Verification of Well-Formed Communicating Recursive State Machines*

Laura Bozzelli[1], Salvatore La Torre[2], and Adriano Peron[1]

[1] Università di Napoli Federico II , Via Cintia, 80126 - Napoli, Italy
[2] Università degli Studi di Salerno, Via S. Allende, 84081 - Baronissi, Italy

Abstract. In this paper we introduce a new (non-Turing powerful) formal model of recursive concurrent programs called well-formed communicating recursive state machines (*CRSM*). *CRSM* extend recursive state machines (*RSM*) by allowing a restricted form of concurrency: a state of a module can be refined into a finite collection of modules (working in parallel) in a potentially recursive manner. Communication is only possible between the activations of modules invoked on the same fork. We study the model checking problem of *CRSM* with respect to specifications expressed in a temporal logic that extends CaRet with a parallel operator (ConCaRet). We propose a decision algorithm that runs in time exponential in both the size of the formula and the maximum number of modules that can be invoked simultaneously. This matches the known lower bound for deciding CaRet model checking of *RSM*, and therefore, we prove that model checking *CRSM* with respect to ConCaRet specifications is Exptime-complete.

1 Introduction

Computer programs often involve the concurrent execution of multiple threads interacting with each other. Each thread can require recursive procedure calls and thus make use of a local stack. In general, combining recursion and task synchronization leads to Turing-equivalent models. Therefore, there have been essentially two approaches in the literature that address the problem of analyzing recursive concurrent programs: (i) abstraction (approximate) techniques on 'unrestricted models' (e.g. see [5, 14]), and (ii) precise techniques for 'weaker models' (with decidable reachability), obtained by imposing restrictions on the amount of parallelism and synchronization.

In the second approach, many *non* Turing-equivalent formalisms, suitable to model the control flow of recursive concurrent programs, have been proposed. One of the most powerful is constituted by *Process Rewrite Systems* (*PRS*, for short) [13], a formalism based on term rewriting, which subsumes many common infinite–state models such as Pushdown Processes, Petri Nets, and PA processes. *PRS* can be adopted as a formal model for programs with dynamic creation and (a restricted form of) synchronization of concurrent processes, and with recursive

* This research was partially supported by the MIUR grant ex-60% 2003-2004 Università degli Studi di Salerno.

E.A. Emerson and K.S. Namjoshi (Eds.): VMCAI 2006, LNCS 3855, pp. 412–426, 2006.

procedures (possibly with return values). *PRS* can accommodate both *parallel-call* commands and *spawn* commands for the dynamic creation of threads: in a parallel-call command, the caller thread suspends its activation waiting for the termination of all called processes, while in a spawn command, the caller thread can pursue its execution concurrently to the new activated threads. However, there is a price to pay to this expressive power: for the general framework of *PRS*, the only decidability results known in the literature concern the reachability problem between two given terms and the *reachable property* problem [13]. Model checking against standard propositional temporal logics is undecidable also for small fragments. Moreover, note that the best known upper bound for reachability of Petri nets (which represent a subclass of *PRS*) requires non primitive recursive space [9]. In [6], symbolic reachability analysis is investigated and the given algorithm can be applied only to a strict subclass of *PRS*, i.e, the *synchronization–free PRS* (the so–called PAD systems) which subsume Pushdown Processes, *synchronization–free* Petri nets, and PA processes. In [10, 11], symbolic reachability analysis of PA processes is used to allow the interprocedural data-flow analysis of programs represented by systems of *parallel flow graphs* (parallel FGS, for short), which extend classical sequential flow graphs by parallel-call commands. In [7], Dynamic Pushdown Networks (DPN) are proposed for flow analysis of multithreaded programs. DPN allows spawn commands and can encode parallel FGS. An extention of this model that captures the modelling power of PAD is also considered.

In this paper, we consider a different abstract model of concurrent programs with finite domain variables and recursive procedures, the *well-formed communicating recursive state machines* (*CRSM*). A *CRSM* is an ordered collection of finite–state machines (called *modules*) where a state can represent a call, in a potentially recursive manner, to a finite collection of modules running in parallel. A parallel call to other modules models the activation of multiple threads in a concurrent program (*fork*). When a fork happens, the execution of the current module is stopped and the control moves into the modules activated in the fork. On termination of such modules, the control return to the calling module and its execution is resumed (*join*). *CRSM* allow the communication only between module instances that are activated on the same fork and do not have ongoing procedure calls. In our model, we allow multiple entries and exits for each module, which can be used to handle finite–domain local variables and return values from procedure calls (see [1]).

Intuitively, *CRSM* correspond to the subclass of *PRS* obtained by disallowing rewrite rules which model spawn commands. Also, note that *CRSM* extend parallel FGS since they also allow (a restricted form of) synchronization and return values from (parallel) procedure calls. With respect to DPN, *CRSM* allow synchronization. Moreover, *CRSM* extend both the recursive state machines (*RSM*) [1] by allowing parallelism and the well-structured communicating (finite-state) hierarchical state machines [3] by allowing recursion. We recall that *RSM* correspond to pushdown systems, and the related model checking problem has been extensively studied in the literature [17, 4, 1]. *CRSM* are strictly more expressive

than *RSM*. In fact, it is possible to prove that *synchronization-free CRSM* correspond to a complete normal form of *Ground Tree Rewriting systems* [8] and thus the related class of languages is located in the Chomsky hierarchy strictly between the context-free and the context-sensitive languages [12].

In this paper, we address the model-checking problem of *CRSM* with respect to specifications expressed in a new logic that we call ConCaRet. ConCaRet is a linear-time temporal logic that extends CaRet [2] with a parallel operator. Recall that CaRet extends standard *LTL* allowing the specification of non-regular context-free properties that are useful to express correctness of procedures with respect to pre- and post-conditions. The model checking of *RSM* with respect to CaRet specification is known to be Exptime-complete [2].

The semantics of ConCaRet is given with respect to (infinite) computations of *CRSM*. In our model, threads can fork, thus a global state (i.e., the stack content and the current local state of each active thread) of a *CRSM* is represented as a ranked tree. Hence, a computation corresponds to a sequence of these ranked trees. As in CaRet, we consider three different notions of local successor, for any local state along a computation, and the corresponding counterparts of the usual temporal operators of *LTL* logic: the *abstract successor* captures the local computation within a module removing computation fragments corresponding to nested calls within the module; the *caller* denotes the local state (if any) corresponding to the "innermost call" that has activated the current local state; a (local) *linear successor* of a local state is the usual notion of successor within the corresponding thread. In case the current local state corresponds to a fork, its successors give the starting local states of the activated threads. With respect to the paths generated by the linear successors, we allow the standard *LTL* modalities coupled with existential and universal quantification. Note that ConCaRet has no temporal modalities for the global successor (i.e., the next global state in the run). In fact, this operator would allow us to model unrestricted synchronization among threads and thus, the model checking of ConCaRet formulas would become undecidable. In ConCaRet we also allow a parallel operator that can express properties about communicating modules such as "every time a resource p is available for a process I, then it will be eventually available for all the processes in parallel with I" (in formulas, $\Box(\, p \rightarrow \| \, \Diamond^a \, p)$).

We show that model-checking *CRSM* against ConCaRet is decidable. Our approach is based on automata-theoretic techniques: given a *CRSM* $\mathcal{S}$ and a formula φ, we construct a *Büchi CRSM* $\mathcal{S}_{\neg\varphi}$ (i.e., a *CRSM* equipped with generalized Büchi acceptance conditions) such that model checking $\mathcal{S}$ against φ is reduced to check the emptiness of the Büchi *CRSM* $\mathcal{S}_{\neg\varphi}$. We solve this last problem by a non-trivial reduction to the emptiness problem for a straightforward variant of classical Büchi Tree Automata. Our construction of $\mathcal{S}_{\neg\varphi}$ extends the construction given in [2] for a *RSM* and a CaRet formula. Overall, our model checking algorithm runs in time exponential in both the maximal number ρ of modules that can invoked simultaneously and the size of the formula, and thus matches the known lower bound for deciding CaRet model checking. Therefore, we prove that the model-checking problem of *CRSM* with respect to ConCaRet

specifications is EXPTIME-complete. The main difference w.r.t. *RSM* is the time complexity in the size of the model that for *RSM* is polynomial, while for *CRSM*, it is exponential in ρ.

Due to the lack of space, for the omitted details we refer the reader to [18].

2 Well-Formed Communicating Recursive State Machines

In this section we define syntax and semantics of *well-formed Communicating Recursive State Machines* (*CRSM*, for short).

Syntax. A *CRSM* is an ordered collection of finite–state machines (*FSM*) augmented with the ability of refining a state with a collection of *FSM* (working in parallel) in a potentially recursive manner.

Definition 1. *A* CRSM $\mathcal{S}$ *over a set of propositions AP is a tuple* $\langle (S_1, \ldots, S_k), start \rangle$*, where for* $1 \leq i \leq k$, $S_i = \langle \Sigma_i, \Sigma_i^s, N_i, B_i, Y_i, En_i, Ex_i, \delta_i, \eta_i \rangle$ *is a* module *and* $start \subseteq \bigcup_{i=1}^{k} N_i$ *is a set of* start nodes*. Each module* S_i *is defined as follows:*

- Σ_i *is a finite alphabet and* $\Sigma_i^s \subseteq \Sigma_i$ *is the set of* synchronization symbols*;*
- N_i *is a finite set of* nodes *and* B_i *is a finite set of* boxes (*with* $N_i \cap B_i = \emptyset$)*;*
- $Y_i : B_i \to \{1, \ldots, k\}^+$ *is the* refinement function *which associates with every box a sequence of module indexes;*
- $En_i \subseteq N_i$ (*resp.,* $Ex_i \subseteq N_i$) *is a set of* entry nodes (*resp.,* exit nodes)*;*
- $\delta_i : (N_i \cup Retns_i) \times \Sigma_i \to 2^{N_i \cup Calls_i}$ *is the* transition function*, where* $Calls_i = \{(b, e_1, \ldots, e_m) \mid b \in B_i,\ e_j \in En_{h_j}$ *for any* $1 \leq j \leq m$*, and* $Y_i(b) = h_1 \ldots h_m\}$ *denotes the set of* calls *and* $Retns_i = \{(b, x_1, \ldots, x_m) \mid b \in B_i,\ x_j \in Ex_{h_j}$ *for any* $1 \leq j \leq m$*, and* $Y_i(b) = h_1 \ldots h_m\}$ *denotes the set of* returns *of* S_i*; we assume w.l.o.g. that exits have no outgoing transitions, and entries have no incoming transitions, and* $En_i \cap Ex_i = \emptyset$*;*
- $\eta_i : V_i \to 2^{AP}$ *is the* labelling function*, with* $V_i = N_i \cup Calls_i \cup Retns_i$ (V_i *is the set of* vertices)*.*

We assume that $(V_i \cup B_i) \cap (V_j \cup B_j) = \emptyset$ for $i \neq j$. Also, let $\Sigma = \bigcup_{i=1}^{i=k} \Sigma_i$, $\Sigma^s = \bigcup_{i=1}^{i=k} \Sigma_i^s$, $V = \bigcup_{i=1}^{i=k} V_i$, $Calls = \bigcup_{i=1}^{i=k} Calls_i$, $Retns = \bigcup_{i=1}^{i=k} Retns_i$, $N = \bigcup_{i=1}^{i=k} N_i$, $B = \bigcup_{i=1}^{i=k} B_i$, $En = \bigcup_{i=1}^{i=k} En_i$, and $Ex = \bigcup_{i=1}^{i=k} Ex_i$. Functions $\eta : V \to 2^{AP}$, $Y : B \to \{1, \ldots, k\}^+$, and $\delta : (N \cup Retns) \times \Sigma \to 2^{N \cup Calls}$ are defined as the natural extensions of functions η_i, Y_i and δ_i (with $1 \leq i \leq k$).

The set of states of a module is partitioned into a set of nodes and a set of boxes. Performing a transition to a box b can be interpreted as a (parallel) procedure call (*fork*) which simultaneously activates a collection of modules (the list of modules given by the refinement function Y applied to b). Since Y gives a list of module indexes, a fork can activate different instances of the same module. Note that a transition leading to a box specifies the entry node (initial state) of each activated module. All the module instances, which are simultaneously activated in a call, run in parallel, whereas the calling module instance suspends its activity waiting for the return of the (parallel) procedure call. The return

of a parallel procedure call to a box b is represented by a transition from b that specifies an exit node (exiting state) for each module copy activated by the procedural call (a synchronous return or *join* from all the activated modules).

Figure 1 depicts a simple *CRSM* consisting of two modules S_1 and S_2. Module S_1 has two entry nodes u_1 and u_2, an exit node u_4, an internal node u_3 and one box b_1 that is mapped to the parallel composition of two copies of S_2. The module S_2 has an entry node w_1, an exit node w_2, and two boxes b_2 and b_3 both mapped to one copy of S_1. The transition from node u_1 to box b_1 in S_1 is represented by a *fork* transition having u_1 as source and the entry nodes of the two copies of S_2 as targets. Similarly, the transition from box b_1 to node u_4 is represented by a *join* transition having the exit nodes of the two copies of S_2 as sources and u_4 as target.

In our model, the communication is allowed only between module instances that are activated on the same fork and are not busy in a (parallel) procedure call. As for the communicating (finite-state) hierarchical state machines [3], the form of communication we allow is synchronous and maximal in the sense that if a component (module) takes a transition labelled by a synchronization symbol σ, then each other parallel component which has σ in its synchronization alphabet must be able to take a transition labelled by σ. For instance, assuming that the symbols σ_1 and σ_2 in Figure 1 are synchronization symbols, the two copies of S_2 which refine box b_1 either both take the transition labelled by σ_1 activating a copy of S_1 with start node u_1 or both take the transition labelled by σ_2 activating a copy of S_1 with start node u_2. Transitions labelled by symbols in $\Sigma \setminus \Sigma^s$ are instead performed independently without any synchronization requirement. A *synchronization-free CRSM* is a *CRSM* in which $\Sigma^s = \emptyset$.

The *rank* of $\mathcal{S}$, written $rank(\mathcal{S})$, is the maximum of $\{|Y(b)| \mid b \in B\}$. Note that if $rank(\mathcal{S}) = 1$, then $\mathcal{S}$ is a *Recursive State Machine* (*RSM*) as defined in [1].

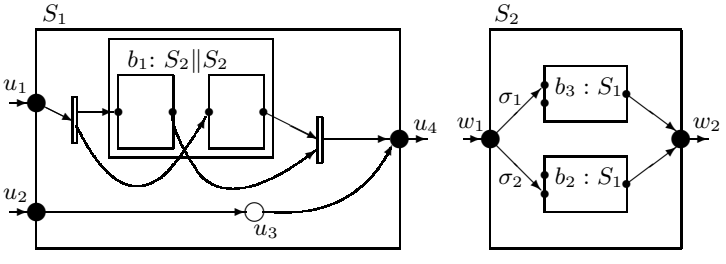

Fig. 1. A sample *CRSM*

Semantics. We give some notation first. A *tree* t is a prefix closed subset of $\mathbb{N}^*$ such that if $y \cdot i \in t$, then $y \cdot j \in t$ for any $0 \leq j < i$. The empty word ε is the *root* of t. The set of *leaves* of t is $leaves(t) = \{y \in t \mid y \cdot 0 \notin t\}$. For $y \in t$, the set of *children* of y (in t) is $children(t, y) = \{y \cdot i \in t\}$ and the set of *siblings* of y (in t) is $siblings(t, y) = \{y\} \cup \{y' \cdot i \in t \mid y = y' \cdot j \text{ for some } j \in \mathbb{N}\}$. For $y, y' \in \mathbb{N}^*$, we write $y \prec y'$ to mean that y is a proper prefix of y'.

The semantics of a $CRSM$ $\mathcal{S}$ is defined in terms of a Labelled Transition System $K_{\mathcal{S}} = \langle Q, R \rangle$. Q is the set of (global) states which correspond to activation hierarchies of instances of modules, and are represented by finite trees whose locations are labelled with vertices and boxes of the $CRSM$. Leaves correspond to active modules and a path in the tree leading to a leave y (excluding y) corresponds to the local call stack of the module instance associated with y. Formally, a state is a pair of the form (t, D) where t is a (finite) tree and $D : t \to B \cup V$ is defined as follows:

- if $y \in \mathit{leaves}(t)$, then $D(y) \in V$ (i.e. a vertex of $\mathcal{S}$);
- if $y \in t \setminus \mathit{leaves}(t)$ and $\mathit{children}(t, y) = \{y \cdot 0, \ldots, y \cdot m\}$, then $D(y) = b \in B$, $Y(b) = h_0 \ldots h_m$, and $D(y \cdot j) \in B_{h_j} \cup V_{h_j}$ for any $j = 0, \ldots, m$.

The *global transition relation* $R \subseteq Q \times (2^{\mathbb{N}^*} \times 2^{\mathbb{N}^*}) \times Q$ is a set of tuples of the form $\langle (t, D), (\ell, \ell'), (t', D') \rangle$ where (t, D) (resp., (t', D')) is the source (resp., target) of the transition, ℓ keeps track of the elements of t corresponding to the (instances of) modules of $\mathcal{S}$ performing the transition, and: for an *internal move* $\ell' = \ell$, for a *call*, ℓ' points to the modules activated by the (parallel) procedure call, and for a *return from a call*, ℓ' points to the reactivated caller module. Formally, $\langle (t, D), (\ell, \ell'), (t', D') \rangle \in R$ iff one of the following holds:

Single internal move: $t = t'$, there is $y \in \mathit{leaves}(t)$ and $\sigma \in \Sigma \setminus \Sigma^s$ such that $\ell = \ell' = \{y\}$, $D'(y) \in \delta(D(y), \sigma)$, and $D'(z) = D(z)$ for any $z \in t \setminus \{y\}$.

Synchronous internal move: $t' = t$, there are $y \in t$ with $\mathit{siblings}(t, y) = \{y_1, \ldots, y_m\}$, $\sigma \in \Sigma^s$, and indexes $k_1, \ldots, k_p \in \{1, \ldots, m\}$ such that the following holds: $\ell = \ell' = \{y_{k_1}, \ldots, y_{k_p}\} \subseteq \mathit{leaves}(t)$, $D'(z) = D(z)$ for any $z \in t \setminus \{y_{k_1}, \ldots, y_{k_p}\}$, $D'(y_{k_j}) \in \delta(D(y_{k_j}), \sigma)$ for any $1 \leq j \leq p$, and for any $j \in \{1, \ldots, m\} \setminus \{k_1, \ldots, k_p\}$, σ is *not* a synchronization symbol of the module associated with $D(y_j)$.

Module call: there is $y \in \mathit{leaves}(t)$ such that $D(y) = (b, e_0, \ldots, e_m) \in \mathit{Call}$, $t' = t \cup \{y \cdot 0, \ldots, y \cdot m\}$, $\ell = \{y\}$, $\ell' = \{y \cdot 0, \ldots, y \cdot m\}$, $D'(z) = D(z)$ for any $z \in t \setminus \{y\}$, $D'(y) = b$, and $D'(y \cdot j) = e_j$ for any $0 \leq j \leq m$.

Return from a call: there is $y \in t \setminus \mathit{leaves}(t)$ such that $\ell = \mathit{children}(t, y) = \{y \cdot 0, \ldots, y \cdot m\} \subseteq \mathit{leaves}(t)$, $\ell' = \{y\}$, $(D(y), D(y \cdot 0), \ldots, D(y \cdot m)) \in \mathit{Retns}$, $t' = t \setminus \{y \cdot 0, \ldots, y \cdot m\}$, $D'(z) = D(z)$ for any $z \in t' \setminus \{y\}$, and $D'(y) = (D(y), D(y \cdot 0), \ldots, D(y \cdot m))$.

For $v \in V$, we denote with $\langle v \rangle$ the global state $(\{\varepsilon\}, D)$ where $D(\varepsilon) = v$. A *run* of $\mathcal{S}$ is an infinite path in $K_{\mathcal{S}}$ from a state of the form $\langle v \rangle$.

2.1 Local Successors

Since we are interested in the local transformation of module instances, as in [2], we introduce different notions of local successor of module instances along a run. We fix a $CRSM$ $\mathcal{S}$ and a run of $\mathcal{S}$ $\pi = q_0 \xrightarrow{(\ell_0, \ell'_0)} q_1 \xrightarrow{(\ell_1, \ell'_1)} q_2 \ldots$ with $q_i = (t_i, D_i)$ for any i. We denote by Q_π the set $\{(i, y) \mid y \in \mathit{leaves}(t_i)\}$. An element (i, y) of Q_π, called *local state* of π, represents an instance of a module that at state q_i is

active and in the vertex $D_i(y)$. Note that the set $\{(i, y') \mid y' \prec y\}$ represents the (local) stack of this instance. Now, we define two notions of *next* local state (w.r.t. a run).

For $(i, y) \in Q_\pi$, $next^\ell_\pi(i, y)$ gives the set of module instances, called *linear successors* of (i, y), that are obtained by the first transition affecting (i, y). Note that such a transition may occur at $j \geq i$ or may not occur at all. In the former case $next^\ell_\pi(i, y)$ is a singleton unless $D_i(y) \in Calls$, and then the linear successors correspond to the entry nodes of the called modules. Formally, if $\{m \geq i \mid y \in \ell_m\} = \emptyset$ then $next^\ell_\pi(i, y) = \emptyset$, otherwise $next^\ell_\pi(i, y)$ is given by:

$$\{(h + 1, y') \mid h = min\{m \geq i \mid y \in \ell_m\}, y' \in \ell'_h, \text{ and either } y' \preceq y \text{or} y \preceq y'\}.$$

Note that if $rank(\mathcal{S}) = 1$ (i.e. $\mathcal{S}$ is a *RSM*), then the *linear* successor corresponds to the (standard) *global* successor.

For each $(i, y) \in Q_\pi$, we also give a notion of *abstract successor*, denoted $next^a_\pi(i, y)$. If (i, y) corresponds to a call that returns, i.e., $D_i(y) \in Calls$ and there is a $j > i$ such that $y \in leaves(t_j)$ and $D_j(y) \in Retns$, then $next^a_\pi(i, y) = (h, y)$ where h is the smallest of such j (the local state (h, y) corresponds to the matching return). For internal moves, the abstract successor coincides with the (unique) linear successor, i.e., if $next^\ell_\pi(i, y) = \{(j, y)\}$, then $next^a_\pi(i, y) = (j, y)$ (note that in this case $D_i(y) \in Retns \cup N \setminus Ex$). In all the other cases, the abstract successor is not defined and we denote this with $next^a_\pi(i, y) = \perp$. The abstract successor captures the local computations inside a module A skipping over invocations of other modules called from A.

Besides linear and abstract successor, we also define a caller of a local state (i, y) as the 'innermost call' that has activated (i, y). Formally, the *caller* of (i, y) (if any), written $next^-_\pi(i, y)$, is defined as follows (notice that only local states of the form (i, ε) have no callers): if $y = y' \cdot m$ for some $m \in \mathbb{N}$, then $next^-_\pi(i, y) = (j, y')$ where j is the maximum of $\{h < i \mid y' \in t_h \text{ and } D_h(y') \in Calls\}$; otherwise, $next^-_\pi(i, y)$ is undefined, written $next^-_\pi(i, y) = \perp$.

The above defined notions allow us to define sequences of local moves (i.e. moves affecting local states) in a run. For $(i, y) \in Q_\pi$, the set of *linear paths* of π *starting from* (i, y) is the set of (finite or infinite) sequences of local states $r = (j_0, y_0)(j_1, y_1) \ldots$ such that $(j_0, y_0) = (i, y)$, $(j_{h+1}, y_{h+1}) \in next^\ell_\pi(j_h, y_h)$ for any h, and either r is infinite or leads to a local state (j_p, y_p) such that $next^\ell_\pi(j_p, y_p) = \emptyset$. Analogously, the notion of *abstract path* (resp. *caller path*) of π *starting from* (i, y) can be defined by using in the above definition the abstract successor (resp. caller) instead of the linear successor. Note that a caller path is always finite and uniquely determines the content of the call stack locally to the instance of the module active at (i, y).

For module instances involved in a call, i.e., corresponding to pairs (i, y) such that $y \in t_i \setminus leaves(t_i)$, we denote the local state (if any) at which the call pending at (i, y) will return by $return_\pi(i, y)$. Formally, if $y \in leaves(t_j)$ for some $j > i$, then $return_\pi(i, y) = \{(h, y)\}$ where h is the smallest of such j, otherwise $return_\pi(i, y) = \perp$. Also, we denote the local state corresponding to the call activating the module instance at (i, y) by $call_\pi(i, y)$. Formally, $call_\pi(i, y) := (h, y)$ where h is the maximum of $\{j < i \mid y \in t_j \text{ and } D_j(y) \in Calls\}$.

Let *Evolve* be the predicate over sets of vertices and boxes defined as follows: $Evolve(\{v_1, \ldots, v_m\})$ holds iff *either* (1) there are $\sigma \in \Sigma \setminus \Sigma^s$ and $1 \leq i \leq m$ such that $v_i \in N \cup Retns$ and $\delta(v_i, \sigma) \neq \emptyset$ (single internal move), *or* (2) there is $\sigma \in \Sigma^s$ such that the set $H = \{1 \leq i \leq m \mid v_i \in N \cup Retns$ and $\delta(v_i, \sigma) \neq \emptyset\}$ is *not* empty and for each $i \in \{1, \ldots, m\} \setminus H$, σ does not belong to the synchronization alphabet of the module associated with v_i (synchronized internal move). In the following, we focus on *maximal runs* of *CRSM*. Intuitively, a maximal run represents an infinite computation in which each set of module instances activated by the same parallel call that may evolve (independently or by a synchronous internal move) is guaranteed to make progress.
Formally, a run π is *maximal* if for all $(i, y) \in Q_\pi$, the following holds:

- if $D_i(y) \in Calls$, then $next^\ell_\pi(i, y) \neq \emptyset$ (a possible module call must occur);
- if $y \neq \varepsilon$ and $D_i(y') \in Ex$ for all $y' \in siblings(t_i, y)$, then $next^\ell_\pi(i, y) \neq \emptyset$ (i.e., if a return from a module call is possible, then it must occur);
- if $y \neq \varepsilon$, $siblings(t_i, y) = \{y_0, \ldots, y_m\}$ and for each $0 \leq j \leq m$, $next^\ell_\pi(i, y_j) = \emptyset$ if $(i, y_j) \in Q_\pi$ and $return_\pi(i, y_j) = \perp$ otherwise, then the condition $Evolve(\{D_i(y_0), \ldots, D_i(y_m)\})$ does *not* hold.

3 The Temporal Logic ConCaRet

Let AP be a finite set of *atomic propositions*. The logic ConCaRet over AP is the set of formulas inductively defined as follows:

$$\varphi ::= p \mid call \mid ret \mid int \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc^b \varphi \mid \varphi \mathcal{U}^b \varphi \mid \|\varphi$$

where $b \in \{\exists, \forall, a, -\}$ and $p \in AP$.

A formula is interpreted over runs of an *CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start \rangle$. Let $\pi = q_0 \xrightarrow{(\ell_0, \ell'_0)} q_1 \xrightarrow{(\ell_1, \ell'_1)} \ldots$ be a run of $\mathcal{S}$ where $q_i = (t_i, D_i)$ for $i \geq 0$. The truth value of a formula w.r.t. a local state (i, y) of π is defined as follows:

- $(i, y) \models_\pi p$ iff $p \in \eta(D_i(y))$ (where $p \in AP$);
- $(i, y) \models_\pi call$ (resp. ret, int) iff $D_i(y) \in Calls$ (resp. $D_i(y) \in Retns$, $D_i(y) \in N$);
- $(i, y) \models_\pi \neg\varphi$ iff it is not the case that $(i, y) \models_\pi \varphi$;
- $(i, y) \models_\pi \varphi_1 \vee \varphi_2$ iff either $(i, y) \models_\pi \varphi_1$ or $(i, y) \models_\pi \varphi_2$;
- $(i, y) \models_\pi \bigcirc^b \varphi$ (with $b \in \{a, -\}$) iff $next^b_\pi(i, y) \neq \perp$ and $next^b_\pi(i, y) \models_\pi \varphi$;
- $(i, y) \models_\pi \bigcirc^\exists \varphi$ iff there is $(j, y') \in next^\ell_\pi(i, y)$ such that $(j, y') \models_\pi \varphi$;
- $(i, y) \models_\pi \bigcirc^\forall \varphi$ iff for all $(j, y') \in next^\ell_\pi(i, y)$, $(j, y') \models_\pi \varphi$;
- $(i, y) \models_\pi \varphi_1 \mathcal{U}^a \varphi_2$ (resp. $\varphi_1 \mathcal{U}^- \varphi_2$) iff given the *abstract* (resp. *caller*) path $(j_0, y_0)(j_1, y_1) \ldots$ starting from (i, y), there is $h \geq 0$ such that $(j_h, y_h) \models_\pi \varphi_2$ and for all $0 \leq p \leq h - 1$, $(j_p, y_p) \models_\pi \varphi_1$;
- $(i, y) \models_\pi \varphi_1 \mathcal{U}^\exists \varphi_2$ (resp. $\varphi_1 \mathcal{U}^\forall \varphi_2$) iff for some linear path (resp., for all linear paths) $(j_0, y_0)(j_1, y_1) \ldots$ starting from (i, y), there is $h \geq 0$ such that $(j_h, y_h) \models_\pi \varphi_2$ and for all $0 \leq p \leq h - 1$, $(j_p, y_p) \models_\pi \varphi_1$;
- $(i, y) \models_\pi \|\varphi$ *iff* for all $y' \in siblings(t_h, y) \setminus \{y\}$ with $h = min\{j \leq i \mid (j, y) \in Q_\pi$ and $next^\ell_\pi(j, y) = next^\ell_\pi(i, y)\}$, it holds that:

– if $D_h(y') \in V$, then $(h, y') \models_\pi \varphi$;
– if $D_h(y') \in B$, then $call_\pi(h, y') \models_\pi \varphi$.

We say the run π *satisfies* a formula φ, written $\pi \models \varphi$, if $(0, \varepsilon) \models_\pi \varphi$ (recall that $t_0 = \{\varepsilon\}$). Moreover, we say $\mathcal{S}$ satisfies φ, written $\mathcal{S} \models \varphi$, iff for any $u \in \mathit{start}$ and for any *maximal* run π of $\mathcal{S}$ starting from $\langle u \rangle$, it holds that $\pi \models \varphi$. Now, we can define the model-checking question we are interested in:

Model checking problem: *Given a* CRSM $\mathcal{S}$ *and a* ConCaRet *formula* φ*, does* $\mathcal{S} \models \varphi$*?*

For each type of local successor (forward or backward), the logic provides the corresponding versions of the usual (global) next operator $\bigcirc$ and until operator $\mathcal{U}$. For instance, formula $\bigcirc^- \varphi$ demands that the caller of the current local state satisfies φ, while $\varphi_1 \mathcal{U}^- \varphi_2$ demands that the caller path (that is always finite) from the current local state satisfies $\varphi_1 \mathcal{U} \varphi_2$. Moreover, the linear modalities are branching-time since they quantify over the possible linear successors of the current local state. Thus, we have both existential and universal linear versions of the standard modalities $\bigcirc$ and $\mathcal{U}$. Finally, the operator $\|$ is a new modality introduced to express properties of parallel modules. The formula $\| \phi$ holds at a local state (i, y) of a module instance I *iff*, being $h \leq i$ the time when vertex $D_i(y)$ was first entered and such that I has been idle from h to i, any module instance (different from I) in parallel with I and not busy in a module call (at time h) satisfies φ at time h, and any module instance in parallel with I and busy in a module call (at time h) satisfies φ at the call time.

Note that the semantic of the parallel operator ensures the following desirable property for the logic ConCaRet: for each pair of local states (i, y) and (j, y) such that $i < j$ and $next^\ell_\pi(j, y) = next^\ell_\pi(i, y)$ (i.e., associated with a module instance which remains idle from i to j), the set of formulas that hold at (i, y) coincides with the set of formulas that hold at (j, y).

We conclude this section illustrating some interesting properties which can be expressed in ConCaRet. In the following, as in standard *LTL*, we will use $\Diamond^b \varphi$ as an abbreviation for $\mathit{true}\, \mathcal{U}^b \varphi$, for $b \in \{\exists, \forall, a, -\}$. Further, for $b \in \{a, -\}$, let $\Box^b \varphi$ stand for $\neg \Diamond^b \neg \varphi$, $\Box^\forall \varphi$ stand for $\neg \Diamond^\exists \neg \varphi$, and $\Box^\exists \varphi$ stand for $\neg \Diamond^\forall \neg \varphi$.

Besides the stack inspection properties and pre/post conditions of local computations of a module as in CaRet [2], in ConCaRet, we can express pre- and post- conditions for multiple threads activated in parallel. For instance, we can require that whenever module A and module B are both activated in parallel and pre-condition p holds, then A and B need to terminate, and post-condition q is satisfied upon the synchronous return (*parallel total correctness*). Assuming that parallel-calls to the modules A and B are characterized by the proposition $p_{A,B}$, this requirement can be expressed by the formula $\Box^\forall \big[(\mathit{call} \wedge p \wedge p_{A,B}) \rightarrow \bigcirc^a q\big]$. Linear modalities refer to the branching-time structure of *CRSM* computations. They can be used, for instance, to express invariance properties of the kind "every time a call occurs, then each activated module has to satisfy property φ". Such a property can be written as $\Box^\forall(\mathit{call} \rightarrow \bigcirc^\forall \varphi)$. We can also express simple global

eventually properties of the kind "every time the computation starts from module A, then module B eventually will be activated", expressed by the formula $t_A \rightarrow \Diamond^{\exists} t_B$. However, we can express more interesting global properties such as *recurrence* requirements. For instance, formula $\Diamond^{\forall}\Box^{\forall}(call \rightarrow \bigcirc^{\forall}\neg t_A)$ asserts that module A is activated a finite number of times. Therefore, the negation of this formula requires an interesting global recurrence property: along any maximal infinite computation module A is activated infinitely many times.

The parallel modality can express alignment properties among parallel threads. For instance, formula $\Box^{\forall}\big[call \rightarrow \bigcirc^{\exists}(\Diamond^{a}\ (\phi \wedge \parallel \phi))\big]$ requires that when a parallel-call occurs, there must be an instant in the future such that the same property ϕ holds in all the parallel threads activated by the parallel-call. In particular, with such formula we could require the existence of a future time at which all threads activated by the call will be ready for a maximal synchronization on a symbol. More generally, with the parallel operator we can express reactivity properties of modules, namely the ability of a module to continuously interact with its parallel components. We can also express mutual exclusion properties: among the modules activated in a same procedure call, at most a module can access a shared resource p (in formulas, $\Box^{\forall}(p \rightarrow\!\parallel \neg p)$).

4 Büchi *CRSM*

In this section, we extend *CRSM* with acceptance conditions and address the emptiness problem for the resulting class of machines (i.e., the problem about the existence of an *accepting* maximal run from a *start* node). Besides standard acceptance conditions on the finite linear paths of a maximal run π, we require a synchronized acceptance condition on modules running in parallel, and a generalized Büchi acceptance condition on the infinite linear paths of π. We call this model a Büchi *CRSM* (*B-CRSM* for short). Formally, a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_f, \mathcal{P}_{sync} \rangle$ consists of a *CRSM* $\langle (S_1, \ldots, S_k), start \rangle$ together with the following acceptance conditions:

- $\mathcal{F} = \{F_1, \ldots, F_n\}$ is a family of accepting sets of vertices of $\mathcal{S}$;
- $\mathcal{P}_F$ is the set of *terminal* vertices;
- $\mathcal{P}_{sync}$ is a predicate defined over pairs (v, H) such that v is a vertex and H is a set of vertices such that $|H| \leq rank(\mathcal{S})$.

Let $\pi = q_0 \overset{(\ell_0,\ell'_0)}{\longrightarrow} q_1 \overset{(\ell_1,\ell'_1)}{\longrightarrow} q_2 \ldots$ be a run of $\mathcal{S}$ with $q_i = (t_i, D_i)$ for $i \geq 0$. For each $i \geq 0$ and $y \in t_i$, we denote by $v(i, y)$ the vertex of $\mathcal{S}$ defined as follows: if $(i, y) \in Q_\pi$ (i.e., (i, y) is a local state), then $v(i, y) := D_i(y)$; otherwise, $v(i, y) := D_h(y)$ where $(h, y) := call_\pi(i, y)$. We say that the run π is *accepting* iff the following three conditions are satisfied:

1. for any infinite linear path $r = (i_0, y_0)(i_1, y_1) \ldots$ of π and $F \in \mathcal{F}$, there are infinitely many $h \in \mathbb{N}$ such that $D_{i_h}(y_h) \in F$ (*generalized Büchi acceptance*);
2. for any local state $(i, y) \in Q_\pi$ such that $next^{\ell}_{\pi}(i, y) = \emptyset$, condition $D_i(y) \in \mathcal{P}_F$ holds (*terminal acceptance*);

3. for any $i \geq 0$ and $y \in \ell'_i$, $\mathcal{P}_{sync}(v(i+1,y), \{v(i+1,y_1), \ldots, v(i+1,y_{m_i})\})$ holds,where $\{y_1, \ldots, y_{m_i}\}$ is $siblings(t_{i+1}, y) \setminus \{y\}$ (*synchronized acceptance*)

We say the run π is *monotone* iff for all $i \geq 0$, q_{i+1} is obtained from q_i either by a module call or by an internal move. Note that in a monotone path either the tree t_{i+1} is equal to t_i (for an internal move) or it is obtained from t_i by adding some children to a leaf (for a module call).

We decide the emptiness problem for *B-CRSM* in two main steps:

1. First, we give an algorithm to decide the problem about the existence of accepting *monotone* maximal runs starting from a given vertex;
2. Then, we reduce the emptiness problem to the problem addressed in Step 1.

4.1 Deciding the Existence of Accepting Monotone Maximal Runs

We show how to decide the existence of accepting *monotone* maximal runs in *B-CRSM* by a reduction to the emptiness problem for *Invariant Büchi tree automata*. These differ from the standard formalism of Büchi tree automata [15] for a partitioning of states into *invariant* and *non-invariant* states, with the constraint that transitions from non-invariant to invariant states are forbidden. Also, the standard Büchi acceptance condition is strengthened by requiring in addition that an accepting run must have a path consisting of invariant states only.

Formally, an (alphabet free) *invariant Büchi tree automaton* is a tuple $\mathbb{U} = \langle \mathcal{D}, P, P_0, M, F, Inv \rangle$, where $\mathcal{D} \subset \mathbb{N} \setminus \{0\}$ is a finite set of branching degrees, P is the finite set of states, $P_0 \subseteq P$ is the set of initial states, $M : P \times \mathcal{D} \to 2^{P^*}$ is the transition function with $M(s,d) \in 2^{P^d}$, for all $(s,d) \in P \times \mathcal{D}$, $F \subseteq P$ is the Büchi condition, and $Inv \subseteq P$ is the invariance condition. Also, for any $s \in P \setminus Inv$ and $d \in \mathcal{D}$, we require that if s' occurs in $M(s,d)$, then $s' \in P \setminus Inv$. A *complete* $\mathcal{D}$-tree is an infinite tree $t \subseteq \mathbb{N}^*$ such that for any $y \in t$, the cardinality of $children(t,y)$ belongs to $\mathcal{D}$. A *path* of t is a maximal subset of t linearly ordered by $\prec$. A run of $\mathbb{U}$ is a pair (t,r) where t is a complete $\mathcal{D}$-tree, $r : t \to P$ is a P-labelling of t such that $r(\varepsilon) \in P_0$ and for all $y \in t$, $(r(y \cdot 0), r(y \cdot 1), \ldots, r(y \cdot d)) \in M(r(y), d+1)$, where $d+1 = |children(t,y)|$. The run (t,r) is *accepting* iff: (1) there is a path ν of t such that for every $y \in \nu$, $r(y) \in Inv$, and (2) for any path ν of t, the set $\{y \in \nu \mid r(y) \in F\}$ is infinite.

The algorithm in [16] for checking emptiness in Büchi tree automata can be easily extended to handle also the invariance condition, thus we obtain the following.

Proposition 1. *The emptiness problem for invariant Büchi tree automata is logspace-complete for* PTIME *and can be decided in quadratic time.*

In the following, we fix a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_F, \mathcal{P}_{sync} \rangle$.

Remark 1. Apart from a preliminary step computable in linear time (in the size of $\mathcal{S}$), we can restrict ourselves to consider only accepting monotone maximal runs π of S starting from call vertices. In fact, if π starts at a non-call vertex v of a module S_h, then either π stays within S_h forever, or π enters a call v' of S_h

that never returns. In the first case, one has to check the existence of an accepting run in the *generalized* Büchi (word) automaton given by $A_h = \langle V_h, \delta_h, \mathcal{F}_h \rangle$, where $\mathcal{F}_h$ is the restriction of $\mathcal{F}$ to the set V_h. This can be done in linear time [15]. In the second case, one has to check that there is a call v' reachable from v in A_h, and then that there is an accepting monotone maximal run in $\mathcal{S}$ from v'.

Now, we construct an invariant Büchi tree automaton $\mathbb{U}$ capturing the monotone accepting maximal runs of $\mathcal{S}$ starting from calls. The idea is to model a monotone run π of $\mathcal{S}$ as an infinite tree (a run of $\mathbb{U}$) where each path corresponds to a linear path of π. There are some technical issues to be handled.

First, there can be finite linear paths. We use symbol $\top$ to capture terminal local states of π. Therefore, the subtree rooted at the node corresponding to a terminal local state is completely labelled by $\top$. Also, since we are interested in runs of $\mathcal{S}$, we need to check that there is at least one infinite linear path in π. We do this using as invariant set the set of all $\mathbb{U}$ states except the state $\top$.

Second, when a module call associated with a box b occurs, multiple module instances $I_1, \ldots, I_m$ are activated and start running. We encode these local runs (corresponding to linear paths of π) on the same path of the run of $\mathbb{U}$ by using states of the form $(b, v_1, \ldots, v_m, i_1, \ldots, i_m, j)$, where $v_1, \ldots, v_m$ are the current nodes or calls of each module, and $i_1, \ldots, i_m, j$ are finite counters used to check the fulfillment of the Büchi condition (see below). Since in monotone runs there are no returns from calls, when a module I_j (with $1 \leq j \leq m$) moves to a call vertex v (by an internal move), we can separate the linear paths starting from v from the local runs associated with all modules $I_1, \ldots, I_m$ except I_j. Therefore, in order to simulate an internal move (in the context of modules $I_1, \ldots, I_m$), $\mathbb{U}$ nondeterministically splits in $d+1$ copies for some $0 \leq d \leq m$ such that d copies correspond to those modules (among $I_1, \ldots, I_m$) which move to call vertices, and the $d+1$-th copy goes to a state s of the form $(b, v'_1, \ldots, v'_m, i'_1, \ldots, i'_m, j')$ which describes the new status of modules $I_1, \ldots, I_m$. Note that in s, we continue to keep track of those modules which are busy in a parallel call. This is necessary for locally checking the fulfillment of the synchronized acceptance condition $\mathcal{P}_{sync}$.

The Büchi condition $\mathcal{F}$ of $\mathcal{S}$ is captured with the Büchi condition of $\mathbb{U}$ along with the use of finite counters implemented in the states. For the ease of presentation, we assume that $\mathcal{F}$ consists of a single accepting set F. Then, we use states of the form (v, i) to model a call v, where the counter i is used to check that linear paths (in the simulated monotone run of $\mathcal{S}$) containing infinite occurrences of calls satisfy the Büchi condition. In particular, it has default value 0 and is set to 1 if either $v \in F$ or a vertex in F is visited in the portion of the linear path from the last call before entering v. In the second case, the needed information is kept in the counters i_h of the states of the form $(b, v_1, \ldots, v_m, i_1, \ldots, i_m, j)$. Counter i_h has default value 0 and is set to 1 if a node in F is entered in the local computation of the corresponding module. Counter $j \in \{0, \ldots, m\}$ is instead used to check that the Büchi condition of $\mathcal{S}$ is satisfied for linear paths corresponding to infinite local computations (without nested calls) of the modules refining the box b. Moreover, in order to check that a node v_h corresponds to a terminal node (i.e., a node without linear successors in the simulated monotone run of

$\mathcal{S}$), $\mathbb{U}$ can choose nondeterministically to set the corresponding counter i_h to -1. Consistently, $\mathbb{U}$ will simulate only the internal moves from vertices $v_1, \ldots, v_m$ in which the module instance associated with v_h does not evolve. Thus, we have the following lemma.

Lemma 1. *For a call v, there is an accepting monotone maximal run of $\mathcal{S}$ from $\langle v \rangle$ iff there is an accepting run in $\mathbb{U}$ starting from $(v, 0)$.*

When the Büchi condition consists of $n > 1$ accepting sets, the only changes in the above construction concern the counters: we need to check that each set is met and thus the $0-1$ counters become counters up to n and the other counter is up to $m \cdot n$. Therefore, denoting $\rho = rank(\mathcal{S})$, n_V the number of vertices of $\mathcal{S}$ and n_δ the number of transitions of $\mathcal{S}$, we have that the number of $\mathbb{U}$ states is $O(\rho \cdot n^{\rho+1} \cdot n_V^{\rho+1})$ and the number of $\mathbb{U}$ transitions is $O(\rho^2 \cdot n^{2\rho+2} \cdot n_V \cdot (n_V + n_\delta)^\rho)$. Thus, by Proposition 1, Remark 1, and Lemma 1 we obtain the following result.

Lemma 2. *The problem of checking the existence of accepting monotone maximal runs in a* B-CRSM $\mathcal{S}$ *can be decided in time $O(\rho^4 \cdot n^{4\rho+4} \cdot (n_V + n_\delta)^{2\rho+2})$.*

4.2 The Emptiness Problem for Büchi *CRSM*

In this subsection, we show that the emptiness problem for Büchi *CRSM* can be reduced to check the existence of accepting monotone maximal runs. We fix a *B-CRSM* $\mathcal{S} = \langle (S_1, \ldots, S_k), start, \mathcal{F}, \mathcal{P}_F, \mathcal{P}_{sync} \rangle$. Moreover, n_V (resp., n_δ) denotes the number of vertices (resp., transitions) of $\mathcal{S}$. Also, let $\rho := rank(\mathcal{S})$.

For $F \subseteq V$, a *finite* path of $K_\mathcal{S}$ $\pi = q_0 \xrightarrow{(\ell_0, \ell'_0)} q_1 \xrightarrow{(\ell_0, \ell'_0)} \ldots q_n$ (with $q_i = (t_i, D_i)$ for any $0 \leq i \leq n$, and $t_0 = \{\varepsilon\}$) is *F-accepting* iff π satisfies the synchronized acceptance condition $\mathcal{P}_{sync}$ and all the linear paths of π starting from the local state $(0, \varepsilon)$ contain occurrences of local states (i, z) such that $D_i(z) \in F$. For a box $b \in B$, we say π is a *b-path* if $D_i(\varepsilon) = b$ for all $1 \leq i \leq n-1$.
We need the following preliminary result.

Lemma 3 (Generalized Reachability Problem). *Given $F \subseteq V$, the set of pairs (v, v') such that $v = (b, e_1, \ldots, e_m)$ is a call, $v' = (b, x_1, \ldots, x_m)$ is a matching return, and there is an F-accepting b-path from $\langle v \rangle$ to $\langle v' \rangle$, can be* computed *in time $O(n_V^2 \cdot 4^\rho \cdot (n_V + n_\delta)^\rho)$.*

Now, we show how to solve the emptiness problem for *B-CRSM* using the results stated by Lemmata 2 and 3. Starting from the *B-CRSM* $\mathcal{S}$ with $\mathcal{F} = \{F_1, \ldots, F_n\}$, we construct a new *B-CRSM* $\mathcal{S}'$ such that emptiness for $\mathcal{S}$ reduces to check the existence of accepting *monotone* maximal runs in $\mathcal{S}'$.

$\mathcal{S}' = \langle (S'_1, \ldots, S'_k), start, \mathcal{F}', \mathcal{P}'_f, \mathcal{P}'_{sync} \rangle$, with $\mathcal{F}' = \{F'_1, \ldots, F'_n\}$, is defined as follows. For $1 \leq i \leq k$, S'_i is obtained extending the set of nodes and the transition function of S_i as follows. For any call $v = (b, e_1, \ldots, e_m)$ of S_i and matching return $v' = (b, x_1, \ldots, x_m)$ such that there is a V-accepting b-path in $\mathcal{S}$ from $\langle v \rangle$ to $\langle v' \rangle$, we add two new nodes u^c_{new} and u^r_{new}, and the edge $(u^c_{new}, \perp, u^r_{new})$, where $\perp$ is a fresh non-synchronization symbol. We say u^c_{new}

(resp., u^r_{new}) is *associated* with the call v (resp., return v'). Moreover, for any edge in S_i of the form (u, σ, v) (resp., of the form (v', σ, u)) we add in S'_i the edge (u, σ, u^c_{new}) (resp., the edge (u^r_{new}, σ, u)). Also, for $1 \leq i \leq n$, if there is an F_i-accepting b-path from $\langle v \rangle$ to $\langle v' \rangle$, then we add u^r_{new} to F'_i (F'_i also contains all elements of F_i). Still, if $v' \in \mathcal{P}_f$, then we add u^r_{new} to $\mathcal{P}'_f$ ($\mathcal{P}'_f$ also contains all elements of $\mathcal{P}_f$). Note that $u^c_{new} \notin \mathcal{P}_f$. In fact, if an accepting maximal run of $\mathcal{S}'$ has a local state labelled by u^c_{new}, then the linear successor of this local state is defined and is labelled by u^r_{new}. Finally, $\mathcal{P}'_{sync}(v'_0, \{v'_1, \ldots, v'_m\})$ (with $m \leq rank(\mathcal{S})$) holds iff there are $v_0, \ldots, v_m \in V$ such that $\mathcal{P}_{sync}(v_0, \{v_1, \ldots, v_m\})$ holds and for all $0 \leq j \leq m$, either $v'_j = v_j$, or v_j is a return (resp., a call) and v'_j is a "new" node associated with it. Thus, we obtain the following result.

Lemma 4. *For any node u of $\mathcal{S}$, there is an accepting maximal run in $\mathcal{S}$ from $\langle u \rangle$ iff there is an accepting* monotone *maximal run in $\mathcal{S}'$ from $\langle u \rangle$.*

Note that the number of new nodes is bounded by $2n_V^2$, the number of new edges is bounded by $n_V \cdot n_\delta + n_V^2$, and by Lemma 3, $\mathcal{S}'$ can be constructed in time $O(|\mathcal{F}| \cdot n_V^2 \cdot 4^\rho \cdot (n_V + n_\delta)^\rho)$. Thus, by Lemmata 2 and 4 we obtain the main result of this section.

Theorem 1. *Given a* B-CRSM $\mathcal{S}$*, the problem of checking the emptiness of $\mathcal{S}$ can be decided in time $O\big((|\mathcal{F}| \cdot (n_V + n_\delta))^{O(\rho)}\big)$.*

5 Model Checking *CRSM* Against ConCaRet

In this section, we solve the model-checking problem of *CRSM* against ConCaRet using an automata-theoretic approach: for a *CRSM* $\mathcal{S}$ and a ConCaRet formula φ, we construct a *B-CRSM* $\mathcal{S}_\varphi$ which has an accepting maximal run iff $\mathcal{S}$ has a maximal run that satisfies φ. More precisely, an accepting maximal run of $\mathcal{S}_\varphi$ corresponds to a maximal run π of $\mathcal{S}$ where each local state is equipped with the information concerning the set of subformulas of φ that hold at it along π.

The construction proposed here follows and extends that given in [2] for CaRet. The extensions are due to the presence of the branching-time modalities and the parallel operator $\|$. For branching-time modalities we have to ensure that the existential (resp. universal) next requirements are met in some (in each) linear successor of the current local state. This is captured locally in the transitions of $\mathcal{S}_\varphi$. Parallel formulas are handled instead by the synchronization predicate.

The generalized Büchi condition is used to guarantee the fulfillment of liveness requirements φ_2 in until formulas of the form $\varphi_1 \, \mathcal{U}^b \, \varphi_2$ where $b \in \{a, \exists, \forall\}$ (caller-until formulas do not require such condition since a caller-path is always finite). For existential until formulas φ', when φ' is asserted at a local state (i, y), we have to ensure that φ' is satisfied in at least one of the linear paths from (i, y). In order to achieve this and ensure the acceptance of all infinite linear paths from (i, y) we use a fresh atomic proposition $\tau_{\varphi'}$.

For every vertex/edge in $\mathcal{S}$, we have $2^{O(|\varphi| \cdot rank(\mathcal{S}))}$ vertices/edges in $\mathcal{S}_\varphi$. Also, the number of accepting sets in the generalized Büchi conditions is at most $O(|\varphi|)$ and $rank(\mathcal{S}_\varphi) = rank(\mathcal{S})$. Since there is a maximal run of $\mathcal{S}$ satisfying formula φ

iff there is an accepting maximal run of $\mathcal{S}_\varphi$, model checking $\mathcal{S}$ against φ is reduced to check emptiness for the Büchi *CRSM* $\mathcal{S}_{\neg\varphi}$. For $rank(\mathcal{S}) = 1$, the considered problem coincides with the model checking problem of *RSM* against CaRet that is Exptime-complete (even for a fixed *RSM*). Therefore, by Theorem 1, we obtain the following result.

Theorem 2. *For a* CRSM $\mathcal{S}$ *and a formula* φ *of* ConCaRet*, the model checking problem for* $\mathcal{S}$ *against* φ *can be decided in time exponential in* $|\varphi|\cdot(rank(\mathcal{S}))^2$*. The problem is* Exptime*-complete* (*even when the* CRSM *is fixed*).

References

1. R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. Reps, and M. Yannakakis. Analysis of recursive state machines. *To appear in ACM Transactions on Programming Languages and Systems*, 2005.
2. R. Alur, K. Etessami, and P. Madhusudan. A Temporal Logic of Nested Calls and Returns. In *Proc. of TACAS'04*, pp. 467–481, 2004.
3. R. Alur, S. Kannan, and M. Yannakakis. Communicating hierarchical state machines. In *Proc. of ICALP'99*, LNCS 1644, pp. 169–178, 1999.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proc. of CONCUR'97*, LNCS 1243, pp. 135–150, 1997.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proc. of POPL'03*, pp. 62–73, 2003.
6. A. Bouajjani and T. Touili. Reachability Analysis of Process Rewrite Systems. In *Proc. of FSTTCS'03*, LNCS 2914, pp. 74–87, 2003.
7. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems. In *Proc. of CONCUR'05*, LNCS 3653, pp. 473–487, 2005.
8. W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14: pp. 217–231, 1969.
9. J. Esparza, and M. Nielsen, Decidability Issues for Petri Nets.In J. Inform. Process. Cybernet., EIK 30 (1994) 3, pp. 143–160.
10. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural parallel flow graphs. In *Proc. of FoSSaCS'99*, LNCS 1578, 1999.
11. J. Esparza and A. Podelski. Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs. In *Proc. of POPL'00*, pp. 1–11, 2000.
12. C. Löding. Infinite Graphs Generated by Tree Rewriting. Doctoral thesis, RWTH Aachen, 2003.
13. R. Mayr. Process Rewrite Systems. In *Information and Computation*, Vol. 156, 2000, pp. 264–286.
14. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *Proc. of TACAS'05*, LNCS 3440, pp. 93–107, , 2005.
15. W. Thomas. Automata on infinite objets. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, Vol. B, pp. 133–191, 1990.
16. M.Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, 1986.
17. I. Walukiewicz. Pushdown processes: Games and Model Checking.In *Int. Conf. on Computer Aided Verification*, LNCS 1102, pages 62–74. Verlag, 1996.
18. URL: www.dia.unisa.it/professori/latorre/Papers/concaret.ps.gz