Synchronous Programming

Properties within a Reaction

Charles André* — Robert de Simone**

Laboratoire I3S, UMR 6070, Université de Nice/CNRS
BP 121
06903 Sophia Antipolis Cedex, France

{andre@unice.fr}

**INRIA Sophia Antipolis*
BP 93
06902 Sophia Antipolis Cedex, France

{rs@sophia.inria.fr}

ABSTRACT. *Complex reactive behaviours can be expressed by synchronous imperative formalisms like* ESTEREL *or* SYNCCHARTS. *To make the best of these models and to avoid the pitfall of "causality cycles", the user has to understand the underlying semantics, known as the "constructive semantics". The first part of this paper is an informal introduction to this semantics. In the second part, this semantics is used to analyze "intra-instant properties" (partial ordering of simultaneous action executions). It appears that the compiler, which implements the constructive semantics, can carry out such analyses. Our method is illustrated by a zFIFO (0 fall-through time FIFO queue), which is a system envolving numerous simultaneous actions whose execution order may be critical.*

RÉSUMÉ. *Les formalismes impératifs synchrones (*ESTEREL *et* SYNCCHARTS*) permettent d'exprimer des comportements réactifs complexes. La connaissance de la sémantique de ces modèles est indispensable si on veut exploiter pleinement leur richesse expressive et éviter les "cycles de causalité". La première partie de ce papier rappelle les grandes lignes de la sémantique constructive. Cette sémantique est utilisée dans la seconde partie pour analyser des propriétés dans l'instant portant sur les ordres partiels d'exécutions d'actions simultanées. Il est montré comment les fonctionnalités du compilateur, issues de la sémantique constructive, permettent une telle analyse. La technique est illustrée par un exemple de zFIFO (*0 fall-through time FIFO queue*) un système qui présente de nombreuses actions simultanées dont l'ordonnancement est critique.*

KEYWORDS: *Reactive systems, synchrony,* ESTEREL, SYNCCHARTS, *constructive semantics, model checking.*

MOTS-CLÉS : *systèmes réactifs, synchrone,* ESTEREL, SYNCCHARTS, *sémantique constructive, vérification de modèle.*

1. Introduction

The *Synchronous Paradigm*, introduced in the mid 80's, has been succesfully applied to control-dominated systems, in which safety and predictability are essential. The synchronous approach relies on the notion of *instant*. Time is considered as discrete. The system performs a series of *reactions*. With each reaction is associated one instant. Given a sequence of input events (stimuli), a synchronous program generates a fully deterministic sequence (total order) of reactions. An imperative synchronous language, like ESTEREL (Boussinot and *al.*, 1991), introduces another kind of ordering: a causal order *within* one instant: elementary actions that compose a reaction are *partially ordered*. Usually, a reaction is considered as atomic, so that instantaneous broadcasting of signals, and even instantaneous protocolar dialogs are meaningful. This is a key point that explains both the expressiveness and the simplicity of the language.

The verification of (logical) temporal properties of a synchronous program has become a classical and useful process. For instance, the symbolic model-checker XEVE (Bouali, 1998), available in the ESTEREL programming environment, allows the designer to formally establish safety properties of his/her program. These properties can be either combinatorial (*e.g.,* the mutual exclusion of two actions, at each instant), or sequential (*i.e.,* properties envolving the instant ordering). In both cases, reactions are taken as atomic.

Analyzing the behaviour *within* a reaction (intra-instant properties) is a much less explored domain, up to now almost restricted to the designers of synchronous language compilers. One of their major concern is about the existence of *causality cycles*. This is an issue inherent to the synchronous approach: several actions can be simultaneously (*i.e.,* at the same instant) performed, but all orderings are not necessarily acceptable. Some orderings must be rejected because they lead to inconsistent behaviour (*e.g.,* a signal that should be both present and absent at one instant). Other orderings are troubling for they violate the causality principle: the effect preceds its cause in the reaction. To cope with these problems, G. Berry has introduced the *constructive semantics* for the ESTEREL language (Berry, 1996). Our study on intra-instant properties relies on Berry's results.

Why to study the ordering of a set of actions that are perceived simultaneous? The interest of this study is obvious when actions assign values to variables, call functions or procedures (all 0-duration actions in ESTEREL). Reading and writing a memory cell during a reaction is typical: according to the ordering the result is different. With in mind a concern for producing safe code, the designer should be given a mean to check if the ordering chosen by the compiler respects the semantics of his/her application [1].

---

1. This is about the semantics of the appication, not the semantics of the language. We assume that the compiler generates code in accordance with the constructive semantics.

To prove the intra-instant properties, we adopt a classical approach by refutation: The program is augmented with explicit causalities that may conflict with the causality structure of the program.

The paper is organized as follows:

– In the first section, a small example illustrates the interests and the drawbacks of mixing parallel evolutions with pre-emptions and immediate reactions.

– The second section shows how the constructive semantics gives unambiguous interpretation to complex reactive behaviour. The constructive semantics itself is beyond the scope of this paper. The *causality refutation* is then explained and applied in the next section.

– The third part presents a case study taking the best of the synchronous approach without falling in the trap of causality cycle. The chosen example is a z-FIFO (zero fall-through time FIFO queue): it is a queue applying a first-in, first-out policy, with possible simultaneous put and get actions, even on an empty or a full queue.
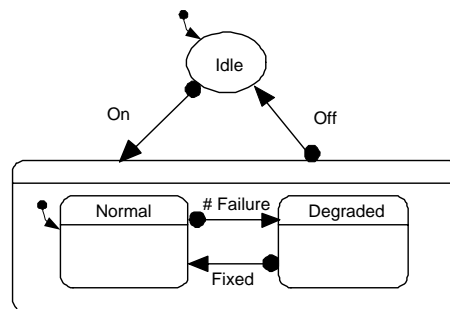
2. Complex reactive behaviours

Figure 1. *A plain controller*

The syncChart displayed in figure 1 represents high-level control for a system with modes `Idle`, `Normal`, and `Degraded`. The `Degraded` mode is instantly reached upon occurrence of signal `Failure`. SYNCCHARTS (André, 1996) are a graphical description model of synchronous systems close to Harel's Statecharts (Harel and *al.*, 1985), except with a richer expressive power concerning prioritary preemptions, and a full synchronous semantics. In particular SYNCCHARTS make a clear syntactic distinction between instantaneous presence testing (where a # symbol prefixes the triggering event), and next-instant state transitions (without this # symbol on labels). It is here important to note that *several* instantaneous transitions can be fired in sequel in the same reaction. A looping such behaviour has to be forbidden.

We should further emphasize here the expressive power of the model (figure 1), including hierarchic macrostates (`Normal`, `Degraded`), instantaneous preemption (for

instance from mode `Normal` to `Degraded`). Instantaneous transitions (with # event triggers) eliminate the need for transient behaviours which might prejudice the clarity of modelisation. Something similar exists in the semantics of Grafcet with the search-for-stability interpretation : intermediate steps are not externally visible then. One last advantage is that translations can generally be modular, in the sense that sub-state refinement due not contradict the translation part obtained from the unrefined system.

compilers currently produce semantically equivalent ESTEREL programs. Figure 2 displays the skeleton of such a program, showing in particular how local signals (`goto1`, `goto2`, `goto3`) are introduced. The % sign begins a one-line comment. This program contains altogether

– an iterative construct (`loop ... end loop`),
– concurrent instructions ($\|$),
– signal preemptions (`await`, `abort ... when`)
– immediate signal testing (`immediate Failure`).

This code is automatically produced, and so perhaps not easily readable. Consider the system being in its `Idle` mode. Control then resides at line 6 on the `await` instruction. All other branches of the parallel construct are idling. If then signals `On` and `Failure` occur simultaneously, signal `goto2` (ligne 6) gets emitted, then the parallel block terminates, and control reaches loop termination (line 28). The loop is thus instantly restarted (ligne 4), and so are the 4 concurrent branches. The present instruction at line 9 passes control to the following `abort` (line 10), which gets instantly preempted (line 14), triggering emission of signal `goto3`. The associated parallel branch then reaches completion (through lines 15 and 16). Signal `goto3` makes the `present` instruction at line 18 handle control to the `Degraded` mode (lines 19 and 20). The `pause` statement at line 27 stops the reaction of the fourth parallel branch. Then signal `goto1` cannot be emitted any longer in the current reaction, so the `present` statement at line 5 handles control to the implicit `else` part, terminating line 7 `present` statement, and thus the first parallel branch.

The previous description shows how complex instantaneous reactive behaviours can be, involving several (finite) subprogram executions, and inducing a partial order on simultaneously performed data operations. This expressive richness *inside* instant reactions avoids transient states and some inherent difficulties of *asynchronous* concurrency (nondeterminism most notably), by encapsulating reaction protocols inside given instants. But it may turn to be *too powerful*, when simultaneous causality relations can build up to a contradiction. Berry's *constructive semantics* allows to build an effective static scheduling between operations for each reaction, or prove that none is feasible (programs are then considered causally incorrect).

In the next section we shall only sketch the basic principles of this semantics, focusing on how it provides a partial ordering of data actions.

```
1        ... % declarative part
2       signal goto1, goto2, goto3 in
3         emit goto1; %initial state
4         loop % forever
5            present goto1 then
6                await On; emit goto2
7            end present
8            ||
9            present goto2 then
10               abort
11                   run Normal % Normal mode
12               when
13                   case Off do emit goto1
14                   case immediate Failure do emit goto3
15               end abort
16           end present
17           ||
18           present goto3 then
19               abort
20                     run Degraded % Degraded mode
21               when
22                   case Off do emit goto1
23                   case Fixed do emit goto2
24               end abort
25           end present
26           ||
27           pause
28        end loop
29      end signal
```

Figure 2. *Esterel program skeleton*

3. The constructive approach

3.1. *General notions*

The operational semantics of the language must provide a way to determine *unique* output event and next-state configuration of a program from *any given* current state and input event provided by the environment. It must also provide a structural interpretation of programming constructs (in accordance with their intuitive meaning), and in a *causal* way: no "look-aheads" or "oracle guesses" should be allowed. While the regular control flow provides an obvious precedence ordering between instructions put in sequence, signal communication between parallel components may require more sophisticated scheduling needs, and even at times build up causality contradictions.

Consider the following program:

```
1 signal S in
2    present S then emit Yes else emit No end present;
3    emit S
4 end signal
```

It allows only one logically consistent solution: `Yes` present, `No` absent, and `S` present. And yet, this solution is disturbing because the test of the presence of `S` (line 2) uses the fact that `S` is emitted (line 3). In other words, the effect precedes its cause, and the behaviour is not constructed in accordance with the intuitive sequencing semantics of the semicolon. ESTEREL's compilers reject this program as a non-constructive one.

In constructive computations, signals other than input signals, are declared *present* when execution provokes an emission, **but** are declared *absent* whenever the progress of control flow provably discards all further emissions from this point on because of choices made in the current reaction so far (and none were performed in that same reaction). This last feature is very important to keep in mind, as it departs from an execution scheme tighty ressembling classical simulation (of active parts of programs). Syntactical acyclic programs form a subset in which more regular, "positive" simulation is feasible.

In case of valued signals the rule becomes that the value can only be used (and thus considered stable) when no further emissions could be performed in the current reaction (again according to control flow choices already established). If emissions on that signal were performed in the reaction then their compound value is used, else if the signal is currently absent then the value inherited from the previous instant is preserved.

3.2. *Circuit translation and formal semantics*

The semantics of the language is most simply explained on *schematic gate logics* circuit description, obtained by translation of Esterel control flow and signal propagation into explicit Boolean variables in a system of Boolean equations. In this representation there is a symmetry between presence (Boolean value 1) and absence (Boolean value 0) of activity. For instance, a signal status is represented by a Boolean variable which is the disjunction of (variables representing) all possible emissions on that signal. This disjunction becomes true whenever a conjunct is, but will become false only if all emission variables are.

Formaly, we let $\mathcal{I}$ (resp., $\mathcal{O}$) represent the set of input (resp. output) signals of a given ESTEREL program. We call *input event* a subset of $\mathcal{I}$, representing signals that are considered present, others being absent. We define *output events* similarly. Also we let $\mathcal{L}$ represent the set of *local signals* (we shall assume for simplicity that all local signal declarations provide distinct names). We shall often confuse an input event I with its characteristic function (or Boolean vector) $I : \mathcal{I} \rightarrow \{0,1\}$, defined by: $\forall s \in \mathcal{I}, I(s) = 1$ iff $s \in I$. We define similarly Boolean vectors O, L (local

signals), and X (Boolean vectors of currently active pause control points). A circuit can be seen as a system of Boolean equations:

$$
\begin{aligned}
O &= F(X, I, O, L) \\
L &= G(X, I, O, L) \\
X' &= H(X, I, O, L)
\end{aligned}
$$

The system is not always monotonic in the classical Boolean domain if there are loops in the dependency relation between variables (a variable depends on another if it uses it in its definition). But it is monotonic in the 3-valued domain ($\perp < 0$, $\perp \leq 1$) with adequate interpretation of Boolean connectives, and thus admits a least fixed-point solution. In essence this solution allows simplification of the form $(1 \vee exp) \rightarrow 1$, $(0 \wedge exp) \rightarrow 0$, **but not** $(exp \vee \neg exp) \rightarrow 1$, which would undo a dependency on a variable not yet fully defined. If this solution is such that completely defined input events (no $\perp$ values for inputs) provide completely defined outputs and next states when applied *inside* the reachable state space, then the system is called *constructively causal*. Further details about this semantics are beyond the scope of this paper. The reader may refer to the paper of G. Berry on *The Constructive Semantics of Pure Esterel* (Berry, 1996).

In what follows, we informally present an example of fact propagation. We focus on the `present ...then ...else ...end` construct, which will be used in the analysis of intra-instant properties (Section 3.3). Consider the instruction:

```
present S then statement1 else statement2 end present
```

1) If `S` is (certainly) present then `statement1` *must* be executed, and `statement2` *cannot* be executed at the current instant.

2) If `S` is (certainly) absent then `statement2` *must* be executed, and `statement1` *cannot* be executed at the current instant.

3) If the presence status of `S` is not known yet, *nothing* can be said about `statement1` and `statement2`. The compiler has to explore other parts of the program, hoping to become certain of either the absence, or the presence of `S`. If it fails to get this information, the program is said to be non-constructive, and therefore rejected.

3.3. *Causality refutation*

Let a and b be two signals in a constructive program P. Let $\sqsubset$ be a dependency relation between signals, within an instant. Let P' be an instrumented version of P, such that $a \sqsubset b$ is imposed in P'. This is done by guarding the emissions of b. The guard is passing when the presence status of a is known *prior to* the emission of b. A practical solution is to replace "`emit b`" by "`present a then emit b else emit b end present`" in P.

If P' is not constructive (whereas P is constructive), then there exists at least one reaction involving a and b in a dependency relation conflicting with $a \sqsubset b$.

If P' is constructive (like P), then in P, for all reactions involving the modified code, a and b are such that

– either a and b are not simultaneously present in the reaction,

– or a and b are simultaneously present in the reaction and $a \sqsubset b$,

– or a and b are simultaneously present in the reaction and independent (written $a \smile b$).

This refutation technique is applied in the following example (Section 4.3.2).

4. Illustrating example: the zFIFO

4.1. *Informal description*

A zero-fall-through time FIFO queue (zFIFO for short) is a memory cell array, with directed shift ability (values are "pushed" from the entrance of the FIFO to the exit at the other side of the array). This is a fairly standard FIFO description. What is special here is that data values flow instantly to the empty cell closest to the exit, and even straight across the FIFO to the exit in case it is all empty (hence the name *0 fall-through time*).

Data input (`Put`) and data withdraw (`Get`) operations can be simultaneous, which leads to special "borderline" cases when the FIFO is full or empty. These cases must be accepted, and this poses a strand on data operations ordering.

4.2. *Formal specification*

We consider zFIFOs accepting data objects of a given type T. We note $f_{n,k}$ for a zFIFO of size n with k occupied cells; $f_{n,0}$ is empty, $f_{n,n}$ is full. Cells are numbered from the exit to the entry point of the zFIFO, so that, in $f_{n,k}$ with $k > 0$ cells 0 to $k-1$ are occupied. We note $f_{n,k}(j)$ the content of cell $j, j \leq k$.

We introduce two operations on zFIFO :

– append : zFIFO $\times T \rightarrow$ zFIFO is such that, if $k < n$, then $\mathrm{append}(\mathrm{f}_{n,k}, v)$ is $\mathrm{f}'_{n,k+1}$ with $\mathrm{f}'_{n,k+1}(k) = v$ and $\forall j : 0..k-1, \mathrm{f}'_{n,k+1}(j) = \mathrm{f}_{n,k}(j)$.

– shift : zFIFO $\rightarrow$ zFIFO is such that, if $k > 0$ then $\mathrm{shift}(\mathrm{f}_{n,k})$ is $\mathrm{f}'_{n,k-1}$ with $\forall j : 0..k-2, \mathrm{f}'_{n,k-1}(j) = \mathrm{f}_{n,k}(j+1)$.

We now consider a zFIFO as a reactive system with two input event signals `Put:T` and `Get`, triggering data operations with the same name, and two corresponding output signals `Taken` and `Got:T` notifying back success of operations.

The following SOS rewrite rules describe the zFIFO semantic behaviour. Input signals appear on top of the transition arrows, and output signals below.

$$0 < k \leq n \Longrightarrow f_{n,k} \xrightarrow[\{Put(v),Get\}]{\{Got(f_{n,k}(0)),Taken\}} \text{append}(\text{shift}(f_{n,k}),v) \qquad (1)$$

$$f_{n,0} \xrightarrow[\{Put(v),Get\}]{\{Got(v),Taken\}} f_{n,0} \qquad (2)$$

$$0 \leq k < n \Longrightarrow f_{n,k} \xrightarrow[\{Put(v)\}]{\{Taken\}} \text{append}(f_{n,k},v) \qquad (3)$$

$$f_{n,n} \xrightarrow[\{Put(v)\}]{\emptyset} f_{n,n} \qquad (4)$$

$$0 < k \leq n \Longrightarrow f_{n,k} \xrightarrow[\{Get\}]{\{Got(f_{n,k}(0))\}} \text{shift}(f_{n,k}) \qquad (5)$$

$$f_{n,0} \xrightarrow[\{Get\}]{\emptyset} f_{n,0} \qquad (6)$$

Our solution has been designed as a collection of properly connected memory cells (Cells) whose behaviour is prescribed by syncCharts and ESTEREL modules. Details about the implementation are available in an extended version of this paper (André and *al.*, 2002).

4.3. *Formal verification*

The correctness of our specification (with respect to the semantics rules) was established in two steps :

1) Validation of behaviours across instants,

2) Validation of action ordering *inside* the instants.

For the first phase we applied the now casual technique of observers (Halbwachs, 1993), which are reactive programs embodying safety properties, set in parallel with the observed programs. The global system is checked using classical symbolic model-checking. This step is a preliminary to the second stage of verification.

The second phase needs to check that in each single reaction, the order between concurrent actions performing data read and write respects the FIFO discipline (first-in, first-out). We shall resort to constructive semantics here to prove our properties, again on a system composed with an observer.

During a reaction, a Cell can be read (signal R) by a `Load`; written (signal W) par un `Store`; read and then written (signal RbW, Read before Written); or even written and then read (signal WbR, Written before Read). Signals R, W, RbW, WbR are added to the program to make observations easier.

4.3.1. *Classical properties*

Thanks to XEVE the symbolic model-checker part of the ESTEREL platform, we have established several properties. The first two properties are about Cells' behaviour:

P1: For all Cells and all instants, R, W, RbW et WbR are exclusive.

P2: For all Cells, execution traces on $\{R,W,RbW,WbR\}$ are

- prefixes of (WbR*;W;RbW*;R)* , for the Cell with index 0;
- prefixes of (W;RbW*;R)*, for Cells with index $j > 0$.

These properties guarantee that there is no loss of information by overwriting. Moreover, **P2** ensures that each read value has been written beforehand. Note that "before" refers either to an earlier instant, or to a previous microstep within the instant. The latter is only possible through code instrumentation (introduction of the new signals RbW, WbR).

The next 5 properties concern the behaviour of a zFIFO $f_{n,k}$:

P3: For all Cells with index j, $0 < j < n$, $\texttt{Occupied}[j] \implies \texttt{Occupied}[j-1]$.

P4: For all Cells with index j, $0 \leq j < l < k$, $\texttt{Occupied}[l]$ and $\texttt{Get} \implies \mathrm{RbW}[j]$.

P5: An item can be withdrawn from the zFIFO if and only if `Get` is present, and either the file is not empty, or the file is empty but `Put` is present.

P6: An item can be put into the zFIFO if and only if `Put` is present, and either the file is not full, or the file is full but `Get` is present.

P7: not Occupied[0] and Put(v) and Get $\implies$ WbR[0]

Property **P3** is a structural property stating that the information is "packed" in low indices Cells. Property **P4** expresses information shifting when a `Get` is performed. Properties **P5** and **P6** are typically the expected behaviour for `Get` and `Put` operations. Finally, property **P7** shows that specification 2 is satisfied.

The other specifications (1,3,5), which refer to functions `append` and `shift`, can not be directly checked by XEVE: XEVE deals with Booleans but not with arbitrary types like T. Checking these properties is the duty of the second phase.

4.3.2. *Properties established by causal analysis*

Proving that adding information to a zFIFO by a `Put` operation is always done at the "tail" of the zFIFO is relatively easy. This is a safety property that can be expressed with auxiliary signals and checked by XEVE.

Only the shift operation is a really problematic. Let r_j (w_j respectively) be the reading (the writing respectively) of Cell j. Let $\prec$ be the (partial) order relation over actions within an instant.

Thanks to properties given in section 4.3.1 and especially property **P4**, it follows that: "For each occupied Cell with index j, the last one excepted, $r_j \prec w_j$ when executing a Get operation".

Now, consider the shift operation. It is not clear whether the semantics of this operation is respected by the concurrent executions of Cells or not. We have to prove that when executing a Get operation, for all occupied Cells, the following property holds: $\forall j, 1 \leq j < k, r_j \prec w_{j-1}$

That is the information written in Cell $j-1$ has been read in Cell j beforehand (during the same instant).

In order to establish this property we first note that action r_j (w_j respectively) can be executed only by an instruction belonging to the body of the module associated with Cell j. This strong "localization" of actions owes much to our approach based on $\ll$capsule$\gg$.

```
1 trap occupied in % Cell[j]
2     loop
3         await Load;
4         % our test insertion point
5         emit Read(vⱼ); % rⱼ
6         present Store then vⱼ := ?Store % wⱼ
7         else exit occupied end present
8     end loop
9 end trap
```

Figure 3. *Excerpt from the module of the Dual-Access Elementary Memory (DAEM)*

Figure 3 shows an excerpt from the code of the module DAEM, focusing on the behaviour of an occupied Cell. The program that contains this code successfully passes the compilation: it is constructive. Let call P this program.

Now, the comments line 4 is substituted by the ESTEREL's instruction:
```
present Check then nothing else nothing end present;
```
Check is the identifier of a dummy signal that will be replaced by an actual signal of the application when instantiating the module. The sequence operator (;) between instructions (4) and (5) imposes a causality relation: "$Check \prec r_j$".

We then carry out two compilations associating $Check$ with r_{j+1} in the first, and with r_{j-1} in the second. The former is constructive, the latter is not.

From the first compilation it follows that in P, for each reaction emitting r_j,

– either r_{j+1} and r_j are not simultaneously present,

– or r_{j+1} and r_j are simultaneously present, and $r_{j+1} \prec r_j$,

– or r_{j+1} and r_j are simultaneously present, and $r_{j+1} \smile r_j$.

From the second test we conclude that P contains a relation which is conflicting with $r_{j-1} \prec r_j$. The conjunction of the two conclusions, up to a change in indices, shows that in P, when r_{j+1} and r_j are simultaneously present, $r_{j+1} \prec r_j$ holds. This is precisely the case when executing the `shift` operation.

The relation $r_{j+1} \prec r_j$ holds for $0 \leq j < k - 1$, with k the number of occupied Cells. Moreover, we already know that $r_j \prec w_j$ for an occupied Cell in the presence of the `Get` signal. We infer that $r_{j+1} \prec w_j$ pour $0 \leq j \leq k - 1$, or with a change of variable $r_l \prec w_{l-1}$ pour $1 \leq l < k$. This is the ordering expected to guarantee a correct shift.

Note that the compiler has imposed $r_{j+1} \prec r_j$, while $r_{j+1} \prec w_j$ would have been sufficient. But anyway, this solution is acceptable. On complex programs, it is difficult for the programmer to guess (and check) the effective ordering of actions during a reaction. On the other hand, the ESTEREL's compiler has to do this job, and it does it very well. Thus, our technique to check intra-instant properties has been to use the compiler as a constraint checker.

5. Conclusion

We highlighted the expressive strongpoints of synchronous reactive modeling on the zFIFO case study. We also showed how expressive power then requires careful analysis, especially of correct ordering of data operations. We showed how to conduct such analysis using constructive semantics, and proper observers to record contradiction when good order fails.

Concerning the design specification, combination of graphical SYNCCHARTS and textual ESTEREL proved especially valuable. Hierarchical AND/OR states, parallelism and preemption allowed to do away with transient, unstable behaviours in a deterministic manner. The formal constructive semantics of both formalisms allows clear specification, even on complex reactions. The industrial plateform solution Esterel Studio [2] combines these approaches.

The ultimate goal behind formal (constructive) semantics of synchronous reactive languages with all its intricacies was to allow for safe, unambiguous programming of embedded systems. We performed successfully a number of verifications of classical safety requirements on our zFIFO. Then we turned to a more novel use of observers and constructive semantics for the verification of proper *partial ordering of actions inside an instant*. This type of verification was in fact mandatory to establish the basic property of the FIFO (first-in, first-out) discipline. It is important here to note that classical observer verification between instants could not establish such properties. To

---

2. http://www.esterel-technologies.com

the best of our knowledge this is the first attempt at proving correct ordering of actions inside a reaction by parallel observers that will introduce a constructive causality contradiction if and only if the property can be refuted.

The current work opens further research directions:

– *Other properties inside instants*, apart from action ordering, could be investigated in the context of safety-critical systems;

– *Methodological aspects*, as examplified by the design style of the model and its properties which was largely inpired by OO-modeling concepts. The application was designed as a collection of cooperating *capsules*. Nevertheless, we departed from such approaches (as ROOM for instance) by insisting on synchronous semantics, including instantaneous broadcast of signals and immediate reactive answers. An ambitious goal behind the current example is to provide for a fully deployed methodological framework for OO synchronous design;

– *Verification methods*, which can be developed by using the compiler itself. This is not entirely surprising as the compiler often needs powerful analysis, of the same nature as model-checking, to perform constructive causality of programs. Symbolic computation of the reachable state space is a simple example of this. Extending this approach to more properties is a current goal.

6. References

André C., "Representation and analysis of reactive behaviors: A synchronous approach", In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC.

André C. and De Simone R., "Synchronous Programming: Properties within a Reaction (extended version)", I3S Research Report 2002-5, Sophia Antipolis (F), March 2002.

Boussinot F. and De Simone R., "The ESTEREL language", *Proceeding of the IEEE*, 79(9):1293–1304, September 1991.

Berry G., *The Constructive Semantics of pure Esterel*, not yet published, available on the web, www.inria.fr/equipes/meije/esterel, Sophia Antipolis (F), 1996.

Bouali A., XEVE: An esterel verification environment. Int'l Conf. on Computer-Aided Verification (CAV'98), LNCS, vol. 1427, Vancouver (BC, Canada), 1998, also available as a technical report INRIA RT-214, 1997.

Halbwachs N., *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Amsterdam, 1993.

Harel D. and Pnueli A., "On the development of reactive systems in logic and models of concurrent systems", *NATO ASI Series, K.R Apt Ed., Springer-Verlag*, 13:477–498, 1985.