Behavioral Specification of a Circuit using SyncCharts:

A Case Study

Charles André, Marie-Agnès Peraldi-Frati
I3S Laboratory - University of Nice Sophia-Antipolis / CNRS
2000 route des Lucioles, BP 121
06903 Sophia Antipolis cédex - France
{andre,map}@i3s.unice.fr

Abstract

In this paper we propose a high-level description of the behavior of digital systems. Behaviors are specified with a graphical synchronous model: "SyncCharts". SyncCharts supports hierarchical descriptions, concurrency and preemption. It is fully compatible with the programming environment of the Esterel synchronous language and can generate output formats understandable by synthesis tools. Thanks to the mathematical semantics of the model, the correctness of the design can be formally established. Taking the example of a non-trivial binary encoder/decoder, we show how our approach makes the design easier, without loss of rigour or efficiency.

Keywords:

System specification and modeling, Validation, Synchronous programming.

Presented at:

Euromicro'2000 - Digital System Design 2000 (DSD'2000), September 5-7, 2000
Maastricht (NL)

I3S Research Report : #00-06 , May 2000

Behavioral Specification of a Circuit using SyncCharts: a Case Study

Charles André, Marie-Agnès Peraldi-Frati
I3S Laboratory - University of Nice Sophia-Antipolis / CNRS
2000 route des Lucioles, BP 121
06903 Sophia Antipolis cédex - France
{andre,map}@i3s.unice.fr

In this paper we propose a high-level description of the behavior of digital systems. Behaviors are specified with a graphical synchronous model: "SyncCharts". SyncCharts supports hierarchical descriptions, concurrency and preemption. It is fully compatible with the programming environment of the Esterel synchronous language and can generate output formats understandable by synthesis tools. Thanks to the mathematical semantics of the model, the correctness of the design can be formally established. Taking the example of a non-trivial binary encoder/decoder, we show how our approach makes the design easier, without loss of rigour or efficiency.

1. Introduction

Different approaches can be adopted in digital circuit design. Depending on his/her scientific background, the designer may prefer either abstract specifications (equational or functional) or state-based specifications with graphical representation. In both cases, there exist pros and cons. Mathematical expressions lend themselves to advanced compilation techniques and easy connections to formal verification tools. Unfortunately, mathematical formulations are often reserved to experts. A non mathematically-inclined customer is never sure that the proposed mathematical expression fully addresses his/her requirements. The gap between requirements and specifications also exists in state-based descriptions, but to a lesser extent. Many people feel more comfortable with graphical representations. Explicit representation of states facilitates the understanding of behaviors; possible animation of the model makes it still easier. A danger of graphical representations may be a weak, or even worse, the absence of, semantics. Too many graphical models are semi-formal, indeed even informal. Ambiguity disqualifies such models in digital circuit design. "State Transition Graphs" do not suffer from weak foundations and they are often used in the design of simple sequential circuits. Their drawback is that they are "flat" models, liable to explosion of the state space for complex applications. To sum up, a mathematical approach allows a precise and powerful expression of behaviors, and easy connections to synthesis and verification tools. Graphical state-based representations of behaviors may be less abstract and thus within the reach of a larger audience. However, complex systems may lead to huge and useless graphs. Whatever the representation, it should be readable, easy to extend and to maintain, with possibilities of simulations and formal verifications.

What we propose is a behavioral description based on a state-transition model, mathematically well-founded. State transition graph is a low-level model not suited to complex system design. Instead, we opt for a higher-level model, like Statecharts [1], able to deal with hierarchy, concurrency and pre-emption. The actual model we use is "SyncCharts" [2], clearly inspired by Statecharts. The two models differ in their underlying semantics. The SyncCharts[1] semantics is fully synchronous and perfectly fits Esterel's semantics [3]. The semantics of Statecharts, such as the one adopted in Statemate [4], is more complex (micro-step semantics). Moreover, SyncCharts offers richer constructions for preemption. Being akin to Esterel, SyncCharts may also include textual descriptions written in Esterel: the designer may choose textual or graphical descriptions for different parts of his/her design.

In fact, SyncCharts is now fully integrated in the "Esterel Studio" platform, marketed by Simulog [5]. As a consequence, SyncCharts has direct access the whole programming platform developed for Esterel: compilers, simulators(XES), model-checkers (XEVE [6]) and circuit optimizers that rely on SIS [7] and TiGeR [8] (an efficient BDD-based tool).

SyncCharts can be used to specify the behavior of any control-dominated discrete-event system. In this paper we present the use of SyncCharts in the design of a binary stream encoder/decoder. Through this example we try to draw advantages of our approach, and contrast it with the

[1] "SyncCharts" is the name of the model, whereas a "syncChart" is an instance of the model.

classical approach proposed in Zahnd's book on Sequential Machines [9].

The next section describes the encoder problem. This small example contains several pitfalls that are analyzed. The solution proposed in Zahnd's book is then recalled and commented. This is the occasion to point out strengths and limitations of finite state machine modeling. The next section illustrates our approach. Main features of SyncCharts are introduced through the example. We explain how to specify the expected behavior in a modular way. The fifth section considers optimization issues and formal proofs of properties.

2. Example of an Encoder/Decoder

The Encoder/Decoder system, represented on Figure 1, is classical in the field of data transfer. It has been devised for electrical transmission by wire. Even if wireless communication has lessened the significance of this coding technique, it is still worth studying it because it raises several interesting algorithmic issues.

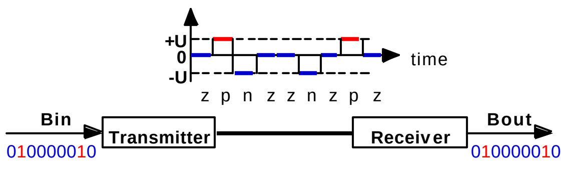


Figure 1. Encoder/decoder

Informal presentation

This encoding/decoding system is used to transmit binary streams. Bits are encoded into a three-level valued electrical voltage: positive ($p=+U$), negative ($n=-U$), or null ($z=0$), for a given constant positive voltage U.
Two requirements are imposed:

1. The mean voltage must be 0, and at each instant the accumulated voltage must stay between $-U$ and $+U$. This avoids electronic problems due to bias polarity.
2. The transmitted code shall never contains more than three consecutive null values. This prevents from clock de-synchronization and misinterpretation of a line break as a continuous stream of z's.

Typically these requirements are imposed for physical/electrical reasons. The first requirement is easily captured by a simple encoding technique: z for 0, and either n or p, in alternation, for 1. The second requirement is trickier: sub-sequences of four consecutive 0's are also encoded with n or p. In order to set apart "true" 1's from "false" 1's (series of four 0's) "Violation" of polarity is used, instead of "Alternation".

Formal specification

Encoding. Let $\boldsymbol{X} = \{0,1\}$ be the input alphabet, $\boldsymbol{U} = \{n,z,p\}$ the output alphabet; the encoding function is a mapping from $\boldsymbol{X}$-sequences onto $\boldsymbol{U}$-sequences:

$\mathbf{encoding} : \boldsymbol{X}^* \rightarrow \boldsymbol{U}^*$.

This mapping is length-preserving:

$\forall x \in \boldsymbol{X}^* \quad |x| = |\,\mathbf{encoding}\,(x)\,|$ where $|x|$ denotes the length of sequence x.

Let $x = \big(x(1),\, x(2),\, \ldots,\, x(n),\, \ldots\big)$ be the input sequence, where $x(k) \in \boldsymbol{X}$, and $u = \big(u(1),\, u(2),\, \ldots,\, u(n),\, \ldots\big)$ be the associated encoded sequence, where $u(k) \in \boldsymbol{U}$. $x(1..k)$ denotes the sub-sequence $\big(x(1),\, x(2),\, \ldots,\, x(k)\big)$, $\bullet$ the sequence concatenation, and λ the empty sequence. Recall that λ is neutral element for $\bullet$.

The first requirement is expressed by

$$\forall n \geq 1 \left| \sum_{k=1}^{k=n} u(k) \right| \leq U$$

the second one by

$$\forall n > 3 \;\neg \big(u(n) = u(n\text{-}1) = u(n\text{-}2) = u(n\text{-}3) = z\big)$$

The encoding functions defined below are *supposed* to respect these requirements. Establishing this property is one of our challenge.

Encoding functions:

- *Normal encoding*:
  $\mathbf{encoding}(0) = z$ and $\mathbf{encoding}(1)$ = either p, or n, alternately.

- *Exceptional encoding*:
  $\mathbf{encoding}(0000) = P\bullet z\bullet z\bullet V$, where P stands for "Parity" and V for "Violation". $P,V \in \boldsymbol{U}$ are computed with the auxiliary functions Π and ∇ defined below.

Let $\Pi : \boldsymbol{U}^* \times \text{Nat} \rightarrow \text{Boolean}$, such that

$\Pi(u,k) = \Big(\mathbf{card}\big(\{u(j) \mid (u(j) \neq z) \wedge (1 \leq j \leq k)\ \}\big) = 0 \ \mathbf{mod}\ 2\Big)$.

Given a $\boldsymbol{U}$-sequence u and an index k, this function returns the (even) parity of the number of $u(j)$ components before or at index k, which are different from z.

Let ∇: $\boldsymbol{U}^* \times \mathrm{Nat} \rightarrow \mathrm{Nat}$, such that
$\nabla(u,k) = \mathrm{Max}\{j \mid (1 \le j \le k) \wedge (u(j) \ne z)\}$ if defined, 0 otherwise.
Given a $\boldsymbol{U}$-sequence u and an index k, this function returns the index j of the nearest $\boldsymbol{U}$-component, before or at index k such that $u(j)$ is different from z.

By convention, we assume that $u(0) = n$, $\Pi(\lambda,0)$ = true and **encoding**$(\lambda)=\lambda$.
Let – be a unary operation on $\boldsymbol{U}$, such that $-n = p$, $-p = n$, $-z = z$.
Standard encoding is such that :
$\forall\, x(1..k)$ such that $\neg\big(x(k) = x(k\text{-}1) = x(k\text{-}2) = x(k\text{-}3) = 0\big)$
let $y = x(1..k\text{-}1)$, v = **encoding**(y), and u = **encoding**(x)

u = **encoding** $(y \bullet 0) = v \bullet z$

u = **encoding** $(y \bullet 1) = v \bullet -v\big(\nabla(v,k\text{-}1)\big)$ (Alternation)

Exceptional encoding is such that :
$\forall\, x$ ending with 0000 , $x(k) = x(k\text{-}1) = x(k\text{-}2) = x(k\text{-}3) = 0$
let $y = x(1..k\text{-}4)$, v = **encoding** (y), and $\varpi = v\big(\nabla(v,k\text{-}4)\big)$
u = **encoding** $(y \bullet 0 \bullet 0 \bullet 0 \bullet 0) = v \bullet P \bullet z \bullet z \bullet V$ where

P = if $\Pi(v,k\text{-}4)$ then z else $-\varpi$

and V = if $\Pi(v,k\text{-}4)$ then ϖ else $-\varpi$ (Violation)

For a $\boldsymbol{U}$-sequence $u(1..k)$, a *violation* occurs when $u(k) = u\big(\nabla(u,k\text{-}1)\big)$, an *alternation* when $u(k) = -u\big(\nabla(u,k\text{-}1)\big)$.
So, P is z if Parity is even, or Alternation if odd; whereas V is always a Violation.

This formal description is all but obvious (recursion and case separations). It needs some familiarity with notations, mathematics, and above all, the links between the requirements and this specification have to be justified. This is definitely beyond the scope of this paper. A solution to the implementation of such an encoder should be a compiler that directly generates a circuit from this mathematical specification. What is great with mathematics is that the satisfaction of the requirements might be automatically checked. We did not do that for two reasons: 1) we do not have access to such checkers, 2) we adopt another approach, less formal, albeit rigorous.

Table 1. Encoding example

k	1	2	3	4	5	6	7	8	9	10	11
x	0	1	**0**	**0**	**0**	**0**	1	**0**	**0**	**0**	**0**
u	*z*	*p*	*n*	*z*	*z*	*n*	*p*	*z*	*z*	*z*	*p*
Π	t	f	t	t	t	f	t	t	t	t	f
A/V		A	A			**V**	A				**V**
Acc	0	+U	0	0	0	-U	0	0	0	0	+U
N/E	N	N	**E**	**E**	**E**	**E**	N	**E**	**E**	**E**	**E**

k: instant; *x:* input; *u:* output ; Π: Π(*u*,*k*);
A/V: Alternation/Violation; Acc: Accumulated value;
N/E: Normal/Exception.

Table 1 illustrates the encoding of a sequence. The different lines of the array represent from top to bottom: the instant (k), the input sequence (x), the encoded output sequence (u), the Parity status, the Violation and Alternation occurrences, the accumulated value of the voltage on the wire showing that requirement 1 is fulfilled. Last line indicates N(ormal) or E(xceptional) encoding. Bold **0**'s highlight "exceptional" sub-sequences.

Causality. The specification shows that the value of $u(k)$ depends on values of $x(k)$, but also of $x(k+1)$, $x(k+2)$, $x(k+3)$, that is, on the future. Therefore, this system is *not causal*, and so, it cannot be realized, as such, by a sequential machine. The classical solution to make a system causal, is to introduce a bounded delay (a 3-delay is needed in this application). Figure 2 shows the classical solution. The synthesis effort is then in the design of a standard Mealy machine (box on the right in Figure 2). u' is the new output signal, such that $\forall\, k$, $u'(k) = u(k\text{-}3)$. Of course the first three values of u' are meaningless.

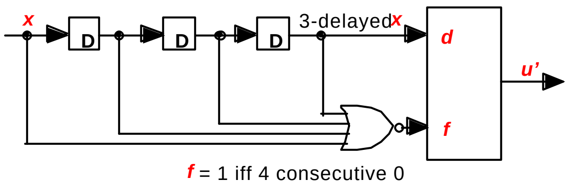


Figure 2. Solving the causality problem.

Decoding. The decoding problem is analogous to the encoding one. Due to the symmetry of the problem, decoding requires 3 delays to make the system causal and the algorithm is a bit simpler than for encoding: Parity has not to be taken into account. Detailed design of the decoder is omitted in this presentation.

3. A classical solution

In his book, Zahnd chose a Mealy machine as a model to represent the encoder example. Figure 3 specifies the behavior of the Mealy machine whose inputs are d and f, and output u'.

Graphical representation may have a great explanatory power. What is difficult is to choose a "good" layout. The one in Figure 3, is specially effective.

Annotations (text, dashed lines and line thickness) may be very useful to improve understanding:

- Two dashed lines delimit 4 quadrants. The vertical line is associated with "Parity", the horizontal line with "Violation".

- Transitions that cross the horizontal line correspond to violations.
- In states on the left-side of the vertical line, parity is even; in states on the right-side, parity is odd. All transitions between states in the two upper quadrants, or the two lower quadrants correspond to alternations.
- Exceptional sub-sequences are distinguished by thicker lines.

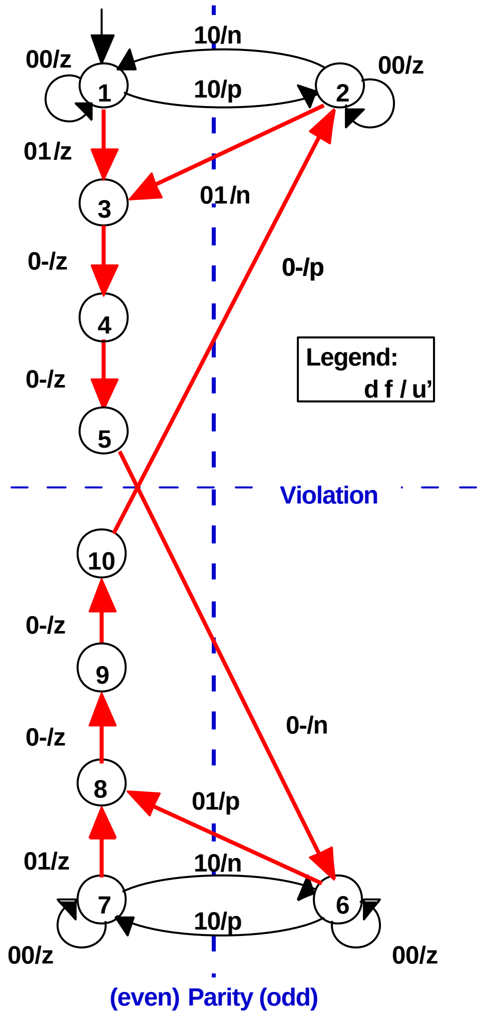


Figure 3. Mealy machine of the Encoder

In spite of its qualities this graph has several drawbacks:

- The graph is not the state graph of the encoder, but only of a part. The actual graph is a composition of this one with a 3-delay machine. Besides its number of states too large to be reasonably displayed on a single sheet, we did not succeed in drawing an "easy-to-interpret" graph.
- The graph Figure 3 may be misleading: variables *f* and *d* are not independent inputs: $f = 1$ implies $d = 0$. A careless user should believe that, according to the graph, a sequence of 4 consecutive inputs $df = 00$ lets the machine in the initial state, which violates requirement 2.
- Even a small change in the specifications could jeopardize this bright design. This is a well-known problem with automata. This is a serious impediment to incremental design.

4. The SyncCharts Approach

We illustrate the use of SyncCharts through the "Encoder" example. A detailed presentation of the syntax and the semantics of SyncCharts has been published elsewhere [2].

Signals

In synchronous modeling, *communications* are abstracted as "signals". With each signal is associated a presence attribute and optionally a value attribute. A signal that conveys a value is said to be a "valued signal", otherwise it is a "pure signal". For binary circuit, Boolean valued signals can be used. A better choice is to use pure signals with the convention that a present signal is "true", an absent signal is "false". Since this is a matter of interpretation, the designer is free to assign a different meaning to the presence status. Having only pure signals is important when we plan to use symbolic model-checkers like XEVE.

With respect to the environment two kinds of signals are distinguished: **input** signals that convey information to the controller, and **output** signals that export information.

For the Encoder, we have :

- input **Bin**;
- output **Minus**, **Zero**, **Plus**;

There is a third kind of signals: the **local** signals. They are not visible to the outside, they convey internal information and are used in synchronization.

Agents

The Encoder is decomposed into several cooperating agents. Each agent has a simple and well-defined mission. For the Encoder, we chose 4 agents (Figure 4):

- The detector of 4 consecutive 0's (**DETECTOR**),
- The parity manager (**PARITY**),
- The output manager that decides to emit either Plus or Minus (**NONZERO**),
- The sequence manager that is the core of the encoding algorithm (**SEQUENCER**).

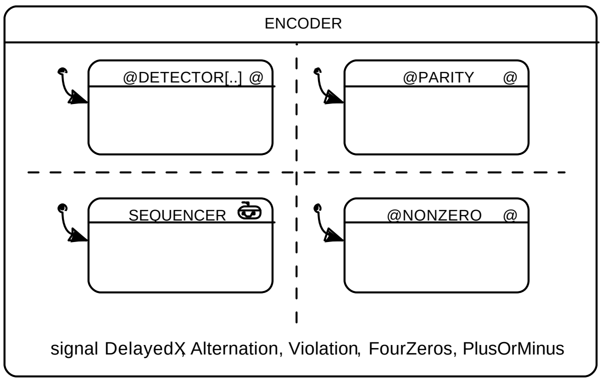

ENCODER
@DETECTOR[..] @
@PARITY @
SEQUENCER
@NONZERO @
signal DelayedX, Alternation, Violation, FourZeros, PlusOrMinus

Figure.4: SyncChart of the encoder (level 0)

In SyncCharts, the behavior of an agent is specified with a macro-state. A macro-state is translated into a module in Esterel. A macro-state is drawn as a rounded-corner rectangle. It is advisable to give a name to a macro-state. An icon at the upper right corner of a macro-state indicates the type of macro-state (@ stands for reference, i.e., the body of the macro-state is defined elsewhere; the icon in **SEQUENCER** says that the graphical representation is defined "in-place"; a third icon, not used in Figure 4, indicates a textual body, directly written in Esterel). Dashed lines mean that all the macro-states are composed in parallel.

Interactions

According to the synchronous hypotheses, communications among agents are modeled by instantaneous broadcasting of signals. Local signals **DelayedX, Alternation, Violation, FourZeros, PlusOrMinus** have been introduced to support communications among the Encoder's agents. The text at the bottom of the macro-state Figure 4 is a declaration of these local signals.

Agents' behavior

Macro-states are recursive structures: the body of a macro-state is itself a syncChart. That is a way to deal with hierarchy. SyncCharts also applies the "Write Once, Read Many" principle: a macro-state defined once, can be re-used several times with possible signal renamings.

To leave a macro-state, SyncCharts uses pre-emptions. Pre-emption is the possibility for an agent to prevent other agents from executing. A pre-emption may be temporary (suspension) or definitive (abortion). In imperative synchronous programming, pre-emption is used for synchronization. There exist two kinds of abortion: weak and strong, and also a special form called "normal termination". SyncCharts supports all of them. The shape of the transition reflects the type of pre-emption: a simple arrow stands for weak abortion, an arrow with a small circle at the end is a strong abortion, and an arrow ending with a triangle is a normal termination. As in Statecharts, arrows are labeled with triggering events (signals whose presence causes the transition), effects (signals emitted during the transition) and possibly a guard (pre-condition for the transition to be fired).

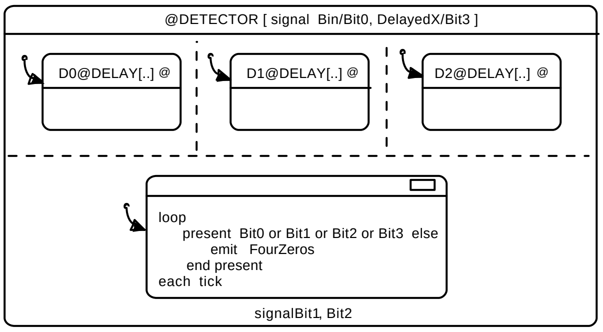


Figure 5. SyncChart of the Detector.

Figure 5 is the syncChart for the **DETECTOR**. The 3-stage shift register is made by the parallel composition of three instances of the **DELAY** macro-state (the **DELAY** macro-state and the renamings of the **DELAY** instances are omitted). The detection is made by a textual macro-state: the Esterel code says that signal **FourZeros** is emitted whenever all 4 bits (**Bit0**, …, **Bit3**) are absent. Note that **Bit1** and **Bit2** are local signals. **Bit0** is renamed as **Bin** (the input of the Encoder) and **Bit3** as **DelayedX**.

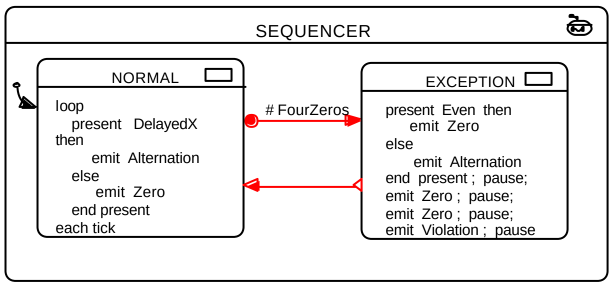
Figure 6 is the syncChart for the **SEQUENCER**. This syncChart expresses the algorithm. The two cases are set apart; The treatment in each case is clearly identified. Switching from mode "Normal" to mode "Exception" is instantaneous: as soon as **FourZeros** is present (this signal is emitted by **DETECTOR**) the control leaves the **NORMAL** macro-state to enter the **EXCEPTION** macro-state by a strong pre-emption. When the exceptional sequence is finished, spontaneously (normal termination) the control returns to the **NORMAL** macro-state.

Compilation

The outlines of the compilation chain are:

- SyncCharts compiler: From a syncChart to a semantically equivalent Esterel program;
- Esterel compiler: From an Esterel program to output code:
  - C programs for simulation with XES
  - Blif description for optimization (with SIS) and verification (XEVE).

  Blif (Berkeley Logic Interchange Format) is a textual representation of a circuit. It is an input format to SIS.



```
loop
  present  DelayedX
then
    emit  Alternation
  else
     emit  Zero
  end present
each tick
```

```
present  Even  then
    emit  Zero
else
     emit  Alternation
end  present ;  pause;
emit  Zero ;  pause;
emit  Zero ;  pause;
emit  Violation ;  pause
```

Figure 6. SyncChart of the sequencer

5. Validation and Performance

To validate the design, we proceed in two steps: simulation and formal verification.

Test of scenarios

XES is an interactive simulator, which is part of the Esterel distribution. Given an Esterel program, XES automatically builds simulation panels that show the status of input signals (set by the user) and output signals (set by the program under test). Execution is traced in the source program, so that the user can visualize concurrent evolutions and pre-emptions. This possibility is now extended to SyncCharts. "Esterel Studio" can do the animation of the syncChart of the controller. Active macro-state are colored red, fired transitions are colored green. This is an invaluable aid to the understanding of the behavior. A software "tape recorder" allows the user to record and play sequences of inputs (scenarios). Testing of scenarios reveals many misconceptions. Since SyncCharts is a high-level description of the behavior, it is often easy to find out the bug and correct it.

Safety properties

Even if the design passes successfully all the tests, it is not sure that all the cases have been covered. In order to establish a safety property, we have to check this property in all reachable states of the controller. The size of the actual reachability set can make the analysis untractable. Fortunately, there exist symbolic computations of the reachability set that allow for state abstraction without loss of exhaustivity. XEVE, a symbolic model-checker available in the Esterel platform, is able to compute (symbolically) the state space of a given program. XEVE can formally establish whether or not a safety property is satisfied. Safety properties can be expressed by temporal logic formulas. We prefer to use the *same formalism* to express both behaviors and properties: a property is given as an Esterel module or a syncChart.

The principle of the proof is to associate an **observer** with the property and compose this observer in parallel with the controller to check. An observer is a reactive agent that "observes" input and output signals of the program and emits a "violation signal" as soon as the property is not satisfied. XEVE symbolically executes the program composed with the observer. If the violation signal is never emitted, then the property is satisfied, otherwise XEVE returns a sequence leading to the counter-example. This is a very effective way to find out deeply hidden errors. Of course, several properties can be checked at once if you use several observers. Note that the safety property observers are used during the verification phase (i.e., the symbolic execution of the controller augmented with the observers). There are needless at run-tine for a guaranteed property.

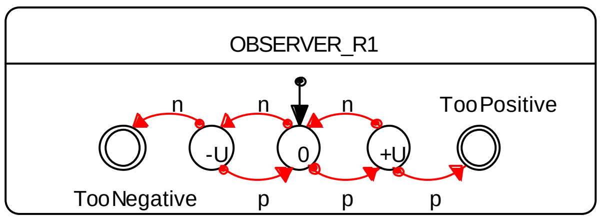


Figure 7. Observer for requirement 1

For the Encoder, a simple state transition graph (a special case of SyncCharts) is sufficient to prove that requirement 1 is satisfied (see Figure 7). For requirement 2, a syncChart with a strong abortion is more suitable (see Figure 8).

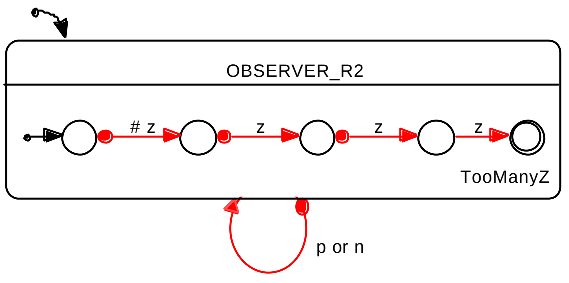


Figure 8. Observer for requirement 2

Other interesting safety properties:
A first property concerns the behavior of the encoder: at each instant one and only one signal out of *n, z, p* is emitted. This kind of property is expressed by a combinatorial formula, easily captured by a textual Esterel module.

```
module OBSERVER_EXCLUSION :

input n, z, p;          % emitted by the controller
output non_exclusive;        % violation signal

loop
  present  (n and not z and not p)
          or (not n and z and not p)
          or (not n and not z and p)
  else
      emit  non_exclusive
  end present
each tick

end module
```

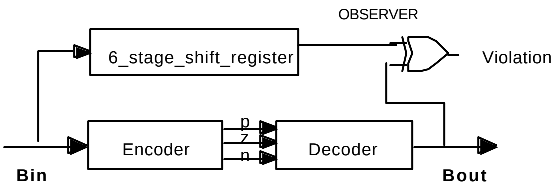


Figure 9. Observer for sequence preserving

Another essential property to prove is that the pair “Encoder Decoder” works correctly. This property is a safety property, too. You compose in parallel the syncChart of the Encoder, and the syncChart of the Decoder. Then you build an observer that compares the input of the Encoder (signal **Bin**) with the output of the Decoder (signal **Bout**). They should be identical, up to some delay (here a 6 instant delay). The observer puts **Bin** as input to a 6-stage shift register (see the **DETECTOR**) and compares the output of the shift register with **Bout**. Fortunately the property is satisfied. Figure 9 outlines the solution and clearly shows that an observer modifies in no way the behavior of the controller

Performance

The translation from SyncCharts to Esterel is structural: The translation is fully automatic but not always clever. In many cases an expert in Esterel language, can find more efficient translations. However, this is not a problem because there exist tools able to optimize the generated code, at the circuit level.

Efficiency must be assessed for the whole process, from the requirements to the implementation.

Examples

Encoder || Observer for requirement 1: reachability set: 159 states.
Encoder || Observer for requirement 2: reachability set: 175 states.
Encoder: reachability set: 105 states.
But Encoder and optimization with SIS: 35 states, that is what you obtain with much effort using the classical approach.
Decoder and optimization with SIS: 31 states.
Encoder || Decoder || Observer for identical bit streams: 623 states.

6. Conclusion

State transition graphs are often used to express the behavior of sequential systems. They are understandable even by non-specialists, but this flat model is not suitable for complex systems. Moreover, state machine descriptions are very sensitive to small changes in the specifications. This is detrimental to incremental design. Since systems involve more and more concurrency and exceptional behaviors, there is a demand for hierarchical models, dealing with concurrency and pre-emption. In this paper, we have proposed SyncCharts, a model derived from Statecharts, to address this problem.

SyncCharts allows high-level specification of reactive behaviors. It is a synchronous model, fully compatible with the Esterel language, so that, a syncChart can mix graphical and textual descriptions sharing a common semantics.

Taking an example of Binary stream Encoder/Decoder we have shown how SyncCharts favors decomposition of the system into interacting agents. These agents are tightly coupled. Because synchronous operations compose well, the emergent behavior remains tractable.

SyncCharts takes advantage from the programming environment developed for Esterel. Given a syncChart, an interactive simulator can be automatically built. This simulator is used to test scenarios. Moreover, programming facilities allows the user to visualize the execution at the source level, i.e., by animation of the syncChart.

The mathematically defined semantics of the model makes it easy to conduct formal verifications on SyncCharts. Safety properties are checked by symbolic model-checking. The expression of the properties can be made in SyncCharts itself, or in Esterel.

Finally, we consider the possibility to use SyncCharts as a language for the design of digital circuits. After compilation, a subclass of syncChart can be compiled into a circuit (blif file) and therefore be further optimized. The results obtained with the Encoder/Decoder are encouraging, since after optimization, our high-level approach gives the result that could have been obtained by hand-coding, but at the price of much effort. Moreover the design was proven correct.

We have, now, to tackle larger designs, for which hand-coding is ruled out. The Esterel compiler, on which SyncCharts indirectly relies, has been developed to cope with large problems. In order to improve the efficiency of the circuit design we plan to explore two ways:

- First, better structural translations of SyncCharts into Esterel without any change in the Esterel language itself,
- Second, cooperative development[2] of SyncCharts translator and Esterel compiler so that behavioral information present in SyncCharts can be directly used by the Esterel compiler (e.g., mutual exclusion between sub-systems, sequential execution of sub-systems…).

References

[1] D. Harel. “Statecharts: a Visual Formalism for Complex Systems”, *Science of Computer programming*, 1987, vol 8, pp 231-274.

[2] C. André. “Representation and Analysis of Reactive Behaviors: A Synchronous Approach” IEEE-SMC Computational Engineering in Systems Applications (CESA), Lille (F), July 1996, pp 19—29.

[3] G. Berry, G. Gonthier. “The Esterel Synchronous Programming Language: Design, Semantics, Implementation” *Science of Computer Programming,* 1992, vol. 19, n°2, pp 87-152. (current version of Esterel v5_21: http://www.inria.fr/meije/verification/esterel)

[4] I-Logix, “Statemate MAGNUM”, http://www.ilogix.com

[5] Simulog. “Esterel Studio”, http://www.simulog.fr

[6] A.Bouali. “Xeve: an Esterel Verification Environement”, *Int'l Conference on Computer-Aided Verification (CAV'98),* june/july 1998, Vancouver, BC Canada. Also available as a technical report INRIA RT-214, 1997.

[7] E.M Sentovitch, K.J Singh, et al. “SIS: a System for sequential circuit synthesis”. Technical report, *UCB/ERL M92/41,* U.C. Berkeley, May 1992.

[8] O. Coudert, J.C Madre, H. Touati. “TiGeR version 1.0, User Guide”, Digital Paris Research Lab. Dec 93, commercialized by Xorix.

[9] J. Zahnd. “Machines séquentielles“. *Presses Polytechniques Romandes*. 1987.

[2] Both are developed at Sophia-Antipolis (France).