the simple and generic language in which we synthesize monitors. One can very easily translate this language into standard languages, such as C, C++, C#, Java, or even into native machine code. For each ptCaRet formula φ, we are going to generate (in Section 4.2) a monitor $\mathcal{M}_\varphi$ as a statement in a language $\mathcal{L}_\varphi$. The only difference between the languages $\mathcal{L}_\varphi$ is the set of variables that one can assign values to; the rest of the language constructs are the same for all φ. The language $\mathcal{L}_\varphi$ has the following simple syntax (note that $\mathcal{L}_{\varphi_1} \subseteq \mathcal{L}_{\varphi_2}$ whenever φ_1 is a subformula of φ_2):

$$
\begin{array}{rl}
\mathit{Var} ::= & \alpha_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \circ \text{ or } \mathcal{S}) \\
\mid & \beta_\phi \text{ (one for each subformula } \phi \text{ of } \varphi \text{ rooted in } \overline{\circ} \text{ or } \overline{\mathcal{S}}) \\
\mathit{Exp} ::= & \mathit{true} \mid A \mid \mathit{Var} \mid \neg\, \mathit{Exp} \mid \mathit{Exp} \wedge \mathit{Exp} \\
\mathit{Stm} ::= & \mathit{Var} := \mathit{Exp} \mid \mathsf{if\ begin\ then\ push} \mid \mathsf{if\ end\ then\ pop} \mid \mathsf{output}(\mathit{Exp}) \mid \mathit{Stm}\, \mathit{Stm}
\end{array}
$$

Therefore, programs in $\mathcal{L}_\varphi$ can use predicates in A (the atomic predicate set of ptCaRet) as ordinary (Boolean) expressions, together with Boolean variables α_ϕ and β_ϕ, one per standard and abstract temporal operator in φ, respectively, and together with Boolean constructs such as complement and conjunction. Statements can be composed using juxtaposition, and can be: α_ϕ or β_ϕ variable assignment, output of a Boolean expression, or conditional push/pop, the latter pushing or popping, by convention, precisely the bit vector $\boldsymbol{\beta}$. We assume a (rather conventional) denotational semantics for $\mathcal{L}_\varphi$ as follows:

Definition 7. *If φ has k_1 standard temporal operators and k_2 abstract temporal operators, then let $MonState_\varphi$ (we think of $\mathcal{L}_\varphi$ programs as monitors) be the state space of $\mathcal{L}_\varphi$, that is, the domain $Bool^{k_1} \times Bool^{k_2} \times Stack \times Output$, where $Bool$ is the set $\{true, false\}$, $Stack$ is the domain $(Bool^{k_2})^*$ of stacks, or lists, over bit vectors of size k_2, and $Output$ is the domain $Bool^*$ of bit lists. Let the functions*

$$[\![\_]\!] : Exp \to MonState_\varphi \to ProgState \to Bool$$
$$[\![\_]\!] : Stm \to MonState_\varphi \to ProgState \to MonState_\varphi$$

be defined as follows:

$$[\![true]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = true, \quad [\![a]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = s(a),$$
$$[\![\alpha_\phi]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = \boldsymbol{\alpha}(i), \text{ where } i \leq k_1 \text{ is the } \boldsymbol{\alpha}\text{-index corresponding to } \phi,$$
$$[\![\beta_\phi]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = \boldsymbol{\beta}(j), \text{ where } j \leq k_2 \text{ is the } \boldsymbol{\beta}\text{-index corresponding to } \phi,$$
$$[\![b_1 \wedge b_2]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = [\![b_1]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) \text{ and } [\![b_2]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s),$$
$$[\![\alpha_\phi := b]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = (\boldsymbol{\alpha}[\boldsymbol{\alpha}(i) \leftarrow [\![b]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s)], \boldsymbol{\beta}, \sigma, \omega),$$
$$[\![\beta_\phi := b]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = (\boldsymbol{\alpha}, \boldsymbol{\beta}[\boldsymbol{\beta}(j) \leftarrow [\![b]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s)], \sigma, \omega),$$
$$[\![\textsf{if begin then push}]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = \begin{cases} (\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\beta} \cdot \sigma, \omega) \text{ if } s(\textsf{begin}), \\ (\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega) \text{ otherwise}, \end{cases}$$
$$[\![\textsf{if end then pop}]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = \begin{cases} (\boldsymbol{\alpha}, \boldsymbol{\beta}', \sigma', \omega) \text{ if } s(\textsf{end}) \text{ and } \sigma = \boldsymbol{\beta}' \cdot \sigma', \\ (\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega) \text{ otherwise}, \end{cases}$$
$$[\![\textsf{output}(b)]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = (\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega \cdot [\![b]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)),$$
$$[\![stm\ stm']\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = [\![stm']\!]([\![stm]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s)).$$

We can now associate a function $[\![\mathcal{M}_\varphi]\!] : MonState_\varphi \to ProgState \to MonState_\varphi$ to each program $\mathcal{M}_\varphi$ in $\mathcal{L}_\varphi$. For a monitor state $(\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega) \in MonState_\varphi$ and a program state $s \in ProgState$, $[\![\mathcal{M}_\varphi]\!](\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)(s) = (\boldsymbol{\alpha}', \boldsymbol{\beta}', \sigma', \omega')$ if and only if the monitor $\mathcal{M}_\varphi$ executed in state $(\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega)$ when program state s is observed, produces monitor state $(\boldsymbol{\alpha}', \boldsymbol{\beta}', \sigma', \omega')$.

Definition 8. *By abuse of notation, we also let $[\![\mathcal{M}_\varphi]\!] : ProgState^* \to MonState_\varphi$ be the function ($false^{k_1}$ is the vector of k_1 false bits, and ϵ is the empty list):*

$$\begin{cases} [\![\mathcal{M}_\varphi]\!](\epsilon) = (false^{k_1}, false^{k_2}, \epsilon, \epsilon) \quad \text{— the "initial" monitor state —} \\ [\![\mathcal{M}_\varphi]\!](ws) = [\![\mathcal{M}_\varphi]\!]([\![\mathcal{M}_\varphi]\!](w))(s) \end{cases}$$

4.2 The Monitor Synthesis Algorithm

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as in [9], because different applications may choose different implementation paradigms. For example, our implementation of the ptCaRet logic plugin in the context of the context of the MOP

INPUT: A ptCaRet formula φ
OUTPUT: Code that monitors φ

Step 1 Allocate a bit α_ϕ, initially *false*, for each subformula ϕ of φ rooted in a standard temporal operator. The intuition for this bit is as follows:
- if $\phi = \circ\psi$ then α_ϕ says if ψ (no typo!) was satisfied at the previous state;
- if $\phi = \psi \,\mathcal{S}\, \psi'$ then α_ϕ says if ϕ was satisfied at the previous state.

Step 2 Allocate a bit β_ϕ, initially *false*, for each subformula ϕ of φ rooted in an abstract temporal operator. The intuition for this bit is as follows:
- if $\phi = \overline{\circ}\psi$ then β_ϕ says if ψ was satisfied at the abstract previous state;
- if $\phi = \psi \,\overline{\mathcal{S}}\, \psi'$, β_ϕ says if ϕ was satisfied at the abstract previous state.

Step 3 Initialize $Code^{\varphi}_{before}$ and $Code^{\varphi}_{after}$ as follows:
- $Code^{\varphi}_{before}$ to the code "`if begin then push`", and
- $Code^{\varphi}_{after}$ to the code "`if end then pop`".

Notation: For subformulae ϕ of φ, let $\overline{\phi}$ be the Boolean expression replacing in ϕ each temporal-operator-rooted subformula ψ which is not a subformula of another temporal-operator-rooted subformula of ϕ, by either α_ψ when ψ is rooted in a standard temporal operator, or by β_ψ when ψ is rooted in an abstract operator. For example, $a \wedge \circ b \,\overline{\mathcal{S}}\, c \wedge \circ(d \,\mathcal{S}\, \overline{\circ} e)$ is $a \wedge \beta_{\circ b \,\overline{\mathcal{S}}\, c} \wedge \alpha_{\circ(d \,\mathcal{S}\, \overline{\circ} e)}$.

Step 4 Following a depth-first-search (DFS) traversal of φ, for each subformula ϕ of φ rooted in a temporal operator do:
- if $\phi = \circ\psi$ then $Code^{\varphi}_{after} \leftarrow (\alpha_\phi := \overline{\psi})\ Code^{\varphi}_{after}$
- if $\phi = \overline{\circ}\psi$ then $Code^{\varphi}_{after} \leftarrow (\beta_\phi := \overline{\psi})\ Code^{\varphi}_{after}$
- if $\phi = \psi \,\mathcal{S}\, \psi'$ then $Code^{\varphi}_{before} \leftarrow Code^{\varphi}_{before}\ (\alpha_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_\phi)$
- if $\phi = \psi \,\overline{\mathcal{S}}\, \psi'$ then $Code^{\varphi}_{before} \leftarrow Code^{\varphi}_{before}\ (\beta_\phi := \overline{\psi'} \vee \overline{\psi} \wedge \beta_\phi)$

Step 5 Output monitor $\mathcal{M}_\varphi$ as the code "$Code^{\varphi}_{before}$ `output(`$\overline{\varphi}$`)` $Code^{\varphi}_{after}$"

Fig. 6. The monitor synthesis algorithm for ptCaRet

system [6, 7], discussed in Section 4.3, uses term rewriting techniques. The monitoring code for a ptCaRet formula φ can be split into three pieces: code to be executed before the monitor outputs the satisfaction status of the formula, the outputting code, and code to be executed after the output. Let $Code^{\varphi}_{before}$ denote the former and let $Code^{\varphi}_{after}$ denote the latter.

$Code^{\varphi}_{before}$ is concerned with updating the status of the "since" operators in a bottom-up fashion, while $Code^{\varphi}_{after}$ with updating the status of the "previously" operators. Indeed, in order to output the satisfaction status of φ, one needs to know the status of all the "since" operators, which may depend upon values in the current state as well as upon values of nested "since" operators, so the inner "since" operators need to be processed before the outer ones. On the other hand, one need not know the particular details (values of atomic predicates) of the current state in order to know the status of the "previously" operators; all one needs to make sure of is that the status of the "previously" operators has been updated at the appropriate previous state (or states in the case of "abstract previously"), after the monitor output. Interestingly, note that, unlike the "since" operators, the "previously" operators need to be processed in a top-down fashion, that is, the outer ones need to be processed before the inner ones.

Note that the monitors $\mathcal{M}_\varphi$ generated in Figure 6 are well-defined, in the sense that each time a generated Boolean expression $\overline{\psi}$ is executed, all the α and β bits that are needed have been calculated. That is because the code is generated following a DFS traversal of the original ptCaRet formula. $\mathcal{M}_\varphi$ is run at each newly generated event, or program state, and outputs either *true* or *false*. Note that each $\mathcal{M}_\varphi$ has the form "(if begin then push) C_1^φ output(O^φ) C_2^φ (if end then pop)", for some potential statements C_1^φ and C_2^φ, and for some Boolean expression O^φ. To simplify notation, we introduce the following:

Definition 9. *Let $\langle C_1^\varphi, O^\varphi, C_2^\varphi\rangle$ be a shorthand for (we use $\emptyset$ for C_1^φ or C_2^φ when they do not exist): "(if begin then push) C_1^φ output(O^φ) C_2^φ (if end then pop)".*

The following result structurally relates monitors generated for formulae φ to monitors generated for its subformulae. One can use this proposition as an equivalent, recursive way to synthesize monitors for ptCaRet:

Proposition 3. *If $\mathcal{M}_\psi = \langle C_1^\psi, O^\psi, C_2^\psi\rangle$ and $\mathcal{M}_{\psi'} = \langle C_1^{\psi'}, O^{\psi'}, C_2^{\psi'}\rangle$ then:*
- $\mathcal{M}_{true} = \langle \emptyset, true, \emptyset\rangle$
- $\mathcal{M}_a = \langle \emptyset, a, \emptyset\rangle$
- $\mathcal{M}_{\neg\psi} = \langle C_1^\psi, \neg O^\psi, C_2^\psi\rangle$
- $\mathcal{M}_{\psi\wedge\psi'} = \langle C_1^\psi\ C_1^{\psi'},\ O^\psi \wedge O^{\psi'},\ C_2^{\psi'}\ C_2^\psi\rangle$
- $\mathcal{M}_{\odot\psi} = \langle C_1^\psi,\ \alpha_{\odot\psi},\ (\alpha_{\odot\psi} := \overline{\psi})\ C_2^\psi\rangle$
- $\mathcal{M}_{\psi\,\mathcal{S}\,\psi'} = \langle C_1^\psi\ C_1^{\psi'}\ (\alpha_{\psi\,\mathcal{S}\,\psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \alpha_{\psi\,\mathcal{S}\,\psi'}),\ \alpha_{\psi\,\mathcal{S}\,\psi'},\ C_2^{\psi'}\ C_2^\psi\rangle$
- $\mathcal{M}_{\overline{\odot}\psi} = \langle C_1^\psi,\ \beta_{\overline{\odot}\psi},\ (\beta_{\overline{\odot}\psi} := \overline{\psi})\ C_2^\psi\rangle$
- $\mathcal{M}_{\psi\,\overline{\mathcal{S}}\,\psi'} = \langle C_1^\psi\ C_1^{\psi'}\ (\beta_{\psi\,\overline{\mathcal{S}}\,\psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi\,\overline{\mathcal{S}}\,\psi'}),\ \beta_{\psi\,\overline{\mathcal{S}}\,\psi'},\ C_2^{\psi'}\ C_2^\psi\rangle$

To prove the correctness of our monitor synthesis algorithm, we need to show that after observing any sequence of program states w, a synthesized monitor $\mathcal{M}_\varphi$ outputs the same result as the satisfaction status of $w \models \varphi$. Therefore, we need to define "the output of the monitor $\mathcal{M}_\varphi$ after observing w":

Definition 10. *Let $(\!|\mathcal{M}_\varphi|\!) : ProgState^+ \to Bool$ be defined for each (non-empty) $w \in ProgState^+$ as $(\!|\mathcal{M}_\varphi|\!)(w) = b$ iff $[\![\mathcal{M}_\varphi]\!](w) = (\boldsymbol{\alpha}, \boldsymbol{\beta}, \sigma, \omega \cdot b)$. For uniformity, let us extend $(\!|\mathcal{M}_\varphi|\!)$ to a function $ProgState^* \to Bool$ (as in Definitions 2 and 5):*
- $(\!|\mathcal{M}_\varphi|\!)(\epsilon) = false$ *when* $\varphi = a, \odot\psi, \psi\,\mathcal{S}\,\psi', \overline{\odot}\psi, \psi\,\overline{\mathcal{S}}\,\psi'$;
- $(\!|\mathcal{M}_{\neg\psi}|\!)(\epsilon) = \neg(\!|M_\psi|\!)(\epsilon)$;
- $(\!|\mathcal{M}_{\psi\wedge\psi'}|\!)(\epsilon) = (\!|M_\psi|\!)(\epsilon) \wedge (\!|M_{\psi'}|\!)(\epsilon)$.

Proposition 4. *The following hold for any $w \in ProgState^*$:*
- $(\!|\mathcal{M}_{true}|\!)(w)$ *is always true,*
- $(\!|\mathcal{M}_a|\!)(w)$ *iff* $w \neq \epsilon$ *and* $a \in last(w)$,
- $(\!|\mathcal{M}_{\neg\psi}|\!)(w)$ *iff not* $(\!|\mathcal{M}_\psi|\!)(w)$,
- $(\!|\mathcal{M}_{\psi\wedge\psi'}|\!)(w)$ *iff* $(\!|\mathcal{M}_\psi|\!)(w)$ *and* $(\!|\mathcal{M}_{\psi'}|\!)(w)$,
- $(\!|\mathcal{M}_{\odot\psi}|\!)(w)$ *iff* $w \neq \epsilon$ *and* $(\!|\mathcal{M}_\psi|\!)(prefix(w))$,
- $(\!|\mathcal{M}_{\psi\,\mathcal{S}\,\psi'}|\!)(w)$ *iff* $w \neq \epsilon$ *and* $((\!|\mathcal{M}_{\psi'}|\!)(w)$ *or* $(\!|\mathcal{M}_\psi|\!)(w)$ *and* $(\!|\mathcal{M}_{\psi\,\mathcal{S}\,\psi'}|\!)(prefix(w)))$,
- $(\!|\mathcal{M}_{\overline{\odot}\psi}|\!)(w)$ *iff* $w \neq \epsilon$ *and* $(\!|\mathcal{M}_\psi|\!)(\overline{prefix}(w))$,
- $(\!|\mathcal{M}_{\psi\,\overline{\mathcal{S}}\,\psi'}|\!)(w)$ *iff* $w \neq \epsilon$ *and* $((\!|\mathcal{M}_{\psi'}|\!)(w)$ *or* $(\!|\mathcal{M}_\psi|\!)(w)$ *and* $(\!|\mathcal{M}_{\psi\,\overline{\mathcal{S}}\,\psi'}|\!)(\overline{prefix}(w)))$.

Proof. The non-trivial ones are those for temporal operators. We only discuss $\overline{\mathcal{S}}$, because the others follow the same idea and are simpler. The monitors for $\psi \overline{\mathcal{S}} \psi'$, ψ, and ψ', respectively, following the notations in Proposition 3 are:

	$\mathcal{M}_{\psi \overline{\mathcal{S}} \psi'}$	$\mathcal{M}_{\psi}$	$\mathcal{M}_{\psi'}$
1.	if begin then push	if begin then push	if begin then push
2.	C_1^{ψ} $C_1^{\psi'}$	C_1^{ψ}	$C_2^{\psi'}$
3.	$\beta_{\psi \mathcal{S} \psi'} := \overline{\psi'} \vee \overline{\psi} \wedge \beta_{\psi \mathcal{S} \psi'}$		
4.	output($\beta_{\psi \mathcal{S} \psi'}$)	output($\overline{\psi}$)	output($\overline{\psi'}$)
5.	$C_2^{\psi'}$ C_2^{ψ}	C_2^{ψ}	$C_2^{\psi'}$
6.	if end then pop	if end then pop	if end then pop

Note that the property holds vacuously if $w = \epsilon$. Assume now that $w = w's$, for some $s \in \mathit{ProgState}$. An interesting and useful property of the generated monitors is that their semantics is very modular, and that pushing or popping $\boldsymbol{\beta}$ does not affect the modular semantics. For example, note that C_1^{ψ} in $\mathcal{M}_{\psi \mathcal{S} \psi'}$ uses no variables defined in $C_1^{\psi'}$ or in $C_2^{\psi'}$, and the bit $\beta_{\psi \mathcal{S} \psi'}$ is only defined in line 3. and used in lines 3. and 4. This modularity guarantees that, if we were to output $\overline{\psi}$ or $\overline{\psi'}$ at line 3. or 4. in $\mathcal{M}_{\psi \mathcal{S} \psi'}$, then its output after processing w would be nothing but $(\!|\mathcal{M}_{\psi}|\!)(w)$ or $(\!|\mathcal{M}_{\psi'}|\!)(w)$, respectively. That means that the $\overline{\psi}$ and $\overline{\psi'}$ in the expression assigned to $\beta_{\psi \mathcal{S} \psi'}$ at line 4. when processing the last state in w are $(\!|\mathcal{M}_{\psi}|\!)(w)$ and $(\!|\mathcal{M}_{\psi'}|\!)(w)$, respectively. We claim that $\beta_{\psi \mathcal{S} \psi'}$ in the assigned expression at line 4. is $(\!|\mathcal{M}_{\psi \mathcal{S} \psi'}|\!)(\overline{\mathit{prefix}}(w))$. There are two cases to analyze. (1) if s is not a return state, then $\beta_{\psi \mathcal{S} \psi'}$ was assigned at line 3. in the previous execution of the monitor, when processing the last state in w', so it is nothing but $(\!|\mathcal{M}_{\psi \mathcal{S} \psi'}|\!)(\mathit{prefix}(w))$; and (2) if s is a return state, then it means that the last state in w' was an end state, so the vector $\boldsymbol{\beta}$ was popped from the stack at the end of the previous step. The only thing left to note is that our push on begins and pop or ends correctly match begin and end states; this follows from the fact that we assume traces complete and well-formed (Definition 4).

Theorem 1. *The monitor synthesis algorithm in Figure 6 is correct, that is, for any* ptCaRet *formula φ and for any $w \in \mathit{ProgState}^*$, $(\!|\mathcal{M}_{\varphi}|\!)(w)$ iff $w \models \varphi$.*

Proof. Straightforward, by induction on both the structure of φ and the length of w, noticing that there is a one-to-one correspondence between the definition of satisfaction in Definitions 2 and 5, and the properties in Proposition 4.

4.3 Implementation as Logic Plugin, Optimizations, Example

MOP [6, 7] is a configurable runtime verification framework, in which specification requirements formalisms can be added modularly, by means of *logic plugins*. A logic plugin essentially encapsulates a monitor synthesis algorithm for a formalism that one can then use to specify properties of programs. The current JavaMOP tool has logic plugins for future time LTL, past time LTL, Allen algebra, extended regular expressions, JML, JASS. JavaMOP takes a Java application to be monitored and specifications using any of the included formalisms together with validation and/or violation handlers (saying what to do if property

validated or violated, in particular nothing), and then waves them together in a runtime verified application by first generating monitors for all the properties using their corresponding logic plugins, and then generating and compiling an AspectJ extension of the original program (runtime monitors are "aspects"). To maintain a reduced runtime overhead (shown on large benchmarks to be, on average, below 10%), MOP piggybacks monitor states onto object states.

The* ptCaRet *MOP logic plugin. We implemented the ptCaRet monitor synthesis algorithm in Section 4.2 as an MOP logic plugin. Our implementation can be found and experimented with online at [3]. Large-scale experiments are still to be performed; we are currently engineering the MOP system to allow monitor states to piggyback not only object states, but also the program stack. In short, our implementation uses *term rewriting* and the Maude system [8], and follows the monitor synthesis algorithm in Figure 6 and its "equivalent", recursive formulation in Proposition 3. Implementations in other languages are obviously also possible; however, term rewriting proved to be an elegant means to synthesize monitors from logical formulae in several other contexts (the other MOP plugins, as well as in JPaX [13]), and so seems to be here.

Our implementation starts by defining the Boolean expressions as an algebraic specification using Maude's mixfix notation (equivalent to context-free grammars); derived Boolean operators are also defined, together with several simplification rules ($\neg\ true = false$, etc.). Boolean expressions are imported both in the target language module and in the ptCaRet module. Both the target language and the ptCaRet modules are defined as algebraic signatures, enriched with structural equalities which turn into simplification rules when executed; this way, for example, each ptCaRet derived operator is defined with one equation capturing its definition. Several other derived operators are defined in addition to those discussed in Section 3. The monitor generation module imports both the target language and the ptCaRet modules, and adds two equations per temporal logic operator; e.g., the equations below process the "abstract since":

```
eq form(F1 Sa F2) = [form(F1), form(F2)] -> Sa .
eq k([exp(B1),exp(B2)] -> Sa -> K) code(I,C1,C2) nextBeta(N)
 = k(exp(beta[N]) -> K) code(I beta[N] := false,
                             C1 beta[N] := B2 or B1 and beta[N], C2) nextBeta(N + 1) .
```

First equation says that subformulae should be processed first (DFS traversal). The second equation combines the codes generated from the subformulae as shown in Proposition 3, appending the assignment for the corresponding bit to `C1`. Note that `C1` here accumulates the "code before" of both subformulae; in terms of Proposition 3, it is "$C_1^{\psi}\ C_1^{\psi'}$". `I` accumulates the monitor initialization code. Finally the optimizations below are implemented also as rewrite rules.

Optimizations. Term-rewriting-based code-generation algorithms can be easily extended with optimizations, because these can be captured as rewrite rules. We discuss some of the optimizations enabled in our implementation. First, we perform Boolean simplifications when calculating $\overline{\psi}$ to reduce runtime overhead ($\neg\neg\psi = \psi$, $\mathsf{true} \wedge \psi = \mathsf{true}$, etc.). Another immediate optimization is the following. The generated code originally has the form (see Fig. 6) "(if begin then push)

C (if end then pop)", for some code C. However, since a program state can only contain at most one of the special predicates, this can be optimized into (syntax of target language needs to be slightly extended):

if begin then (push; C[begin $\leftarrow$ *true*, end $\leftarrow$ *false*, call $\leftarrow$ *false*, return $\leftarrow$ *false*]; exit)
if end then (C[begin $\leftarrow$ *false*, end $\leftarrow$ *true*, call $\leftarrow$ *false*, return $\leftarrow$ *false*]; pop; exit)
C[begin $\leftarrow$ *false*, end $\leftarrow$ *false*];

After the substitutions above, further Boolean simplifications may be triggered. Also, some assignments may become redundant, such as, for example, "beta[3] := beta[3]"; rules to eliminate such assignments are also given. A further optimization on the generated code is possible, but we have not implemented it yet: some subformulae can repeat in different parts of the original formula; the current implementation generates monitoring code for each repeating instance, which is redundant and can be reduced using a smarter optimization algorithm.

Example. We here show the monitor generated by our implementation for a more complex ptCaRet specification. Suppose that a program carries out a critical multi-phase task and the following safety properties must hold when execution enters the second phase:

1. Execution entered the first phase within the same procedure;
2. Resource acquired within same procedure since first phase must be released;
3. Caller of current procedure must have had approval for the second phase;
4. Task is executed directly or indirectly by the procedure *safe_exec*.

These can be captured as the following ptCaRet formula:

$$\begin{aligned} \mathit{enter\_phase\_2} \rightarrow\ & (\neg(\neg \mathit{enter\_phase\_1}\, \overline{\mathcal{S}}\, \mathsf{begin}) \\ & \wedge(\neg \mathit{acquire}\, \overline{\mathcal{S}}\, \mathit{enter\_phase\_1} \vee \neg(\neg \mathit{release}\, \overline{\mathcal{S}}\, \mathit{acquire})) \\ & \wedge @_c(\mathit{has\_phase\_2\_pass}) \\ & \wedge \overline{\diamond}_b(\mathit{safe\_exec}) \end{aligned}$$

Our implementation generates the following monitor for this specification:

```
if begin then {push(beta);
  beta[0] := safe_exec or beta[0]; beta[1] := enter_ph1 or not acquire and beta[1];
  beta[2] := acquire or not release and beta[2]; beta[3] := true; beta[4] := true;
  output(not enter_ph2 or not beta[4] and alpha[0] and beta[0] and (not beta[2] or beta[1]));
  alpha[3] := true; alpha[2] := alpha[1]; alpha[1] := has_ph2_pass; alpha[0] := has_ph2_pass;
  exit}
if end then {
  beta[1] := enter_ph1 or not acquire and beta[1]; beta[2] := acquire or not release and beta[2];
  beta[3] := beta[3] and (not alpha[3] or alpha[2]); beta[4] := not enter_ph1 and beta[4];
  output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
  alpha[3] := false; alpha[2] := alpha[1]; alpha[1] := has_ph2_pass; alpha[0] := has_ph2_pass;
  pop(beta); exit}
beta[1] := enter_ph1 or not acquire and beta[1]; beta[2] := acquire or not release and beta[2];
beta[3] := beta[3] and (not alpha[3] or alpha[2]); beta[4] := not enter_ph1 and beta[4];
output(not enter_ph2 or not beta[4] and beta[0] and beta[3] and (not beta[2] or beta[1]));
alpha[3] := false; alpha[2] := alpha[1]; alpha[1] := has_ph2_pass; alpha[0] := has_ph2_pass
```

The formula contains derived operators, e.g., $@_c$, which are first expanded. The monitoring code uses four α bits and five β bits (the expanded formula contains four concrete temporal operators and five abstract ones). For example, $\overline{\diamond}_b(\mathit{safe\_exec})$ is expanded into $(\mathsf{begin} \rightarrow \mathsf{true})\, \overline{\mathcal{S}}\, (\mathsf{begin} \wedge \mathit{safe\_exec})$, which is then simplified to $\mathsf{true}\, \overline{\mathcal{S}}\, (\mathsf{begin} \wedge \mathit{safe\_exec})$, equivalent to $\overline{\diamond}(\mathsf{begin} \wedge \mathit{safe\_exec})$. `beta[0]` in the generated code is used to check this operation; it only needs to be updated at the begin state, where it becomes true if *safe_exec* holds.

5 Conclusion and Future Work

We presented the logic ptCaRet and a monitor synthesis algorithm for it. ptCaRet includes abstract variants of past temporal operators. It can express safety properties about procedural programs which cannot be expressed using conventional ptLTL. The generated monitors contain both a local state and a stack. The local state is encoded on as many bits as concrete temporal operators the original formula had, while the stack pushes/pops bit vectors of size the number of abstract temporal operators the original formula had. An optimized implementation of the monitor synthesis algorithm has been organized as an MOP logic plugin, and is available to download from [3]. There is room for further optimizations of the generated code. An extensive evaluation of the effectiveness of ptCaRet runtime verification on large programs needs to be conducted. On the theoretical side, it would be interesting to explore the relationship between our monitors generated for ptCaRet and the nested word automata in [5]; [5] gives an operational monitoring language for nested words based on BLAST's specification language. In contrast, our language is declarative and an operational encoding synthesized automatically.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA'05*, 2005.
2. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 467–481. Springer, 2004.
3. F. S. L. at UIUC. ptCaRet MOP Logic Plugin. `http://fsl.cs.uiuc.edu/index.php/Special:JavaMOPPTCARETOnline`.
4. P. Avgustinov, J. Tibble, and O. de Moor. Making Trace Monitors Feasible. In *OOPSLA'07*, 2007. to appear.
5. S. Chaudhuri and R. Alur. Instrumenting C programs with nested word monitors. In *14th Workshop on Model Checking Software(SPIN)*, volume 4595 of *Lecture Notes in Computer Science*, pages 279–283. Springer, 2007. Tool paper.
6. F. Chen and G. Roşu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specif. and Implementation. In *RV'03*, volume 89(2) of *ENTCS*, 2003.
7. F. Chen and G. Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, 2007. to appear.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual. `http://maude.cs.uiuc.edu`.
9. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Technology Transfer*, 6(2):158–173, 2004. (also TACAS'02, LNCS 2280).
10. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
11. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
12. G. Roşu. On Safety Properties and Their Monitoring. Technical Report UIUCDCS-R-2007-2850, Dept. of Comp. Sci., Univ. of Illinois at Urbana-Champaign, 2007.
13. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.