Syntax-driven behavior partitioning for model-checking of ESTEREL programs

Eric Vecchié and Robert de Simone [1]

INRIA, Sophia Antipolis, France

Abstract

We consider the issue of exploiting the structural form of ESTEREL programs to partition the algorithmic RSS (reachable state space) fix-point construction used in model-checking techniques. 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 transition relation part. Here a brute-force symbolic breadth-first search would have mixed the exploration of P and Q instead, in case P had different behaviors of various lengths, and that would result in irregular BBD representation of temporary state spaces, a major cause of complexity in symbolic model-checking.

Difficulties appear in our decomposition approach when scheduling the different transition parts in presence of parallelism and local signal exchanges. Program blocks (or "Macro-states") put in parallel can be synchronized in various ways, due to dynamic behaviors, and considering all possibilities may lead to an excessive division complexity. The goal is here to find a satisfactory trade-off between compositional and global approaches. Concretely we use some of the features of the TIGER BDD library, and heuristic orderings between internal signals, to have the transition relation progress through the program behaviors to get the same effect as a global RSS computation, but with much more localized transition applications. We provide concrete benchmarks showing the usefulness of the approach.

Key words: Esterel, model-checking, BDD, reachability, partitioning, program-blocks, frontier, high-level, syntax, cofactoring

1 Introduction

In the last decade the advent of BDD-based implicit state-space representation [Bry86] allowed to scale up various analysis techniques ("model-checking", in a wide acceptation of the term) 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,ISS+03]. In

[1] Eric.Vecchie@sophia.inria.fr, Robert.De_simone@sophia.inria.fr

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

the context of ESTEREL [Ber92] 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 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].

For expository reasons we shall focus in the current paper on a tiny kernel version of the language, still sufficient to present our techniques (which can handle the full language). In particular we shall only consider specifications consisting of a general parallel "network" of otherwise sequential components, with global scoping of internal signals. Still, components can exchange signals to start, freeze or abort one another, and change macro-state configurations.

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, so that the transition relation always grows. But 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 informally motivate our framework on a simple example. Then 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 an abstract description of our approach, first on sequential components and then on parallel systems with local signal exchanges, followed by 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, followed by suggestions for further improvement on the treatment of $loop$ constructs.

An example : the good old digital wristwatch.
The design as shown in picture 1 consists of several modules, and an interface of 4 input buttons and an output LCD screen (with also an audio buzz) :

an ALARM **(counter) module** computes the time and date by adding up input quartz ticks; it sends information to the DISPLAY when change occurs (increment);

a TIME_SET **module** allows to change and update the time and date values using proper input buttons;

an ALARM_SET **module** allows to set an alarm time, and to toggle the on/off alarm mode;

a STOPWATCH **module** allows to set/reset or stop a chronometer; it should continue running if needed even when not on display;

a DISPLAY **module** displays proper information on the wristwatch LCD screen or audio buzz, according to the current modes set;

a BUTTON_DECODER **module** should link the actual wristwatch buttons to the proper signals entering submodules according to the current mode(s). In particular the role of the upper-right button, hereafter named "*Mode_Select*", will be to alternate active mode between TIME_SET, ALARM_SET and STOPWATCH modes.

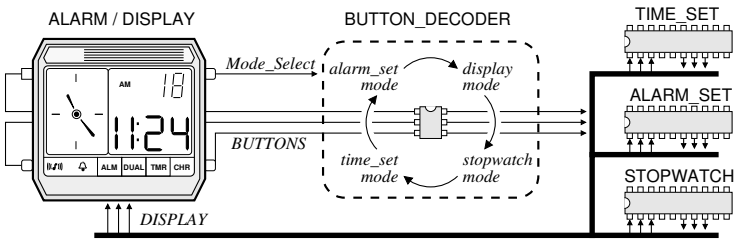

Fig. 1. The wristwatch design.

The reactive behaviors of the components could not be included here for sake of room. But the reader can easily convince him/herself that the three submodules amongst TIME_SET, ALARM_SET, and STOPWATCH have to be put in parallel, but remain largely exclusive concerning their response to most *button* events (in fact only the currently selected mode answers those impulses, safe for the button which switches modes). The basic breadth-first search analysis of such a program does not take advantage of the fact that submodules are exclusive and computes the reachable state space on the whole program. The analysis of this program could be divided into three parts instead : the first part computing the reachable states in the TIME_SET mode, then in the ALARM_SET mode and finally in the STOPWATCH mode. Thus, the state search of each mode could largely be done independently of the two others. The gain in space of such a method is obvious since the analysis of the original program can be assimilated to the analysis of three programs, all smaller than the original one as local transitions are used in place of global ones.

2 μ-ESTEREL

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 :

$$
\begin{array}{lll}
P ::= \texttt{pause} & \mid P \parallel P & \mid \texttt{present}\ S\ \texttt{then}\ P\ \texttt{else}\ P\ \texttt{end} \\
\mid P\ ;\ P & \mid \texttt{emit}\ S & \mid \texttt{abort}\ P\ \texttt{when}\ S \\
\mid \texttt{loop}\ P\ \texttt{end} & \mid \texttt{signal}\ S\ \texttt{in}\ P\ \texttt{end} &
\end{array}
$$

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). The topic of constructive causality (for the determination of signal presence values) is a large body of ESTEREL semantic theory, but we shall not address it here; instead we shall assume that there is **no** cyclic combinational dependencies between signals.

While 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 analyses at the circuit level. In our case we try to lift them some more by exploiting high-level structuring information from the source syntax.

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

3 Symbolic next-state operation, and optimizations

3.1 Symbolic state space computation

The basic breadth-first search Reachable State Space algorithm can be written:

```
1 reachable ← INIT
2 new ← INIT
3 while ( new ≠ ∅ ) do
4     new ← Image_Δ(new, INPUTS) ⧵ 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 R and I :

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

where $B = \{0, 1\}$, m is the number of registers and n is the number of input signals 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 :

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

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

3.2 Set encoding

Given a set of BDD variables $\mathcal{R} = \{r_1, \ldots r_n\}$, we introduce the operator $\lfloor \mathcal{R} \rfloor = \lambda X \to \neg r_1 \wedge \ldots \wedge \neg r_n$. If $r_1, \ldots r_n$ are variables representing boolean registers $R_1, \ldots R_n$ then $\lfloor \mathcal{R} \rfloor$ represents the set of states in which all registers R_i are inactive for all $i \in [1..n]$.
We notice that $\overline{\lfloor \mathcal{R} \rfloor} = \lambda X \to r_1 \vee \ldots \vee r_n$ represents the set of states in which at least one register R_i is active for $i \in [1..n]$. We have $S \cap \overline{\lfloor \mathcal{R} \rfloor} = S \setminus \lfloor \mathcal{R} \rfloor$ for all set S.

3.3 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 $f_{\uparrow S}$ the cofactoring of f by the set S :

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

The value of $f_{\uparrow S}$ out of S is not used and can be anything. It is set in order to minimize the size of the BDD representing $f_{\uparrow S}$. 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 $\lambda X \rightarrow \neg r$. In other words, the BDD encoding the transition function of registers that will not be activated in the next instant is very small.

4 General description of the method

At the heart of the method is the division of the program body into blocks (or macro-states) of proper granularity. In sequential subcomponents macro-states will be combined in sequence or as alternative choices *if-then-else*. State search will be performed inside each block until stabilization, before moving to the next one. The next iterative step will take as new initial states those *"pending"*, which were obtained as end frontier states from the proper previous local fix-point searches. 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.

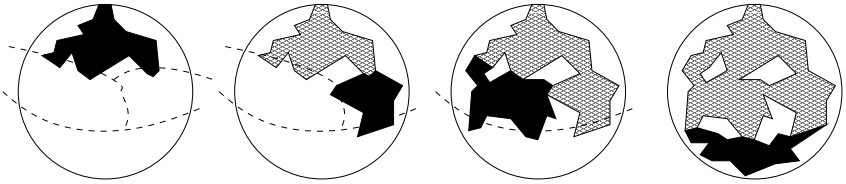

Fig. 2. Partitioning method according to four blocks of program. Frontiers between blocks (drawn in dashed line) are opened one by one.

This scheme raises a problem with parallelism, and the case where two local frontiers can be traversed concurrently in parallel components. Taking all possible combinations of blocks into account would lead to a Cartesian product explosion of cases. So we choose instead to follow the following strategy: first, find a "good" ordering of frontiers, likely to match the progress of state creation. Then, we start with a minimal number of active blocks, and we only *add up* new blocks when passing frontiers, without closing any back. So the transition relation will grow from initial to the global full one. But, meanwhile, we only apply growing transition relations to states that could provably create new ones outside the previous scope, so that states that were reached and contributed only *inside a previous step* were safely computed using only a restricted version. So, our *new* set of states on which transition relation is applied will always lay outside the previous combinations of blocks, and the various operations of cofactoring will (hopefully) leave out much of the transition relation description. This "wave" of progressing blocks is shown in figure 2, while figure 3 shows the details of behaviors at the frontiers.

The division in between blocks and the definition of relevant frontiers of course rely 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. The graph is built on top of our syntax tree, using the same nodes. It describes all possible paths followed by the control between each instruction of an ESTEREL program, especially between registers. The frontier between blocks will then be described by selecting a dedicated subset of edges. The selection varies dynamically as less and less frontier edges are preserved, causing the extension of the transition relation described before. Then,

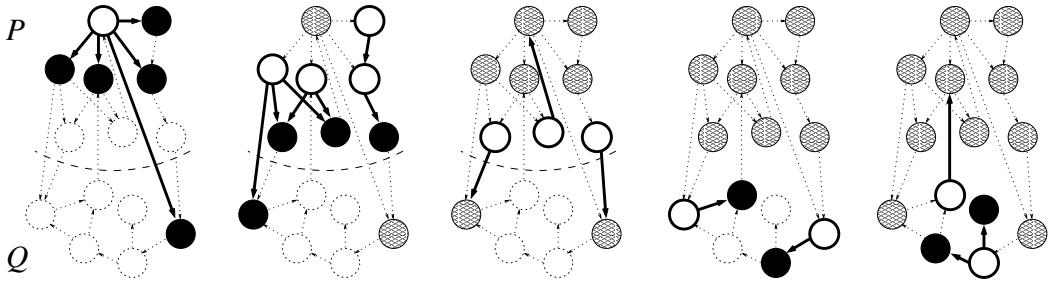

Fig. 3. Detail of our partitioning method on a frontier between two blocks P and Q. In the first three steps, the saturation of P is performed. States which overflow outside of the P area are not used in the image computation. In the last three steps, the saturation of P and Q is performed starting with the former pending states. Since whole P has been analyzed, the exploration of the reachable states only concerns Q.

from the current graph containing locked and unlocked edges, each iterative macro-step of the algorithm consists in computing the set of inactive registers, build the proper BDD description of the considered area, select the proper set of next-step initial configurations from states *pending*. In the next section we shall describe our choice of *frontier* from the structural syntax, together with the control flow graph creation.

5 Partitioning into "macrostates" according to syntax

There are two aspects that will be considered here. The first obvious one is how to partition the transition relation application according to syntax. The second is how to figure when the decomposition is indeed beneficial (because no subpart is in fact degraded to the point that considering it in isolation would be a waste of energy); we shall remain elusive on that second aspect for the time being.

Sequence statement.

Consider a program consisting of two components put together in sequence : P`;`Q. If the reachable state space is computed in a breadth-first search manner on a global transition relation, then states in Q will be considered while possibly further states in P are still not reached, in which 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).
There are two cases where partitioning is a waste of energy. The first is the obvious case where P or Q contains no pause statement. The second occurs when P is a constant-length program. For example, if P is of the form **`pause;pause`** then each execution of P is spread on two instants. In other words, the partitioning of P`;`Q is naturally performed by the breadth-first search algorithm.

Choice operator.

If we now consider **`present`** `S` **`then`** P **`else`** Q **`end`** alternative choice, the situation is very similar. Reachable state spaces in P and Q can be built independently if one assumes that both branches do not terminate instantly, and thus contain `pause` statements.

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.

Loops.

In a sequential ("parallel-free") context, the exploration of a **loop** P **end** program breaks down to the exploration of P, since the loop only leads back to the initial configurations of P, already reached. In the (more common) parallel setting, though, the problem comes from the fact that which blocks can be active in parallel is in general dynamically obtained through successive synchronizations (this is in a large part why RSS construction can be so hard). Our current solution is to *only increase* the register support for transition relations used in successive fix-points, and to rely *hopefully* on the fact that actual synchronizations will only allow state creation of such shape that the regular cofactoring of TIGER will clean up the excess of transition relations (when configurations are uniformly inactive, leading to *false*-valued registers, the corresponding transition parts are discarded). In the future, further studies of the control-flow graph structure should help us figure which frontiers can be seen as globally synchronizing the full system's pattern of loops, so that it can be preserved as a frontier *each time* loops are "unrolled".

Parallel networks and signal synchronizations.

As already mentioned, 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 4. The only syntactic element at

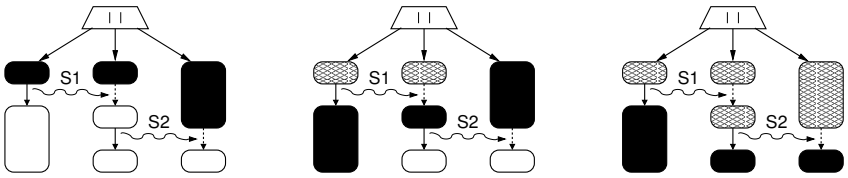

Fig. 4. Partitioning method for a parallel component. There are two signals synchronizing three parallel components. Our technique aims at partitioning according to the black-colored blocks. Hatched blocks should be removed by cofactoring methods.

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.

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 7.1. In the future we intend to investigate possible refinements of this choice (of more static nature), specially looking for frontiers with global effects that could be preserved across loops.

6 Control flow graph

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

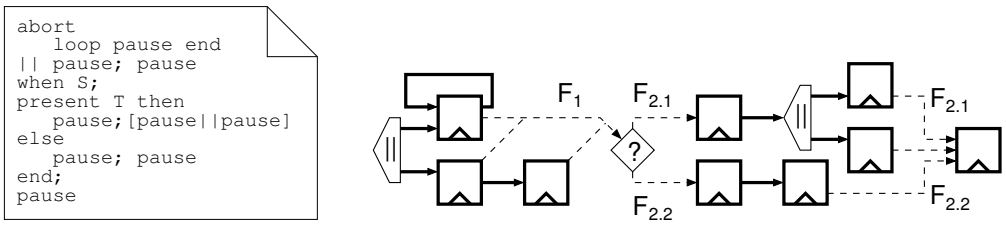

Fig. 5. Example of an ESTEREL program with its control graph. Frontiers F_1, $F_{2.1}$ and $F_{2.2}$ in dashed line have been produced by the `abort` and the `present` statements.

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 $J = \{J_1, \ldots J_n\}$.

Atomic instructions produce graphs containing a single node and no edge :

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

In the following statements, we suppose that an instruction I produces a graph $\mathcal{G}(I) = (\mathcal{I}, \mathcal{O}, \mathcal{N}, \mathcal{E}, \mathcal{F})$. As well, for $i \in [1, 2]$ we have $\mathcal{G}(I_i) = (\mathcal{I}_i, \mathcal{O}_i, \mathcal{N}_i, \mathcal{E}_i, \mathcal{F}_i)$. In our graph, we can abstract the beginnings and the ends of the scope. The graph of a signal declaration is thus the same as for I :

$$\mathcal{G}(\mathbf{signal}^L\ s\ I\ \mathbf{end}^{L'}) = (\mathcal{I}, \mathcal{O}, \mathcal{N}, \mathcal{E}, \mathcal{F})$$

In a binary sequence, final nodes of the first graph are linked to start nodes of the second graph :

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

A loop never terminates, thus its set of final nodes is empty. The final nodes of $\mathcal{G}(I)$ are linked to its entries.

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

Both branches of a parallel are started in the same instant. Thus the start point of a parallel is a unique node linked to the entries of its both branches.

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

In a `present` statement, we want to put frontiers in order to explore I_1, then I_2 and then, anything which is executed after this statement in the program. Frontiers are thus placed before and after the "then" branch and the "else" branch.

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

Each `pause` instruction may lead to the end of the `abort` instruction that encloses it. Such transitions are frontiers which will help us split the RSS computation and thus are put in the set $\mathcal{F}$.

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

7 The precise algorithm and its BDD implementation

We shall introduce useful notations. $Closure_{(\mathcal{N},\mathcal{E})}(\mathcal{I})$ represents the set of states reachable from $\mathcal{I}$ through edges in $\mathcal{E}$. We write $\epsilon(X) = \{j \in \mathcal{N}\ /\ i \in X, \exists i \rightarrow j \in \mathcal{E}\}$ the set of target nodes of edges of $\mathcal{E}$ whose source belongs to X :

$$Closure_{(\mathcal{N},\mathcal{E})}(\mathcal{I}) = (\mu X\ .\ \mathcal{I} \cup \epsilon(X))$$

The following function computes the "surface" of a program block. Given a set $\mathcal{R} \subset \mathcal{N}$ 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 $\mathcal{R}$. If $\mathfrak{R}$ is the set of nodes of type "pause", then :

$$Surface_{(\mathcal{N},\mathcal{E})}(\mathcal{R}) = Trans(\mu X\ .\ \mathcal{R} \cup (\epsilon(X) \setminus \mathfrak{R}))$$

where $Trans(X)$ is the set of edges in $\mathcal{E}$ whose source belongs to X. Given a set S of graph nodes, we introduce the operator $Register\langle S \rangle$ which returns the set of register BDD variables in S : $Register\langle S \rangle = \{r_i\ /\ (\mathbf{pause}\ r_i) \in S\}$. This operator will help us to make the link between our control flow graph and the symbolic BDD-based computations.

7.1 Partitioned algorithm

Our partitioned algorithm is guided by the control flow graph where edges are progressively unlocked. The BDD restrictedArea represents the set of all states (reachable or not) lying inside the frontier. At each step of the algorithm, the image computation is performed only on the pending reachable states lying inside restrictedArea (line 7). Cofactoring according to the current domain is implicitely done in the image computation (line 8). At the end of each step, the new-found states are stored in the pending set (line 9). No unlocking is needed as long as new states are found inside restrictedArea (lines 4, 5, 6).
This first algorithm does not describe the way restrictedArea is initialized and enlarged (this will be explained later).

```
1  reachable ← INIT, pending ← INIT
2  −− 1. Initialize the set encoded by "restrictedArea"
3  while ( pending ≠ ∅ ) do
4      if ( (pending ∩ restrictedArea) = ∅ ) then
5          −− 2. Unlock some edges and enlarge "restrictedArea"
6      end if
7      currentDomain ← pending ∩ restrictedArea
8      new ← Image_Δ(currentDomain, INPUTS) ⧵ reachable
9      pending ← (pending ⧵ currentDomain) ∪ new
10     reachable ← reachable ∪ new
11 end while
```

Control flow graph and restricted area initializations (1).

We assume that the syntax tree of the analyzed program is given in $\mathcal{T}$. The initialization process consists in building the graph to obtain an initial set of locked edges and then build the set restrictedArea with respect to these initial conditions.

1. Initialize the set encoded by "restrictedArea"

```
1  (I, O, N, E, F) ← G(T)
2  allRegs ← Register⟨N⟩
3  reachableRegs ← Register⟨Closure_(N,E) (I)⟩
4  restrictedArea ← ⌊allRegs ⧵ reachableRegs⌋
```

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

Restricted area enlargement (2).

When restrictedArea 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 restrictedArea set. Such edges can only be found in the surface of $reachableRegs$. 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 restrictedArea, a new edge is analyzed in order to decide whether it should be unlocked or not. In fact, edges whose origin belongs to $Closure_{(\mathcal{N},\mathcal{E})}(\mathcal{I})$ must be analyzed first, which does not appear in our algorithm.

2. Unlock some edges and enlarge "restrictedArea"

```
1  surface = F ∩ Surface_(N,E∪F) (reachableRegs), i ← 1
2  while ( (pending ∩ restrictedArea) = ∅ )
3      frontier ← surface[i], i ← i + 1
4      -- 2.1. Check if "frontier" should be opened
5      if ( unlock? ) then
6          -- 2.2. Unlock "frontier"
7      end if
8  end while
```

Edge crossing (2.1).

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

2.1. Check if "$frontier$" should be opened

```
1  newRegs ← Register⟨Closure_(N,E∪frontier) (I)⟩ \ reachableRegs
2  if ( newRegs = ∅ ) then
3      unlock? ← true
4  else if ( ( pending \ ⌊newRegs⌋ ) ≠ ∅ ) then
5      unlock? ← true
6  else
7      unlock? ← false
8  end if
```

First, we compute the set of nodes in the graph that would be reached if the edge $frontier$ was unlocked. We just need to know the new-found registers which are stored in $newRegs$ at line 1. If $frontier$ leads to no register, it can be unlocked but this will have no effect on the set restrictedArea (line 2, 3). If $newRegs$ is not empty, we check if there are some states in pending that have activated one or more new registers contained in $newRegs$ (line 4, see section 3.2). In this case, the edge can be unlocked.

Unlocking compatible frontiers (2.1').

An example : R_1, R_2 and R_3 are three inactive registers locked by three distinct edges. The set pending contains two states : the first in which only R_1 and R_3 are active and the second where only R_2 and R_3 are active. We unlock a first edge that lets us activate the register R_1. Then, at this point of the algorithm nothing forbids us to activate R_2 before R_3 whereas we would prefer to activate only R_3.
The solution consists in making a copy of the set pending called pending' before starting to unlock edges. Each time an edge is unlocked, we reduce the set pending' in order to keep only "compatible" states activating new-allowed registers.

```
1  pending' ← pending
2      ...
3      else if ( ( pending' \ ⌊newRegs⌋ ) ≠ ∅ ) then
4          pending' ← pending' \ ⌊newRegs⌋
5          unlock? ← true
6      ...
```

In our previous example, once R_1 has been allowed to be activated, R_2 cannot be activated before R_3 any more.

Unlocking an edge (2.2).

Once an edge has been decided to be unlocked, we just have to perform the following updates : first, the unlocked edge is moved from $\mathcal{F}$ to $\mathcal{E}$. Then, the set restrictedArea is enlarged.

2.2. Unlock "$frontier$"

1 $\mathcal{E} \leftarrow \mathcal{E} \cup \{frontier\}$, $\mathcal{F} \leftarrow \mathcal{F} \setminus \{frontier\}$
2 $reachableRegs \leftarrow reachableRegs \cup newRegs$
3 restrictedArea $\leftarrow \lfloor allRegs \setminus reachableRegs \rfloor$

7.2 Correctness arguments (hints)

We shall give informal arguments to justify our claim that all states will be reached by our partitioned technique.

In the end, the ever-growing transition relation will reach the form of the global one 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 *all states* reachable only from there. But, importantly, the new states reached inside a fix-point search at a given stage of transitions selection that are **not** put inside the pending set are those *which cannot produce any further successors* (because we have reached a fix-point of that restricted relation transition). So, a reachable state will eventually be reached when frontier unlocking will open a path to it.

8 Prototype implementation and benchmarks

We implemented our method with the help of the TIGER BDD package and we tested it on some ESTEREL designs. 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.
As our current prototype model-checker can only handle the μ-ESTEREL restricted syntax (without data handling), we were not yet able to parse and analyze large programs from the ESTEREL benchmark suites. Results are still promising on small, hand-written programs. For instance, on the largest benchmark example which passed our syntactic criteria (named **sequencer** in the benchmark suite), we decreased the peak memory usage due to BDD consumption by about 60% (17 Mbytes vs 40 Mbytes). Figures 6 and 7 show the evolution of the algorithm on this example.

On another large design (named **cabin** in the benchmark suite), the default method using a global transition relation is not able on our workstation to produce more than 534 states, following three iteration steps in the search (in 11.85 seconds), and collapsing in the fourth step with over 900 Mbytes of memory consumed. Our method was able to produce **135 441 875** states (after 35 hours 40 minutes) achieving with success 123 iterations.

Although inspiring, the wristwatch design presented in section 1 is too small to present significant results.

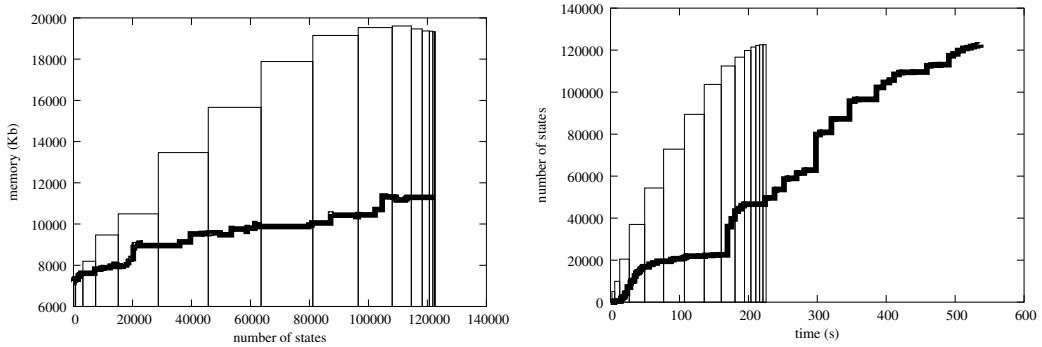

Fig. 6. In the both graphs, boxes represent the default algorithm. The solid-line represents the partitioned algorithm. The first graph shows the memory used with respect to the number of states found (partitioned is better). The second shows the number of states found with respect to the computation time (default method is faster, as expected).

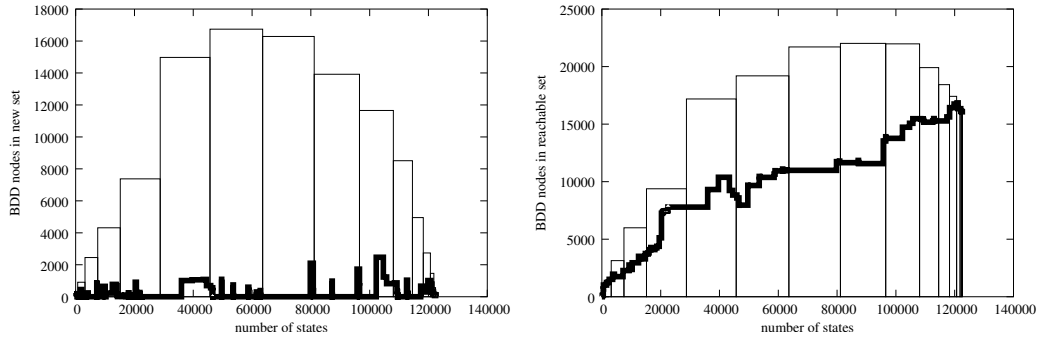

Fig. 7. The first graph shows the number of nodes in the BDD encoding the new-found states with respect to the number of states found. The second shows the number of nodes in the BDD encoding the reachable states with respect to the number of states found.

9 Conclusion and future work

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 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 emanated 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 "good" order, so that all reachable states can be captured. The ever-increasing aspect of the transitions allow to avoid the potential blow-up in various cases of pairing blocks active in parallel. But, as a corollary, the method still suffers from relative inefficiencies in the treatment of loops, which cannot really be divided in a succession of steps. We intend in the future to study whether closer inspection of the graph can lead to cases where a frontier (say, a signal reception) can be seen to have a global synchronization feature, so that a number of looping components put in parallel cannot progress at respective irregular "speeds". This is for instance the case in our sketched WRISTWATCH example, where the BUT-

TON_DECODER main loop has the effect of serially activating the apparently parallel TIME_SET, ALARM_SET, and STOPWATCH modules, while most of their behaviors can only be performed in a round-robin fashion, commanded by the BUTTON_DECODER main loop.

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, 1991. North-Holland.

[BCL+94] J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, 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, 1994.

[Ber92] Gérard Berry. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Ber99] G. Berry. The constructive semantics of pure Esterel. Draft version 3. http://www-sop.inria.fr/meije/, 1999.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proc. Workshop on Automatic Verification Methods for Finite State Machines*, volume LNCS 407, 1989.

[CMT93] O. Coudert, J. C. Madre, and H. Touati. Tiger version 1.0 user guide, 1993. Digital Paris Research Lab memorandum.

[Cou91] Olivier Coudert. *SIAM: Une Boite à Outils Pour la Preuve Formelle de Systèmes Séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Octobre 1991.

[GB94] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 299–310, Standford, California, USA, 1994. Springer-Verlag.

[ISS+03] S. Iyer, D. Sahoo, Ch. Stangier, A. Narayan, and J. Jain. Improved Symbolic Verification using Partitioned Techniques. LNCS ??, 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.