A Temporal Logic of Nested Calls and Returns *

Rajeev Alur[1], Kousha Etessami[2], and P. Madhusudan[1]

[1] University of Pennsylvania
[2] University of Edinburgh

Abstract. Model checking of linear temporal logic (LTL) specifications with respect to pushdown systems has been shown to be a useful tool for analysis of programs with potentially recursive procedures. LTL, however, can specify only regular properties, and properties such as correctness of procedures with respect to pre and post conditions, that require matching of calls and returns, are not regular. We introduce a *temporal logic of calls and returns* (CaRet) for specification and algorithmic verification of correctness requirements of structured programs. The formulas of CaRet are interpreted over sequences of propositional valuations tagged with special symbols *call* and *ret*. Besides the standard global temporal modalities, CaRet admits the *abstract-next operator* that allows a path to jump from a call to the *matching* return. This operator can be used to specify a variety of non-regular properties such as partial and total correctness of program blocks with respect to pre and post conditions. The abstract versions of the other temporal modalities can be used to specify regular properties of *local* paths within a procedure that skip over calls to other procedures. CaRet also admits the *caller* modality that jumps to the most recent pending call, and such caller modalities allow specification of a variety of security properties that involve inspection of the call-stack. Even though verifying context-free properties of pushdown systems is undecidable, we show that model checking CaRet formulas against a pushdown model is decidable. We present a tableau construction that reduces our model checking problem to the emptiness problem for a Büchi pushdown system. The complexity of model checking CaRet formulas is the same as that of checking LTL formulas, namely, polynomial in the model and singly exponential in the size of the specification.

1 Introduction

Propositional linear temporal logic (LTL) is a popular choice for specifying correctness requirements of reactive systems [23, 22]. LTL formulas are built from atomic propositions using temporal modalities such as "next," "always," and "until," and are interpreted over infinite sequences of states that assign values to atomic propositions. The LTL model checking problem is to determine whether all the computations of a system satisfy a given LTL specification. In

* Supported in part by ARO URI award DAAD19-01-1-0473 and NSF award CCR-0306382.

traditional model checking [11, 21, 16], the model is a finite state machine whose vertices correspond to system states and whose edges correspond to system transitions. However, model checking is also feasible when the model is a *recursive state machine* (or equivalently, a *pushdown system*), in which vertices can either be ordinary states or can correspond to invocations of other state machines in a potentially recursive manner. Recursive state machines (RSMs) can model the control flow in typical sequential imperative programming languages with recursive procedure calls. Model checking of LTL specifications with respect to RSMs can be solved in time polynomial in the size of the model and exponential in the size of the specification [7, 5, 12, 1, 3, 20]. This problem has been well studied over the last few years leading to efficient implementations and applications to program analysis as well as model checking of C or Java programs [24, 2, 13, 10].

While LTL is an attractive specification language for capturing regular sequencing requirements such as "between successive write operations to a variable, a read operation should occur," it cannot express requirements such as "if the pre-condition p holds when a module is invoked, the post-condition q should hold when the module returns." This requires matching of calls and returns, and is a context-free property if calls are nested (recall that the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is a non-regular context-free language [17]). Correctness of program blocks with respect to *pre* and *post* conditions has been emphasized in the verification literature since the early days of logics for structured programs [15], and also forms an integral part of modern interface specification languages for object oriented programming such as JML [6]. In this paper, we introduce CaRet —a temporal logic that can express requirements about matching calls and returns, along with the necessary tools for algorithmic reasoning. Algorithmic verification of nonregular specifications have been considered previously [4, 14, 19], but to the best of our knowledge, this is the first specification language that allows specification of partial and total correctness with respect to pre and post conditions, and has a decidable model checking problem with respect to boolean abstractions of recursive sequential programs.

The formulas of our logic are interpreted over *structured computations*. A structured computation is an infinite sequence of states, where each state assigns values to atomic propositions, and can be additionally tagged with *call* or *ret* symbols. A call denotes invocation of a (sequential) program module, and the matching return denotes the exit from this module, where a module may correspond to a procedure or a function in structured imperative languages such as C, or methods in object-oriented languages such as Java, or remote invocations of components in a distributed environment. Given a structured computation, the *abstract-successor* of the i-state is defined to be the matching return position if the i-th state is a call, and $i+1$ otherwise. Besides the global temporal modalities, CaRet admits their *abstract* counterparts. For example, $\bigcirc^a \varphi$ holds at a position if φ holds at its abstract-successor position. Consequently, if the state formula $call_A$ denotes the invocation of a module A, then the CaRet formula $\Box(call_A \wedge p \rightarrow \bigcirc^a q)$ specifies the total correctness with respect to the pre-condition p and post-condition q. An abstract path is obtained by applying the abstract-successor operation repeatedly, and captures the *local* computation within a module that removes computation fragments corresponding to calls to other blocks. The abstract-versions of operators such as *always* and *until* can be used to specify regular requirements about such local paths.

In a structured computation, for every position, besides the global and abstract successors, there is also a natural notion of the *caller* position which gives the most recent unmatched call position. The caller path, obtained by repeated applications of the *caller* operator can be interpreted as the contents of the call-stack at a position. Our logic admits the *caller* counterparts of all the temporal modalities. These modalities allow specification of properties that require inspection of the call-stack such as "a module A should be invoked only if the module B belongs to the call-stack," or "the number of interrupt-handlers in the call-stack should never exceed 10." It is worth noting that the relevance of stack inspection for specifying correctness of security properties and calling sequences of interrupt handlers has been identified by many researchers, and decision procedures for checking specific properties already exist [18, 10, 13, 9]. In particular, [13] uses LTL on pushdown systems but allows the atomic propositions of the LTL formula to correspond to any regular language evaluated over the call stack. Our logic mixes global, abstract, and temporal modalities allowing integrated specifications, and is more expressive allowing specification of properties such as "variable x remains unchanged after the current call returns."

Given an RSM (or a pushdown system) whose vertices are labeled with atomic propositions, there is a natural way to associate a set of structured computations by tagging invocations and returns with *call* and *ret* symbols. The model checking problem, then, is to check whether all computations of an RSM S satisfy a CaRet specification φ. Note that both S and φ define context-free languages, and problems such as inclusion and emptiness of intersection are undecidable for context-free languages [17]. However, in our setting, the model and the specification are synchronized on the *call* and *ret* symbols, and as a result we show that the model checking problem becomes decidable. Our decision procedure generalizes the tableau-based construction for LTL model checking: given an RSM S and a specification φ, we show how to construct another RSM S_φ with generalized Büchi conditions such that S_φ has an accepting run iff S has a computation that violates φ. The time complexity of the decision procedure is polynomial in the size of S and exponential in the size of φ. This is identical to the time complexity of model checking of LTL specifications with respect to RSMs, and the model complexity is exactly the same as that of the reachability problem for RSMs (cubic in general, and linear if the number of entries denoting the inputs to a module, or the number of exits denoting the outputs of a module, is bounded). As in the case of LTL, model checking of CaRet with respect to RSMs is Exptime-complete, and its model complexity is Ptime-complete.

Related Work: In [4], the authors show a restricted but decidable temporal logic which allows variables to count the number of occurrences of states in intervals which then can be compared using Presburger constraints. The work in [19] deals with model checking regular systems against pushdown treeautomata specifications. Propositional dynamic logic can be augmented with some restricted classes of context-free languages (those accepted by simple-minded PDAs) such that validity is decidable [14]. Stack inspection is considered in [18] where the authors propose a logic that can specify properties of the stack at any point using temporal operators; in [13] a more general way of accessing the stack is provided using propositions that are interpreted using automata that run over the stack.

2 Computation Model

2.1 Structured Computations

Let us fix a finite set Γ of symbols. The augmented alphabet of Γ is the alphabet: $\hat{\Gamma} = \Gamma \times \{call, ret, int\}$. The symbol *call* denotes the invocation of a module, *ret* denotes the exit or the return from a module and *int* stands for internal actions of the current module. For $\sigma \in \Gamma$, we call the symbols of the form $(\sigma, call)$, (σ, ret) and (σ, int), *calls*, *returns* and *internal symbols*, respectively.

For an infinite word α and an integer $i \geq 0$, we use α_i to denote the i-th symbol in α and α^i to denote the suffix of α starting at the i-th symbol.

For a word α over $\hat{\Gamma}$, there is a natural notion of a matching between calls and returns: if $\alpha_i = (\sigma, call)$ and $\alpha_j = (\sigma', ret)$, we say that j is the *matching return* for i if j is the return corresponding to the call at i. Formally, we define a more general partial function R_α that maps any $i \in \mathbb{N}$ to the first unmatched return after i: if there is a j' such that j' is greater than i and j' is a return and the number of calls and returns in $\alpha_{i+1} \ldots \alpha_{j'-1}$ are equal, then $R_\alpha(i) = j$, where j is the smallest such j'; else $R_\alpha(i) = \perp$. If α_i is a call, then $R_\alpha(i)$ will be its corresponding return.

In the logic we define we have three notions of successor:

- The global-successor ($succ^g$) is the usual successor function. The global-successor of i in α, denoted $succ^g_\alpha(i)$, is $i + 1$.
- The abstract-successor ($succ^a$) points to the next "local" successor. If the current symbol is a call, then it skips the entire computation within the call and moves to the matching return. Otherwise, it is the global-successor provided the global-successor is not a return. If the global successor is a return, then the abstract-successor is defined as $\perp$. Formally, if α_i is an internal symbol or a return, then: if α_{i+1} is not a return, then $succ^a_\alpha(i)$ is $i + 1$, otherwise it $\perp$. If α_i is a call, then $succ^a_\alpha(i) = R_\alpha(i)$. Note that if α_i is a call that has no matching return, then $succ^a_\alpha(i)$ is $\perp$.
- The caller ($succ^-$) is a "past" modality that points to the innermost call within which the current action is executed. If there is a $j' < i$ such that $\alpha_{j'}$ is a call and $R_\alpha(j') > i$ or $R_\alpha(j') = \perp$, then $succ^-_\alpha(i) = j$, where j is the greatest such j'. Otherwise, $succ^-_\alpha(i) = \perp$.

Notice that we do not demand that calls have matching returns, or even that returns have matching calls. In the models of programs that we consider, it is true that every return has a matching call (but calls to a module may never return). However, since our logic does not require this assumption, we have chosen to present it in this general framework.

2.2 Recursive State Machines

Recursive state machines (RSMs) were introduced in [1], and independently in [3] under a different name, to model the interprocedural control flow in recursive programs. RSMs are expressively equivalent to pushdown systems, but they more tightly capture the control flow graphs of procedural programs and enable us to more directly reason about them. We therefore adopt RSMs as system models in this paper.

Syntax. A *recursive state machine* (RSM) S over a set of propositions AP is a tuple $(M, \{S_m\}_{m \in M}, \mathit{start})$, where M is a finite set of module names, for every $m \in M$, S_m is a *module* $S_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m, \eta_m)$, and $\mathit{start} \subseteq \bigcup_{m \in M} N_m$, is a set of start nodes. Each module S_m consists of the following components:

- A finite nonempty set of *nodes* N_m and a finite set of *boxes* B_m.
- A labeling $Y_m : B_m \to M$ that assigns to every box a module name.
- A nonempty set of *entry* nodes $En_m \subseteq N_m$ and a nonempty set of *exit* nodes $Ex_m \subseteq N_m$.
- Let $\mathit{Calls}_m = \{(b, e) \mid b \in B_m, e \in En_{Y_m(b)}\}$ denote the set of *calls* of m and let $\mathit{Retns}_m = \{(b, x) \mid b \in B_m, x \in Ex_{Y_m(b)}\}$ denote the set of *returns* in m. Then, $\delta_m : N_m \cup \mathit{Retns}_m \to 2^{N_m \cup \mathit{Calls}_m}$ is a *transition function*.
- Let $V_m = (N_m \cup \mathit{Calls}_m \cup \mathit{Retns}_m)$. We refer to V_m as the set of *vertices* of S_m. η_m is a labeling function $\eta_m : V_m \to 2^{AP}$ that associates a set of propositions to each vertex, i.e., to nodes, calls and returns.

We let $V = \bigcup_{m \in M} V_m$ denote the set of all vertices of S and let $\eta : V \to 2^{AP}$ be the extension of all the functions η_m, $m \in M$. Also, let $B = \bigcup_{m \in M} B_m$ and let $Y : B \to M$ denote the function that extends all the functions Y_m $(m \in M)$.

Figure 1 depicts an example RSM in which there are two modules, S_1 and S_2. The calls and returns have been identified explicitly by labeling them with c and r, respectively. Module S_1, for example, has two entries labeled by p and by q, only one exit, labeled x, and two boxes b_1 and b_2. Note that calls and returns can also have other propositions labeling them, and these may differ from the labels of the corresponding entries and exits. For example the call of box $b2$ has label t, and this differs from the label z of the entry to S_2, which box $b2$ maps to. Assume that the entry of S_1 labeled q is the unique start node. In such a case, a sample computation of the RSM, annotated with call and return information, consists of the sequence of labels:
$(\{q\}, \mathit{int})\ (\{d\}, \mathit{int})\ (\{t\}, \mathit{call})\ (\{z\}, \mathit{int})\ (\{t\}, \mathit{int})\ (\{y\}, \mathit{int})\ (\{w\}, \mathit{ret}) \ldots$

Semantics. From an RSM S we define a (infinite) global Kripke structure $K_S = (Q, \mathit{Init}, \kappa, \delta)$. The (global) *states*, denoted Q, are elements $(\gamma, u) \in B^* \times V$ such that either

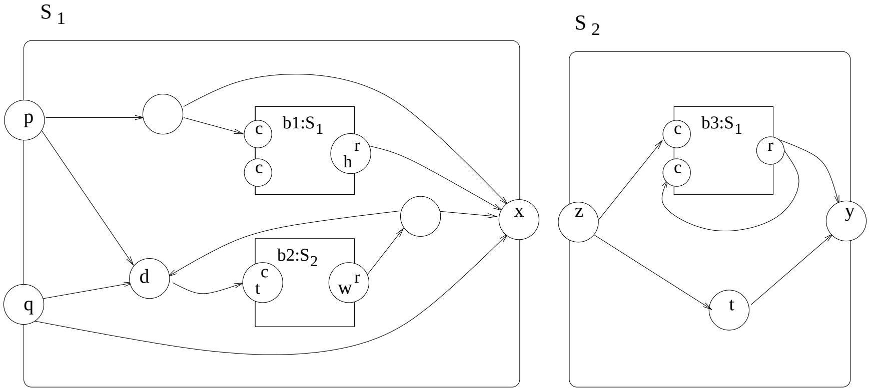

Fig. 1. A sample RSM

– $\gamma = \epsilon$ and $u \in V$, or,
– $\gamma = b_1 \ldots b_k$ (with $k \geq 1$) and $\forall i \in [1, k-1]$, $b_{i+1} \in B_{Y(b_i)}$ and $u \in V_{Y(b_k)}$.

The initial states are $Init = \{(\epsilon, u) \in Q \mid u \in start\}$. The labeling function is $\kappa((\gamma, u)) = (\eta(u), z)$, where $z = int$ if u is a node, $z = call$ if u is a call, and $z = ret$ if u is a return.

The global transition relation $\delta : Q \rightarrow 2^Q$, is defined as follows. For $s = (\gamma, u)$ and $s' = (\gamma', u')$, $s' \in \delta(s)$ if and only if one of the following holds:

– **Internal move:** $u \in (N_m \cup Retns_m) \setminus Ex_m$, $u' \in \delta_m(u)$, and $\gamma' = \gamma$
– **Call a module:** $u = (b, e) \in Calls_m$, $u' = e$ and $\gamma' = \gamma.b$
– **Return from a call:** $u \in Ex_m$, $\gamma = \gamma'.b$, and $u' = (b, u)$

Let $\Gamma = 2^{AP}$ and $\hat{\Gamma} = \Gamma \times \{call, ret, int\}$. For a word $\alpha = \alpha_0\alpha_1 \ldots \in \hat{\Gamma}^\omega$, a *run* of K_S on α is a sequence of states $\pi = s_0, s_1, \ldots$ where $\kappa(s_i) = \alpha_i$, for all $i \in \mathbb{N}$, and such that $s_0 \in Init$ and for every $i \in \mathbb{N}$, $s_{i+1} \in \delta(s_i)$.
For an RSM S, let $\mathcal{L}(S) = \{\alpha \in \hat{\Gamma}^\omega \mid$ there is a run of K_S on $\alpha\}$.

3 Linear Temporal Logic of Calls and Returns

3.1 Syntax and Semantics

Let $\Gamma = 2^{AP}$, where AP is a finite set of *atomic propositions*. Let the augmented alphabet of Γ be $\hat{\Gamma} = \Gamma \times \{call, ret, int\}$. Let P denote the set $AP \cup \{call, ret, int\}$. The models of our logic are the words in $\hat{\Gamma}^\omega$.

The *Propositional Linear Temporal Logic of Calls and Returns* (CaRet) over AP is the set of formulas defined by:

$$\varphi := \ p \mid \varphi \vee \varphi \mid \neg\varphi \mid \bigcirc^g \varphi \mid \varphi \mathcal{U}^g \varphi \mid \bigcirc^a \varphi \mid \varphi \mathcal{U}^a \varphi \mid \bigcirc^- \varphi \mid \varphi \mathcal{U}^- \varphi$$

where $p \in P$.

For a word $\alpha \in \hat{\Gamma}^\omega$, we define the semantics by inductively defining when (α, n) satisfies a formula φ, where $n \in \mathbb{N}$. A word α satisfies φ iff $(\alpha, 0)$ satisfies φ.

For a word α over $\hat{\Gamma}$, $n \in \mathbb{N}$, the semantics is defined as:

- $(\alpha, n) \models p$ iff $\alpha_0 = (X, d)$ and $p \in X$ or $p = d$ (where $p \in P$)
- $(\alpha, n) \models \varphi_1 \vee \varphi_2$ iff $(\alpha, n) \models \varphi_1$ or $(\alpha, n) \models \varphi_2$
- $(\alpha, n) \models \neg\varphi$ iff $(\alpha, n) \not\models \varphi$
- $(\alpha, n) \models \bigcirc^g \varphi$ iff $(\alpha, succ^g_\alpha(n)) \models \varphi$, i.e., iff $(\alpha, n+1) \models \varphi$.
- $(\alpha, n) \models \bigcirc^a \varphi$ iff $succ^a_\alpha(n) \neq \perp$ and $(\alpha, succ^a_\alpha(n)) \models \varphi$.
- $(\alpha, n) \models \bigcirc^- \varphi$ iff $succ^-_\alpha(n) \neq \perp$ and $(\alpha, succ^-_\alpha(n)) \models \varphi$.
- $(\alpha, n) \models \varphi_1 \mathcal{U}^b \varphi_2$ (for any $b \in \{g, a, -\}$) iff there is a sequence of positions $i_0, i_1, \ldots, i_k$, where $i_0 = n$, $(\alpha, i_k) \models \varphi_2$ and for every $0 \leq j \leq k-1$, $i_{j+1} = succ^b_\alpha(i_j)$ and $(\alpha, i_j) \models \varphi_1$.

The operators $\bigcirc^g$ and $\mathcal{U}^g$ are the usual global-next and global-until operators of LTL. The $\bigcirc^a$ and $\mathcal{U}^a$ operators are the abstract versions of the next and until operators— $\bigcirc^a \varphi$ demands that the abstract successor state satisfy φ while $\varphi_1 \mathcal{U}^a \varphi_2$ demands that the abstract path from the current position (i.e. the path formed by successive abstract successors) satisfy $\varphi_1 \mathcal{U} \varphi_2$.

The formula $\bigcirc^- \varphi$ demands that the caller of the current position satisfies φ while $\varphi_1 \mathcal{U}^- \varphi_2$ demands that the backward path of successive caller states (which is always finite) satisfies $\varphi_1 \mathcal{U} \varphi_2$.

As in standard linear temporal logic, we will use $\diamond^b \varphi$ as an abbreviation for $True\, \mathcal{U}^b \varphi$, and $\square^b \varphi$ for $\neg \diamond^b \neg \varphi$, for $b \in \{a, g, -\}$. While writing specifications, we will omit the superscript g as the global scope coincides with the classical interpretation of temporal operators, and we will also use logical connectives such as conjunction and implication freely.

Given an RSM S and a formula φ of CaRet, both over AP, define $S \models \varphi$ to mean that for every $\alpha \in \mathcal{L}(S)$, $\alpha \models \varphi$. We are ready to define the model-checking question we are interested in:

Model-checking problem:

Given an RSM S and a formula φ of CaRet*, does $S \models \varphi$?*

Consider the RSM S, of Figure 1. Assume $start = En_1$, then $S \models \square^g(d \rightarrow \diamond^g z)$ because every global path starting at the vertex labelled d leads to the entry of S_2, which is labeled z (assuming there are no other vertices labeled d). However, $S \not\models \square^g(d \rightarrow \diamond^a z)$ because abstract runs starting in S_1 stay only within S_1 and do not go inside a box to visit the entry of S_2, and hence do not encounter the label z. Also note that $S \models \square^g(y \rightarrow \bigcirc^- t)$ because if we are at the exit of the module S_2, then the last call (which must exist because executions begin in S_1) must have occured from the call of box b_2 (the only box labeled by S_2).

3.2 Specifying Requirements

Pre and Post Conditions: In the classical verification formalisms such as Hoare logic, correctness of procedures is expressed using pre and post conditions [15]. Partial correctness of a procedure A specifies that if the pre-condition p holds when the procedure A is invoked, then if the procedure terminates, the post-condition q is satisfied upon return. Total correctness, in addition, requires the procedure to terminate. Assume that all calls to the procedure A are characterized by the proposition p_A. Then, the requirement

$$\varphi_{total} : \ \Box \, [\, (call \wedge p \wedge p_A) \ \rightarrow \bigcirc^a q \,]$$

expresses the total correctness, while

$$\varphi_{partial} : \ \Box \, [\, (call \wedge p \wedge p_A) \ \rightarrow \neg \bigcirc^a \neg q \,]$$

expresses the partial correctness.

Boundedness: The number of unmatched calls at a position in a word corresponds to the height of the stack at that position. The requirement that "every call must return," or equivalently, "the stack should be empty infinitely often", is expressed by the specification

$$\varphi_{empty} : \ \Box \ (\, call \rightarrow \bigcirc^a ret \,)$$

A weaker requirement is that the stack should be repeatedly bounded, that is, there exists a natural number n such that infinitely often the stack is of height at most n. These kinds of specifications have been studied for pushdown games [8]. This property can be specified in CaRet by the formula:

$$\varphi_{rep\text{-}bounded} : \ \Diamond\Box \ (\, call \rightarrow \bigcirc^a ret \,)$$

Even though this specification does not rule out the possibility that the stack grows unboundedly, an RSM S satisfies the requirement $\varphi_{rep\text{-}bounded}$ iff there exists a natural number n such that the number of pending calls at any position is at most n. The boundedness requirement itself is not expressible in our logic.

Local Properties: The abstract path starting at a node inside a module A is obtained by successive applications of $\bigcirc^a$ operator, and skips over invocations of other modules called from A. CaRet formulas can specify properties of such abstract paths. For example, if the proposition t_A denotes that the control is within a module A, then the formula

$$\varphi_{local\text{-}response} : \ \Box \, [(t_A \ \wedge \ p) \ \rightarrow \ \Diamond^a q \,]$$

specifies the *local* version of the response property "every request p is followed by a response q." In general, any LTL expressible local property can be specified in CaRet.

Stack Inspection Properties: The caller path starting at a node inside a module A is obtained by successive applications of $\bigcirc^-$ operator, and encodes the stack at that position. As shown in [18, 13], stack inspection can specify a variety of security properties. For instance, the requirement that a module A should be invoked only within the context of a module B, with no intervening call to an overriding module C, is expressed by the formula

$$\varphi_{stack} : \ \Box \ (call \wedge p_A \rightarrow (\neg p_C)\, \mathcal{U}^- p_B \,).$$

In general, any property that can be formulated as a star-free regular property of the stack content can be expressed in CaRet. For example, when a critical procedure is invoked, one can require that all the procedures in the call stack have the necessary privilege. We refer the reader to [18, 13] for the relevance of such specifications for capturing security domains, authority delegation, and stack inspection policies in modern programming languages such as Java. Since CaRet can state properties of the stack as well as the global evolution, it can express dynamic security policy constraints, where the permissions change depending upon what privileges have been used thus far (see [18] where such constraints are motivated but cannot be expressed in their logic).

It is worth noting that CaRet allows nesting of different types of modalities. The requirement that a temporal property φ holds when the current module returns is expressed by the formula

$$\varphi_{upon\text{-}return} : \bigcirc^{-} \bigcirc^{a} \varphi$$

This property is not expressible in existing approaches such as augmenting LTL with regular stack valuations [13].

Interrupt-driven sequences: Interrupt-driven software are prevalent in the embedded software domain where the stack-size is very limited. In this setting, while the system is handling an interrupt, the same interrupt could occur, causing the stack to get large. Estimating the maximum stack content is thus a relevant question (see for example [9]). The property that states that "in computations where an interrupt is not interrupted by itself, the formula φ holds" can be expressed in CaRet as:

$$\varphi_{no\text{-}rec\text{-}int} : \Box\,((\mathit{call} \wedge p_{int}) \to \neg \bigcirc^{-} \Diamond^{-} p_{int}) \to \varphi$$

We can also write CaRet formulas that are true only where the stack depth reaches n, for some constant n.

RSM Encoding: Our logic is rich enough to encode the computations of an RSM in a manner similar to the encoding of finite-state machines in LTL. For an RSM S, we write the formula φ_S by introducing a proposition for every vertex of S and ensuring local rules of the evolution of S using the global-next modalities. To ensure that a call at (b, e) returns to some return of the form (b, x), we can assert the following: whenever $p_{(b,e)}$ holds (where $p_{(b,e)}$ is the proposition for (b, e)), either $\neg \bigcirc^{a}$ *true* holds or for precisely one return of the form (b, x), both $\bigcirc^{a} p_{(b,x)}$ and $\bigcirc^{g} \Diamond^{a} p_x$ hold. Then for any S, φ_S is such that for any formula φ, $S \models \varphi$ iff $\varphi_S \to \varphi$ is valid.

It is worth noting that if S and S' are RSMs (or pushdown automata) then $\varphi_S \wedge \varphi_{S'}$ does not represent the intersection of their languages in the usual sense due to the shared *call* and *ret* tags; it represents the synchronized product where the two are required to synchronize on when to call (i.e. push) and when to return (i.e. pop).

4 Model checking

In this section we show how to solve the model checking problem for RSMs against CaRet specifications. We first define the notion of recursive generalized Büchi automata which our decision procedure will use.

4.1 Recursive Generalized Büchi Automata

Our automata-based algorithms will use RSMs augmented with acceptance conditions: both ordinary and generalized Büchi conditions. A *recursive generalized Büchi automaton* (RGBA) $S = (M, \{S_m\}_{m \in M}, start, \mathcal{F})$ consists of an RSM together with a family $\mathcal{F} = \{F_1, \ldots, F_r\}$ of accepting sets of vertices of S where $F_j \subseteq V$, for $j \in \{1, \ldots, r\}$. When there is only one accepting set, $\mathcal{F} = \{F\}$, we have a *Recursive Büchi Automaton* (RBA).

For an RGBA S, the acceptance condition $\mathcal{F} = \{F_1, \ldots, F_r\}$, induces an acceptance condition on the Kripke structure K_S: $\mathcal{F}^\# = \{F_1^\#, \ldots, F_r^\#\}$, where $F_i^\# = \{(\gamma, u) \in Q \mid u \in F_i\}$. We say a run π of K_S is an *accepting* run iff for all $F \in \mathcal{F}$, for infinitely many $i \in \mathbb{N}$, $s_i \in F^\#$. For an RGBA S, let $\mathcal{L}(S) = \{\alpha \in \hat{\Gamma}^\omega \mid$ there is an accepting run of K_S on $\alpha\}$. Note that when there is no acceptance condition, i.e., when $\mathcal{F}$ is the empty set, every run is accepting, and thus such RGBAs correspond to ordinary RSMs.

An important parameter of the size of an RSM S, introduced in [1], is $\theta_S = \max_{m \in M} \min\{|En_m|, |Ex_m|\}$, that is, each module has at most θ_S entries or θ_S exits. It was shown in [1] that reachability analysis for RSMs and language emptiness for RBAs can be performed in time $O(|S|\theta_S^2)$ and space $O(|S|\theta_S)$. This construction can be generalized to obtain the following:

Proposition 1. *Given an RGBA S, with acceptance condition $\mathcal{F} = \{F_1, \ldots, F_r\}$, checking $\mathcal{L}(S) = \emptyset$ can be solved in time $O(r|S|\theta_S^2)$ and space $O(r|S|\theta_S)$.*

To see why the proposition holds, let us recall the algorithm of [1] for the analysis of RBAs with one acceptance condition F. That algorithm proceeds in two phases. In the first phase, for each component S_m, we compute for every entry en and every exit ex of S_m whether there is a path from (ϵ, en) to (ϵ, ex) in the global Kripke structure K_S, and if so, whether there is a path that goes through an accepting state in $F^\#$. This involves solving reachability in an And-Or graph and takes time $O(|S|\theta_S^2)$ and space $O(|S|\theta_S)$. We then augment the RSM with "summary edges" between calls and returns of boxes, to indicate reachability from entries to exits, and we label these summary edges as "accepting edges" if it is possible to get from the entry to the exit via an accepting state. In addition, we also add edges from a call to the corresponding entry of the component that labels the call's box. Once all these edges are added, we are left with an ordinary (flat) Büchi automaton of size $O(|S|\theta_S)$, in which we must detect the existence of a reachable accepting cycle (which we can in linear time).

To generalize this construction to RGBAs with acceptance condition $\mathcal{F} = \{F_1, \ldots, F_r\}$, we first observe that we can do the first phase of "summary edge" calculation separately with respect to every accepting set $F_i \in \mathcal{F}$. We then label each summary edge from a call to a return with the set $C \subseteq \{1, \ldots, r\}$ of "colors" corresponding to those accepting sets which can be visited on some path from the respective entry to the exit. The computation takes time $O(r|S|\theta_S^2)$, and the resulting flat generalized Büchi automaton H has $O(|S|\theta_S)$ edges, but summary edges can each be labeled with $O(r)$ colors. To check that $\mathcal{L}(H) = \emptyset$, we can use a slightly modified version of the standard algorithm for conversion of the generalized Büchi automaton H to a Büchi automaton, to produce a Büchi automaton H' of size $O(r|S|\theta_S)$ that accepts the same language as H. We then run a linear time emptiness test on this Büchi automaton.

4.2 The Decision Procedure

The main construction here will show how to build, for any RSM S and a formula φ over AP, a recursive generalized Büchi automata (RGBA) that accepts exactly the set of words in $\mathcal{L}(S)$ that satisfy φ. For simplifying the proof, we assume without loss of generality that in the RSMs we consider, entries have no incoming transitions, exits have no outgoing transitions and there are no transitions from returns to calls nor exits.

Let φ be a formula over AP. The closure of φ, $Cl(\varphi)$, is the smallest set that contains φ, contains *call*, *ret* and *int*, and satisfies the following properties:

- If $\neg\varphi' \in Cl(\varphi)$ or $\bigcirc^b\varphi' \in Cl(\varphi)$ (for some $b \in \{g, a, -\}$), then $\varphi' \in Cl(\varphi)$.
- If $\varphi' \vee \varphi'' \in Cl(\varphi)$, then $\varphi', \varphi'' \in Cl(\varphi)$.
- If $\varphi' \mathcal{U}^b \varphi'' \in Cl(\varphi)$, where $b \in \{g, a, -\}$, then φ', φ'', and $\bigcirc^b(\varphi' \mathcal{U}^b \varphi'')$ are in $Cl(\varphi)$.
- If $\varphi' \in Cl(\varphi)$ and φ' is not of the form $\neg\varphi''$ (for any φ''), then $\neg\varphi' \in Cl(\varphi)$.

It is straightforward to see that the size of $Cl(\varphi)$ is only linear in the size of φ. Henceforth, we identify $\neg\neg\varphi$ with the formula φ.

An *atom* of φ is a set $Y \subseteq Cl(\varphi)$ that satisfies the following properties:

- For every $\varphi' \in Cl(\varphi)$, $\varphi' \in Y$ iff $\neg\varphi' \notin Y$.
- For every formula $\varphi' \vee \varphi'' \in Cl(\varphi)$, $\varphi' \vee \varphi'' \in Y$ iff ($\varphi' \in Y$ or $\varphi'' \in Y$).
- For every formula $\varphi' \mathcal{U}^b \varphi'' \in Cl(\varphi)$, where $b \in \{a, g, -\}$, $\varphi' \mathcal{U}^b \varphi'' \in Y$ iff either $\varphi'' \in Y$ or ($\varphi' \in Y$ and $\bigcirc^b(\varphi' \mathcal{U}^b \varphi'') \in Y$).
- Y contains exactly one of the elements in the set $\{call, ret, int\}$.

Let $Atoms(\varphi)$ denote the set of atoms of φ; note that there are $2^{O(|\varphi|)}$ atoms of φ. To handle formulas in LTL, we can build a pushdown automaton whose states are of the form (u, A), where u is the current node of the RSM that is being simulated and A is an atom that represents the set of formulas true at u. We can use the stack for storing the names of the boxes, pushing in the box name at a call and popping it at the return.

The main difference in the construction for CaRet formulas is that at a call, the atom A true at the call is also pushed onto the stack along with the box b. When the call returns, we pop b and A and make sure that the abstract-next requirements in A get satisfied at the return-node. Note that we cannot keep track of A using the finite-state control because recursive calls will make the set of atoms to be remembered unbounded.

The caller modality is easier to handle. If we are at a call (b, e) where the set of formulas A is true, then a formula $\bigcirc^-\varphi'$ is true in the module being called only if $\varphi' \in A$. The caller formulas are hence passed down from the caller to the called module. The above two ideas are the crux of the construction.

There are several other technical issues to be handled. When an until-formula $\varphi_1\,\mathcal{U}^b\varphi_2$ is asserted at a node, we must make sure the liveness requirement φ_2 is eventually satisfied. This is done (as for LTL) using a generalized Büchi condition, one for each until formula. Caller-until formulas do not even require such a condition as the caller-path from any node is finite.

If an abstract-until formula $\varphi_1\,\mathcal{U}^a\varphi_2$ is asserted at a node u in a module, its liveness requirement φ_2 must be met in the abstract path from u and not in a node belonging to an invocation from the current module. In order to handle this we also keep track in the state whether the current node belongs to an invocation that will eventually return or not. For an abstract-until formula, the Büchi condition corresponding it has only states that correspond to invocations that do not return.

Let us now describe the construction formally. Let the given RSM be $(M, \{S_m\}_{m\in M}, start)$. Let $M' = \{m' \mid m \in M\}$ be a new set of module names, one for each module $m \in M$. The recursive generalized Büchi automaton we construct is $(M', \{S_{m'}\}_{m\in M'}, start', \mathcal{F})$ which is defined below.

Let $Tag = \{inf, fin\}$. For every node $u \in N_m$ that is not an exit there are nodes of the form (u, A, t) in $S_{m'}$, where A is an atom that represents the set of formulas that hold at u and $t \in Tag$ is a tag that signifies whether the run in the current module is infinite (inf–will never exit) or is finite (fin–will exit eventually). Similarly, for every $b \in B_m$, there are boxes of the form (b, A, t) in $S_{m'}$ where t is a tag and A is an atom containing formulas true at entries (b, e).

For any vertex v of S, we say an atom A is propositionally consistent with v if $\eta(v) \cap AP = A \cap AP$ and, further, if v is a node, then $int \in A$, if v is a call, then $call \in A$ and if v is a return then $ret \in A$.

For every $m \in M$, if $S_m = (N_m, B_m, Y_m, En_m, Ex_m, \delta_m, \eta_m)$, then $S'_m = (N_{m'}, B_{m'}, Y_{m'}, En_{m'}, Ex_{m'}, \delta_{m'}, \eta_{m'})$ where

- $N_{m'} = \{\ (u, A, t) \mid u \in N_m \setminus Ex_m, A \in Atoms(\varphi), t \in Tag$ and A is propositionally consistent with $u\ \} \cup$
  $\{\ (x, A, R) \mid x \in Ex_m, A, R \in Atoms(\varphi),$ and A is propositionally consistent with $x\ \}$
- $B_{m'} = B_m \times Atoms(\varphi) \times Tag$; $Y_{m'}(b, A, t) = (Y_m(b))'$
- $En_{m'} = \{(e, A, t) \mid e \in En_m, A \in Atoms(\varphi), t \in Tag\}$
- $Ex_{m'} = \{(x, A, R) \mid x \in Ex_m, A, R \in Atoms(\varphi)\}$
- $\eta_{m'}((u, A, t)) = \eta_m(u)$, $\eta_{m'}(((b, A, t), (e, A', t'))) = \eta_m((b, e))$, and $\eta_{m'}(((b, A, t), (x, A', R))) = \eta_m((b, x))$

Notice that calls are of the form $((b, A, t), (e, A', t'))$. In this A is the set of formulas true at this call and A' is the set of formulas true at the next vertex which will be the entry e. Hence, since the box-name (b, A, t) is pushed on the stack, the formulas A true at the call is remembered across the invocation.

Exits are of the form (x, A, R); here A denotes the formulas that are true at the exit x while R denotes the formulas true when the control returns to the called module. At a return $((b, A, t), (x, A', R))$, since the set of formulas true at the call was A and the set of formulas true at return is R, we will require that the abstract-next requirements in A are met in R.

For atoms A and A', we define a relation $AbsNextReq(A, A')$ that is true iff the abstract-next requirements in A are exactly the ones that hold in A', i.e. for each $\bigcirc^a \varphi' \in Cl(\varphi)$, $\bigcirc^a \varphi' \in A$ iff $\varphi' \in A'$. Similarly, we define a relation $GlNextReq(A, A')$ that is true iff the global-next requirements in A are exactly the ones that hold in A', i.e. for each $\bigcirc^g \varphi' \in Cl(\varphi)$, $\bigcirc^g \varphi' \in A$ iff $\varphi' \in A'$.

Also, let the caller formulas in A be denoted by $CallerFormulas(A) = \{\bigcirc^- \varphi' \mid \bigcirc^- \varphi' \in A\}$. The transition relation $\delta_{m'}$ is defined as follows:

(T1) From nodes to non-exit nodes: $\delta_{m'}((u, A, t))$ contains (u', A', t') iff:
- $u' \in \delta_m(u)$; $t = t'$
- $GlNextReq(A, A')$ and $AbsNextReq(A, A')$
- $CallerFormulas(A) = CallerFormulas(A')$

(T2) From nodes to calls: $\delta_{m'}((u, A, t))$ contains $((b, A', t'), (e, A'', t''))$ iff:
- $(b, e) \in \delta_m(u)$; $t' = t$
- A' is propositionally consistent with (b, e).
- $GlNextReq(A, A')$ and $AbsNextReq(A, A')$
- $CallerFormulas(A) = CallerFormulas(A')$
- $GlNextReq(A', A'')$; $CallerFormulas(A'') = \{\bigcirc^- \varphi' \in Cl(\varphi) \mid \varphi' \in A'\}$
- If $t'' = inf$, then $t = inf$ and there is no formula of the kind $\bigcirc^a \varphi$ in A'.

(T3) From nodes to exits:
$\delta_{m'}((u, A, t))$ contains (x, A', R), where $x \in Ex_m$, iff:
- $x \in \delta_m(u)$; $t = fin$
- $GlNextReq(A, A')$; $AbsNextReq(A, A')$; $GlNextReq(A', R)$
- $CallerFormulas(A) = CallerFormulas(A')$
- There is no formula of the kind $\bigcirc^a \varphi$ in A'.

(T4) From returns to nodes: $\delta_{m'}((b, A, t), (x, A', R))$ contains (u, A'', t'') iff
- $u \in \delta_m((b, x))$; $t'' = t$
- $AbsNextReq(A, R)$; $CallerFormulas(A) = CallerFormulas(R)$
- R is propositionally consistent with (b, x).
- $GlNextReq(R, A'')$ and $AbsNextReq(R, A'')$
- $CallerFormulas(R) = CallerFormulas(A'')$

The set of initial nodes is the set $start' = \{(u, A, t) \mid u \in start, \varphi \in A, A$ does not contain any formulas of the form $\bigcirc^- \varphi', t = inf\}$.

We say an atom A *momentarily satisfies* an abstract or global until formula $\varphi_1 \,\mathcal{U}^b \varphi_2$ (where $b = g$ or $b = a$) if either $\varphi_2 \in A$ or $\varphi_1 \,\mathcal{U}^b \varphi_2 \notin A$.
The generalized Büchi condition $\mathcal{F}$ is given by the following sets:
- A set containing all vertices of the form (u, A, t), where $t = inf$.
- For every global-until formula $\varphi_1 \,\mathcal{U}^g \varphi_2$ in $Cl(\varphi)$, there is a set in $\mathcal{F}$ containing all vertices of the form (u, A, t) or (x, A, R) or $((b, A, t, R), (e, A', t'))$ or $((b, A', t, A), (x, A'', A))$ where A momentarily satisfies $\varphi_1 \,\mathcal{U}^g \varphi_2$.
- For every abstract-until formula $\varphi_1 \,\mathcal{U}^a \varphi_2$ in $Cl(\varphi)$, there is a set in $\mathcal{F}$ containing all vertices of the form (u, A, t), $((b, A, t, R), (e, A', t'))$ or $((b, A', t, A), (x, A'', A))$ where A momentarily satisfies $\varphi_1 \,\mathcal{U}^a \varphi_2$ *and* $t = inf$.

The first set ensures that the tags were guessed correctly. The second class of states ensure that global-until formulas get satisfied eventually. For abstract-until formulas that get asserted at nodes where the abstract-path is infinite, the third class of sets ensure that they get eventually satisfied. Hence:

Theorem 1. *Given an RSM S and a formula φ, the model-checking problem for S against φ can be solved in time $|S| \cdot \theta_S^2 \cdot 2^{O(|\varphi|)}$, i.e., in time polynomial in S and exponential in the size of the formula. The problem is* EXPTIME*-complete (even when the RSM is fixed).*

Proof: Given RSM S and a formula φ, construct the RGBA $S_{\neg\varphi}$ for the RSM S and the negation of the formula φ. This RGBA generates exactly the runs of S that do not satisfy φ. Note that for every vertex/edge in S, we have $2^{O(|\varphi|)}$ vertices/edges in the RGBA; similarly $\theta_{S_{\neg\varphi}}$ is $\theta_S \cdot 2^{O(|\varphi|)}$. Also, the number of generalized Büchi sets is at most $|\varphi|+1$. By Proposition 1, the complexity follows and the problem is in EXPTIME. It is known that checking RSMs against LTL is already EXPTIME-hard, even when the RSM is fixed (this follows from the proof in [5]). Hence the problem is EXPTIME-complete. $\square$

5 Conclusions

We have proposed a notion of structured computations that abstractly captures reactive computations with nested calls and returns of program modules. We have introduced a temporal logic that allows specification of requirements of such computations, and a decision procedure to model check such specifications with respect to recursive state machines. This leads to a rich and unified framework for algorithmic reasoning about temporal requirements, stack inspection properties, and classical correctness requirements of structured programs. While our technical focus has been on model checking, CARET can be used in other applications such as simulation and runtime monitoring where LTL has been established to be fruitful.

The temporal modalities presented in this paper are natural for structured computations, but are not exhaustive. For example, one can define *global-predecessor*, *abstract-predecessor*, and *next-return* as the temporal duals of the global-successor, abstract-successor, and last-caller modalities, respectively. These can be added to the logic at no extra cost. On the other hand, consider the *within* modality I: $I\varphi$ holds at a call position i iff the computation fragment from position i to j, where j is the matching return, satisfies the temporal formula φ. Adding this modality raises the complexity of model checking to 2EXPTIME. A related question concerns the *expressive completeness* of the logic. We are currently studying the problem of characterizing the subclass of context-free properties that can be algorithmically checked against an RSM model.

Acknowledgement: We thank Mihalis Yannakakis for fruitful discussions.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Proc. of CAV'01*, LNCS 2102, pages 207–220. Springer, 2001.
2. T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. *SPIN Workshop on Model Checking of Software*, LNCS 1885, pages 113–130, 2000.

3. M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In Proc. ICALP, volume LNCS 2076, pages 652–666. 2001.
4. A. Bouajjani, R. Echahed, and P. Habermehl. On the verification problem of nonregular properties for nonregular processes. In *Proc., 10th Annual IEEE Symp. on Logic in Computer Science*, pages 123–133. IEEE, 1995.
5. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Applications to model checking. In *CONCUR'97: Concurrency Theory, Eighth International Conference*, LNCS 1243, pages 135–150. Springer, 1997.
6. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, R. Leino, and E. Poll. An overview of JML tools and applications. In *Proc. 8th International Workshop on Formal Methods for Industrial Critical Systems*, pages 75–89, 2003.
7. O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92: Concurrency Theory*, LNCS 630, pages 123–137. Springer, 1992.
8. T. Cachat, J. Duparc, and W. Thomas. Solving pushdown games with a Σ_3 winning condition. In *Proc. of CSL 2002*, LNCS 2471, 322-336. Springer, 2002.
9. K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T.A. Henzinger, and J. Palsberg. Stack size analysis for interrupt driven programs. In *Proceedings of the 10th International Symposium on Static Analysis*, volume LNCS 2694, pages 109–126, 2003.
10. H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 235–244, 2002.
11. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification, 12th International Conference*, LNCS 1855, pages 232–247. Springer, 2000.
13. J. Esparza, A. Kucera, and S. S. Schwoon. Model-checking LTL with regular valuations for pushdown systems. *Information and Computation*, 186(2):355–376, 2003.
14. D. Harel, D. Kozen and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
16. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
17. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
18. T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Proc. of the IEEE Symp. on Security and Privacy*, 89–103, 1999.
19. O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown Specifications. In *Proc. of LPAR 02*, LNCS 2514, pages 262–277. Springer, 2002.
20. O. Kupferman, N. Piterman, and M.Y. Vardi. Model checking linear properties of prefix-recognizable systems. In Proc. of *CAV 02*, LNCS 2404, 371–385, 2002.
21. O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *Proc., 12th ACM POPL*, pages 97–107, 1985.
22. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification.* Springer-verlag, 1991.
23. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977.
24. T. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. ACM POPL*, pages 49–61, 1995.