Compositional semantics of ESTEREL and verification by compositional reductions. [*]

R. de Simone and A. Ressouche

INRIA Sophia-Antipolis
B.P.93
06902 Sophia-Antipolis Cdx
FRANCE

Abstract. We present a compositional semantics of the ESTEREL synchronous reactive language, in the process algebraic style of *Structured Operational Semantics*. We then study its interplay with various reductional transformations on the underlying automata model, focusing on compositionality and congruence properties. These properties allow early nested reductions to take place at intermediate stages during the construction of a (reduced) model, a key point in cutting down the combinatorial explosion which plagues verification of parallel programs.
We consider the following transformations: bisimulation minimisation (state quotient), hiding of signals made invisible (abstraction), trimming of behaviours disallowed from external context (filtering). We illustrate part of the approach on a simple hardware bus arbiter specification. The verification method was implemented in STRL-MAUTO, a version of the AUTO tool customized to the "synchronous reactive" structure of actions.

1 Introduction

Verification of parallel systems is often plagued by the famous combinatorial blow-up arising in the representation of the global state space model. Many approaches have been proposed to cut down this complexity. One of the most fruitful is *compositional reduction*, by which subsystems minimisation is carried as early as possible along the process structure. This is made possible when the parallel systems are expressed in a process algebraic syntax and endowed with a compositional form of *Structured Operational Semantics*. Such an approach has been followed for the "classical" process algebras CCS, MEIJE, LOTOS amongst others, leading to tool support.

We want here to adapt these techniques -known as *verification by reductions*- to the specific case of *synchronous reactive systems*, and more specifically to the ESTEREL language (in its abstract, process-algebraic form). The main distinction in terms of *transition system* interpretation in between such formalisms and usual process algebras lays in the structure of action labels, which are now triples of: *necessarily present* signals; *necessarily absent* signals; emitted signals (reaction). An event needs to obey certain coherency properties, but must not necessarily be complete, in the sense that some signals may remain undetermined in it, indifferently present or absent. This allows economic representation of several possible input flows at once. We call *reactive*

[*] This work was partly supported by the ESPRIT BRA project Concur2

automata the finite transition systems bearing such type of labels. They correspond esentially to a rephrasing of *multiple input/output sequential machines* to an algebraic formulation of behaviour labels.

We provide the behavioural semantic interpretation of ESTEREL into reactive automata in a *SOS* style that matches our compositional purposes. This semantics differs in presentation from the original [1]. Here the input flow is synthesized upward from components to parallel products, rather than *inherited* by subprocesses from a global one. A keypoint in this compositional approach resides in the proper definition of *well-caused* processes, answering the well-known problem of signals being emitted as a result of their own presence/absence. To this end we add to each transition of reactive systems a *causal relation*, to remain acyclic, thereby completing the full definition of our reactive automata.

Next we define a number of reduction transformations on reactive automata, including *signal hiding* (abstraction), *symbolic bisimulation* (equivalence quotient), and *context filtering* (extended restriction), and study their compositional properties w.r.t. our operational semantics. We take *compositionality* in a broader sense, wondering *how much* reduction can actually be distributed along components prior to composition by a language operator (typically the `parallel` construct).

Symbolic bisimulation of reactive automata is more refined than usual plain bisimulation, and as a new notion seems very interesting in its own rights: consider the simple case of a state p, with $p \xrightarrow{i?.o!} p_1$ and $p \xrightarrow{i\#.o!} p_2$ while p_1 and p_2 are bisimulating each other (here $i\#$ means: *i is tested as absent*). This state p is somehow bisimilar to the one with a single transition labeled simply $o!$ leading to a state equivalent to p_1 and p_2, as any presence value for i has the same effect.

Finally we illustrate part of the approach on a simple hardware *bus arbiter* modeled in our language.

2 SOS behavioural semantics of ESTEREL

We now provide a compositional structured operational semantics for a large subset of (pure) ESTEREL. Extension to the full language is straightforward. We start by defining *events*, to become then the behavioural labels of *reactive automata*'s transitions. A structural interpretation of the process algebraic operators into reactive automata transformers is then given.

2.1 Reactive & broadcast structure of actions

In ESTEREL, as other synchronous reactive formalisms, behaviours are composed of events: an *event* E consists of a set of occurring input signals together with a set of output signals emitted back *in reaction*, synchronously with the inputs. Input signals not occurring in an event are supposed to be absent. We shall depart from this usual representation in that our events will explicitly specify *absent* inputs in addition to present ones, and leave unspecified signals when their presence value is immaterial for the transition to be taken. It should be clear that this approach allows to *factor* many possible input behaviours in a single transition description.

Signals will be supposed in the sequel to belong to some *finite* interface set $\mathcal{A}$. We use a predefined specific signal `_exit` to indicate (sequential) termination, which is thus dealt with internally much as other signal. Still, it can only be emitted, never received.

Definition 2.1 *An* event *E is a triple $(I,J,O) \in 2^{\mathcal{A}} \times 2^{\mathcal{A}} \times 2^{\mathcal{A}}$ such that I and J are disjoint, and $(O \setminus \{\_exit\}) \subset I$. An event consists of:* received *signals in I (which were tested as present in the course of the transition);* forbidden *signal in J (tested as absent);* emitted *signals in O. We let $\mathcal{E}$ be the set of events.*

An event is called saturated *if $I \cup J = \mathcal{A}$.*

An input event *is simply a couple $(I,J) \in 2^{\mathcal{A}} \times 2^{\mathcal{A}}$ with $I \cap J = \emptyset$. We let $\mathcal{IE}$ be the set of input events.*

Notation 2.1 *We typically write $E = I?.J\#.O!$ instead of (I,J,O) to reinforce type discrimination, and further, e.g. $i_1?.i_2?.j\#$ instead of $\{i_1,i_2\}?.\{j\}\#.\emptyset!$ (the product notation indicates simultaneity). We note $E \subset E'$ when $I \subset I', J \subset J', O \subset O'$, and $E \setminus \{S\}$ for $(I \setminus \{S\})?.(J \setminus \{S\})\#.(O \setminus \{S\})!$.*

We chose to include O in I. As an alternative one could use $I' = I \setminus O$ instead. This latter representation is more compact, but certain logical analogies we shall need in the sequel fit better the first choice.

ESTEREL (introduced below) is an imperative language containing an explicit parallel operator. This operator is *synchronous*, in the sense that both subprocesses proceeds at the same pace. Actions are broadcast, so that a given signal must be perceived consistently (as present or absent) by all parallel processes. At the event level, this introduces a specific definition of simultaneous product.

Definition 2.2 *Two events $E_1 = I_1?.J_1\#.O_1!$ and $E_2 = I_2?.J_2\#.O_2!$ are* compatible, *noted $E_1 \uparrow E_2$ iff:*

$$(J_1 \cap I_2) = \emptyset = (J_2 \cap I_1)$$

The simultaneity product *of two compatible events E_1 and E_2, noted $E_1.E_2$ is the event $E = (I_1 \cup I_2)?.(J_1 \cup J_2)\#.\,(O_1 \setminus \{\_exit\} \cup O_2 \setminus \{\_exit\} \cup (\{\_exit\} \cap O_1 \cap O_2))$*

The treatment of *_exit* in products insures *distributed termination*: a parallel process terminates only when both sides do. Note in particular that $(E \subset E') \Rightarrow (E \uparrow E')$ and $E_1, E_2 \subset E_1.E_2$.

With a logical *and* interpretation of product, the input part of an event (the I and J sets) builds exactly a characteristic propositional formula for saturated input events, as a conjunction of literals and negated literals. The simultaneity product respects this interpretation. On the other hand the analogy is much less clear on the output part of the event: a signal not in O is *not* irrelevant (it is certainly not emitted yet), but not incompatible in simultaneous product with this emission on the other part. We shall pursue this logical interpretation further, but *on signal receptions* only.

Definition 2.3 *A* logical event expression *is a couple (f,O) where f is a propositional formula (based on input signal names as atomic propositions), and O is a finite set of emitted signals. We note $L_{\mathcal{A}}$ the class of logical event expressions on $\mathcal{A}$.*

A logical event expression represents a finite set of reactive events (as a characteristic function). They can be retrieved back by simple prenex disjunctive normal form expansion of f, under simultaneous product interpretation of each summand.

Definition 2.4 *By extension, an event $E = I?.J\#.O$ is called* compatible *with a saturated imput event $E_0 = I_0?.J_0\#$ if $I_0 \cap J = \emptyset = (J_0 \cap I) \setminus O$*

In short, compatibility here only allows E to suppose **more** present signals than provided from outside, and only when emitted by the process itself, which builds partly its own environment.

2.2 Reactive automata

Definition 2.5 *A* reactive automaton *is a structure* $(Q, \mathcal{A}, T, init)$ *where* Q *is a (finite) set of* states, $init \in Q$ *is the* initial *state,* $\mathcal{A}$ *is a (finite) signal interface (or* sort*), and* $T \subset (Q \times Ev_{\mathcal{A}} \times Q)$ *is a transition relation labeled by reactive events on* $\mathcal{A}$.

A logical reactive automaton *is the similar structure, only with the transition labels in* $L_{\mathcal{A}}$ *instead of* $Ev_{\mathcal{A}}$.

Reactive automata are tightly linked to *sequential machines* in hardware theory, or various models of reactive formalisms [3]. Sequential machines are automata equipped with two functions, *Out* and *Next*, mapping respectively a couple $(State, IE)$, $IE \in \mathcal{IE}$ to a (synchronous) output event and to a next state. Our reactive automata, while modeling in essence the same objects, insist on algebraic structure of events in a way that will prove useful to define the semantics. A reactive automaton defines executions, from saturated input events describing the environment's offer as follows: a compatible input event occurring in an outgoing transition is selected, which sets an output event and a resulting state, these of the transition. A specific form of determinism is called for here.

Definition 2.6 *A reactive automaton is* input-deterministic *iff two distinct transitions leaving the same state cannot be compatible. Formally:*

$$\forall p \in Q, \forall t = (p, E, p_1), t' = (p, E', p_2) \in T, \neg(E \uparrow E')$$

Definition 2.7 *A reactive automaton is* input-complete *iff it can react to any incoming input event, up to the fact that* the process can itself partly build its own environment, by raising signals. *Formally:*

$$\forall E_0 \ \textit{saturated} \ \in \mathcal{E}, \forall p \in Q, \exists E, \ (p \xrightarrow{E} \ ), \ E \uparrow E_0$$

One should note that input-determinism does **not** imply the uniqueness of such E in the definition of input-completeness (different solutions may not be mutually compatible). Uniquenes will be gained from additional causality requirements later.

2.3 Syntax of ESTEREL as a Process Algebra

Esterel is a programming language designed for structured modeling of reactive systems (or sequential machines), as described in the previous section. Reactive systems are open systems, with successive event reactions taking place in (discrete) instants of time. The main novel features of Esterel are: explicit signal handling (raising/testing); explicit parallelism for modular decomposition; explicit atomicity defining successive (logical discrete) instants; internal cooperation by local signaling; priority handling by watchdog constraints (again on the signals themselves). Therefore Esterel turns the concept of *signals*, central to the model, into its main structuring paradigm for programming. Apart from these novel features, Esterel contains more classical procedural constructs, sequential composition and loops. General recursion is disallowed to guarantee the finite state model property. We left out the `trap/exit` and value-passing mechanisms of the full language.

The syntax of the Esterel algebra is the following:

$$\begin{aligned} P = \ & \mathtt{stop} \mid \mathtt{nothing} \mid \mathtt{emit\ S} \mid P \| P \mid P;P \mid \mathtt{present\ S\ then}\ P\ \mathtt{else}\ P\ \mathtt{end} \mid (P) \\ & \mid \mathtt{do}\ P\ \mathtt{watching\ S} \mid \mathtt{signal\ S\ in}\ P\ \mathtt{end} \mid \mathtt{loop}\ P\ \mathtt{end} \mid P[\mathsf{S}/\mathsf{S}] \end{aligned}$$

where S figures a syntactic class of mere signal names, to be instantiated in $\mathcal{A}$..

Formal semantics is provided below. We give now some informal concerns that led to its design. First of all, *sequential composition* is **not** an atomicity operator; a module can proceed in sequence inside the same instant. In Esterel jargon it is said that *semicolon is infinitely fast*. This assumption corresponds in hardware circuits to the fact that, at this logical-gate level of description, the output(s) of a gate is synchronous with its input(s), and can be "instantly" wired to other gates. Given this, each module is possibly endowed with quite complex behaviours in a single atomic step, raising and testing several signals in causal fashion. An informal reaction description may start like: *provided signal I occurs, then test signal J, and if J is here then emit O in answer, else if K* .... Of course signals may also be tested independently, if in parallel. As a limitation to this interpretation, it must discard ill-caused modules, in which a signal emission would result from its own presence value. This corresponds to *races* in circuits.

The `stop` construct implements atomicity. A module, or better said all its parallel branches, will maximally progress in sequence until next `stop` points, providing a common *end-of-reaction*. The `do/watching` construct implements priority preemption: the body is executed at successive instants *until* the guarding signal occurs, killing the body. For technical reasons this preemption is not active at the very same instant when control reaches this (sub)term.

2.4 Ill-caused processes

As already mentioned, internal signal chatters may cause ill-caused situations. We describe below three generic pathological examples, which must be eliminated as causally incorrect. This calls for the introduction of an additional *dependence* relation in between signals, which must be kept acyclic as a strict partial order.

`signal S in (present S then emit S else emit a)` This process has two external behaviours: $\emptyset$ and $a!$, depending on non-causal choices about S below the signal declaration. It is therefore not *input-deterministic*;

`signal S in (present S then nothing else emit S)` This process has no external behaviour, as S is emitted iff it is not here! It is therefore not *input-complete*;

`signal S in signal T in (present S else emit T|| present T else emit S)` One cannot find here any notion of smaller or greater fixpoint solution, due to possible reaction to signal absence.

Events and dependencies could be represented in a single, partially commutative structure. We prefer our interpretation using separate *events/dependence relations* objects since the dependence flow relation is not based on signals names, but occurrences of them (that is, the dependence may vary from transitions to transitions). Also, sufficient criteria for uniform causal correctness are known at static semantic level, which solve in practice the causality problem (by approximation) at a different level. We shall not enter details here.

In the sequel we let R be a binary relation on signals, and R^+ its non-reflexive transitive closure. Only a signal emitted as a consequence of its own testing will introduce reflexive couples in R^+, breaking the partial strict order property.

2.5 SOS behavioural semantics of ESTEREL as a Process Algebra

We let S, T, ... be variables ranging over signal names in $\mathcal{A}$.. We let $P, Q, \ldots$ be variables ranging over programs (syntaxic states), and E, E_1, E_2 range over events. The sets I,

J and O always refer implicitly to an event E, possibly with matching subscripts. The deduction rules will "prove" behaviours of the shape $P \xrightarrow{E, R} Q$ in a natural deduction style.

We now proceed with the semantic rules for each language construct.

$$\texttt{stop} \xrightarrow{\emptyset?.\emptyset\#.\emptyset!\ ,\ \emptyset} \texttt{nothing} \tag{1}$$

$$\texttt{nothing} \xrightarrow{\emptyset?.\emptyset\#.\{\_exit\}!\ ,\ \emptyset} \texttt{nothing} \tag{2}$$

$$\texttt{emit S} \xrightarrow{\{S\}?.\emptyset\#.\{S, \_exit\}!\ ,\ \emptyset} \texttt{nothing} \tag{3}$$

$$\frac{P \xrightarrow{E\ ,\ R} P',\quad \_exit \notin O}{P;Q \xrightarrow{E\ ,\ R} P';Q} \tag{4}$$

$$\frac{P \xrightarrow{E_1\ ,\ R_1} P', Q \xrightarrow{E_2\ ,\ R_2} Q', \_exit \in O_1\ ,\ E_1 \uparrow E_2\ ,\ O_2 \cap I_1 = \emptyset}{P;Q \xrightarrow{(E_1 \setminus \{\_exit\}).E_2\ ,\ (R_1 \cup R_2 \cup ((I_1 \cup J_1) \times O_2)^+)} Q'} \tag{5}$$

$$\frac{P \xrightarrow{E_1\ ,\ R_1} P',\quad Q \xrightarrow{E_2\ ,\ R_2} Q',\quad E_1 \uparrow E_2}{P||Q \xrightarrow{E_1.E_2\ ,\ (R_1 \cup R_2)^+} P'||Q'} \tag{6}$$

$$\frac{P \xrightarrow{E\ ,\ R} P',\quad S \notin J\#}{\texttt{present S then } P \texttt{ else } Q \texttt{ end} \xrightarrow{(I \cup \{S\})?.J\#.O!\ ,\ (R \cup (\{S\} \times O))^+} P'} \tag{7}$$

$$\frac{Q \xrightarrow{E\ ,\ R} Q',\quad S \notin I?}{\texttt{present S then } P \texttt{ else } Q \texttt{ end} \xrightarrow{I?.(J \cup \{S\})\#.O!\ ,\ (R \cup (\{S\} \times O))^+} Q'} \tag{8}$$

$$\frac{P \xrightarrow{E\ ,\ R} P',\quad (S \in I? \Rightarrow S \in O!),\quad \neg R^+(S,S)}{\texttt{signal S in } P \xrightarrow{E \setminus \{S\}\ ,\ R \setminus (\{S\} \times \mathcal{A} \cup \mathcal{A} \times \{S\})} \texttt{signal S in } P'} \tag{9}$$

$$\frac{P \xrightarrow{E\ ,\ R} P',\quad \_exit \notin O}{\texttt{loop } P \texttt{ end} \xrightarrow{E\ ,\ R} P'; \texttt{loop } P \texttt{ end}} \tag{10}$$

$$\frac{\texttt{P} \xrightarrow{E\ ,\ R} \texttt{P}'}{\texttt{do } P \texttt{ watching S} \xrightarrow{E\ ,\ R} \texttt{present S then nothing else do } P' \texttt{ watching S}} \tag{11}$$

The form of rule 9 is sufficient because all operators maintain *coherency* of events, implying already that $(S \in J\#) \Rightarrow \neg(S \in O!)$. Rules 10 and 11 only perform unfolding, in a tail-recursion way that respects rationality, and thus finite state representation. The `loop` construct assumes its body not to terminate in its initial reaction, to avoid unguardedness. The `watching` construct does not become active at the instant the instruction gets control.

The previous semantics associates a (finite) *reactive automaton* $\tilde{P}$ with every process term P.

Definition 2.8 *An* ESTEREL *process is* well-caused *if for all deduction of a global transition in the reactive automaton, the causal relation R^+ remains acyclic at all nodes of the deduction tree.*

This definition in essence asks for R to remain a strict partial order for all subterms of a given precess, except for those which are never used in deductions (dead code).

Proposition 2.1 *Let P be a* well-caused ESTEREL *process and $\tilde{P}$ its associated reactive automaton. Then $\forall p$ state of $\tilde{P}$, $\forall E_0$ saturated input event ,$\exists E$ unique, $p \xrightarrow{E}$ and $E \uparrow E_0$. Then of course $\tilde{P}$ is in particular input-complete and input-deterministic.*

Proof (sketch) By structural case analysis. The proof uses the following lemma

Lemma 2.1 *Let $(E,R),(E',R')$ label transitions outgoing from the same state in $\tilde{P}$. Suppose $\exists s \in \dot{A}$, $E = s\#.(E \setminus \{s\})$, $E' = s?.s!.(E' \setminus \{s\})$ and $(E \setminus \{s\}) \uparrow (E' \setminus \{s\})$. Then either s is a minimal point in both R and R', or R' has a cycle on s*

The lemma allows to settle the problem of input-determinism for the **signal** declaration operator. Other cases of interest are the **present** test operator, where it is proved that to refute input-completeness in the global automaton, one must run into a cycle in the dependence because of the phenomenon encountered in the previous counterexamples.

3 Compositionality of reductions

3.1 General framework

We now introduce 3 distinct types of reductions/transformations on reactive automata, and study the corresponding compositionality properties of the language constructs. We take here the word *compositionality* in a broad sense: our reductions will in general include parameters, and these will adapt from global processes to subterms. In fact the study of *compositionality* will largely amount to the characterisation of parameter transformations which allow "good" subautomata reductions prior to composition. These transformations may also rely on static semantics information gathered from the subterms themselves (typically, their external signal interface). A formal definition of optimality of reductions in general is out of scope. Instead we will argue in each case.

While compositionality will indicate where reductions *may* take place to simplify subsystems before combining them, there is a general trade-off between the expected gain and the time spent in the reduction phase. In general the parallel constructs are the most critical syntaxic locations where prior simplification is helpful, while other operators will only pass transformed parameters down the syntax tree. We will not study further such strategies.

We now introduce our three classes of reductions:

behaviour abstraction identifying similar (sets of sequences of) behaviours, for instance by *hiding* signals. In this paper we shall only deal with this latter special case. The parameters are thus *sets of signals* (to retain visible). Different choices of visible signals will provide different *partial views* of a system.

state minimisation according to behavioural equivalences, or bisimulations [6]. This reduction does not involve parameters.

behaviour filtering constraining (sequences of) behaviours according to some environment, itself described as a finite state structure. Here parameters are quite complex, consisting of automata with logical event acceptors as labels.

3.2 Hiding and Compositionality

Let V be a set of *visible signals*, and H_V the transformation function which erases signals not in V from transition events. Obviously H_V may decrease the number of transitions by merging these which only differ on invisible signals. It can also loose determinism, and keeps the number of states unchanged.

The basic compositionality problem with this transformation is that a signal declared hidden could in general be tested several times in the deduction tree of a given transition. Identical choices are imperative then, so the signal should be reintroduced as visible (but only in the necessary scope).

A signal is called free when occurring outside the scope of a local declaration `signal S in` .We note $\mathcal{I}_P$ (respectively $\mathcal{O}_P$) the sets of *free* signals occurring in a `present S` or `watching S` subterm of P(respectively in an `emit S`).

We let $Com(P,Q) \equiv_{def} ((\mathcal{I}_P \cup \mathcal{O}_P) \cap \mathcal{I}_Q) \cup ((\mathcal{I}_Q \cup \mathcal{O}_Q) \cap \mathcal{I}_P)$. This auxiliary function provides the list of signals whose presence value is shared by P and Q.

Proposition 3.1

$$\begin{aligned}
H_V(\texttt{stop}) &= \texttt{stop}\\
H_V(\texttt{nothing}) &= \texttt{nothing}\\
H_V(\texttt{emit S}) &= \texttt{nothing} \textit{ if } S \notin V\\
&\ \texttt{emit S} \textit{ otherwise}\\
H_V(\texttt{loop } P) &= \texttt{loop } H_V(P)\\
H_V(\texttt{signal S in } P) &= H_V\ (\texttt{signal S in } H_V(P))\\
H_V(P||Q) &= H_V\left(H_{V\cup Com(P,Q)}(P)||H_{V\cup Com(P,Q)}(Q)\right)\\
H_V(P;Q) &= H_V\left(H_{V\cup Com(P,Q)}(P);H_{V\cup Com(P,Q)}(Q)\right)\\
H_V(\texttt{do } P \texttt{ watching S}) &= \texttt{do } H_V(P) \texttt{ watching S } \textit{if } \texttt{S} \notin (\mathcal{I}_P \cup \mathcal{O}_P)\\
&\ H_V\left(\texttt{do } H_{V\cup\{\texttt{S}\}}(P) \texttt{ watching S}\right) \textit{ otherwise}\\
H_V(\texttt{present S then } P \texttt{else } Q) &= \texttt{present S then } H_V(P)\texttt{else } H_V(Q)\\
&\ \textit{if } \texttt{S} \notin (\mathcal{I}_P \cup \mathcal{O}_P \cup \mathcal{I}_Q \cup \mathcal{O}_Q)\\
&\ H_V\left(\texttt{present S then } H_{V\cup\{\texttt{S}\}}(P)\texttt{else } H_{V\cup\{\texttt{S}\}}(Q)\right)\\
&\ \textit{otherwise.}
\end{aligned}$$

According to these identities, one may choose to descend some hidings down the syntax tree. It may come as a surprise that `signal S` construct does not add `S` to `V`, but this shall in general be dealt with by nested parallel constructs. A related problem lies in the "best binary" parallel division of n subterms in parallel, which we shall ignore in this paper.

3.3 Symbolic Bisimulation and Compositionality

Plain strong bisimulation on automata is simply the coarsest equivalence in between states that respects event abilities. It is now a well-established notion, see e.g. [6]. Plain strong bisimulation is trivially a congruence w.r.t. all ESTEREL operators, as their semantics is given in a purely behavioural format. Also it permutes with the *hiding* reduction defined above, and is compatible with non-determinism.

In *reactive* automata, an event may represent *several* possible reactions due to unsaturated input signals. This makes room for a new *symbolic* (strong) bisimulation, when different combination of events would actually build up plain bisimulation on saturated reactions.

As an example, consider the very simple reactive automaton
$P = (\{s_0, s_1\}, s_0, \{i, o\}, T = \{(s_0, i?.o!, s_1)\,,\,(s_0, i\#.o!, s_1)\})$.
Its output and resulting state are the same no matter which transition is taken, no matter which presence value the signal i may take. We want to identify P with the simpler
$Q = (\{s_0, s_1\}, s_0, \{i, o\}, T = \{(s_0, o!, s_1)\})$.
About the dependence relation here: any potential dependence (i, o) should be discarded. As Q is equally determisitic or complete as P, this is harmless, and can only safely allow more processes to be causally "correct".

This example can be generalized to all cases where transitions have labeling events with identical outputs and *symbolic bisimilar* resulting states (not just identical). Then the input events can be grouped in *sums of products*, and more generally into propositional formulas (justifying the introduction of *logical reactive automata*). The equality problem for propositional boolean functions have been of course extensively studied, either recently through *Binary Decision Diagrams*'s canonical forms, or in the past with *Prime Implicants* theory.

Notation 3.1 *Let B be a (symmetric) binary relation on states. We note IE_{B,O,p_0} the function $S \rightarrow \mathcal{IE}$ defined by*

$$IE_{B,O,p_0}(p) = \{I?.J\# \;/\; \exists p', B(p_0, p') \text{ and } p \xrightarrow{I?.J\#.O!} p'\}$$

Definition 3.1 *Let $A = (S, s_{init}, Ev_{\mathcal{A}}, T)$ be a reactive automaton. A (symmetric) binary relation $B \subset S \times S$ is a symbolic bisimulation iff:*

$$\forall (p,q) \in B, \forall O \subset \mathcal{A}_{out}, \forall r \in S, \quad \sum_{I_k?.J_k\# \in IE_{B,O,r}(p)} I_k?.J_k\# = \sum_{I_k?.J_k\# \in IE_{B,O,r}(q)} I_k?.J_k\#$$

Proposition 3.2 *(Strong) Symbolic Bisimulation is a congruence with respect to all operators of* ESTEREL *on well-caused processes.*

Logical reactive automata have canonical form w.r.t. symbolic bisimulation up to *equivalence of boolean formula, and the corresponding minimisation is compositional w.r.t. all operators of* ESTEREL *(on well-caused processes).*

As already mentioned, BDDs provide a framework where equivalence of boolean formulae becomes identity. The situation is more complex for simple reactive automata, since there is no unique minimal "sum of products" form for a boolean formula.

3.4 Context Filtering and Compositionality

Often in practice one is only interested to the behaviour of a given reactive system according to a supposed *context*, representing all possible reaction offers from a supposed environment. Such features are partially provided in ESTEREL or LUSTRE [2] at programming level.

Also, when putting two reactive processes in parallel, it can be highly beneficial to first create each only in a context compatible with the other, to limit the size of subcomponents. This involves a creative step, where one guesses context properties of sibling processes. Nevertheless our approach will formally validate these assumptions in a second phase.

Distributing global contexts *Context-dependent* bisimulations [4] introduced a notion of relativity of behaviours in a process algebraic setting. Similarly, so-called *Don'tCare* sets provided relativity of behaviours, this time for hardware circuit descriptions. Inspiring from these two sources, we introduce *context filtering* as a reduction of behaviours.

Definition 3.2 *A* reactive context *C is an automaton $(S, s_{init}, \mathcal{A}, T)$ where the transition labels are (satisfiable) propositional formulae based on the basic predicates $S?, S \in \mathcal{A}_{in}$ and $S!, S \in \mathcal{A}_{out}$*

Thus, the only difference with logical reactive automata is the symmetric treatment of output and input signals. In particular a context may impose that a signal *cannot* be emitted in a given reaction. We do not require a reactive context in general to be input-complete or input-deterministic.

We now proceed to define the F_C filtering transformation.

Definition 3.3 *Let C be a context and P a logical reactive automaton. The* filtering *of P by C (noted $F_C(P)$) is the automaton defined by the* SOS *rule:*

$$\frac{P \xrightarrow{(f,O!)} P' \quad C \xrightarrow{g} C' \quad (f \wedge g/O) \text{ satisfiable}}{F_C(P) \xrightarrow{(f \wedge g/O, O!)} F_{C'}(P')}$$

where g/O is the propositional formula in which all basic predicates $S!$ have been replaced by: `true` *if $S \in O$,* `false` *otherwise.*

Notice that while filtering may restrict transitions and reachable states, it constructs an automaton based on the *cartesian product* of the process with the context. The definition could be carried to simple reactive automata by expanding the resulting labels into sums of products, then sums into several transitions.

Our compositionality concern now is to distribute C along the structure of the process. Due to the complexity of the parameters here (operational contexts), we shall only show how a context C can be distributed from a parallel process to a context $Left(C)$ for its left subcomponent (*Right* case is symmetric). The transformation uses static information on the right components (its *output* sort, noted $\mathcal{O}_{Right}$).

We call a basic predicate *in positive position* in a propositional formula if it appears inside an *even* number of negations.

Definition 3.4
$$Left(C) = (S, s_{init}, \mathcal{A}, T')$$
where T' is deduced from T by replacing in each labeling formula all basic predicates in $\mathcal{O}_{Right}$ in positive position by `true`.

In essence, our definition consists in weakening the demand for outputs, to those which *may not* be left to the other side to emit.

We briefly sketch the transformation for other operators. Distributing contexts alongside sequential products is hard somehow: the head process should live with the same context, while the tail processes should respect *any* postfix part of it (making *any* context state potentially initial). Positive outputs are dealt with in the same fashion as for parallel construct. The **present** test operator selects parts of the context according to compatibiliy of the initial transitions. The **watching** operator "cuts" the context after non-initial transitions assuming presence of the signal.

Local contexts As previously mentioned, providing "local" contexts for subcomponents in parallel can be very efficient in cutting down complexity (an illustration for the Bus arbiter example described below is provided in figure 4). But validity of these "creative" local assumptions must then be checked

A general way to check this validity, proposed by Larsen, is to keep track of - only- these transitions which cross from the "allowed" part to the "inaccessible" part of the filtered reactive automaton. These transitions are recorded *without* their target states (and further behaviours). The validity check consists in verifying that no such transitions is then used as part of a filtered global behaviour.

Formally, we complete the specification of F_C with another rule.

Definition 3.5 *Let C be a context and note Ref_C (for "refused by C") be the formula $\neg \bigvee g_i$, over all g_i such that $C \xrightarrow{g_i}$. Let* `Error` *be a new distinguished "syntactic" state. Then*
$$\frac{P \xrightarrow{(f, O!)} P' \quad (f \wedge Ref_C/O) \ satisfiable}{F_C(P) \xrightarrow{(f \wedge Ref_C/O, O!)} \texttt{Error}}$$

Checking for presence of constant `Error` in the global system will then indicate inappropriate local context assumptions.

In the *deterministic, causally correct* reactive case an even simpler solution exists: if a behaviour which could effectively be part of a global behaviour is improperly filtered at a subsystem level, then no other global behaviour can make up for the loss, so that the global reactive automaton becomes input-incomplete. This provides a simple test, to be performed only once, for the validity of local filter applications. Of course the determinacy requirements implies that this improved method cannot be combined with *hiding* for instance.

3.5 A Bus Arbiter example

"Gate-Level" description The purpose of this synchronous circuit is to select a single output `AckOut`$_i$ amongst possibly several `RequestIn`$_j$ inputs received from n users. It chooses basically the one of lowest index, but ensures fairness through use of a token ring: the owner of the ring gets priority over the regular previous selection system, which it must therefore disallow (or "override"). The circuit "Gate-Level" description is provided in figures 1 (for the basic component) and 2 for the ring, here of 4 components ($n = 4$). The **Override(In/Out)** wire design instantly propagates down the cells and, in the end, aborts the **GrantIn** signals through an auxiliary **SignalInverter** module. Another auxiliary **Init** module provides the seminal **Token**.

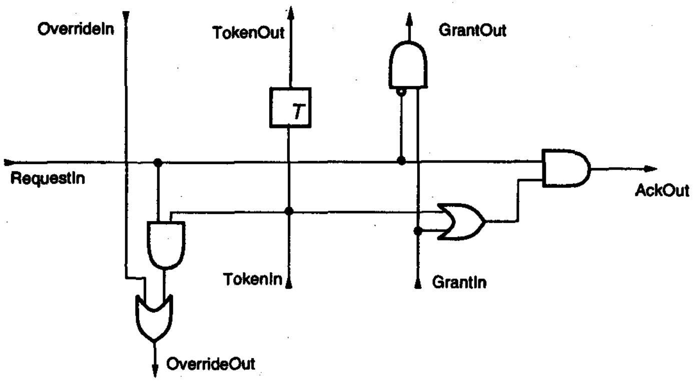

Fig. 1. The basic element

The programs in figure 3 provides an ESTEREL encoding of the description. We shall not carry the equivalence proof here, which would require recalling the formal definition of sequential machine interpretation of circuits. We added derived constructs to ESTEREL for sake of concision:
halt ≡ loop stop end,
do P watching immediate S ≡ present S then nothing else do P watching S,
await [immediate] S ≡ do halt watching [immediate] S,
every S do P ≡ await S; loop (do (P;halt) watching S).
Also, the keyword **run** introduces non-recursive module instantiation, possibly involving alphabetic renaming.

4 Conclusions

We presented a compositional semantics of ESTEREL and studied its combination with several transformation and reductions operations on labeled transition systems, allowing the same verificational approach on reactive systems as we knew worked well on "classical" CCS-like process algebras. We obtained this through a structure of behaviours taking signal absence into account. This work could be compared with compositional semantics of ARGOS [5], a formalism close to STATECHARTS. Our approach allows non-determinism in some constructions.

The structural syntax of ESTEREL allows to state (and solve) the problem of compositional reduction, and thus to deal with state explosion in a way orthogonal to symbolic model-checking techniques. These symbolic structures (*BDD*s) could still serve to

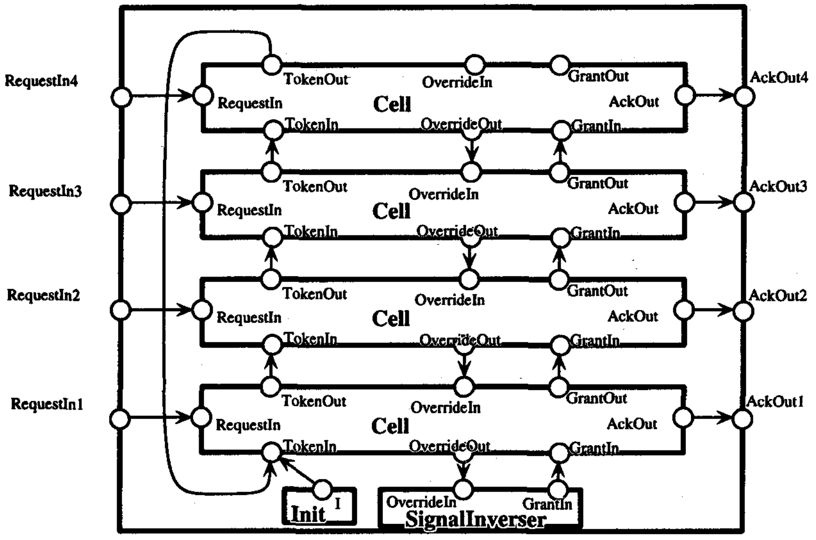

Fig. 2. The Token ring network

implement event labels in processes and contexts, and support symbolic bisimulation checking.

Finally we should stress that during our early experiments the local filterings proved perhaps the most efficient technique for cutting down complexity, while not automatically sound. There we started viewing our approach as a useful *user-guided proof-assistant*, with human creativity providing extra assumptions (on local signals) to speed up construction and cut down space requirements, while the system still performs quite fancy semantic automatic reductions (such as *bisimulation minimisation* or *signals hiding abstraction*).

References

1. G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 1992.
2. N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 1992.
3. D. Harel. Statecharts: A visual approach to complex systems. *SCP*, 1987.
4. K. Larsen. *Context-dependent Bisimulation between Processes.* PhD thesis, Edinburgh University, 1986.
5. F. Maraninchi. Operational and Compositional Semantics of Synchronous Automaton Generation. In *CONCUR'92*, volume LNCS 630, 1992.
6. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

```
module Cell:
input RequestIn, GrantIn, TokenIn, OverrideIn;
output GrantOut, TokenOut, AckOut, OverrideOut;
 every immediate GrantIn do
   present RequestIn then emit AckOut else emit GrantOut end
 end
||
 loop
   await immediate TokenIn;
   present RequestIn then emit AckOut; emit OverrideOut end;
   await tick; emit TokenOut
 end
||
 every immediate OverrideIn do emit OverrideOut end
.
module Init:
output Token;
emit Token
.
module SignalInverser:
input Override;
output Grant;
loop
  present Override else emit Grant end;
  await tick
end.
module Arbiter4:
input RequestIn1, RequestIn2, RequestIn3, RequestIn4;
output AckOut1, AckOut2, AckOut3, AckOut4;
signal G1, G2, G3, G4, O1, O2, O3, O4, T1, T2, T3, T4, Gnull, Onull in
     run Init[signal T1/Token]
  || run SignalInverser[signal O1/Override, G1/Grant]
  || run Cell[signal RequestIn1/RequestIn, AckOut1/AckOut,
                     G1/GrantIn, G2/GrantOut, T1/TokenIn, T2/TokenOut,
                     O1/OverrideOut, O2/OverrideIn]
  || run Cell[signal RequestIn2/RequestIn, AckOut2/AckOut,
                     G2/GrantIn, G3/GrantOut, T2/TokenIn, T3/TokenOut,
                     O2/OverrideOut, O3/OverrideIn]
  || run Cell[signal RequestIn3/RequestIn, AckOut3/AckOut,
                     G3/GrantIn, G4/GrantOut, T3/TokenIn, T4/TokenOut,
                     O3/OverrideOut, O4/OverrideIn]
  || run Cell[signal RequestIn4/RequestIn, AckOut4/AckOut,
                     G4/GrantIn, Gnull/GrantOut, T4/TokenIn, T1/TokenOut,
                     O4/OverrideOut, Onull/OverrideIn]
end.
```

Fig. 3. Elementary modules and 4-cells network.