State Abstraction Techniques for the Verification of Reactive Circuits

Yannis Bres[1], Gérard Berry[2], Amar Bouali[2], and Ellen M. Sentovich[3]

[1] CMA-EMP/INRIA (Yannis.Bres@cma.inria.fr)
[2] Esterel Technologies ([Gerard.Berry,Amar.Bouali]@esterel-technologies.com)
[3] Cadence Berkeley Labs (EllenS@cadence.com)

Abstract. Several techniques for formal verification of synchronous circuits depend on the computation of the reachable state space (RSS) of the circuit. Computing the exact RSS may be prohibitively expensive. In order to simplify the computation, the exact RSS can be replaced by an over-approximation of it, called the ORSS. The resulting verification computation will be conservative, and the larger the ORSS, the more conservative the approximation. A common technique for computing the ORSS is to replace some of its state variables by inputs. In this paper, we present a new approach based on variable abstraction using a three-valued logic. We also present a way to reduce the over-approximation by using structural information given by compilers of high-level languages like Esterel, ECL or SyncCharts. A real example of an avionic system is used to show the improvements that variable abstraction can bring.

1 Introduction

This paper deals with formal verification of synchronous designs derived from programs written either in Esterel [4, 5], ECL [15] or SyncCharts [1] languages. These languages are well suited for control-dominated programs, both for hardware and software targets. ECL and SyncCharts programs can be translated into Esterel. The Esterel compiler translates such programs into the pair of a sequential circuit and a data path.

Formal verification is currently performed on the control part of the program, by XEVE [2], a BDD-based verifier publicly available, or the verifier built-in the Esterel Studio tool [14]. The properties are expressed by *synchronous bug observers* [20], i.e. auxiliary signals that are emitted by the circuit in case of a safety property violation. Verification amounts to checking that observer signals can never be emitted. To check observers, XEVE uses a forward reachability technique well-adapted to Esterel control-dominated programs: it iteratively computes the reachable state space of the circuit, or *RSS*, checking at each step that observers cannot be emitted for any reachable state and any legal input. Although this approach has proved successful in handling quite large designs, it is limited by the potential explosion of BDDs during the computation of the RSS.

This paper is devoted to improvements of the verification algorithm based on *variable abstraction*. The global idea is to use over-approximations (ORSS) of the exact RSS, which is usually an overkill to prove safety properties. Verification using the ORSS is *conservative*: if a property is true for an ORSS, it is true for the original circuit, but a given ORSS may not prove the desired property. An ORSS can be obtained directly from the structure of the source program, as explained in [25], but its impact on verification performance is relatively limited. A better approach to simplify verification is to *reduce the number of variables and functions* occuring during the BDD computations. We study two techniques for this purpose: register inputization, in which a state variable is simply made free in the RSS, and register abstraction, in which we use a three-valued logic. Register inputization views some registers as free combinational variables, losing their state-holding contents. Register abstraction uses a three-valued logic and makes some register variables completely disappear from the BDD, which is attractive to improve computation times, but is a stronger abstraction. Both techniques can be combined with the aforementioned structural ORSS ones. We show the efficiency of our method on a real avionics system, the fuel management of a twin-engine jet aircraft from Dassault Aviation, and present two other experiments.

Section 2 presents the algorithms for RSS computation. Section 3 presents the ORSS abstraction techniques. Section 4 presents the application example, and section 5 concludes.

1.1 Related Work

One of the most studied approach to ORSS computation is based on FSM decomposition: in [11], Cho et al. proposed approximate RSS computation algorithms that decompose the set of state variables into disjoint subsets. Each subset is used to compute a portion of the RSS, and the cross-product is taken afterwards for an ORSS. Extension to non-disjoint subsets was described by Govindaraju et al. in [16], and refined in [17] through addition of auxiliary state variables that increase correlation between subsets. Such techniques perform *a posteriori* quantification, as state variables from other subsets are replaced by inputs, which can turns out to be very expensive.

Three-valued logic are often used in model checking partial or approximated systems. For instance, [6] (refined in [7]) used three-valued logic in order to interprete modal logic formulas on partial Kripke structures. However, this work and its refinement ([19], [18], ...) operates on labeled transition systems which are explicitly explored, while our analysis are performed on systems represented as Boolean circuits, symbolically explored using BDD-based techniques. Although applications to symbolic techniques were considered, this has not yet been done to the best of our knowledge.

2 Background

2.1 Finite State Machines

Let $\mathbf{B} = \{0, 1\}$ be the Boolean set. The FSMs we consider are completely specified Mealy machines, defined as tuples $(m, n, p, \delta, \omega, \mathcal{I}, \mathcal{J})$, where:

- m is the number of inputs.
- n is the number of state variables (registers).
- p is the number of outputs.
- $\delta : \mathbf{B}^m \times \mathbf{B}^n \to \mathbf{B}^n$ is the vector of elementary register transition functions.
- $\omega : \mathbf{B}^m \times \mathbf{B}^n \to \mathbf{B}^p$ is the vector of elementary output functions.
- $\mathcal{I} : \mathbf{B}^n \to \mathbf{B}$ is the characteristic function of the set of initial states.
- $\mathcal{J} : \mathbf{B}^m \to \mathbf{B}$ is the characteristic function of the valid input space. For instance, if some inputs are implied by others, or if some pairs of inputs are mutually exclusive, the whole input space would not be valid.

We use the same notation for a set or its characteristic function. Thus, $\mathcal{J}(x) = 1$ means $x \in \mathcal{J}$. Also, for the sake of clarity, we omit the arrow on top of vectorial functions or variables. Negated expressions are either prefixed by $\neg$ or overlined.

2.2 Standard RSS Computation

The usual way to compute the RSS of a FSM symbolically [9, 12], is to find the limit of the converging sequence of finite sets defined by the following equations:

$$\begin{aligned} \mathrm{RSS}_0 &= \mathcal{I} \\ \mathrm{RSS}_{k+1} &= \mathrm{RSS}_k \cup \delta(\mathcal{J}, \mathrm{RSS}_k) \end{aligned} \tag{1}$$

where we use the standard extension of function to sets:

$$\delta(X, Y) = \{\delta(x, y) \mid x \in X, y \in Y\}$$

Using BDDs for characteristic functions, (1) becomes:

$$\mathrm{RSS}_{k+1} = \mathrm{RSS}_k \cup \{\, r' \in \mathbf{B}^n \mid \exists r \in \mathrm{RSS}_k\,,\ \exists i \in \mathbf{B}^m \,.\, \mathcal{J}(i) \,\wedge\, r' = \delta(i, r) \,\} \tag{2}$$

In [12], Coudert and Madre introduced the *image* operator $\mathrm{Img}(f, \chi)$, which computes the image of the vectorial function f on the state set of characteristic function χ[1]:

$$\mathrm{Img}(f, \chi) = \lambda r'. \left(\exists r, i \,.\, \chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k = f_k(i, r) \right) \right) \tag{3}$$

[1] $\lambda r'.E$ is the standard λ-calculus notation for the unnamed function of body E, with argument r'.

Algorithm 1 presents an outline of the computation of RSS for a given FSM. The main iteration that computes successive RSS_k sets is from line 4 to 17. Line 5 builds the domain for each iteration, based on most recently reached states and the set of valid inputs $\mathcal{J}$. Lines 8 to 10 contain the loop that builds the transition function for the current iteration domain. Line 9 builds the function associated with a single register, restricted for the current domain. Line 10 associates this function with its register variable for the next state and combines it with the final transition function. Line 12 applies the transition function to the last reachable state set. Line 13 performs existential quantifications over the set of old register variables and inputs. Line 14 substitutes the new register variables by the old ones, in order to obtain a function over the set of old register variables for the next iteration. Finally, line 15 computes the sets of new states and line 16 adds this set to the final reachable state set. Iteration stops when the set of new states is empty.

Note that this is only a crude implementation. In the next version, currently under development, the complete transition function is actually never built as we do in lines 8 to 10, which may cause the computation to blow-up quickly. The image is computed over partitions of the transition function and existential quantifications are performed *on-the-fly* rather than in a simple pass, as we mention in line 13. However, it is beyond the scope of this article to discuss such improvements.

```
1   function RSS( FSM )
2     Result ← I
3     NewStates ← I
4     repeat
5       Domain ← J ∧ NewStates
7       δ ← 1
8       for i ∈ [1..n]
9         δ_i ← BuildRestrictedRegisterFunction( i, Domain )
10        δ ← δ ∧ (NewRegVariable(i) = δ_i)
11      end for
12      Image ← δ ∧ NewStates
13      Image ← Quantify( Image, OldRegVariables+InputVariables )
14      Image ← Substitute( Image, NewRegVariables, OldRegVariables )
15      NewStates ← Image ∧ ¬Result
16      Result ← Result ∨ Image
17    until NewStates = 0
```

Algorithm 1: RSS fixed-point computation

Example Using Algorithm 1, we can enumerate the reachable states of the circuit of Figure 1. The initial state $(1, 0, 0, 0)$ of the circuit is indicated by the values at the bottom of the registers. The first iteration reveals the new state $(0, 1, 0, 0)$; the second iteration reveals the new state $(0, 0, 1, 0)$; the third iteration reaches the fixpoint: the three registers r_1, r_2 and r_3 are exclusive and r_4 is always 0.

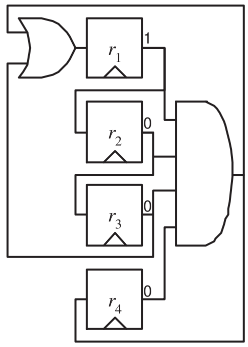

Figure 1: A sequential circuit

RSS Computation Complexity Analysis In this section, the complexity is expressed with respect to the BDD size and in the worst case.

The cost of $\neg$ is constant and the cost of $\vee$, $\wedge$ is polynomial [9, 12]. Unfortunately, the cost of the Img operator, used in line 13 in the Algorithm 1, is exponential with respect to the number of variables, notably because of nested existential quantifications. Informally, while $\exists x \,.\, f(x)$ amounts to computing $f(0)+f(1)$, $\exists x,y \,.\, f(x, y)$ amounts to computing $f(0, 0)+f(0, 1)+f(1, 0)+f(1, 1)$, and so on.

In the sequel, we will study techniques to improve the RSS computation by reducing the number of variables to apply *a posteriori* quantification to, in some case at the expense of over-approximation.

3 ORSS Computation

3.1 Replacing State Variables by Inputs

Replacing state variables by inputs can improve the RSS computation: there are fewer register functions to build, combine and manipulate during the image computation, and fewer register variables to substitute. Replacing state variables by inputs weakens the constraints between these variables, leading to an over-approximated result.

Note that the number of *a posteriori* existential quantifications to perform remains the same.

When a state variable is replaced by an input, the correlation between multiple occurrences of this variable in an expression is maintained. This is the case in reconvergent fanout in a circuit. For instance, in Figure 2, there is a circuit fragment generated from a statement like *present I then ... else ...*. The *go* wire, which determines whether a statement is active, is combined with the input I presence wire. Even if the state variable driving this *go* wire is replaced by an input, we are still able to determine that *then* and *else* branches are exclusive.

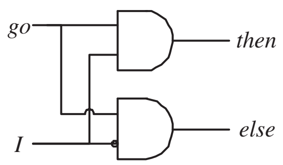

Figure 2: Generated nets for a *present I then ... else ...* statement

Example With the example circuit of Figure 1, suppose we want to check that $r_1 \wedge r_2 = 0$. We can replace r_4 with an input and apply the standard RSS computation algorithm: from the initial state $(1,0,0)$, the first iteration reveals the new state $(0,1,0)$, the second iteration reveals the new state $(0,0,1)$ and the third iteration reaches the fixpoint. We can still prove that $r_1 \wedge r_2 = 0$, but the computation required fewer register functions.

Conversely, if we choose to replace r_3 with an input, starting from the initial state $(r_1, r_2, r_4) = (1,0,0)$, the first iteration reveals the new states $(0,1,0)$ and $(1,1,0)$, the second iteration reveals the new state $(0,0,0)$ and the third iteration reaches the fixpoint. We cannot prove that $r_1 \wedge r_2 = 0$.

3.2 Variable Abstraction using Ternary-Valued Logic

Three-Valued Logic As a refinement of Malik's work [23], Shiple, Berry and Touati [24] used Scott's three-valued logic to analyse cyclic circuits. Scott's three-valued logic is built upon the usual two-valued Boolean logic by adding a third value, noted $\perp$, which means that a variable is *undefined*, and by extending usual Boolean operators.

Similarly, we propose to introduce a third value meaning that a variable is *defined*, i.e. either *true* or *false*, noted d. Indeed, the laws for d are exactly those of $\perp$, and we are simply using standard Scott Logic. However, we prefer to use the d symbol since the intuition is different.

The 3 logic values $\{0,1,d\}$ are respectively encoded by the pairs of Boolean values $\{1,0\}$, $\{0,1\}$ and $\{0,0\}$. In expressions, we encode variables we want to keep by a pair $(x, \overline{x})$, and variables we want to abstract by the constant pair $d = (0,0)$. Three-valued functions (TVFs) are encoded using a pair of Boolean functions $\left(f^0, f^1\right)$, such that f^0 (resp. f^1) is the characteristic function of the set for which f evaluates to 0 (resp. 1). The set f^d of valuations for which f is defined is $f^d = \overline{f^0 + f^1}$ and, by construction, $f^0 \cdot f^1$ is always false. Hence, f does not characterize a partition of two sets $(f, \neg f)$ as in Boolean logic, but a partition of three sets $\left(f^0, f^1, f^d\right)$, as seen on Figure 3.

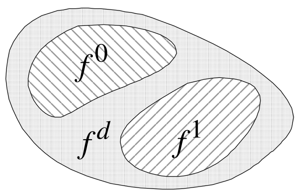

Figure 3: f^0, f^1 and f^d onsets

Standard operators over Boolean functions are extended to TVFs with respect to the following formulas:

$$\begin{aligned} \neg\left(f^0, f^1\right) &= \left(f^1, f^0\right) \\ \left(f^0, f^1\right) + \left(g^0, g^1\right) &= \left(f^0 \cdot g^0, f^1 + g^1\right) \\ \left(f^0, f^1\right) \cdot \left(g^0, g^1\right) &= \left(f^0 + g^0, f^1 \cdot g^1\right) \end{aligned}$$

For instance, $f + g$ is false if both f and g are false, but true as soon as either f or g is true.

The three-valued logic functions are known to be monotonic [8] in the lattice $\{d \leq 0, d \leq 1\}$.

Application to the RSS Computation By abstracting variables, we take the previous technique a step further: while state variables replaced by inputs were still present in intermediate computation, abstracted variable are completely removed from the support of the BDDs.

To achieve this, we need to return to how the equality in (3) is computed. As the equality $a = b$ can be written as $a \cdot b + \overline{a} \cdot \overline{b}$, (3) is internally expanded into:

$$\operatorname{Img}(f, \chi) = \lambda r'. \left(\exists r, i \,.\, \chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k \cdot f_k(i, r) + \overline{r'_k} \cdot \overline{f_k(i, r)} \right) \right) \tag{4}$$

Using three-valued logic, we cannot simply replace f_k by f_k^1 and $\overline{f_k}$ by f_k^0, as we do not have $f_k^1 \vee f_k^0$, unlike $f_k \vee \overline{f_k}$, as represented on Figure 3. Instead of a partition f, $\overline{f}$, we now have three sets, f^0, f^1, and $f^d = \overline{f^0 + f^1}$, the latter being the set of arguments for which we only know that f is defined. Therefore, we must *widen* the positive function f by $\overline{f^0}$, and the negative function $\overline{f}$ by $\overline{f^1}$.

We introduce the OImg operator as the widening of the standard Img operator, defined as:

$$\mathrm{OImg}(f,\chi) = \lambda r'.\left(\exists r,i \,.\, \chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k \cdot \overline{f_k^0(i,r)} + \overline{r'_k} \cdot \overline{f_k^1(i,r)}\right)\right) \qquad (5)$$

Informally, we have replaced the characteristic function of the set "on which f is true" (the *onset* of f), by a superset "on which f is certainly not false", and *vice versa*. However, when applied to concrete variables of the form $(x, \neg x)$, the onsets of f^0 and f^1 forms a partition of the domain on which f is defined, and the result of (5) remains exact.

Since three-valued functions are monotonic, the OImg operator is also monotonic in the complete lattice of sets of states. Hence the algorithm terminates with a unique least fixpoint.

Example Returning to the example circuit of Figure 1, in order to check that $r_1 \wedge r_2 = 0$, we can abstract r_4, and apply the widened RSS computation algorithm: from the initial state $(r_1, r_2, r_3) = (1, 0, 0)$, three iterations reveal the states $(0, 1, 0)$ and then $(0, 0, 1)$. Having abstracted r_4, we could prove that $r_1 \wedge r_2 = 0$ with less functions but also with less intermediate variables.

Conversely, if we choose to abstract r_3, starting from the initial state $(r_1, r_2, r_4) = (1, 0, 0)$, three iterations reveal the states $(0, 1, 0)$ and $(1, 1, 0)$ and then $(0, 0, 0)$. We cannot prove that $r_1 \wedge r_2 = 0$.

Discussion As for the previous technique, there are fewer register functions to combine and manipulate during the image computation, and even fewer variables to substitute. Furthermore, the number of variables that have to be quantified *a posteriori* is reduced: the former formula $\exists x,y \,.\, f(x,y)$ becomes $f(d,0) + f(d,1)$ when x is abstracted, instead of $f(0,0) + f(0,1) + f(1,0) + f(1,1)$. One can argue that the number of register function to build is increased, but this step of the RSS computation is far from being critical.

On one hand, abstraction reduces the number of BDD variables and functions to compute, by early quantification. Of course, if the abstracted variables were really irrelevant, they would have also disappeared from the BDDs, but *during* its construction; our technique removes them *before*.

On the other hand, we have seen that the equality must be *widened*, which leads to an over-approximated result. Furthermore, the information we loose in the abstraction process is the correlation between positive and negative instances of a variable. For instance, $d\overline{d}$ is abstracted to d instead of 0. Returning to Figure 2, choosing to abstract the *test* variable would lead to loose the knowledge that both *true* and *false* branches are exclusive.

So far, the selection of state variables to be abstracted still depends on proper human designer guidance.

3.3 Refinement Using the Esterel Selection Tree

Esterel [4, 5] is a control-dominated language: the control part has a hierarchical structure, reflecting nesting of statements in the original programs, while communication between different parts of the program is handled through instantaneously propagating signals or shared variables. Roughly, every construct in the program has an associated *selection* wire indicating whether this construct is active or not. The value that these selection wires carry comes from combinations of registers, i.e. the current state of the machine. Selection registers are then combined with tests to activate other areas of the program and finally propagated to the registers to determine the next FSM state.

As generated from high-level language, Esterel circuits feature some interesting information concerning their design, notably the hierarchy of *pauses*, i.e. registers generated by explicit or implicit pause statements. For instance, consider Program 1, where declarations are omitted. Square brackets group statements, semi-colons indicate sequence, and || indicates parallelism. The *await* instruction contains an implicit pause: once the first instant an *await* statement was activated is over, the next statement is executed as soon as the awaited signal appears. Therefore, each *await* statement will generate a register in the circuit. Because *await* statements at lines 2 and 4 are executed in sequence, their register are *exclusive*; similarly, because block 1-9 is executed in sequence with the *await* statement at line 10, the register coming from line 10 is *exclusive* with all registers coming from block 1-9. On the other hand, blocks 2-5 and 7-8 are *compatible*, so no relation can be infered.

In Esterel circuits, such an information is stored in the *Selection Tree* [25], where non-terminal nodes indicate either compatibility or exclusivity and terminal nodes are registers. The Selection Tree of Program 1 is represented on the right-hand side, where exclusive nodes are noted with sharps. From this Selection Tree, we can build a BDD that alone gives an over-approximation of the RSS of the circuit, for all the states it denies cannot be reached *by construction*. With adequate variable ordering, the construction of such a BDD is straightforward. In the sequel, we will see that this BDD can be used both as an upper bound for over-approximation, and to maintain some constraints on loosen variables.

Use of the Esterel Selection Tree When replacing state variables by inputs, the Esterel selection tree can be used in two ways.

First, we can enhance the input care set with constraints involving at least one state variable that has been replaced by an input, as the input care set can actually reference both inputs and combinational inputs, i.e. real state variables.

Second, we can build, from the relations involving state variables, an upper bound for over-approximation, or *over-approximation ceiling*. Erroneously discovered states that cannot be reached *by construction* are removed by intersecting the set of new states with the over-approximation ceiling, at the end of each step of the RSS computation process.

When abstracting variables, we cannot refine the input care set with relations from the Esterel selection tree, as there is no variable any more. However, we

```
[
    await I1;                    pause 1 - - - #
    do something;                              # - - -|
    await I2;                    pause 2 - - - #      |
    do something                                      |- - -#
  ||                                                  |     #
    await I3;                    pause 3 - - - - - - -|     # - - - - -
    do something                                            #
];                                                          #
await I4;                        pause 4 - - - - - - - - - -#
do something
```

Program 1: Esterel Example Program

are still able to reference the subset of state variables that are not abstracted, and build an over-approximation ceiling BDD.

4 Experiments

We have implemented the presented technique on top of the TiGeR [13] BDD package. Our tool was run on a 750MHz Pentium III machine with 1GB of memory.

We present data on an industrial Esterel circuit: the fuel-management system of a twin-engine jet aircraft from Dassault Aviation, described in [21]. This system consists in several modules: 2 engines, 2 feeder tanks and several internal and external tanks. The main function of this system is to ensure that engines are properly fed, while managing component failures, fuel load balancing between the two sides of the aircraft, in-flight refueling, etc. Most of these tasks are handled by the two feeder tanks, and several safety properties were written for these modules. The complete design has 9,154 nets and 509 registers. Computing the exact RSS of the complete design is intractable on a 1GB machine. However, when focusing on only one safety property at a time, this become largely feasible after a simple pass of transitive network sweeping, which may remove more than 300 registers.

Tables 1 and 2 shows comparisons of the aforementioned approaches to the RSS computation, for two properties of the design that feature regular behavior of our tool (other properties only show behaviours similar to either one of the featured properties). The first line shows the results for exact computation, the second when some state variables are replaced by inputs, the third when some state variables are abstracted, and the fourth when some state variables are abstracted and the Esterel selection tree is used as an over-approximation ceiling. Time and memory columns do not take into account the file parsing and network construction times and memory usages, as they do not depend on the RSS computation approach, and, on such examples, may become the most expensive part of the process (although their complexity is linear and usually negligible). The ♯L column indicates the number of remaining registers after the transitive network sweeping pass.

Following the advices of the designers, we chose to abstract or replace by inputs the state variables of all of the internal and external tanks but the feeder ones. Note that the hierarchical nature of Syncharts designs allows our tool to work with simple abstraction hints from the designer.

method	time	mem. (MB)	#L	total reachable states at step n				
				1	2	3	4	5
exact	>10mn	79	178	8,749	3.01e8	1.33e13	3.33e13	3.67e13
repl. by inputs	3.8s	6	59	37	341	3,738		
abstracting	1.7s	7	59	37	2.71e5	9.48e6	9.51e6	
abs. + seltree	1.5s	6	59	37	1,670	6,807	7,407	

Table 1: Verification of Property 4

method	time	mem. (MB)	#L	total reachable states at step n						
				1	2	3	4	5	6	7
exact	>2mn	21	120	2	1.66e4	2.41e8	7.03e8	8.85e8		
repl. by inputs	0.6s	5	37	2	70	229	245			
abstracting	0.3s	5	37	2	4.33e3	1.07e6	2.42e6			
abs. + seltree	0.3s	5	37	2	865	7.92e4	1.77e5			

Table 2: Verification of Property 6

By removing state variables, we reduce the number of functions to build and compute the image of. We also reduce the number of existential quantifications to perform. Also, we cut some transitive links between functions, then allowing the transitive network sweeping pass to remove more state variables. When the removed variables are properly chosen, this results in great speed and memory usage improvements: there are several orders of magnitude of differences between the exact RSS computation and the least over-approximation technique. Furthermore, less iterations may be required to reach the fixpoint. Both tables show that abstracting state variables can lead to a greater over-approximation than replacement by inputs; for Property 4, this even require a additional iteration step. However, using relations between state variables expressed by the Esterel selection tree allows us to reduce significantly the over-approximation.

In any case, state variables to be removed must be selected with care. If not, excessive over-approximation may lead to a *snowball effect*: unreachable states are found reachable, then the image of these states must be computed, which may lead to other unreachable states found as reachable, and so on. As variable abstraction computes greater over-approximations than replacement by inputs, we can naturally expect results to be worse when they are already bad with replacement by inputs.

On another example, time improvements due to variable abstraction range from 20X to 70X and memory reduction from 5X to 10X, but the standard technique of replacing state variables by inputs achieves better results. We are currently improving the ORSS computation algorithm in order to obtain better figures.

5 Conclusions

This paper presents a technique to improve the computation of the Reachable State Space of sequential circuits, by computing over-approximations of it through variable abstraction, using a three-valued logic. This approach takes the commonly used technique of replacing state variables by inputs a step further. When state variables to be removed are properly chosen, relevant improvements of both time and memory usage can be noticed in comparison with replacement by inputs. Excessive over-approximation may be confined by using *by construction* RSS over-approximation ceilings, expressed by high-level structural data, like the Esterel selection tree.

References

1. Charles André. *SyncCharts: A Visual Representation of Reactive Behaviors*, I3S, 1996.
2. Amar Bouali. *XEVE, an Esterel Verification Environment. Proceedings of the 10th International Conference on Computer Aided Verification*, CAV'98, 1998.
3. J. R. Burch, E. M. Clarke, D. L. Dill, K. L. McMillan. *Symbolic Model Checking - 10^{20} States and Beyond. Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, June 1990.
4. Gérard Berry. *The Esterel Language Primer.* CMA, Ecole des Mines de Paris and INRIA. Available with the Esterel system and updated for each release.
5. Gérard Berry. *The Constructive Semantics of Pure Esterel.* CMA, Ecole des Mines de Paris and INRIA. July 2, 1999.
6. G. Bruns, P. Godefroid. *Model Checking Partial State Spaces with 3-Valued Temporal Logics. Proceedings of the 11th Computer Aided Verification International Conference, CAV'99*, 1999.
7. G. Bruns, P. Godefroid. *Generalized Model Checking: Reasoning about Partial State Spaces. Proceedings of the International Conference on Concurrency Theory, Concur'00*, August 2000.
8. J.A. Brzozowski, C.-J. H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1996.
9. Olivier Coudert, Christian Berthet, Jean-Christophe Madre. *Verification of Synchronous Sequential Machines Based on Symbolic Execution. Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Sciences*, June 1989.

10. Arnaud Cavani, Patrick Courty. *Implementing Esterel Studio Validation of Call Control under the Bluetooth Protocol*. Esterel Technologies, 2001. Available at http://www.esterel-technologies.com.
11. H. Cho, G. Hatchel, E. Macii, B. Plessier, F. Somenzi. *Algorithms for Approximate FSM Traversal based on State Space Decomposition. Proceedings of the 30th ACM/IEEE Design Automation Conference*, DAC'93, 1993.
12. Olivier Coudert, Jean-Christophe Madre. *Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion. Proceedings of International Worshop on Formal Methods in VLSI Design*, January 1991.
13. O. Coudert, J.-C. Madre, H. Touati. *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab, 1993.
14. Bernard Dion, Sylvan Dissoubray. *Modeling and Implementing Critical Real-Time Systems with Esterel Studio*. Esterel Technologies. In *Real-Time Magazine* 99-1, 1999.
15. Luciano Lavagno, Ellen M. Sentovich. *ECL: A Specification Environment for System-Level Design*. Cadence Berkeley Laboratories. *Proceedings of the 36th Design Automation Conference*, DAC'99, June 1999.
16. Shankar G. Govindaraju, David L. Dill, Alan J. Hu, Mark A. Horrowitz. *Approximate Reachability with BDDs using Overlapping Projections. Proceedings of the 35th Design Automation Conference*, DAC'98, June 1998.
17. Shankar G. Govindaraju, David L. Dill, Jules P. Bergmann. *Improved Approximate Reachability using Auxiliary State Variables. Proceedings of the 36th Design Automation Conference*, DAC'99, June 1999.
18. P. Godefroid, M. Huth, R. Jagadeesan. *Abstraction-based Model Checking using Modal Transition Systems. Proceedings of the 12th International Conference on Concurrency Theory*, Concur'01, August 2001.
19. M. Huth, R. Jagadeesan, D. Schmidt. *Modal Transition Systems, a Foundation for Three-Valued Program Analysis. European Symposium on Programming*, 2000.
20. N. Halbwachs, F. Lagnier, P. Raymond. *Synchronous Observers and the Verification of Reactive Systems. Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology*, June 1993.
21. Yann Le Biannic, Eric Nassor, Emmanuel Ledinot, Sylvan Dissoubray. *UML Object Specification for Real-Time Software. RTS 2000 Show*, March 2000.
22. Luciano Lavagno, Ellen Sentovich. *ECL: A Specification Environment for System-Level Design. Proceedings of the 36th Design Automation Conference*, DAC'99, June 1999.
23. S. Malik. *Analysis of Cyclic Combinational Circuits. IEEE Transactions on Computer-Aided Design*, 13(7), July 1994.
24. Thomas R. Shiple, Gérard Berry, Hervé Touati. *Constructive Analysis of Cyclic Circuits. Proceedings of the International Design and Testing Conference*, IDTC'96, 1996.
25. E. Sentovich, H. Toma, G. Berry. *Efficient Latch Optimization Using Incompatible Sets. Proceedings of the 34th Design Automation Conference*, DAC'97, 1997.