Formal Methods in System Design, V, 1–25 (199X)
© 199X Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

The Synchronous Approach to Designing Reactive Systems

AXEL POIGNÉ, MATTHEW MORLEY, OLIVIER MAFFEÏS {poigne,morley,maffeis}@gmd.de
LESZEK HOLENDERSKI, AND REINHARD BUDDE {holenderski,budde}@gmd.de
GMD – Forschungszentrum Informationstechnik GmbH,
SET – Institut für Systementwurfstechnik, Schloß Birlinghoven, D-53754 Sankt Augustin, Germany

Received December 10, 1996

Editor:

Abstract. Synchronous programming is available through several formally defined languages having very different characteristics: ESTEREL is imperative, while LUSTRE and SIGNAL are declarative in style; STATECHARTS and ARGOS are graphical languages that allow one to program by constructing hierarchical automata. Our motivation for taking the synchronous design paradigm further, integrating imperative, declarative (or dataflow), and graphical programming styles, is that real systems typically have components that match each of these profiles. This paper motivates our interest in the mixed language programming of embedded software around a number of examples, and sketches the semantical foundation of the SYNCHRONIE toolset which ensures a coherent computational model. This toolset supports a design trajectory that incorporates rapid prototyping and systematic testing for early design validation, an object oriented development methodology for long term software management, and formal verification at the level of automatically generated object code.

Keywords: Embedded Software, Reactive Systems, Synchronous Programming, Synchronous Automata.

1. Introduction

Reactive computer systems continuously respond to external stimuli generated by their environments. They are critical components of our technology dominated lives, be they in control systems such as ABS for cars, fly-by-wire in aircraft, railway signalling, power generation, shopfloor automation, or in such mundane things as washing machines and video recorders. Mastering the design of these systems, and reducing the time needed to bring them to market, becomes of utmost economic importance in times of increasing market dynamics. This paper advances the paradigm of *synchronous programming* as a means to match these goals.

Several programming languages that originally emerged as engineering notations are now defined in the IEC Standard 1131-3 [18]. These languages have been designed specifically for embedded software applications, mainly in the process control industry but with increasing influence in other sectors. Unfortunately the IEC 1131 languages have not been designed with the benefit of formal semantics. Yet, to advance the state of the practice of embedded software design it is important to provide tools that support high-level specification and rapid prototyping, integrate testing and formal verification to achieve early design validation, and encourage modular software development—to ease review, maintenance, and certification. For this, languages having precise mathematical semantics are required.

Synchronous programming languages like those discussed in this paper have the potential to introduce such rigour o embedded software design. These languages have very distinctive characteristics: ESTEREL [4] is imperative in style, while LUSTRE [13] and SIGNAL [22] are declarative; STATECHARTS [15] and ARGOS [26] are graphical notations that enable one to program directly by constructing hierarchical automata. These languages share a common communication metaphor, that of synchronously broadcast signals. Sections 2 and 3 introduce the key ideas behind the synchronous approach to embedded software, and outline some of the programming constructs available in LUSTRE, ESTEREL, and ARGOS through a few simple examples. A longer introduction to these languages can be found in [12].

The specific profiles of these languages reflects the fact that they have been developed in response to problems emerging in different application areas. LUSTRE and SIGNAL derive from requirements of industries mainly aware of electrical and electronics engineering methods, who wanted to manage the increasing complexity of their applications, and gain greater flexibility in their design, by introducing software. These languages have therefore been designed for the discrete handling of continuous phenomena. They invoke metaphors commonly used by electrical engineers in control theory, and are thus most suited to signal-based applications such as in navigation or digital signal processing where often a sampling of different related frequencies can be found. In contrast, ESTEREL and graphical languages like ARGOS are better suited for scheduling complex operating modes, handling intricate patterns of events, and describing interrupt-driven behaviours. However, the difference in their best-suited application profiles is only a qualitative assessment since these languages have broadly similar expressive power.

Our motivation for taking the synchronous design paradigm further, with the wholesale integration of imperative, declarative (or dataflow), and graphical programming styles, is that real systems typically have components that match each of these profiles. As argued by Gajski [11], the construction of embedded systems requires a combination of state based and dataflow models which support hierarchical structuring of behaviour, concurrency, and exception handling. We believe a *semantical* combination of the languages described here will definitely satisfy these requirements. That is the unique feature of the SYNCHRONIE workbench which is under active development in the Embedded Software Design group at GMD. The success of this venture rests on the underlying computational model sketched in Section 4, and with greater rigour in Section 5, which provides a coherent mathematical framework, and yields compact, verifiable code. The main functions of the SYNCHRONIE workbench are described in Section 6.

Synchronous programming languages are already being evaluated in some industries, particularly aviation and power generation where the problems are real-time constrained and safety critical. The commercial interest in synchronous languages lies not only in the style, but in their seamless integration with existing software development practices through the programming environments with which they are provided. A short overview of the marketed environments that support each language, and some industrial applications, can be found in [2, 19]. In addition to project management facilities, and editing facilities that mix text and graphics, the commercial environments provide advanced features for design validation, and back-end compilation to various languages like C or VHDL. The SYNCHRONIE

workbench also provides such features, but offers system designers in addition much greater freedom in the choice of programming language, with the unique option to freely mix the various synchronous programming modes.

For the most part synchronous languages provide primitive datatypes and operations on them only: the emphasis is on gaining intellectual control over the program's often intricate control logic, rather than on data processing issues. Where compound datatypes need to be used, the synchronous language is 'hosted' in a common language such as C, Fortran, or Ada. Synchronous languages achieve good separation between concerns of data and control logic; their novel fusion in SYNCHRONIE with object oriented construction techniques [7, 8] offers the right kind of design encapsulation and abstraction mechanisms to achieve much needed transparency in the software development lifecycle.

2. Reactivity and Synchrony

In contrast to an interactive system (say a text editing program), a reactive system is fully responsible for the synchronisation with its environment. A system is reactive when it is fast enough to respond to every input event, and its reaction latency is short enough that the environment is still receptive to its responses. Most control systems, and systems for digital signal processing as used in industry or in telecommunications, are reactive according to this characterisation. Common features of reactive systems are:

concurrency: they typically consist of several concurrent components that cooperate to realize the intended overall behaviour.

real-time: they are supposed to meet strict constraints with regard to timing, such as response time or availability.

determinism: a system's reaction is uniquely determined by the kind and timing of external stimuli.

heterogeneity: they often consist of components implemented in quite different technologies like software, hardware, or on distributed architectures.

reliability: requirements include functional correctness as well as temporal correctness of behaviour; also robustness, and fault tolerance.

The fundamental idea of the synchronous approach is simple: reactive systems are idealised by assuming that stimulation and reaction are simultaneous, or that reaction takes *zero time*—meaning no observable amount of time. A system is stimulated by events from the environment, but responds instantaneously. Physical time does not play any special role: time will be considered as a sequence of a particular kind of external events. From this point of view the statement

"the train must stop within 10 seconds,"

is not essentially different to the statement

> "the train must stop within 100 metres."

In both cases something is said about the occurrence of events [12]:

> "The event *stop* must precede the 10th (or 100th) next occurrence of the event *second* (or *metre*)."

Events are manifest by signals which are *broadcast* throughout the system instantaneously. A system reacts by emitting (that is, broadcasting) signals as well, so the statement above could be modified to:

> "The signal `train_stops` must be emitted before the signal `second` has been emitted 10 times."

Abstracting physical time offers a number of advantages: the granularity of time may be changed without affecting the sequence of events, and system components can be composed and decomposed into subcomponents without changing the observable behaviour. This is particularly beneficial for proving system properties. In practical terms, the *synchrony hypothesis* states that a system reacts fast enough to record all external events in the proper order. This property is realistic only insofar as it can be checked: it corresponds to the hardware point of view that the time needed by an operation is of no importance as long as it does not exceed the duration of a clock cycle, or, vice versa, that the clock cycle is determined by the operation which consumes most time. The synchronous approach therefore advances a two-stage design for reactive systems: physical time is first abstracted to focus on the functions to be maintained, the validity of this abstraction being actually verified when these systems are implemented.

3. Synchronous Languages

In this section we sketch some typical synchronous programming styles, focusing in the subsections that follow on ESTEREL, ARGOS, and LUSTRE. A larger example is described in Section 4.1 that uses these languages together to solve a programming problem in a natural way. Here we wish to give the reader a feeling for synchronous programming in general, so our examples are necessarily rather simple.

3.1. Imperative Programming in ESTEREL

Suppose the following informal specification has been given:

> "If a second mouse click succeeds a first one within 5 milliseconds there is a double click, otherwise there is a single click."

Assuming that time units are specified by the signal `tick`, the required behaviour is captured by the ESTEREL program (not the shortest) in Figure 1. At first the program waits until a `click` signal is produced by the environment. Then two programs are executed in parallel. The upper subprogram broadcasts the `single` signal after five ticks provided that

```
loop
 trap done in
  await click;
  [
   await 5 tick;
   present double else
    emit single; exit done
   end
  ||
   await click;
   emit double; exit done
  ]
 end trap
end loop
```

Figure 1. An ESTEREL mouse controller

double is not emitted simultaneously; it then raises the exception (the *trap signal*) done, and terminates. The conditional programming construct present ... end should be interpreted as 'if the double signal is present do nothing, otherwise emit single (and exit)'. The other subprogram emits the double signal if click occurs a second time, and exits.

Trap signals are exceptions which, when raised, abort all programs within the scope of their declaration. The net effect of raising done in either of the parallel branches in the mouse program is that the body of the loop is terminated; ESTEREL's semantics cause the loop to restart immediately, so the program returns to the await click at the top of the loop. Note the priority given to double over single if the second click should happen at the fifth tick of the clock.

In contrast to the trap statement, the construct

```
do halt watching click
```

(which is what await click actually abbreviates) involves a second, stronger kind of *preemption mechanism*. The halt statement is the only one in ESTEREL to consume time; in fact halt starts but never terminates. However, the body of the watching construct will be preempted whenever the watchdog condition becomes true (that is, whenever the signal click is present in this example).

All language constructs of ESTEREL are *instantaneous* apart from the halt statement, or derived constructs like await. As we saw, the loop restarts immediately it terminates; sequential composition is likewise reckoned to take zero time, as are the tests in conditional statements. Thus, if a second click does coincide with the fifth tick, the present test in the first subprogram in the example above will be executed in the same logical instant as the click which aborts the await in the second subprogram. Of course, no assumptions are made here about behaviour with respect to physical time which is represented explicitly by the tick signal supplied be the environment.

3.2. Graphical Programming in ARGOS

To illustrate a graphical notation the program of the previous section is now coded in ARGOS (see Figure 2). States are represented by rounded boxes. Automata are hierarchic in that states can contain subautomata (for example in state *two* there are two subautomata). Automata can run in parallel, which is indicated by a dashed line. Finally, the scope of signals can be restricted—this is indicated by a square cornered box, instead of a rounded one,

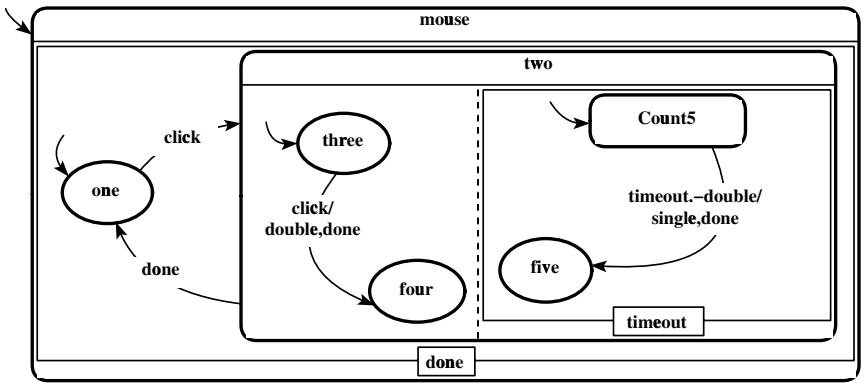


Figure 2. ARGOS program for the mouse controller

with a list of signals glued to it. A *default arrow* indicates the initial state of each subautomaton (e.g., those labelled *one*, *three*, and *Count5*).

When started the mouse automaton enters the initial state (labelled *one* here). Occurrence of the `click` signal causes a jump from there to state *two*, and this initialises both of the parallel subautomata. A label on an arrow that is of the form a/b indicates that the transition should take place if signal a arrives, and that this will cause the signal b to be emitted simultaneously. Generally the *guard* (a) on a transition can be a list which specifies the presence or absence of a number of signals, and the *output action* (b) is a list of signal names (omitted, if the list is empty). So the (subsequent) transition from state *Count5* to state *five* will fire if the `timeout` signal is present and the `double` signal is not present; this causes both `single` and `done` to be emitted.

We have to add that the *refinement* of state *Count5* contains the subautomaton displayed in Figure 3. At the fifth tick the `timeout` signal is emitted and the subautomaton comes to a

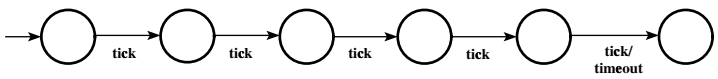


Figure 3. ARGOS 5 tick counter (i.e., `await 5 tick; emit timeout`)

halt (tick is implicit in the guard on every transition in ARGOS, but is mentioned explicitly here if the guard would otherwise be empty). Meanwhile, if a second `click` occurs the transition from state *three* to state *four* takes place causing the simultaneous emission of `double` and `done`. However, if the second `click` occurs *at* the fifth tick the transition to state *five* is not possible because `double` occurs negatively in the guard of the transition.

Hence, exactly one of `single` or `double` will be emitted from state *two*, and in either case this will be accompanied by the (local) signal `done`. This signal causes state *two* to be abandoned when it is emitted because it appears in the guard on the transition out of that

state—whatever individual states the subautomata are in, when this signal occurs they will be abandoned. Such control flow is similar to the preemption mechanisms of ESTEREL. The program therefore returns to state *one*, and waits for the next `click`.

3.3. Declarative Programming in LUSTRE

To illustrate the declarative synchronous programming style we shall implement a recursive digital filter in LUSTRE. A first order digital filter may be specified by a *signal flow graph* [32] such as that in Figure 4. Quantities on incoming edges at the nodes of this graph are summed, and their result is broadcast along the outgoing edges. The labels z^{-1}, a_0, a_1 and $-b_1$ on the arcs denote a delay by a shift register, and multiplications by the respective constants. In linear form we would have the equation

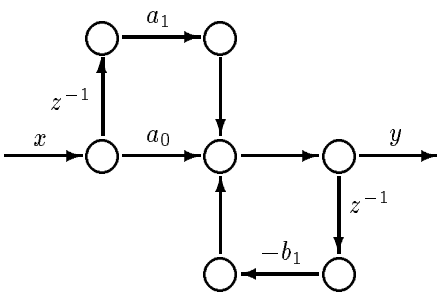


Figure 4. A first order digital filter

$$y_n = a_0 x_n + a_1 x_{n-1} - b_1 y_{n-1}$$

where $n > 0$ denotes the time index. The boundary condition is $y_0 = a_0 x_0$. This equation with its boundary condition translates to the LUSTRE *node*:

```
node FILTER (x : real) returns (y : real);
let
  y = a0 * x -> a0 * x + a1 * pre(x) - b1 * pre(y);
tel;
```

(one has to instantiate the constants to specific real values of course). The equation is evaluated on every program cycle which is marked in LUSTRE by an implicit clock tick, rather than the explicit `tick` seen earlier. Once a LUSTRE program is started it runs forever without terminating, executing every equation once a cycle. In this case only at the first tick will the term `a0 * x` be evaluated; at all later times the term `a0 * x + a1 * pre(x) - b1 * pre(y)` is evaluated. This gives the semantics of the *followed-by* operator `->`. The `pre` operator allows access to the previous value of the expression on which it operates.

LUSTRE has other operators for *upsampling* and *downsampling*. These operations are illustrated in the timing diagram below. Downsampling is by means of the `when` operator, and upsampling is by means of the `current`:

B	f	t	f	t	t	t	f	f	t	.	. .
X	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	.	. .
Z = X when B		X_2		X_4	X_5	X_6			X_9	.	. .
Y = current(Z)	•	X_2	X_2	X_4	X_5	X_6	X_6	X_6	X_9	.	. .

`Z = X when B` takes the value of `X` only if the 'clock' signal `B` is true (it is undefined otherwise), and `current(Z)` latches the value of `Z` up to the next sampling signal, i.e., the next instant `B` becomes true (the • is explained in Section 5). This mechanism allows one to easily define the digital filter with regard to a different base frequency:

```
node BFILTER (b : bool, x : real) returns (y : real);
var z when b : real;
let
  z = (FILTER(x)) when b;
  y = current(z);
tel;
```

Other declarative synchronous languages include SIGNAL and SILAGE [22, 16]. The latter is mainly used as the specification language of the CATHEDRAL [21] toolset for the synthesis of DSP chips. These languages are quite similar to LUSTRE in style but use slightly different mechanisms for upsampling and downsampling.

This concludes our survey of the synchronous programming styles. Most of the features of ARGOS and LUSTRE have been mentioned, but ESTEREL is a slightly richer language than the mouse controller example illustrates since, in particular, signals can carry data which may be tested to modify the control flow. These languages share the communication principle of synchronously broadcast signals, with the scoping mechanisms shown. However, the fact that signals can be tested and emitted simultaneously in the parallel branches of a synchronous program can sometimes give rise to confusion over the cause and effect of a signal's activation. Such causality cycles (they correspond to short circuits in sequential hardware) are programming errors that can be detected statically by the compilers; they do not affect the semantics of the individual languages as such, only the class of acceptable programs. The detection of causality cycles is intricate however, and falls outwith the scope of the present paper (see [25, 35], for example).

4. Integration of Synchronous Languages

Since complex systems often have components which match each of the different profiles sketched above, it is natural to wish to express each in the most appropriate language rather than shoe-horning solutions from a single language. To illustrate the useful interaction between these synchronous languages we reprogrammed the production cell case study [23] that has been used for evaluating and comparing software development methodologies and tools. The successful design and verification of the controller has been discussed independently in the contexts of ESTEREL, SIGNAL and LUSTRE [23], so we do not dwell on the details of the specification in Section 4.1 below, but rather on its overall organisation. Section 4.2 outlines some aspects of the underlying computational model which makes possible the idea of integrating these languages into a single, coherent mathematical framework. The formalities are drawn out in Section 5.

4.1. Multi-modal Programming

The production cell's input arrives via a feed belt that conveys metal plates to an elevating rotary table; the table lifts each plate to a position where a robot picks it up with its first arm, transferring it to a press. When the plate has been forged by the press it is removed to a deposit belt by the second arm of the robot. The circuit completes with a crane that unloads the forged plates at the end of the deposit belt. Although the circuit for one item is quite simple, the design of the cell's control software should maximise the throughput while meeting the various constraints of the devices.

The cell's controller has a short initialisation phase during which the various devices are set to specific (safe) states, followed by an endlessly looping process that controls the actual production cycle. The production phase divides naturally into six components that run in parallel, one for each physical device. To represent the design at this highest level of organisation we used ARGOS as illustrated in Figure 5. The graphical nature of this lan-

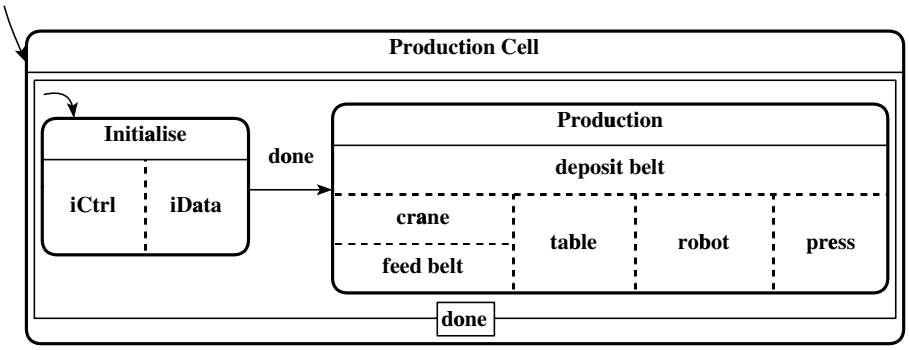
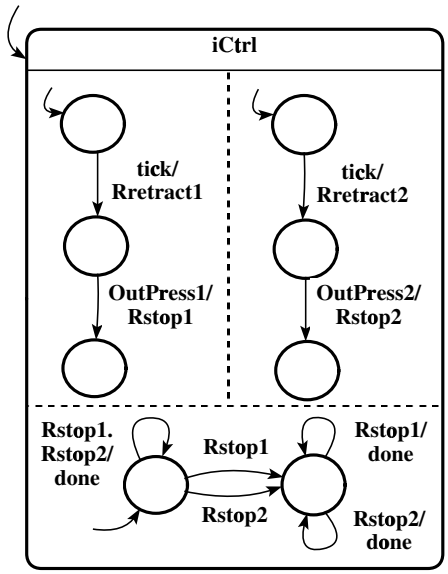


Figure 5. The Production Cell in ARGOS

guage makes it very suitable for presenting high-level design choices, and the reader can appreciate at a glance the overall structure of the control program. The graphical style eases communication between partners in the development of the software, and others involved in constructing the production cell including management, mechanical and electrical engineers designing the physical components, system and safety analysts, and so on.

The initialisation phase sets both arms of the robot to a retracted state, ready to handle the first plate to arrive (for safety since they might otherwise be damaged in the press when it is switched on). This behaviour is implemented by first emitting signals `Rretract1` and `Rretract2` to the actuators of the robot, and then waiting for the arms to reach the desired positions. Reaching these positions is signalled by `OutPress1` and `OutPress2`, and when this occurs a stop command is emitted to the respective arm by means of the signals `Rstop1` and `Rstop2`. When both arms have reached the desired positions the signal `done` is emitted, transferring control to the unending production phase.

If we were programming only in ARGOS (or STATECHARTS, for that matter), there would be little choice but to implement this initialisation logic in a program similar to that in Figure 6(a). A drawback of the graphical formalism becomes apparent. The control flow is



(a) ARGOS initialisation

```
module iCtrl:
 input OutPress1,OutPress2;
 output Rretract1,Rstop1,
        Rretract2, Rstop2,
        done;
 [
   emit Rretract1;
   await OutPress1;
   emit Rstop1
 ||
   emit Rretract2;
   await OutPress2;
   emit Rstop2
 ];
 emit done
end module
```

(b) ESTEREL initialisation

Figure 6. Initialisation phase of the Production Cell

confused by the fact that one has to explicitly manage the synchronisation to emit `done` when both arms have been retracted (via the process in the lower part of the figure). Also, the reader may be forgiven for wondering whether some external process running in parallel with *Initialise* can interfere by emitting an `Rstop1` signal, say. (There is no such process, but this cannot be inferred simply by looking at Figures 5 and 6.) At this level of detail graphical programming becomes cumbersome, and therefore error prone, and one quickly loses sight of the flow of information.

Instead, we can refine the state *iCtrl* in Figure 5 with the ESTEREL program shown in Figure 6(b). This makes the natural control flow explicit (with the semi-colon after the concurrent initialisation of the two arms), and ESTEREL's powerful parallel operator handles the synchronisation on our behalf to ensure that `done` is only emitted when both branches have terminated. This program only handles pure signals (that is to say, those signifying events) and no data—even though the position of the arms are in reality provided by a potentiometer delivering a real value. In the next section we shall describe one method of handling these data (refining state *iData*) to complete the initialisation program.

The rest of the program is implemented in LUSTRE as described by Holenderski [17], and will not be discussed here. The steady state behaviour of the press, the robot, and the other physical components is adequately expressed in LUSTRE since these are interlocked, nonterminating parallel processes. However, specifying the *sequential* composition of the initialisation and production phases in LUSTRE leads to an obscure program since the sequential composition operator is simply not available. To avoid this in his implementation, Holenderski programmed the sequence at the level of the C interface (to the simulator provided for the case study), but this ad hoc approach is highly error prone in general, undermines the formal definition of the synchronous language (LUSTRE, in this case), and preempts our performing any formal verification of the full program.

4.2. Synchronous Automata

Performing the sequential composition in ARGOS is, in contrast, fully formal as long as the meaning of the combination of the synchronous languages is clear. We are aided here by their relative simplicity (e.g., when compared to Ada or C), and though they each have a very different 'look and feel', ESTEREL, LUSTRE, and ARGOS can be interpreted in the same computational model. This section introduces the main notions behind synchronous automata, the more formal presentation being deferred until Section 5.

4.2.1. Boolean Automata

Boolean automata are easier to understand and are, in fact, a particular instance of synchronous automata which capture the essence of the synchronous languages. Boolean automata have two kinds of statements:

$s \Leftarrow \phi$: the signal s is emitted if condition ϕ is satisfied, and

$h \leftarrow \phi$: the control register h is true (or 'set') in the next instant if ϕ is satisfied.

A synchronous program is represented by a collection of such statements defining signals to represent transient information, or registers to represent persistent information. The operational semantics of Boolean automata is defined by two successive phases: given a valuation υ that assigns a truth value, *tt* or *ff*, to each of the registers and inputs (inputs are represented by free variables), a reaction is a solution of the system of equations $\{s \Leftarrow \phi\}$; this solution extends υ to cover all signals, and we use this to compute the assignments $\{h \leftarrow \phi\}$ to yield the next state of the machine. A solution to the signal equations for all input patterns and (reachable) states must be proved to exist at compilation time to guarantee that the program is *reactive* in that it may respond to every input stimulus. In addition, this solution must be unique to guarantee that the program is *deterministic*. These issues are common to all synchronous programming languages.

For an example, the behaviour of the ESTEREL statement `await OutPress1` is defined by the Boolean automaton:

$$\begin{aligned} \omega &\Leftarrow \mathit{OutPress1} \wedge h_1 \\ h_1 &\leftarrow \alpha \vee (h_1 \wedge \neg \mathit{OutPress1}) \end{aligned}$$

The register h_1 captures the pause in this `await` construct. All registers are initially false; h_1 will be set when the Boolean automaton is initialised—that is, when the special *start signal* α is present (it is only present, or true, in the very first clock cycle). Thereafter, h_1 is set every program cycle until the signal `OutPress1` occurs, at which point the automaton terminates (h_1 becomes, and stays, false). Termination is signified by the special signal ω. Note that the `OutPress1` will be ignored in the very first clock cycle—readers familiar with ESTEREL will realise that an '`immediate`' is needed to handle that event.

The Boolean automaton

$$\begin{aligned} \mathit{Rstop1} &\Leftarrow \alpha \\ \omega &\Leftarrow \alpha \end{aligned}$$

captures the behaviour of the statement `emit Rstop1`: this automaton emits the signal and terminates immediately. The sequence `await OutPress1;emit Rstop1` specifies that if the former statement terminates, control passes instantaneously to the latter. This control flow is tracked by the compiler which substitutes for the start condition (α) of the Boolean automaton associated with the second command in sequence, the termination condition of the first. Hence, for the upper parallel branch of Figure 6(b), we obtain:

$$\begin{aligned} \mathit{Rretract1} &\Leftarrow \alpha \\ \mathit{Rstop1} &\Leftarrow \mathit{OutPress1} \wedge h_1 \\ \omega &\Leftarrow \mathit{OutPress1} \wedge h_1 \\ h_1 &\leftarrow \alpha \vee (h_1 \wedge \neg \mathit{OutPress1}) \end{aligned}$$

We cannot sketch the translation of all the language constructs here, but have hopefully provided some feeling for how the translation proceeds. ARGOS and Boolean LUSTRE also have natural and compact interpretations in Boolean automata. The full translation of pure ESTEREL, along with its proof of correctness with respect to the published semantics, is given in [34].

4.2.2. Synchronous Automata

Synchronous automata represent an enhancement to the model to handle data; Boolean automata only capture the pure control of synchronous languages, focusing on synchronisation issues and ignoring the data carried by signals, and actions upon them. Intuitively, this enhancement is achieved by coupling the presence of a signal with a unique datum. The earlier notion is refined thus:

$s \Leftarrow \nu \ @ \ \phi$: when ϕ is satisfied, the signal s is emitted with the value returned by the function ν, and

$h \leftarrow \nu \ @ \ \phi$: when ϕ is satisfied, the register h is set for the next instant with the result of the function ν.

In this framework, Boolean automata are synchronous automata where the domain of values is restricted to a single-point set denoted $\{\bullet\}$. Such signals are referred to as *pure signals* (also pure, or control, registers) because the values are of little interest. So, the earlier Boolean equations ($s \Leftarrow \phi$) and assignments ($h \leftarrow \phi$) have now given way to conditional equations ($s \Leftarrow \nu \ @ \ \phi$) and guarded commands ($h \leftarrow \nu \ @ \ \phi$). Boolean automata are thus *pure* synchronous automata.

To illustrate the idea let us specify the discretisation of the position of the arms of the robot in the production cell in the initialisation phase thus:

```
node iData (Arm1, Arm2 : real)
   returns (OutPress1, OutPress2 : bool);
let
   OutPress1 = (Arm1 <= 0.3);
   OutPress2 = (Arm2 <= 0.1);
tel;
```

This LUSTRE node emits `OutPress1` and `OutPress2` whenever the sensors on the arms of the robot indicate that they are a safe distance from the press. This is defined by the synchronous automaton:

$$
\begin{array}{lcl}
\mathit{OutPress1} & \Leftarrow & \mathit{Arm1} \leq 0.3 \ @ \ \alpha \vee h_{\mathit{idata}} \\
\mathit{OutPress2} & \Leftarrow & \mathit{Arm2} \leq 0.1 \ @ \ \alpha \vee h_{\mathit{idata}} \\
h_{\mathit{idata}} & \leftarrow & \bullet \ @ \ \alpha \vee h_{\mathit{idata}}
\end{array}
$$

The first two statements translate the equations of the LUSTRE node; the latter represents the implicit control in declarative programs. The register h_{idata} is initially *inactive*, gets set (or becomes active) when the program is started with α, and remains set thereafter. This persistence is indicated by h_{idata} appearing in the statement guard where it refers to the active status of the register, not its value. Since it never again changes state, h_{idata} might be thought redundant. Nevertheless, it captures the nonterminating property of declarative programs: once a declarative program gets started via α, it retains control in that it is executed at every later instant of time. We anticipate that it may *lose control* as well. The idea required to integrate declarative and imperative styles is formally captured in synchronous automata by the control axiom.

4.2.3. The Control Axiom

The control axiom states that no synchronous automaton can react if it is not in control—a synchronous automaton has control either if α is present, or some of the registers of the process are set. By way of a non-example, the following statement alone does not define a synchronous automaton since the presence of *OutPress1* neither depends on α, nor any state (the latter dependency exists implicitly in a LUSTRE program's main loop):

$$\mathit{OutPress1} \Leftarrow \mathit{Arm1} \leq 0.3 \ @ \ \mathit{tt}$$

The control axiom appears to contradict the notion that a *reactive* system must maintain an ongoing relationship with its environment, and thus must always react. But even if this is true for a complete reactive system (the whole program), some of its subcomponents may be active for just a while; in particular, they may be preempted. In the production cell, while the ESTEREL module `iCtrl` terminates by itself after emitting `done`, the LUSTRE node `iData` never terminates. But, with respect to the semantics of ARGOS, the *Initialise* state (Figure 5) is preempted by `done`, which therefore aborts the LUSTRE node. This is performed (in the compiler) by redefining h_{idata} thus:

$$h_{idata} \leftarrow \bullet \ @ \ \alpha \vee (\neg done \wedge h_{idata})$$

This *guarding* of register statements translates a weak preemption mechanism [3]. This is similar to ESTEREL's trap construct, and is the only preemption mechanism of ARGOS. The strong preemption mechanism is captured by guarding both kinds of statements. A longer presentation of synchronous automata and their algebra is given in [24], whence the summary of the formal semantics in the next section is drawn.

5. Semantics in a Nutshell

It should be evident by now that the synchronous programming paradigm has a dual nature. On the one hand languages such as LUSTRE are descriptive in that they constrain possible behaviours; one the other hand languages like ESTEREL and ARGOS foster a constructive point of view in that an automaton is specified which prescribes the transformation of a given state into another. We argued informally above that synchronous automata are a useful intermediate representation. These structures are reasonably unsophisticated mathematically, and match both programming styles well enough to serve as a kind of synchronous object code, though of course originating in the tradition of deterministic automata. While not wishing to lose the light presentational style of the previous sections, or burden the reader with undue formality, we shall try to shed some light on the formal semantics in a slightly more rigorous manner. The subsections that follow first address the descriptive, then the prescriptive aspects of synchronous programming, before merging 'dataflow' and 'controlflow' in their synthesis at the end.

5.1. The Declarative Aspect: Constraining Dataflow

Behaviour is manifest in what it is possible to observe. We classify observations by attributing a name which, for the moment, will be referred to as a *signal*. A signal s may be *present* with a value taken from a set $\mathcal{V}$, or it may be *absent*. If a particular signal is observed over time, a flow of values and 'misses' (when the signal is absent) will be obtained:

$\surd$	0	1	2	3	4	5	6	7	.	.	.
s	v_0	v_1		v_2			v_3	v_4	.	.	.

A flow is characterised by a subset $\widehat{s}$ of the natural numbers (the *sampling rate* of s), and a valuation $s : \widehat{s} \to \mathcal{V}$. If $n \in \widehat{s}$, then s is present at instant n, and takes the value $s(n)$. In the example above, $\widehat{s} = \{0, 1, 3, 6, 7, \ldots\}$. We refer to a set of flows as a *synchronous process*, and at each instant of time require that at least one signal of a synchronous process is present. This requirement allows us to identify 'observed' (or external) time with the natural numbers.

Conceptually, synchronous dataflow programs deal with the *specification* of such processes. For example, the LUSTRE equation `Z = X + Y` specifies constraints upon the corresponding flows:

$\widehat{z} = \widehat{x} = \widehat{y}$	the sampling rates are the same, and
$z(n) = x(n) + y(n)$	at each instant $n \in \widehat{z}$ the values are the same.

Taking a second example, the *memoisation* operator `pre` in `Y = pre(X)` introduces a delay so that the value of `X` observed at one sampling point is observed on `Y` at the next:

	0	1	2	3	4	5	6	7	8	.	.
`X`	$\mathtt{X}_0$	$\mathtt{X}_1$			$\mathtt{X}_2$			$\mathtt{X}_3$	$\mathtt{X}_4$	.	.
`Y = pre(X)`	•	$\mathtt{X}_0$			$\mathtt{X}_1$			$\mathtt{X}_2$	$\mathtt{X}_3$	.	.

At the first sampling point no value of `X` can have been observed beforehand so we insert a 'non-value' • which indicates that even if `Y` is present, its value is undefined. Of course, a compiler must guarantee that no program's reaction ever depends on a non-value. This phenomenon is similar to that of program variables (in other languages) which should not be used at run-time before they have been initialised. LUSTRE's 'followed-by' operator, discussed in Section 3.3, allows one to initialise signals properly.

The formal definition (of `pre`) is quite elaborate since it depends on the sampling points. Skipping the exact definition, let us use the notation $\widehat{x}_m$ to refer to the mth sampling point in sequence, counting from 0. Then:

$\widehat{y} = \widehat{x}$, and

$y(\widehat{x}_{m+1}) = x(\widehat{x}_m)$, for all sampling points $\widehat{x}_m$, and $y(\widehat{x}_0) = \bullet$

Synchronous automata are introduced here as a more elementary language for specifying constraints on dataflows. The statement

$$y \Leftarrow x \ @ \ c$$

specifies that, whenever the pure signal c is present, the signal y must be present and its value must be equal to that of x. In more formal terms:

$c \subseteq \widehat{x}$, $c \subseteq \widehat{y}$, and for all $n \in c$, $y(n) = x(n)$

Notation is somewhat overloaded here. To be precise, each subset $c \subseteq \mathbb{N}$ is in one-to-one correspondence with its characteristic function $\tilde{c} : \mathbb{N} \to \{tt, ff\}$ such that $\tilde{c}(n) = tt$ iff $n \in c$. Hence the Boolean operators used in *control expressions* (the guards after the @

symbol) are well justified, and can be unambiguously interpreted as Boolean operators, or as operators on subsets of $\mathbb{N}$.

A point to note is that although the definition above forces y to have a value at instant n only if $n \in c$, it may still have values at other instants. Furthermore, we require the signal x to be present at instants $n \in c$, otherwise the constraint $y(n) = x(n)$ is meaningless. The idea naturally generalises to more complex statements $y \Leftarrow \nu \ @ \ \phi$ where ν is a data expression and ϕ is a control expression.

Closely related to the `pre` operator is the second type of statement found in synchronous automata, namely

$$y \leftarrow x \ @ \ c$$

This delays the observation on x by just one instant, but upsamples at the same time: whenever $n \in c$, whatever was observed on x in instant n will be observed on y at instant $n+1$. The figure

	0	1	2	3	4	5	6
c		•			•	•	
x		x_0		x_1	x_2	x_3	
$y \leftarrow x \ @ \ c$			x_0			x_2	x_3

interprets the more formal definition

$c \subseteq \widehat{x}$, $c^+ \subseteq \widehat{y}$, and for all $n \in c$, $y(n+1) = x(n)$

where $c^+ = \{n+1 \mid n \in c\}$. However, it is not a totally trivial exercise to prove that, provided attention is restricted to signals x and y, the synchronous automaton

$$\begin{array}{lll} y & \Leftarrow & r \ @ \ \widehat{x} \\ r & \leftarrow & x \ @ \ \widehat{x} \\ r & \leftarrow & r \ @ \ \neg\widehat{x} \\ \mathit{init}\ r & = & \bullet \end{array}$$

x			x_0		x_1	x_2	
r	•	•	•	x_0	x_0	x_1	x_2
y			•		x_0	x_1	

is equivalent to the LUSTRE statement `Y = pre(X)`. This introduces the third (and final) clause in synchronous automata, viz

$$\mathit{init}\ r = v$$

which is used to initialise a dataflow that would not otherwise take a value 'at α' (i.e., $r(0) = v$). While the proof that the above automaton implements `pre` is a little tricky, the reader can easily verify that the semantics compute the flows in the adjacent diagram which shows how the 'last' value of x is stored until the next sampling point. Note that $\neg\widehat{x} = \mathbb{N} - \widehat{x}$, so it turns out that $\widehat{r} = \mathbb{N}$. For the diagram we supposed $\widehat{x} = \{2, 4, 5, \ldots\}$.

5.2. The Imperative Aspect: Managing Control

The imperative synchronous languages are inherently based on the idea of a (distributed) state. In ESTEREL, for instance, the `halt` statement is used to indicate local sequence control within the parallel branches of a program. The `halt`'s behaviour is defined by

$$\left.\begin{array}{lcl} h & \leftarrow & \alpha \vee h \\ \omega & \Leftarrow & \mathit{ff} \end{array}\right\} \quad \text{which abbreviates} \quad \left\{\begin{array}{lcl} h & \leftarrow & \bullet \ @ \ \alpha \vee h \\ \omega & \Leftarrow & \bullet \ @ \ \mathit{ff} \end{array}\right.$$

where an h is introduced as a specific *control register* for each such `halt`. Once this synchronous automaton has been activated by α, the register h remains active forever, and the automaton never terminates. This is precisely reflected in the formal semantics of $h \leftarrow \alpha \vee h$ which yields $(\widehat{\alpha} \cup \widehat{h})^+ \subseteq \widehat{h}$; given that $\widehat{\alpha} = \{0\}$ this means $\widehat{h} = \mathbb{N}^+$. Moreover, $\widehat{\omega} = \emptyset$.

In this discussion we revert to Boolean automata because *control* is specified in terms of pure signals only. We apologise for any terminological confusion that may be troubling the reader at this point: we stated earlier that flows represent observations made of certain attributes, namely signals, but now speak in terms of registers. To reconcile the nomenclature we shall also relate flows to registers, but stipulate that registers are 'unobservable' (as one might expect). Registers may be *active* (and thus possessing a value, though it may be '$\bullet$' in the case of control registers), or *inactive*.

In order to model preemption in synchronous automata the simple idea is to guard 'reactivation' in that the above automaton is modified to:

$$\begin{array}{lcl} h & \leftarrow & \alpha \vee \neg s \wedge h \\ \omega & \Leftarrow & s \wedge h \end{array}$$

Once the control register is activated, it remains activated only up to the instant that the signal s is present; the termination signal ω is emitted at the same later instant. Note that $\omega \Leftarrow s \wedge h$ requires that 'control' is with h (i.e., h is active). These clauses capture precisely the semantics of ESTEREL's `await` statement (cf. Section 3.1) where

`await click` is defined as `do halt watching click`

This construct generalises to `do P watching s`, where `P` is an arbitrary program statement. In addition to the preemption of halts, ESTEREL's semantics require that no signal is emitted by `P` if the watchdog signal s is present; also, s is to be ignored by `P` in the instant that it commences execution [4]. The preemption effect can be achieved by guarding the emittance of signals. A typical clause in `P` that depends on a register h may be

$$c \ \Leftarrow \ \nu \ \ @ \ (a \wedge \alpha) \vee (b \wedge h)$$

meaning that c is emitted if a is present in the first (α) instant, or if the control register h is active when b is present. The emission of c is preempted by the watchdog action s thus:

$$c \ \Leftarrow \ \nu \ \ @ \ (a \wedge \alpha) \vee (b \wedge \neg s \wedge h)$$

Note that c is still emitted in the first instant if a is present.

In contrast to this, ARGOS only allows preemption of registers. With regard to Figure 2, emmittance of the signal `done` should preempt all the states inside state *two*, including state

two itself. The reader will note that preemption of signals must be avoided because otherwise the signal `done`, being the cause of the preemption in this case, would not then be emitted. Preemption of registers is achieved by the same guarding mechanism as before, now with the guard $\neg done$. Working through the details, the reader may quickly check that

$$\begin{array}{lcl} h_{one} & \leftarrow & \alpha \vee (done \wedge h_{two}) \vee (\neg click \wedge h_{one}) \\ h_{two} & \leftarrow & (click \wedge h_{one}) \vee (\neg done \wedge h_{two}) \end{array}$$

relates to the transitions from state *one* to state *two*, and conversely. Further, building on the informal description in Section 3.2, it is easy to check that the behaviour of the subprocess in state *two* is captured by the synchronous automaton $\mathcal{A}$

$$\begin{array}{lcl} double & \Leftarrow & (click \wedge h_{three}) \\ single & \Leftarrow & (timeout \wedge \neg double \wedge h_{Count5}) \\ done & \Leftarrow & (timeout \wedge \neg double \wedge h_{Count5}) \vee (click \wedge h_{three}) \\ h_{three} & \leftarrow & \alpha \vee (\neg click \wedge h_{three}) \\ h_{Count5} & \leftarrow & \alpha \vee (\neg(timeout \wedge \neg double) \wedge h_{Count5}) \end{array}$$

(omitting the register clauses refining state *Count5*, and the `timeout` signal). We have not defined h_{four} or h_{five} here since these 'terminal' states in the respective ARGOS subautomata never become activated. Guarding all the registers of $\mathcal{A}$ with $\neg done$ therefore (the '$\Leftarrow$' clauses do not change, so are not repeated), we obtain $\mathcal{A}'$

$$\begin{array}{lcl} h_{one} & \leftarrow & \alpha \vee (done \wedge h_{two}) \vee (\neg click \wedge h_{one}) \\ h_{two} & \leftarrow & (click \wedge h_{one}) \vee (\neg done \wedge h_{two}) \\ h_{three} & \leftarrow & (click \wedge h_{one}) \vee (\neg click \wedge \neg done \wedge h_{three}) \\ h_{Count5} & \leftarrow & (click \wedge h_{one}) \vee (\neg(timeout \wedge \neg double) \wedge \neg done \wedge h_{Count5}) \end{array}$$

which is a fairly precise account of what was described informally in Section 3.2. Of course, a more structural, or compositional treatment is needed for an ARGOS compiler to synthesise $\mathcal{A}'$ from $\mathcal{A}$ in general. Thus, for example, the ARGOS compiler must bind h_{four} and h_{five} since it is not until the context surrounding these subautomata is known that it becomes clear that these registers never become activated.

A compositional semantics for ARGOS in terms of synchronous automata which allows us to make such optimisations as that above is rather easy to establish, but the same cannot be said for the compositional semantics of ESTEREL, which is a much more complex language. However, to give a flavour of the approach let us return to the discussion of sequential composition begun in Section 4.2. Assume that `P` and `Q` are translated to synchronous automata $\mathcal{P}$ and $\mathcal{Q}$ respectively. We use the common trick when modelling `P ; Q` within a parallel calculus (see, e.g., [29]) of placing $\mathcal{P}$ and $\mathcal{Q}$ in parallel, but preventing $\mathcal{Q}$ from starting until $\mathcal{P}$ signifies that it has terminated. The start signal α suspends (by the control axiom) the activation of $\mathcal{Q}$ until $\mathcal{P}$ emits its ω:

$$\mathcal{P}[\gamma /\!\!/_{\omega}] \mid \mathcal{Q}[\gamma /\!\!/_{\alpha}]$$

The notation $\mathcal{P}[\gamma /\!\!/_{\omega}]$ means that ω is given a fresh name γ in $\mathcal{P}$ (the same γ replaces α in $\mathcal{Q}$). The beauty of synchronous automata is that the '$\mid$' operator is just textual juxtaposition (concatenation). Since γ is broadcast, $\mathcal{Q}$ starts in the same instant $\mathcal{P}$ terminates.

Formally, termination (of $\mathcal{P}$) requires that all control registers become inactive at the end of the instant in which $\mathcal{P}$ emits ω. This is a defining property of synchronous automata, like the control axiom discussed in Section 4.2.3. Fortunately, these turn out to be invariants of our compositional semantics of ESTEREL. Specifically, in [34] we prove that given $\mathcal{P}$ is the translation to synchronous automata of the ESTEREL statement `P`:

1. if $\mathcal{P}$ emits ω, then in the next instant no control register h in $\mathcal{P}$ is active, and
2. if no control register h in $\mathcal{P}$ is active, then no signals are emitted from $\mathcal{P}$.

These results support the main theorem in [34] which states that our synchronous automata semantics coincides with the official behavioural semantics of (pure) ESTEREL [4].

5.3. Merging Control and Dataflow

The basic question is: what is control in dataflow? It seems that calling a dataflow program magically transfers 'control' to the program, just as killing the resulting (endless) process withdraws control. To get a grasp of the issue, recall that "a system is reactive when it is fast enough to respond to every input event, and its reaction latency is short enough that the environment is still receptive to its responses." Implicit in this point of view is the notion that without an input event, no 'response' is given—that is: *if some signal is present then some input signal must be present.* Hence, it should be possible to control a synchronous process by guarding the input.

Let `X` and `Y` be input signals to the LUSTRE program (node) consisting of the single equation `Z = X + Y`. Then the behaviour of the LUSTRE program is equivalent to

$$\begin{aligned} z &\Leftarrow x + y \;\; @\; \widehat{x} \wedge (\alpha \vee h) \\ h &\leftarrow \bullet \;\; @\; \alpha \vee h, \end{aligned}$$

on the level of observable signals. Now preemption may be applied since, cheerfully abusing syntax, `do [Z = X + Y] watching s` is just:

$$\begin{aligned} z &\Leftarrow x + y \;\; @\; \widehat{x} \wedge (\alpha \vee \neg s \wedge h) \\ h &\leftarrow \bullet \;\; @\; \alpha \vee \neg s \wedge h \end{aligned}$$

This simple trick of adding control reconciles the declarative and the imperative programming styles, at least with regard to preemption of dataflows.

Another facet of the interaction between dataflow and controlflow emerges in the example discussed in Section 4.2. Recall that a LUSTRE node (`iData`) was placed in parallel with an ESTEREL module (`iCtrl`) in order to realise the initialisation phase of the production cell. We have already seen how to terminate the LUSTRE node when `done` is emitted, but how is the `OutPress1` signal (say) to be handled by the ESTEREL control program? The point is that

$$\mathit{OutPress1} \;\Leftarrow\; \mathit{Arm1} \leq 0.3 \;\; @\; \alpha \vee h_{\mathit{idata}}$$

from Section 4.2.2, is a Boolean flow which is (modulo preemption) *always* present with a Boolean value; it is not a pure signal as required by

$$\omega \Leftarrow \bullet \ @ \ \mathit{OutPress1} \wedge h_1$$

in Section 4.2.1. The difficulty is that LUSTRE does not distinguish between 'clocks' and Boolean flows (although SIGNAL and ESTEREL do, for example). For strict formality we are therefore obliged to sample the Boolean flows:

$$\begin{array}{lll} \omega & \Leftarrow \bullet & @ \ \mathit{true}(\mathit{OutPress1}) \wedge h_1 \\ h_1 & \leftarrow \bullet & @ \ \alpha \vee (\neg \mathit{true}(\mathit{OutPress1}) \wedge h_1) \end{array}$$

where *true* is the natural projection: $\mathit{true}(b) = \{n \in \widehat{b} \mid b(n) = tt\}$. Ultimately, of course, the control expressions will be implemented by Boolean functions in the target (programming) language to which the synchronous automaton is compiled; *true* is thus a detail required for the consistency of the model, rather than an implementation detail.

This closes our discussion of the mathematical underpinnings of the SYNCHRONIE workbench. Of course, many technical details (of the formal translations) have been omitted in this semantics in a nutshell, but we hope that the reader will have grasped the basic, rather simple ideas. However, one important issue in synchronous programming languages that we have not dwelt upon, but which deserves a mention at least, is that of guaranteeing determinism and reactivity. The formal translations may well generate clauses such as

$$\begin{array}{lll} y & \Leftarrow \bullet & @ \ x \wedge \neg y \\ y & \Leftarrow \bullet & @ \ x \vee y \end{array}$$

which are inconsistent, or allow non-deterministic behaviour. Compilers can report such mishaps and indicate what went wrong. If a synchronous automaton is free of such problems (this comes under the name of *causality analysis*, or *clock calculus*), it is guaranteed to specify a deterministic Mealy machine which may be implemented either in hardware (for Boolean automata), or software.

6. The Architecture of SYNCHRONIE

The SYNCHRONIE project at GMD [33] is organised around the construction a workbench for mixed synchronous language programming which combines editors, compilers, code generators and tools for simulation as well as tools for testing, verification and animation. The integration of languages is based on the synchronous automata sketched above; these play a pivotal role in the system architecture of SYNCHRONIE (see Figure 7 on page 21) since all back-end tools are based on this common representation of synchronous programs. An essential tool that is intrinsic to the workbench is the *linker* which merges synchronous automata from various sources; this approach guarantees a maximum of independence with regard to specific language constructs, and improves modularity in translating programs to synchronous automata. The other kernel function is a *causality analyser* which implements sound heuristics that guarantee reactivity and determinism—the algorithm implemented is similar to that of Shiple et al. [35].

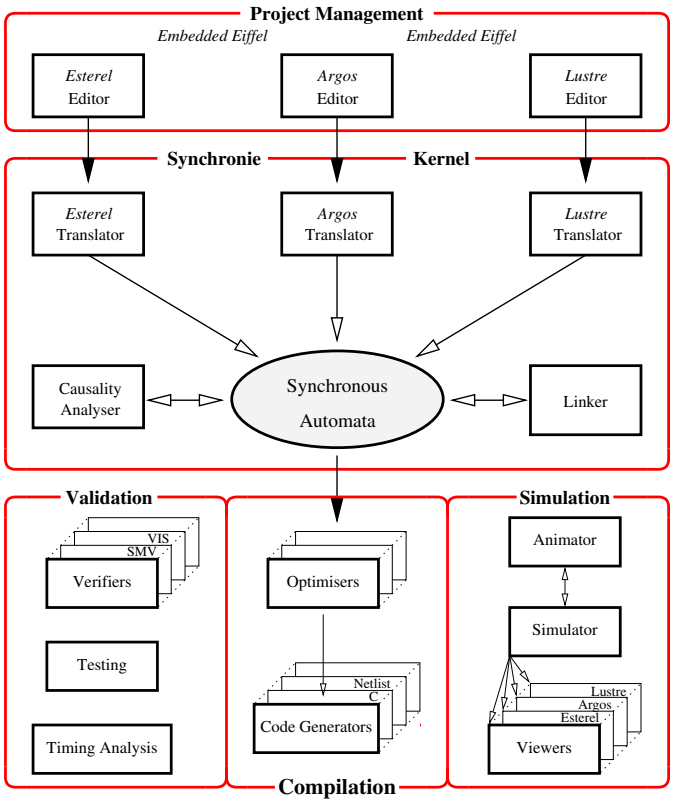


Figure 7. The SYNCHRONIE workbench

Project Management The increasing trend in electro-technical systems of hardware components being replaced by software provides the advantage that a device can be easily adjusted to an individual customer's needs. In this enterprise, synchronous languages themselves address the more delicate parts of the design problem (managing the often convoluted control flow, and synchronisations between distributed elements or control inputs), but this has to be recognised as only a part of the software problem. Overall, the appropriate flexibility in software design is best achieved using object oriented techniques. Although there is no space to elaborate here, this is an important part of the SYNCHRONIE toolset and we are developing an object oriented design environment for real-time applications based on a fusion of ARGOS and ESTEREL (for programming the control flow), and the object oriented language Eiffel (for the arithmetic, and other data manipulations). We call the hybrid language *Embedded Eiffel*.

Thus, a *project* may have multiple components written in a variety of synchronous languages (but 'hosted' only in C or Embedded Eiffel at present). This most visible layer of

the SYNCHRONIE workbench provides programmers with editing, browsing, and project management facilities amongst components written in different languages, and a control front-end to other functions of SYNCHRONIE.

Compilation The compilation functions provide the link towards executing synchronous automata on various platforms. In particular, this function produces code for simulation and analysis tools for rapid prototyping and design validation. This encompasses:

Optimisers may reorganise generated code variously, but the optimisations performed depend on the ultimate destination of the code. For instance, optimisations that eliminate internal signals should be inhibited when code for simulators and symbolic debuggers is required. However, experience indicates that optimisation to minimise the number of registers is desirable for formal verification tools.

Code Generators exploit the close relationship between Boolean automata and sequential circuits which makes it straightforward to generate netlist formats to be exported to hardware synthesis and analysis tools such as SIS [5]. Generating portable executable code like C exploits a hierarchical representation of synchronous automata for runtime efficiency of the resulting code.

Ultimately we aim to produce code for particular target architectures, like the PIC processors for micro-controller applications, but optimisation has to be applied with care, particularly in safety critical applications, as there is often a requirement for *readable* code (meaning that requirements can be traced to the executable code).

Validation Analysis functions are intended to improve confidence throughout the steps of the development chain, helping designers and programmers to reduce the cost of errors by finding them as early as possible. Several types of analysis function are identified with SYNCHRONIE:

Verifiers support design validation through formal proof of logical specifications and requirements. For the most part we provide links and interfaces to third party tools. For instance, Boolean automata equate with sequential circuits so we can use (among many adequate alternatives) the SMV model checker, or the NP-TOOLS propositional logic verifier, on aspects of the design [28, 36].

Testing improves product confidence through simulation. The testing tool allows selection between several criteria like *path* or *boundary values* testing. A prototype test specification environment to support systematic testing in the sense of [30] has been designed around the workbench's graphical ARGOS editor. This second type of verification tool complements the formal verification tools in that it focuses more on behavioural specification [31].

Synchronous automata, and particularly Boolean automata, have a simple execution model that supports fine grained timing analysis. This is important for verifying that a program can meet strong timing constraints (i.e., to satisfy the synchrony hypothesis).

Simulation The simulation functions provide stepwise interpretation of synchronous automata, and a means to effect rapid prototyping. Two simulation tools are distinguished:

Simulators illustrate execution of synchronous programs at the source level by highlighting the syntactical entities of programs which correspond to active signals and registers. Source viewers, like editors, provide browsing facilities among the different components of a program.

Animators support rapid prototyping of simple environments for executing programs. Animators are based on a toolbox of basic components for generating inputs and displaying outputs (via waveform diagrams, etc.).

The production cell control program described in Section 4 has been developed entirely using prototype tools from the SYNCHRONIE toolset (the figures were created using the ARGOS editor). The program itself is moderately complex, having several hundred signals and 140 control registers, but is easily handled by the VIS (a CTL [9] model checker and Verilog simulation and hardware synthesis environment [6]) for which we have a simple back-end code generator.

7. Conclusions and Further Work

Embedded Eiffel has already been used to develop two small-scale industrial products, both now marketed in Germany: the first is a Massflow Meter (a sensitive coriolis device for measuring fluid flow [8]); the second is a small electronic lock system based on Radio Frequency Identification technology [7]. Both were developed using SYNCHRONIE tools, but the high-level (ESTEREL) specification had to be hand translated into assembler in the latter case since the target technology (a PIC16C86 micro-controller with a meager 2k ROM) was not accessible through automatically generated (C) code. Automatically producing code for such limited hardware from high-level specifications is a formidable challenge that represents a next design goal of the SYNCHRONIE project.

The SYNCHRONIE project is a member of the (European) Eureka project SYNCHRON: *Bringing the Synchronous Technology for Real-Time Design* [1]. This is dedicated to promoting synchronous programming languages in industry. The SYNCHRONIE workbench is not therefore being developed in isolation from the current generation of synchronous programming environments (which promote the respective languages independently). Links to export synchronous automata to existing tools, such as model checkers like LESAR [14], and industrial strength verification tools like NP-TOOLS [36], will emerge in due course since we plan to be fully compatible with the DC (for 'declarative code') format [10], principal deliverable of the SYNCHRON project.

The idea of integrating synchronous programming languages is not entirely without precedent therefore, although the efforts to date have mainly focused on providing a common exchange format between downstream analysis tools (Halbwachs gives a good summary of the 'common formats' predating DC [12]). Jourdan and others proposed a semantical integration of ARGOS and LUSTRE, based on a translation of ARGOS into LUSTRE [20], but that first attempt was flawed in that some causally correct ARGOS programs would produce causally incorrect LUSTRE. More recently Maraninchi and Halbwachs have shown how to encode ARGOS in DC [27], and this offers a robust method of merging these two

languages. The 'declarative code', as its name suggests, has been largely influenced by the dataflow synchronous programming community, and there is as yet some doubt as to its suitability for (full) ESTEREL. *Synchronous automata*, in contrast, supply a uniform mathematical framework through which we are able to freely intermix declarative and imperative, textual and graphical programming styles.

Acknowledgments

The work reported in this paper is not that of the named authors only. We should like to take this opportunity to thank all our colleagues in the Embedded Software Design group at GMD, and give them credit for their efforts in guaranteeing the success of SYNCHRONIE. The group has been led throughout by Axel Poigné, with significant input from Reinhard Budde and Leszek Holenderski. Karl-Heinz Sylla (jointly with Budde) designed Embedded Eiffel as a paradigmatic example for the combination of object-oriented design techniques with synchronous programming; their several industrial case studies in the design of real-time systems provided insights from which the whole group has profited. Maciej Kubiczek, working closely with Leszek Holenderski, very professionally wrote compilers for ARGOS and LUSTRE, and all the supporting software in the present version of the workbench. Agathe Merceron did many case studies in verification using the prototype tools being developed for the workbench. Monika Müllerburg takes responsibility for all activities related to testing. Hans Nieters presented the first full version of the presentation layer including an ARGOS editor in Spring 1996 which put all our ideas to work.

References

1. A. Benveniste. SYNCHRON: a project proposal. *ERCIM News*, 12:26, Jan. 1993.
2. A. Benveniste. Synchronous languages provide safety in reactive system design. *Control Engineering*, pages 87–89, Sept. 1994.
3. G. Berry. Preemption in concurrent systems. In R. K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93. Springer–Verlag, Dec. 1993.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
5. R. K. Brayton et al. BLIF-MV: An interchange format for design verification and synthesis. Technical Report UCB/ERL M91/97, University of California, 1991.
6. R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer–Verlag, 1996.
7. R. Budde and K.-H. Sylla. Eingebettete Echtzeitsysteme. *Objekt-Spektrum*, pages 10–16, Apr. 1995.
8. R. Budde and K.-H. Sylla. Objekt-orientierte Echtzeitanwendungen auf Grundlage perfekter Synchronisation. *Objekt-Spektrum*, pages 54–60, Feb. 1995.
9. E. M. Clarke and E. A. Emerson. Design and verification of synchronizing skeletons using branching time temporal logic. In *Logics of Programs Workshop*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer–Verlag, 1981.
10. The common formats of synchronous languages: The declarative code DC. Deliverable of the Eureka SYNCHRON project, Oct. 1995. Available by ftp at `ftp.imag.fr`, `pub/SPECTRE/LUSTRE/DC.tar.gz`.
11. D. D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

12. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht, 1993.
13. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1321, Sept. 1991.
14. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, 1992.
15. D. Harel. STATECHARTS : A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
16. P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *Proceedings of the Custom Integrated Circuits Conference*, pages 213–216, May 1985.
17. L. Holenderski. Production cell in LUSTRE. In *Formal Development of Reactive Systems* [23].
18. Software for computers in the application of industrial safety-related systems. International Electrotechnical Commission, 1994. IEC Standard 1131, Part 3.
19. L. J. Jagadeesan, C. Puchol, and J. E. von Olnhausen. A formal approach to reactive systems software: A telecommunications application in ESTEREL. *Formal Methods in System Design*, 8(2):123–151, Mar. 1996.
20. M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond. A multiparadigm language for reactive systems. In *IEEE International Conference on Computer Languages (ICCL-V), Toulouse, France*. IEEE Computer Society Press, 1994.
21. D. Lanneer, S. Note, et al. Architectural synthesis for medium and high throughput processing with the New CATHEDRAL environment. In R. Camposano and W. Wolf, editors, *High-Level VLSI Synthesis*. Kluwer, 1991.
22. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sept. 1991.
23. C. Lewerentz and T. Linder. *Formal Development of Reactive Systems*, volume 891 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995.
24. O. Maffeïs and A. Poigné. Synchronous automata for reactive, real-time and embedded systems. Arbeitspapiere der GMD 967, Forschungszentrum Informationstechnik GmbH, Jan. 1996.
25. S. Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(7), 1994.
26. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of CONCUR*, volume 630 of *Lecture Notes in Computer Science*, Aug. 1992.
27. F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Formal Techniques for Real-Time and Fault-Tolerance (FTRTFT'96)*, Lecture Notes in Computer Science. Springer Verlag, 1996.
28. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.
29. R. Milner. Interpreting one concurrent calculus in another. *Theoretical Computer Science*, 75:3–13, 1990.
30. M. Müllerburg. Systematic testing: a means for validating reactive systems. *Software Testing, Verification and Reliability*, 5(3), 1995.
31. M. Müllerburg et al. Systematic testing and formal verification to validate reactive systems. *Software Quality Journal*, 4:287–307, Dec. 1995.
32. A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, Eaglewood Cliffs, 1989.
33. A. Poigné. Specification of complex systems. *ERCIM News*, 16:33, Feb. 1994. For up to date information, see `http://set.gmd.de/EES`.
34. A. Poigné and L. Holenderski. Boolean automata for implementing ESTEREL. Arbeitspapiere der GMD 964, Forschungszentrum Informationstechnik GmbH, Dec. 1995.
35. T. R. Shiple, G. Berry, and H. Toutai. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Testing Conference*, pages 328–333. IEEE Computer Society, Mar. 1996.
36. G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In *Proceedings of SAFECOMP'90*, pages 31–6. IFAC, Pergamon Press, 1990.

Received Date
Accepted Date
Final Manuscript Date