Syntax-Driven Reachable State Space Construction of Synchronous Reactive Programs

Eric Vecchié and Robert de Simone

INRIA Sophia-Antipolis, France

Abstract. We consider in the current paper the issue of exploiting the structural form of Esterel programs [BG92] to partition the algorithmic RSS (reachable state space) fix-point construction used in model-checking techniques [CGP99]. The basic idea sounds utterly simple, as seen on the case of sequential composition: in $P;Q$, first compute *entirely* the states reached in P, and then only carry on to Q, each time using only the relevant local transition relation part. Here a brute-force symbolic breadth-first search would have mixed the exploration of P and Q instead. The introduction of parallel (state product) operators, as well as loop iterators and local synchronizing signals make the problem more difficult (and more interesting). We propose techniques to partition statically ("at compile time") the program body, so as to obtain a good trade-off between locality and multiplicity of steps.

1 Introduction

In the last decade the advent of BDD-based implicit state-space representation [Bry86] allowed to scale up various analysis techniques to large realistic synchronous reactive system designs. But BDDs alone cannot be relied upon to cope with all the complexity of the reachable state space construction. Specifically, while the BDD encoding of the final reachable state space may often be very compact, the transition relation and the intermediate steps of next-state computations can be exceedingly larger. Several clever techniques for partitioning the application of transition functions have been proposed, which partially solve the problem [BCL91,BCL$^+$94,HD93,ISS$^+$03]. In the context of Esterel we propose to use the structural syntactic nature of the design to apply transition relations piecewise, only when it may provide further states. Intuitively in a sequential composition $P;Q$ one clearly wants to compute *all* reachable states in P first, then progress to states in Q. While this may seem a trivial idea at first (after all, reachable state space construction can be seen as exhaustive symbolic simulation of all behaviors), care has to be taken, specially in presence of parallel components and internal signal communications, so that the approach retains some of the advantages of symbolic approach, namely that all individual behaviors are not enumerated (or not even nearly so). This is a typical time/space trade-off. Still, using the algorithmic structure of Esterel programs to guide (symbolic, exhaustive, breadth-first search) state space construction is a clear, simple idea that was never tried out before to the best of our knowledge. Other

K. Etessami and S.K. Rajamani (Eds.): CAV 2005, LNCS 3576, pp. 213–225, 2005.

works with similar concern usually attempt to precede the symbolic breadth-first search with partial explicit depth-first search simulations that identify new initial configurations "ahead" in the potential behaviors [GB94,PP03].

In essence our refined algorithm proceeds as follows : initially a very restricted transition relation is applied, with many locations of (internal or external) signal receptions "blocked". Then those signal reception occurrences are progressively "re-allowed", in a heuristically ordered fashion. Some transitions can be blocked again in order to deal with loop constructs but in the general case, as the new extensions are always applied to "most recent" states, the old and already largely searched parts get "cleaned up" by some simplification properties of the `TiGeR` BDD package [CMT93], which "cofactors" out the transition parts found to lay outside the domain of states they are applied to. This operation simplifies drastically the support (i.e, the set of variables that the relation effectively depends upon), and thus the computations. Heuristics for ordering the "reception allowances" are based on a graph structure extracted from the structural syntax, so that it is compliant with the natural precedence that may exist (for instance, when a reception on S causes the emission on T otherwise also expected, it is obviously better to release S before T).

The paper is organized as follows : first we give a brief summary of (a restricted micro-subset of) ESTEREL, as well as technical elements of symbolic model-checking. We focus on how the `TiGeR` BDD package [CBM89] performs transition partitioning and "transition cofactoring" in order to decrease the size of data structures (and optimize the variables support) when applying the next-state computation. These techniques will come handy later on to understand ours. Then we provide a description of our approach with the actual algorithm and its BDD implementation, relying on the already mentioned features of `TiGeR`. We justify the correctness of our partitioned approach to build the full RSS. We close with the description of our prototype implementation and performance benchmarks.

2 Context

ESTEREL is an imperative *synchronous reactive language*. We shall only consider here a simple version, where data variables and data-handling are discarded, as often in model-checking. We shall thus only use *Signals* as (identifier) types. A full program consists of a header (where an interface of *input* and *output* signals are defined), followed by a body. Syntax of program statements is provided by the following simple grammar :

$P ::=$ `pause`	$\mid P \parallel P$	$\mid$ `present` S `then` P `else` P `end`
$\mid P \,;\, P$	$\mid$ `emit` S	$\mid$ `abort` P `when` S
$\mid$ `loop` P `end`	$\mid$ `signal` S `in` P `end`	

with S ranging over signals.

Naive semantics of ESTEREL goes as follows : programs behaviors are discretely divided between instants. Control threads are executed until reaching a

`pause` statement, which is the main statement which cuts behaviors into atomic instants. We call "reaction" the full behavior performed during a given instant. In a reaction *cycle*, input signals are read/sampled, and internal computation takes place until output signals are emitted in answer, and the program state is progressed. Instants are based on a *common* logical clock, which paces all parallel threads. This (the fact that all components proceed with the same atomic steps of instants) is why we call the model "*synchronous*". Of course in a reaction various parallel threads do **not** run independently, as they may synchronize and affect one another causally (hardware people would say "combinationally"). When control reaches a `present S` test statement, it may have to postpone execution until a consistent definitive value (present or absent) is obtained for the signal inside the current reaction (either because it is emitted somewhere in parallel, or because other threads of execution provably progressed to a point where provably *all* potential emissions were discarded).
While being a high-level imperative language, ESTEREL enjoys a semantic-preserving translation to hardware RTL level (net-lists) where causality issue can be more readily dealt with, and a second level of interpretation into Mealy FSMs (again semantically sound). This second level actually looses information on fine causality issues, but makes explicit the actual reachable state space, and thus can be the definitional background for model-checking analysis techniques. Of course the purpose of implicit (or symbolic) BDD-based model-checking is to apply these analysis at the circuit level. In our case we try to lift them some more by exploiting high-level structuring information from the source syntax.

Symbolic next-state operation. Starting from the initial state ι, the basic breadth-first search Reachable State Space algorithm can be written:

Algorithm 1.1. Breadth-first search algorithm

```
1  reachable ← ι
2  new ← ι
3  while ( new ≠ ∅ ) do
4      new ← Image_Δ(new) ⧵ reachable
5      reachable ← reachable ∪ new
6  end while
```

The set of states reached at the n^{th} iteration is built from the set of states reached at the $(n-1)^{\text{th}}$ iteration and the set of valid inputs of the program, by computing the image under a transition relation Δ. The algorithm stops when no new state can be found. Each state of the program is a valuation of the set R of boolean registers of the circuit and each input of the program is a valuation of the set I of input signals. The unique global transition relation Δ let us compute the new states of the program with respect to the value of I and R :

$$\begin{aligned} \Delta \ : \ \mathbb{B}^m \times \mathbb{B}^p &\to \mathbb{B}^p \\ (I, R) &\to R' = \Delta(I, R) \end{aligned}$$

where $\mathbb{B} = \{0, 1\}$, m is the number of input signals and p is the number of registers of the circuit. In fact Δ can be "partitioned" and decomposed into a

vector of functions δ_i, where each δ_i concerns a different image register, and depends only on a subset of the source registers and of the input signals :

$$\begin{aligned} \delta_i : \quad & \mathbb{B}^{m_i} \times \mathbb{B}^{p_i} \to \mathbb{B} \\ & (I_i, R_i) \to r'_i = \delta_i(I_i, R_i) \end{aligned}$$

Vectors I_i and R_i are called the support of these transition functions. m_i and p_i are respectively the number of input signals and the number of registers of this support. Such a partitioning scheme is used to speed up applications of BDDs representing the individual δ_i [BCL91].

Extended cofactoring methods. We shall extensively use some well-known BDD transformations, known in general as *extended cofactoring techniques* [Cou91]. In essence the principle is that, if the value of the BDD is only relevant on a subset of the possible valuations of its variables, then this restricted domain of definition can be used to simplify the expression of the BDD (possibly changing its value outside of it). Generally the domain is itself provided as a BDD. We note $f_{\downarrow S}$ the cofactoring of f by the set S :

$$f_{\downarrow S}(X) = \lambda X \to \begin{cases} f(X) & \text{if } X \in S \\ ? & \text{if } X \notin S \end{cases}$$

The value of $f_{\downarrow S}$ out of S is not used and can be anything. It is set in order to minimize the size of the BDD representing $f_{\downarrow S}$. In our algorithm, this operator is used in the **Image** function. It lets us handle smaller BDDs during the image computation since the transition relation is reduced with respect to the domain it is applied on. More precisely, given a register r, if the activation condition of r (the set of states for which $r = 1$) and the domain of the transition relation are disjoint, then the transition function of r can be reduced to a very simple expression $\lambda X \to \neg\, r$. In other words, the BDD encoding the transition function of registers that will not be activated in the next instant is very small.

3 Partitioned Algorithm

Our partitioned algorithm consists in performing each step of the reachable states exploration in a reduced number of program blocks. State search will be performed inside each block until stabilization, before moving to the next one ; this algorithm is an adaptation of the algorithm 1.1. The BDD area represents the set of all states (reachable or not) lying inside the program blocs we are focusing. At each step of the algorithm, the cofactored image computation is performed only on the pending reachable states lying inside area (line 8). At the end of each step, the new-found states are stored in the pending set (line 9). area is left unchanged as long as new states are found inside it (lines 5, 6, 7).

This algorithm does not describe the evolution of area (this will be developed in section 4).

Algorithm 1.2. Partitioned algorithm

```
1  reachable ← ι
2  pending ← ι
3  area ← area_0        /* area_0 : see algorithm 1.3 */
4  while ( pending ≠ ∅ ) do
5      if ( (pending ∩ area) = ∅ ) then
6          area ← area′        /* area′ : see algorithm 1.4 */
7      end if
8      new ← Image_Δ(pending ∩ area) ∖ reachable
9      pending ← (pending ∖ area) ∪ new
10     reachable ← reachable ∪ new
11 end while
```

Partitioning into "macro-states" according to syntax. At the heart of the method is the division of the program body into blocks (or macro-states) of proper granularity. To disallow search in given blocks, one needs only to remove the part of the transition relation where all registers of these blocks are inactive. The bloc division of course relies heavily on the structural syntax, and mostly on signal receptions (as in `abort P when S`) and, to a lesser extent, on signal emissions. We use a control flow graph data structure to help us with this task. We shall stick to the classical translation from ESTEREL to circuits described in [Ber99], which generates exactly one boolean register for each `pause` statement. In the sequel we shall consider an abstract syntax tree version for ESTEREL programs where `pause` constructs are explicitly labeled by the corresponding register names, providing the necessary association. In fact, we want to recognize each instance of instruction that we identify here with a unique label mentioned as exponent. Each node of the tree is typed with respect to the instruction it represents. Thus, the tree node of an instruction of type **instruction** and labeled by L is written :

$$(\mathbf{instruction}^{L}\ subtree_1{}^{l_1}\ \ldots\ subtree_n{}^{l_n})$$

The control flow graph of a given syntax tree T is defined as follows : $\mathsf{G}(\mathsf{T}) = (\mathsf{I}, \mathsf{O}, \mathsf{N}, \mathsf{E}, \mathsf{F})$ where N is the set of the nodes of the graph. These nodes are the same as those of the syntax tree. I and O are subsets of N and represent respectively the start and final nodes of the graph. The edges of our graph (written $i \mapsto j$) are divided into two categories : E contains "normal" edges and F contains the edges used as frontiers. By construction, the set $\mathsf{E} \cap \mathsf{F}$ is empty. Thus, edges corresponding to `present` and `abort` statements are settled in F. Such edges are called "frontier" edges. Other edges are settled in E.

We describe here the way we build our control flow graph for each ESTEREL instruction. This description uses labels of the syntax tree which are a lighter way to identify the nodes. The usual operator " $\times$ " allows us to join each element of a set $I = \{I_1, \ldots I_m\}$ to each element of a set $J = \{J_1, \ldots J_n\}$.

In this section, we suppose that an instruction I produces a graph $\mathsf{G}(I) = (\mathsf{I}, \mathsf{O}, \mathsf{N}, \mathsf{E}, \mathsf{F})$. As well, for $i \in [1,2]$ we have $\mathsf{G}(I_i) = (\mathsf{I}_i, \mathsf{O}_i, \mathsf{N}_i, \mathsf{E}_i, \mathsf{F}_i)$. Atomic instructions produce graphs containing a single node and no edge :

$$\mathsf{G}(\mathbf{emit}^L\ s) = (\{L\}, \{L\}, \{L\}, \varnothing, \varnothing)$$
$$\mathsf{G}(\mathbf{pause}^L\ r) = (\{L\}, \{L\}, \{L\}, \varnothing, \varnothing)$$

In our graph, we can abstract the beginnings and the ends of the scope. The graph of a local signal declaration is thus the same as for I :

$$\mathsf{G}(\mathbf{signal}^L\ s\ I\ \mathbf{end}^{L'}) = \mathsf{G}(I)$$

Choice operator. Consider a `present S then` P `else` Q `end` statement. If the reachable state space is computed in a breadth-first search manner on a global transition relation, then states in P and Q will be considered at the same time. In this case the intermediate symbolic description is likely to be larger than the final one, if one grants that intermediate forms of partially reached state spaces are more irregular than final ones. Moreover, the sequentially partitioned state space search here allows to use only the relevant part of the transition relation when dealing with each component (P, then Q). Frontiers are thus placed before and after the "then" branch and the "else" branch.

$$\mathsf{G}(\mathbf{present}^L\ s\ I_1\ I_2\ \mathbf{end}^{L'}) = (\{L\}, \{L'\}, \mathsf{N}_1 \cup \mathsf{N}_2 \cup \{L, L'\}, \mathsf{E}_1 \cup \mathsf{E}_2, \mathsf{F}')$$
$$\text{where} \quad \mathsf{F}' = \mathsf{F}_1 \cup \mathsf{F}_2 \cup (\{L\} \times (\mathsf{I}_1 \cup \mathsf{I}_2)) \cup ((\mathsf{O}_1 \cup \mathsf{O}_2) \times \{L'\})$$

Preemption. An `abort` P `when S` statement allows to add abortive transitions to the natural terminations of P. Our partitioning technique will aim at exploring fully P before exploring the next program blocks activated by P's terminations (of course this will have the effect of blocking also the potential emissions causing the abort, that would figure in the same global transition). Therefore, we want to consider each transition exiting P as frontier. Each `pause` instruction may lead to the end of the `abort` instruction that encloses it. Thus :

$$\mathsf{G}(\mathbf{abort}^L\ s\ I\ \mathbf{end}^{L'}) = (\mathsf{I}, \{L'\}, \mathsf{N} \cup \{L'\}, \mathsf{E}, \mathsf{F} \cup \mathsf{F}')$$
$$\text{where} \quad \mathsf{F}' = (\mathsf{O} \cup \{l \ /\ (\mathbf{pause}^l\ r) \in \mathsf{N}\}) \times \{L'\}$$

Sequence statement. Partitioning a $P;Q$ sequence statement is a waste of energy. If P is a constant-length program like `pause;pause` then the partitioning of $P;Q$ is naturally performed by the breadth-first search algorithm. Variable-length programs are already partitioned since containing `present` or `abort` statements.

$$\mathsf{G}(\mathbf{seq}^L\ I_1\ I_2\ \mathbf{end}^{L'}) = (\mathsf{I}_1, \mathsf{O}_2, \mathsf{N}_1 \cup \mathsf{N}_2, \mathsf{E}', \mathsf{F}_1 \cup \mathsf{F}_2)$$
$$\text{where} \quad \mathsf{E}' = \mathsf{E}_1 \cup \mathsf{E}_2 \cup (\mathsf{O}_1 \times \mathsf{I}_2)$$

Parallel networks and signal synchronizations. The problem here is to establish which blocks put in parallel can be active in parallel, so that the global search can be divided with matching progressions. This is shown in figure 1. The

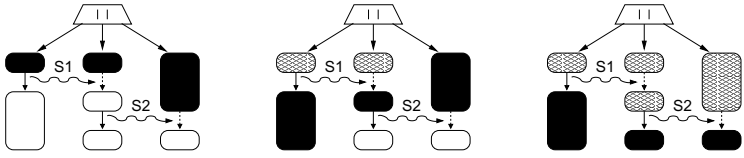

Fig. 1. *Partitioning method for a parallel component. There are two signals synchronizing three parallel components. Our technique aims at partitioning according to the black-colored blocks. Hatched blocks should be removed by cofactoring methods*

only syntactic element at our disposal here to indicate synchronization will of course be signal reception. These receptions must be matched by corresponding emissions when signals are local (otherwise receptions of input signals can occur anytime, but each parallel component must perceive it consistently). Nevertheless it should be noted that, in the synchronous reactive framework, *it is possible* that a local signal emission causes no reception, if none are "actively watching" at the time. So, while we shall use signal receptions to generate frontier transitions, these will automatically generate simultaneous frontiers at *emit* side **when they are enabled**, and otherwise emissions can be passed and go unsynchronized. To clarify further, consider the following simple example : P_1`; emit S;` P_2 `||` Q_1`; await S;` Q_2. If the design of this program is so that any emission of `S` is received by the `await S` statement, then P_2 can not be active if Q_2 is not. Thus partitioning according to Q_1 and Q_2 will partition the first branch according to P_1 and P_2 as well. If some emissions of `S` are not received, then partitioning according to Q_1 and Q_2 will have no precise effect on the first branch. In all case there is a real benefit in partitioning this way. In the best case, the reachable state space computation will concern P_1 and Q_1 first and then, P_2 and Q_2. In the worst case, it will concern P_1, P_2 and Q_1 and then, P_2 and Q_2.

$$\mathsf{G}(\mathbf{par}^{L}\ I_1\ I_2\ \mathbf{end}^{L'}) = (\{L\}, \mathsf{O}_1 \cup \mathsf{O}_2, \mathsf{N}_1 \cup \mathsf{N}_2 \cup \{L\}, \mathsf{E}', \mathsf{F}_1 \cup \mathsf{F}_2)$$
$$\text{where} \quad \mathsf{E}' = \mathsf{E}_1 \cup \mathsf{E}_2 \cup (\{L\} \times (\mathsf{I}_1 \cup \mathsf{I}_2))$$

Loops. In loop constructs a new difficulty arises : whether blocks can be truly concurrent is in general only known dynamically (this is in a large part why RSS construction can be so hard). Loops are the only constructs in which we want to lock frontiers during state space exploration. In ESTEREL programs, registers which are not running in parallel cannot be active at the same time. We can use this static information in order to deactivate registers in loop constructs. Thus, each time a register r is activated we shall deactivate the set of registers incompatible with r and belonging to the same loop as r. We call $Lock\,(r)$ such a set which of course can be refined at will. The graph of a loop statement is the following :

$$\mathsf{G}(\mathbf{loop}^{L}\ I\ \mathbf{end}^{L'}) = (\mathsf{I}, \varnothing, \mathsf{N}, \mathsf{E} \cup \mathsf{E}', \mathsf{F}) \quad \text{where} \quad \mathsf{E}' = \mathsf{O} \times \mathsf{I}$$

Frontier ordering. Currently, the order in which frontiers will be unlocked is defined dynamically, "at run time" during the course of our successive fix-point iterations searching new states in growing support domains. We select each time a frontier that is likely to produce new states, and is not strictly preceded by another one. This relies deeply on the shape of a *pending* set of states that are incompletely processed, and can generate configurations beyond the current frontiers. Details shall be provided in section 4.

This partial order is statically refined according to the syntax of the programs. This static order written "$\prec$" is a guarantee that frontiers will not be opened prematurely. The statement "a frontier $\mathfrak{x}$ should be opened before a frontier $\mathfrak{y}$" is written $\mathfrak{x} \prec \mathfrak{y}$. In fact, defining a static order between frontiers consists in defining an order between the target nodes of the frontiers. Thus, if $\mathfrak{u}$ and $\mathfrak{v}$ are two nodes, $\mathfrak{u} \prec \mathfrak{v}$ means that any frontier leading to $\mathfrak{u}$ should be opened before any frontier leading to $\mathfrak{v}$:

$$\mathfrak{u} \prec \mathfrak{v} \Longleftrightarrow (\mathfrak{x} \mapsto \mathfrak{u}) \ \prec \ (\mathfrak{y} \mapsto \mathfrak{v}) \quad \forall \mathfrak{x}, \forall \mathfrak{y}$$

The definition of "$\prec$" is purely syntactic. In a sequence $(\mathbf{seq}^L \ I_1 \ I_2 \ \mathbf{end}^{L'})$, one wants to open frontiers in I_1 before frontiers in I_2. Thus we have $\mathsf{N}_1 \prec \mathsf{N}_2$.

In an $(\mathbf{abort}^L \ s \ I \ \mathbf{end}^{L'})$ statement, one wants to open frontiers inside I before frontiers leading outside I. This can be written $\mathsf{N} \prec L'$.

4 The Precise Algorithm and Its BDD Implementation

We shall introduce useful notations. Given a set $\mathcal{R} = \{r_1, \ldots r_n\}$ of BDD variables, we introduce the operator :

$$\mathit{NOr}(\mathcal{R}) = \lambda X \rightarrow \neg\,(r_1 \vee \ldots \vee r_n)$$

If $r_1, \ldots r_n$ are variables representing boolean registers $R_1, \ldots R_n$ then $\mathit{NOr}(\mathcal{R})$ represents the set of states in which all registers R_i are inactive for all $i \in [1..n]$. We notice that $\mathit{Or}(\mathcal{R}) = \overline{\mathit{NOr}(\mathcal{R})}$ represents the set of states in which at least one register R_i is active for $i \in [1..n]$. Given a set X of graph nodes, we introduce the operator $\mathit{Register}\,(\mathsf{X})$ which returns the set of register BDD variables in X :

$$\mathit{Register}\,(\mathsf{X}) = \{r \ \mid \ (\mathbf{pause}\ r) \in \mathsf{X}\}$$

This operator will help us to make the link between our control flow graph and the symbolic BDD-based computations. Source and target node of an edge $\mathfrak{u} \mapsto \mathfrak{v}$ are written :

$$\mathit{Src}(\mathfrak{u} \mapsto \mathfrak{v}) = \mathfrak{u} \quad \text{and} \quad \mathit{Dest}(\mathfrak{u} \mapsto \mathfrak{v}) = \mathfrak{v}$$

Given a "classical" directed graph $(\mathsf{N}\ ,\ \mathsf{E})$, we write :

$$\mathit{Succ}_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) = \{j \in \mathsf{N} \ \mid \ i \in \mathsf{X} \ \wedge \ i \mapsto j \in \mathsf{E}\}$$

the set of target nodes of edges of E whose source belongs to X and we write :

$$Out_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) = \{i \mapsto j \in \mathsf{E} \mid i \in \mathsf{X}\}$$

the set of edges of E whose source belongs to X. The operator :

$$Closure_{(\mathsf{N},\mathsf{E})}(\mathsf{Y}) = \mu(\lambda \mathsf{X} \to \mathsf{Y} \cup Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X}))$$

represents the set of nodes reachable from Y through edges in E. The following operator computes the "surface" of a program block. Given a set $\mathsf{Y} \subseteq \mathsf{N}$ of nodes (corresponding to a set of active registers), the surface is the set of edges that can be crossed in the immediate instant following the activation of one or more registers in Y. If P is the set of nodes of type "pause", then :

$$Surface_{(\mathsf{N},\mathsf{E})}(\mathsf{Y}) = \mu(\lambda \mathsf{X} \to \mathsf{Y} \cup (Succ_{(\mathsf{N},\mathsf{E})}(\mathsf{X}) \setminus \mathsf{P}))$$

Given a set $\mathcal{R} = \{r_1, \ldots r_n\}$ of registers, we write :

$$Lock(\mathcal{R}) = Lock(r_1) \cap \ldots \cap Lock(r_n)$$

the set of registers which we want to deactivate when all $r_1, \ldots r_n$ are activated.

4.1 Graph-Guided Algorithm

In this section, we describe the evolution of the set area in the algorithm 1.2 with respect to the control flow graph. We assume that the syntax tree of the analyzed program is given in T.

Control flow graph and restricted area initializations. The initialization process consists in building the graph to obtain an initial set of locked edges and then build the set area_0 with respect to these initial conditions.

Algorithm 1.3. Initialization of area_0

1 $(\mathsf{I}, \mathsf{O}, \mathsf{N}, \mathsf{E}, \mathsf{F}) \leftarrow \mathsf{G}(\mathsf{T})$
2 $\mathtt{inner} \leftarrow Closure_{(\mathsf{N},\mathsf{E})}(\mathsf{I})$
3 $\mathfrak{R} \leftarrow Register(\mathsf{N})$, $\mathfrak{R}^+ \leftarrow Register(\mathtt{inner})$
4 $\mathsf{area}_0 \leftarrow NOr(\mathfrak{R} \setminus \mathfrak{R}^+)$

The first step consists in building the graph (line 1). Then, we need to know the set $\mathfrak{R}^+$ of registers which are allowed to be active (line 3). Finally, area is defined as the set of states such that no register but those in $\mathfrak{R}^+$ is active (line 4).

Restricted area enlargement. When area is required to be enlarged, we want to unlock "good" edges. We only want to unlock edges which allow us to include some pending states inside the growing area set. Such edges can only be found in the surface of inner (line 1) and are sorted according to "$\prec$" (line 2). Furthermore, more than one edge may be required to be unlocked. This is the typical case where two parallel branches are awaiting the same signal. Thus, while no pending state lies inside area, a new edge is analyzed in order to decide whether it should be unlocked or not.

Algorithm 1.4. Enlargement of area'

```
1  surface ← Surface_(N,E∪F) (inner)
2  frontier ← Sort_≺(Out_(N,F)(surface))
3  i ← 1
4  while ( (pending ∩ area) = ∅ ) do
5      f ← frontier[i]
6      /* check if f should be opened, see algorithm 1.5 */
7      if ( open? ) then
8          /* open f, see algorithm 1.6 */
9          /* close some frontiers, see algorithm 1.7 */
10     end if
11     i ← i + 1
12 end while
```

Edge crossing. To determine whether an edge should be unlocked, one has to focus on the new active registers in the set pending.

Algorithm 1.5. Crossing a frontier

```
1  inner^new ← Closure_(N,E) (Dest(f))
2  ℜ^new ← Register (inner^new) ∖ ℜ^+
3  open? ← false
4  if ( ℜ^new = ∅ ) then
5      open? ← true
6  else if ( pending ∩ Or(ℜ^new) ≠ ∅ ) then
7      open? ← true
8  end if
```

First, we compute the set of nodes in the graph that would be reached if the edge f was unlocked. We just need to know the new-found registers which are stored in $\mathfrak{R}^{new}$ at line 2. If f leads to no register, it can be unlocked but this will have no effect on the set area (line 4, 5). If $\mathfrak{R}^{new}$ is not empty, we check if there are some states in pending that have activated one or more new registers contained in $\mathfrak{R}^{new}$ (line 6, 7). In this case, the edge can be unlocked.

Unlocking an edge. Once an edge has been decided to be unlocked, we just have to perform the following updates : first, the unlocked edge is moved from F to E. Then, the set area is enlarged.

Algorithm 1.6. Opening a frontier

```
1  E ← E ∪ {f}, F ← F ∖ {f}
2  inner ← inner ∪ inner^new, ℜ^+ ← ℜ^+ ∪ ℜ^new
3  area' ← NOr(ℜ ∖ ℜ^+)
```

Locking some edges. Finally we close some edges to deal with loop constructs. In this algorithm, graph updates have been discarded.

Algorithm 1.7. Closing frontiers

1 $\mathfrak{R}^+ \leftarrow \mathfrak{R}^+ \setminus Lock\,(\mathfrak{R}^{new})$
2 ...

4.2 Correctness Arguments (Hints)

Formally, one should prove that our new partitioned technique computes the same RSS as the global one. But the correctness assumption relies on a simple argument, that we shall state only informally.

In the last iteration of the algorithm's main loop, the (ever-growing) transition relation will be the global one, as used in the classical single iteration breadth-first search. But it is only applied to a selection of new initial states (those taken from the temporary pending sets), and thus will reach only *all states* reachable from there. But older states were only discarded from the pending potential new state generators when all theirs successors were produced (because they could be so in a more restricted transition relation form. So it is harmless not to consider them any longer.

5 Experimental Results

The results presented here have been obtained by executing our program on a Bi-Pentium III - 550 MHz with 1 GByte of memory and running under the Linux operating system. The memory was limited to 900 MBytes in order to avoid the use of disk swap. These results have been obtained without closing frontiers in loop constructs.

We implemented our method with the help of the `TiGeR` BDD package and we tested it on numerous ESTEREL designs. Still, many were small programs which primarily helped us validate our implementation. Results here are not so significant since memory consumption is not an issue, as intermediate BDDs blow-ups are very limited. Figure 2 presents experimental results obtained on pretty big ESTEREL designs. Concerning computation time, our method was slower on the `sequencer` example as expected since more iterations are required to reach RSS completion. But, surprisingly it appeared to win on bigger designs (`mmid`, `sat`). This is so since each iteration step works on much smaller objects (BDD DAGs). We still need more experiments to be fully conclusive on our findings.

6 Conclusion

To the best of or knowledge our method is the only partitioning method based on syntactic {*sequential/alternative/parallel/synchronized*} structural information drawn from (synchronous) programs. Our method tends to mimic the behavioral progression of control through time, but in a context where all paths

Program		Steps	Found states	Crossed states	Memory	Time
`globalopt`	def.	3	342 858 276 099	583 065 603	$> 900M$	$34m40s$
598 regs.	part.	80	**705 085 932 547**	**5 542 740 483**	$> 900M$	$26h45m32s$
`site`	def.	3	232 705 179	1 049 601	$> 900M$	$22m51s$
308 regs.	part.	91	**2 380 837 289**	**452 110 875**	$> 900M$	$9h58m45s$
`cabin`	def.	3	13 321	534	$> 900M$	$14m22s$
919 regs.	part.	147	**719 031 955**	**484 744 348**	$> 900M$	$18h54m29s$
`sequencer`	def.	18	122 597	all	$40\,359K$	$\mathbf{3m47,22s}$
154 regs.	part.	145	122 597	all	$\mathbf{17\,022K}$	$8m56,59s$
`mmid`	def.	13	10 308 357	all	$205\,214K$	$45m59s$
111 regs.	part.	113	10 308 357	all	$\mathbf{42\,368K}$	$\mathbf{19m38}$
`chorusBin`	def.	6	16 928 480	441 417	$> 900M$	$5h39m35s$
92 regs.	part.	79	**136 329 824**	**all**	$851\,369K$	$238h10m45s$
`cdtmica`	def.	10	12 538 388 785	10 651 674 353	$> 900M$	$15h24m46s$
208 regs.	part.	185	**23 384 736 769**	**all**	$748\,971K$	$36h31m23s$
`steam`	def.	3	3 865 747 524	396 566 399	$> 900M$	$48m36s$
128 regs.	part.	101	**41 774 141 026**	**all**	$762\,153K$	$25h30m21s$
`sat`	def.	17	43 487 202 056	17 566 150 006	$> 900M$	$6h42m50s$
192 regs.	part.	339	**35 740 420 392 968**	**all**	$\mathbf{77\,797K}$	$\mathbf{3h00m56s}$

Fig. 2. *Comparison between the default and the partitioned method : the first column* (Steps) *is the number of computation steps achieved with success, the second column* (Found states) *is the number of found states, the third column* (Crossed states) *is the number of states whose image has been successfully computed, the forth column* (Memory) *shows the memory required and the fifth column* (Time) *shows the computation times.*

have to be followed (exhaustive search, as opposed to single path simulation). We presented a solution to partition the RSS computation, primarily according to signal receptions, and then order the evaluation of blocks according to progression of control. This latter information is drawn from a control-flow graph, itself directly extracted from the abstract syntax tree. The graph is also used to actually build the precise transition relation selected at any given macro-step, by including the parts where registers enclosed inside proper frontiers are found. Frontiers are progressively expanded, in a hopefully "sensible" order, so that all reachable states can be captured. Sometimes, frontiers are closed in order to deal with loop constructs as if they were "unrolled". This method provides good experimental results showing the relevance of the approach.

References

[BCL91] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[Ber99] Grard Berry. *The Constructive Semantics of Pure Esterel*. INRIA, 1999. http://www-sop.inria.fr/esterel.org/.

[BG92] Grard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Sci. of Comput. Program.*, 19(2):87–152, 1992.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[CMT93] Olivier Coudert, Jean-Christophe Madre, and Herv Touati. TiGeR *Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.

[Cou91] Olivier Coudert. *SIAM: Une Boite Outils Pour la Preuve Formelle de Systmes Squentiels*. PhD thesis, Ecole Nationale Suprieure des Tlcommunications, Octobre 1991.

[GB94] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Proc. 6th International Computer Aided Verification Conference*, volume 818, pages 299–310, 1994.

[HD93] Alan J. Hu and David L. Dill. Reducing BDD size by exploiting functional dependencies. In *Design Automation Conference*, pages 266–271, 1993.

[ISS+03] S. Iyer, D. Sahoo, Ch. Stangier, A. Narayan, and J. Jain. Improved symbolic verification using partitioned techniques. In *Proceedings CHARME'03*, pages 410–424. LNCS 2860, 2003.

[PP03] E. Pastor and M.A. Peña. Combining Simulation and Guided Traversal for the Verification of Concurrent Systems. In *Proceedings of DATE'03*. IEEE publisher, 2003.