Stability of Discrete Sampled Systems*

J.-F. Héry, J.-C. Laleuf†
EDF/DER, Clamart - France

N. Halbwachs, F. Lagnier‡
Vérimag, Grenoble - France

Abstract

We consider the wide class of real-time systems that periodically sample their inputs. A desirable property of such systems is that their outputs should be, in some sense, more precise when the sampling period gets shorter. An approximation of this property consists in requiring that, whenever the inputs don't change, the outputs stabilize after a finite number of steps. We present a set of heuristics to check this stability property, in the case of purely Boolean systems. These heuristics have been experimented on a nuclear plant control software, and have been shown to dramatically reduce the cost of stability analysis.

1 Introduction

Many real-time embedded systems, appearing in industrial control (e.g., plant supervision, flight control, ...), are *periodic sampled systems*. The global behavior of such systems is quite simple: it consists in a periodic loop, sampling inputs — from sensors or, more generally, from a shared memory — computing the corresponding outputs, and updating the local memory for the next step. Generally speaking, this periodic behavior is a discrete approximation of an ideal, analog, behavior, which would instantly compute a continuous result from continuous inputs, and instantly react to any relevant discrete change of the inputs. With this intuitive intention in mind, one would naturally expect that the shorter is the period, the more accurate is the approximation. If we restrict ourselves to *discrete* control systems, shortening the period should only involve a more precise perception of input events — i.e., detecting transient changes, suitably ordering events that was previously considered simultaneous, ... — and a faster reaction to these events. Let us call *monotonicity* this intuitive property.

Now, in actual discrete systems, it happens that, in some situations, some Boolean outputs of the system oscillate permanently while the inputs don't change. This behavior obviously violates our intuitive notion of monotonicity, since shortening the period would only make the oscillation faster (see Fig. 1).

Such a phenomenon is generally considered as an error, and appears to be very difficult to detect statically, since it can happen only in very specific states of the system. A system where it cannot happen is called *stable*: in a stable system, whenever the inputs remain unchanged,

*This work has been partially supported by ESPRIT-LTR Project "SYRF".

†{Jean-Francois.Hery,Jean-Claude.Laleuf}@der.edfgdf.fr

‡{Nicolas.Halbwachs,Fabienne.Lagnier}@imag.fr

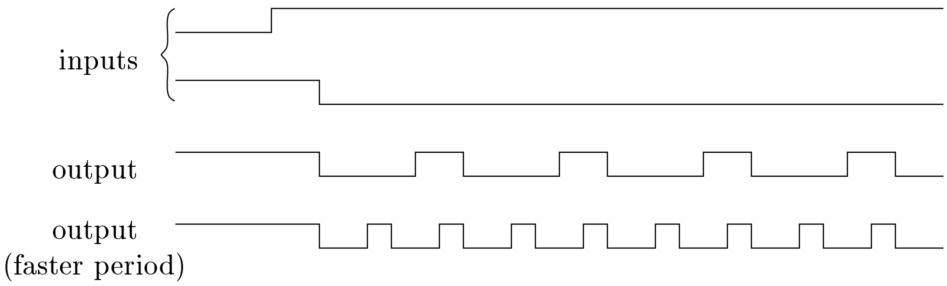


Figure 1: Output oscillation

the outputs reach stable values after a finite number of periods. Deciding system stability can involve the knowledge of its whole state graph.

The goal of this paper is to propose some heuristic techniques allowing, in most practical cases,

- either to ensure the system stability without building its state graph;
- or to focus the problem to some small parts of the system, corresponding to small parts of the state graph, which can be efficiently built.

We restrict ourselves to discrete sampled systems: they are formalized in Section 2, as standard *Mealy machines*, which can classically be represented as *operator networks* (sequential circuits), made of combinational Boolean gates and Boolean memories. In Section 3, we formally define the notion of stability, and the little stronger property that will be actually analyzed: it requires that whenever the inputs stay stable, the outputs *and the state* (i.e., the memories) stabilize after a finite number of steps. In Section 4, we show that each strongly connected component (SCC) of the operator network can be analyzed in turn, and that the stability of some simple SCCs, containing only one memory, can be shown. In Section 5, we use local necessary conditions of unstability, called *local cycle conditions*, the unfeasibility of which is often easy to show. In Section 6, we broaden these cycle conditions, using approximate values of the involved stable variables.

2 Definitions and Notations

Throughout the paper, we will assimilate discrete systems with standard Mealy machines. Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values, and let us note the disjunction additively, the conjunction multiplicatively, and the negation by the "*bar*" notation ($\overline{f} = \neg f$). A *Mealy machine* with n state variables, m input variables and p output variables, is a pair (τ, δ) where

- τ is a total function from $\mathbb{B}^{m+n}$ to $\mathbb{B}^n$ (transition function) given as a vector $(\tau_k)_{k=1\ldots n}$ of n functions from $\mathbb{B}^{m+n}$ to $\mathbb{B}$. If $s \in \mathbb{B}^n$ (state), $\iota \in \mathbb{B}^m$ (input), $\tau(\iota, s)$ denotes the vector $[\tau_k(\iota, s)]_{k=1\ldots n} \in \mathbb{B}^n$ (next state from s for ι).
- δ is a total function from $\mathbb{B}^{m+n}$ to $\mathbb{B}^p$ (output function), also given as a vector $(\delta_\ell)_{\ell=1\ldots p}$ of p functions from $\mathbb{B}^{m+n}$ to $\mathbb{B}$. $\delta(\iota, s)$ is the output in s for ι.

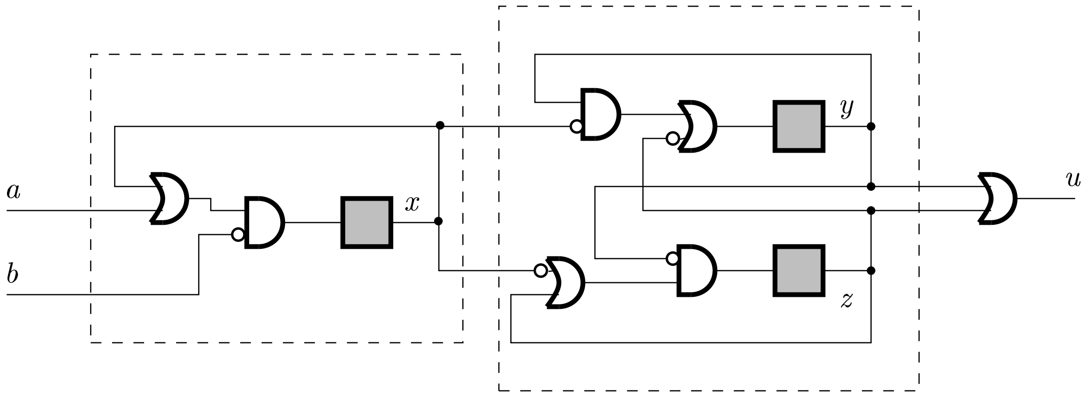


Figure 2: a Mealy machine as an operator network

An *input sequence* to the machine is an infinite sequence $\mathcal{I} = (\iota_0, \iota_1, \ldots, \iota_i, \ldots)$ of vectors of $\mathbb{B}^m$. The *run* and the *image* of the machine on such an input sequence $\mathcal{I}$ are, respectively, the sequence $\mathcal{S} = (s_0, s_1, \ldots, s_i, \ldots)$ of elements of $\mathbb{B}^n$ (states), and the sequence $\mathcal{O} = (\omega_0, \omega_1, \ldots, \omega_i, \ldots)$ of elements of $\mathbb{B}^p$ (outputs), defined by

$$
\begin{array}{l}
s_0 = 0^n \text{ (initial state}^1\text{)} \\
\forall i \geq 1, \quad s_i = \tau(\iota_{i-1}, s_{i-1}) \\
\forall i \geq 0, \quad \omega_i = \delta(\iota_i, s_i)
\end{array}
$$

More concretely, we will name input, output, and state variables by identifiers, and define the transition functions using the "*prime*" notation (read x' as "*next* x").

Example 1: For instance, the system of Boolean equations:

$$
\begin{array}{rclcrcl}
x' & = & \overline{b}\,(a + x) & \qquad & y' & = & \overline{z} + y.\overline{x} \\
z' & = & \overline{y}\,(\overline{x} + z) & \qquad & u & = & y + z
\end{array}
$$

is a convenient way of describing a machine with 2 input variables (a, b), 3 state variables (x, y, z), and 1 output variable (u), with

$$
\begin{array}{rclcrcl}
\tau_1(\iota, s) & = & \neg\iota[2] \wedge (\iota[1] \vee s[1]) & \qquad & \tau_2(\iota, s) & = & \neg s[3] \vee (s[2] \wedge \neg s[1]) \\
\tau_3(\iota, s) & = & \neg s[2] \wedge (\neg s[1] \vee s[3]) & \qquad & \delta(\iota, s) & = & s[2] \vee s[3]
\end{array}
$$

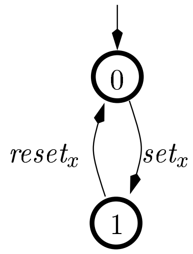
Another classical way of describing a Mealy machine is by an **operator network**. Boolean functions are described by their gate networks, and state variables correspond to memories. Fig. 2 shows a network corresponding to the above machine.

An operator network can be viewed as a directed graph, the nodes of which are the operators and memories of the network, and the edges of which are the "wires" oriented according to the direction of data circulation. In such a graph, we will use the standard notion of "strongly

[0] We could have let the initial state be a parameter of the machine. The choice of 0^n is for simplicity.

connected components[2]" (SCC for short): In Fig. 2, the two SCCs of the network are shown in dashed boxes. Let us recall that SCCs can be determined in linear time [Tar72].

Finally, we will make an extensive use of the following representation of memories by two-state automata: let x be a state variable, whose evolution is defined by the equation "$x' = f_x$". Let us consider the Shannon expansion of the Boolean function f_x according to x: $f_x = \overline{x}.f_x^0 + x.f_x^1$, where f_x^0 and f_x^1 are independent of x. Then f_x^0 (resp., $\overline{f_x^1}$) is the condition that sets the memory x from 0 to 1 (resp., that resets it from 1 to 0); we will note it set_x (resp. $reset_x$). As a consequence, each memory x has a canonical representation as the opposite two-state automaton.



For instance, in Example 1, the state variable z is defined by the equation $z' = \overline{y}\,(\overline{x} + z)$. Its Shannon expansion according to z is $z' = \overline{z}\,\overline{y}\,\overline{x} + z\,\overline{y}$, so $set_z = \overline{y}\,\overline{x}$ and $reset_z = y$.

3 Stability

We formalize, now, the notion of stability we want to analyze: an infinite sequence $\mathcal{I} = (\iota_0, \iota_1, \ldots, \iota_i, \ldots)$ is said to be *ultimately stable* if it is constant from some term, i.e., if

$$\exists i \in \mathbb{N} \text{ such that } \forall j \geq i, \iota_j = \iota_i$$

A machine is said to be *weakly stable* if and only if its image (sequence of outputs) on every ultimately stable input sequence is ultimately stable.

First notice that this property is neither trivial to specify, nor to verify. As a temporal logic formula [Pnu77], it would be written like

$$\Box(\Box(\iota \equiv \bigcirc\iota) \Rightarrow \Diamond\Box(\omega \equiv \bigcirc\omega))$$

and verifying it by model-checking [VW86] is likely to be very expensive.

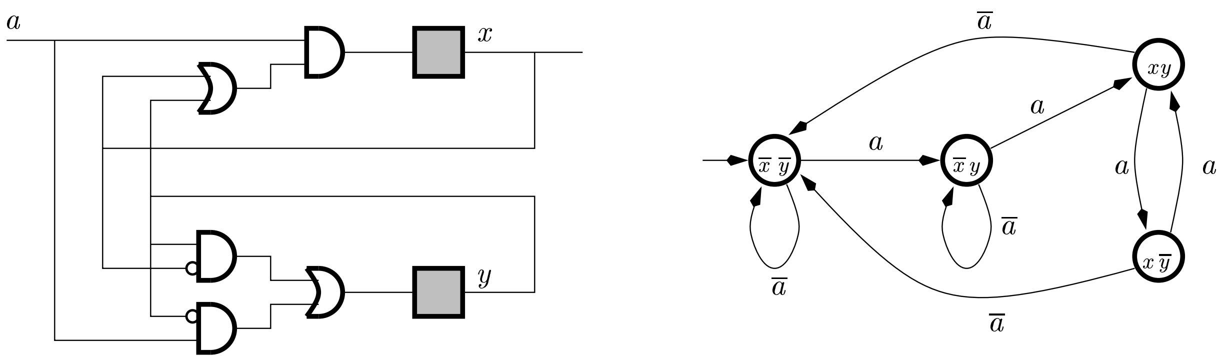
Weak stability is the property we are interested in, in view of the arguments given in the Introduction. In fact, we will consider a stronger property, because it is easier to check, and because experience shows that, on actual applications, it is generally very close to the wanted property. This stronger property states that, in presence of ultimately stable inputs, the *state* stabilizes in a finite number of steps:

A machine is said to be *strongly stable* (or simply *stable*) if and only if its run (sequence of states) on every ultimately stable input sequence is ultimately stable. Obviously, strong stability implies weak stability.

Example 2: Fig. 3.a shows the network of a machine with 1 input a, 2 state variables x, y, and one output x, defined by the system of equations

$$x' = a(x + y) \qquad y' = a\overline{y} + \overline{x}\,y$$

[2]i.e., a subset of operators, each pair of which is connected by a directed path.



(a) (b)

Figure 3: A simple machine and its state graph

Fig. 3.b shows the graph of reachable states of the machine. Possible unstabilities appear on this graph as circuits: apart from self-loops (which cannot correspond to unstabilities), the graph contains 3 elementary circuits:

(1) $(\overline{x}\,\overline{y}) \xrightarrow{a} (\overline{x}\,y) \xrightarrow{a} (xy) \xrightarrow{\overline{a}} (\overline{x}\,\overline{y})$
(2) $(\overline{x}\,\overline{y}) \xrightarrow{a} (\overline{x}\,y) \xrightarrow{a} (xy) \xrightarrow{a} (x\,\overline{y}) \xrightarrow{\overline{a}} (\overline{x}\,\overline{y})$
(3) $(xy) \xrightarrow{a} (x\,\overline{y}) \xrightarrow{a} (xy)$

Circuits (1) and (2) don't correspond to unstabilities, since their traversal involves input changes. Now, circuit (3) corresponds to an oscillation when the input a remains true. This oscillation does not change the output x but makes the state variable y oscillate. As a consequence, the machine is weakly stable, but not strongly stable.

This example shows that both weak and strong stability can be checked on the state graph of a machine, by examining its elementary circuits. The *cycle condition* of a circuit is the conjunction of the input conditions of its transitions. If we define an *unstable circuit* to be such that its cycle condition is feasible, then

- the machine is strongly stable if and only if its state graph does not contain any unstable circuit;

- the machine is weakly stable if and only if its outputs don't change along each unstable circuit of its state graph.

Now, this method is clearly expensive, and unfeasible for complex systems. This is why we investigate, now, heuristic methods to avoid the construction of the state graph. In the current state of our research, these heuristics only apply to strong stability. Let us stress out that, in the rest of the paper, we will never consider state graphs and their circuits, but only the operators networks and their strongly connected components!

4 Strongly Connected Components

A very first (obvious) remark is that a memory cannot oscillate if its input is stable; so, a memory can only cause an unstability if it appears in a loop of the network. An important consequence is that we can consider each strongly connected component of the network separately:

- An SCC of the network is *stable*, if, considered alone, it represents a stable machine. A sufficient condition for a machine to be stable is that all the SCCs of its network be stable.
- A memory that doesn't belong to an SCC cannot introduce unstability.

Another simple case concerns SCCs that contain only one memory: let this memory be associated with a state variable x, defined by the equation "$x' = f_x(x, y_1, \ldots, y_k)$", where the y_i are either inputs or state variables from other SCCs. The function f_x is said to be *monotonic* in x, if, for each valuation of $y_1, \ldots, y_k$, the implication $f_x(0, y_1, \ldots, y_k) \Rightarrow f_x(1, y_1, \ldots, y_k)$ holds.

Condition 1: If an SCC contains only one memory x, and if f_x is monotonic, the considered SCC is stable.

For instance, in our Example 1 (Fig. 2), the network has two SCCs: the first contains only the memory x, and the second contains both y and z. Then, since the function defining x, $\overline{b}\,(a + x)$, is monotonic in x, the former SCC is stable, and we only have to analyze the second SCC, considering x as its only input.

5 Local Cycle Conditions

Our second criterion looks very weak at first glance, but provides surprising good results [HL97]: for a memory x to oscillate, it must be able to change with the same values for input variables. More formally, there should exist two instants t_1 and t_2 at which the inputs are the same, and such that set_x holds at t_1 and $reset_x$ holds at t_2. If $\vec{\iota}$ (resp. $\vec{s}$) represents the vector of input (resp., state) variables, if $\vec{s_1}$ and $\vec{s_2}$ represents the values of state variables at t_1 and t_2, respectively, this condition can be written

$$\exists \vec{\iota}, \vec{s_1}, \vec{s_2},\ set_x(\vec{\iota}, \vec{s_1}).reset_x(\vec{\iota}, \vec{s_2}) = 1$$

Moreover, state variables that have already been found stable can be considered (as inputs) to have the same values in $\vec{s_1}$ and $\vec{s_2}$. So, in general, we will have a set σ of stable variables (including inputs) and a set ξ of remaining variables, and, for each $x \in \xi$, the considered condition will be

$$\exists \vec{\sigma}, \vec{\xi_1}, \vec{\xi_2},\ set_x(\vec{\sigma}, \vec{\xi_1}).reset_x(\vec{\sigma}, \vec{\xi_2}) = 1$$

Let us note $\mathcal{U}_x^\sigma$ the condition $\exists \vec{\xi_1}, \vec{\xi_2},\ set_x(\vec{\sigma}, \vec{\xi_1}).reset_x(\vec{\sigma}, \vec{\xi_2})$ and call it the *local cycle condition* for x in σ.

Condition 2: If $\vec{\sigma}$ is a set of inputs or stable variables, the unsatisfiability of $\mathcal{U}_x^\sigma$ is a sufficient condition for x to be stable.

This provides us with an algorithm to analyze an SCC of the network: let ι and s be, respectively, the set of input and of state variables of the SCC. The following algorithm returns in σ (resp., in xi) the set of memories that have been found stable (resp., the stability of which is not garanteed):

Algorithm 1

start with $\sigma = \iota$; $\xi = s$;
while $\exists x \in \xi$ such that $\mathcal{U}_x^\sigma \equiv 0$ do
  $\xi := \xi \setminus \{x\}$; $\sigma := \sigma \cup \{x\}$;
end while

Example 3: Let us consider an SCC with 2 inputs a and b, and two state variables x and y, defined by

$$x' = \overline{y} + \overline{a} \qquad y' = \overline{b}\,(x + y)$$

We have $set_x = \overline{y} + \overline{a}$, $reset_x = y.a$, $set_y = x.\overline{b}$, $reset_y = b$
Starting the algorithm with $\sigma = \{a, b\}, \xi = \{x, y\}$, we get

$$\begin{aligned}
\mathcal{U}_x^\sigma &= \exists y_1, y_2,\ set_x(a, b, y_1).reset_x(a, b, y_2) \\
&= \exists y_1, y_2,\ (\overline{y_1} + \overline{a})(y_2.a) \\
&= \exists y_1, y_2,\ \overline{y_1}.y_2.a \\
&= a \\
&\not\equiv 0 \quad \text{(so, nothing can be concluded for } x) \\
\mathcal{U}_y^\sigma &= \exists x_1, x_2\ set_y(a, b, x_1).reset_y(a, b, x_2) \\
&= \exists x_1, x_2\ (x_1.\overline{b}).b \\
&= 0
\end{aligned}$$

So, y is stable, and we iterate with $\sigma = \{a, b, y\}$ and $\xi = \{x\}$. We get

$$\begin{aligned}
\mathcal{U}_x^\sigma &= set_x(a, b, y).reset_x(a, b, y) \\
&= (\overline{y} + \overline{a}).(y.a) \\
&= 0
\end{aligned}$$

and x is found stable too.

A more precise condition can be found, by considering *pairs* of depending state variables in the same SCC. Let $\mathcal{D}(x)$ be the set of state variables that belong to the same SCC as x, and appear either in set_x or in $reset_x$. Assume the previous algorithm failed to show the stability of two variables x, y, with $y \in \mathcal{D}(x)$. We already know that, when $\mathcal{U}_y^\sigma$ is false, y is stable. So we can split the definition of $\mathcal{U}_x^\sigma$ into two cases:

- the case where $\mathcal{U}_y^\sigma$ holds
- the case where $\mathcal{U}_y^\sigma = 0$, in which case the value of y can be assumed to be stable (i.e., $y_1 = y_2$).

This leads to a stronger condition of unstability for x:

$$\mathcal{U}_{x,y}^\sigma = \exists \vec{\xi_1}, \vec{\xi_2},\ (\mathcal{U}_y^\sigma.set_x(\vec{\sigma}, \vec{\xi_1}).reset_x(\vec{\sigma}, \vec{\xi_2})) + (\overline{\mathcal{U}_y^\sigma}.set_x(\vec{\sigma}, \vec{\xi_1}).reset_x(\vec{\sigma}, \vec{\xi_2}).(y_1 = y_2))$$

Condition 3: If $\vec{\sigma}$ is a set of inputs or stable variables, if $y \in \mathcal{D}(x)$, the unsatisfiability of $\mathcal{U}_{x,y}^\sigma$ is a sufficient condition for x to be stable.

So, when Algorithm 1 fails to show that all memories of an SCC are stable (i.e., returns $\xi \neq \emptyset$), we can apply the following one:

Algorithm 2

while $\exists x \in \xi$ such that $\exists y \in \mathcal{D}(x)$ such that $\mathcal{U}^{\sigma}_{x,y} \equiv 0$ do
$\quad \xi := \xi \setminus \{x\};\ \sigma := \sigma \cup \{x\};$
end while

Of course, this approach could be continued with 3 variables and more, but would become more and more expensive.

Example: Let's come back to Example 1. In Section 4, we showed that the first SCC of the network — reduced to the state variable x —, is stable. Now, if we apply Algorithm 1 to the second SCC, we get:

$$\sigma = \{x\}\ ,\ \xi = \{y, z\}$$
$$\mathcal{U}^{\sigma}_{y} \;=\; \exists z_1, z_2,\ x.\overline{z1}.z2 \;=\; x$$
$$\mathcal{U}^{\sigma}_{z} \;=\; \exists y_1, y_2,\ \overline{x}.\overline{y_1}.y_2 \;=\; \overline{x}$$

So, nothing can be deduced about the stability of y and z. Now, algorithm 2 provides:

$$\mathcal{U}^{\sigma}_{y,z} \;=\; \overline{x}.x + x.(\exists z,\ x.\overline{z}.z) \;=\; 0$$
$$\mathcal{U}^{\sigma}_{z,y} \;=\; x.\overline{x} + \overline{x}.(\exists y,\ \overline{x}.\overline{y}.y) \;=\; 0$$

from which we can conclude that y and z are stable.

6 Approximation of Stable Values

When the previous heuristics fail to show the stability of an SCC, the SCC contains variables with feasible cycle conditions: they are formulas involving input variables of the SCC, some of which come from other SCCs. We can try to show that no stable values of these input variables can make the cycle condition feasible. Now, we don't know the possible stable values of state variables coming from other SCCs. Here, we propose [HL97] to use an approximation of these values: as a matter of fact, for each state variable x, we have the following information about its stable values:

When x stabilizes to 1, surely $reset_x$ is false
When x stabilizes to 0, surely set_x is false

In other words, the stable values $\hat{x}$ of x satisfy:

$$set_x \Rightarrow \hat{x} \Rightarrow \overline{reset_x}$$

So, if, in a cycle condition $\mathcal{U}$, we replace each occurrence of x (resp., of $\overline{x}$) by $\overline{reset_x}$ (resp., by $\overline{set_x}$) we get a new condition $\mathcal{U}'$ which is weaker than $\mathcal{U}$ ($\mathcal{U} \Rightarrow \mathcal{U}'$). If $\mathcal{U}'$ happens to be identically false, so is $\mathcal{U}$, and the considered cycle cannot introduce unstability. Of course, this can be tried with all state variables appearing in $\mathcal{U}$, or with combinations of these variables.

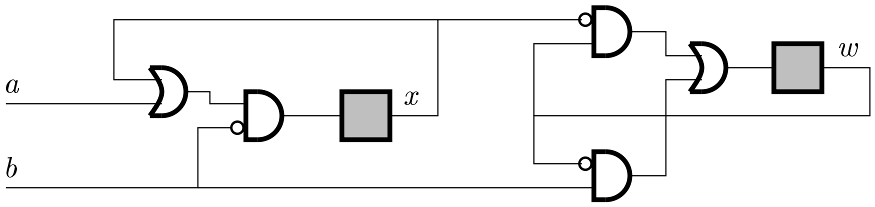


Figure 4: The network of Example 4

Example 4: Let us consider the system represented in Fig. 4, whose system of equations is: $x' = \overline{b}\,(a+x)$, $w' = b.\overline{w} + \overline{x}.w$. There are two SCCs, one containing x and the other containing w. The first one is found stable, as in Section 4. In the second SCC, none of our previous heuristics works:

- $f_w = b.\overline{w} + \overline{x}.w$ is not monotonic in w
- $\mathcal{U}_w^{\{a,b,x\}} = b.x \not\equiv 0$
- $\mathcal{D}(w) = \emptyset$

Now the local cycle condition for w in $\{a,b,x\}$ is $\mathcal{U}_w^{\{a,b,x\}} = b.x$. Since $reset_x = b$, we get $\mathcal{U}' = b.\overline{b}$, which is identically false. This shows that w is stable.

7 Application and Future Work

From an industrial point of view, stability is by no means a theoretical problem.

The first reason is that oscillating Booleans are frequently observed in nuclear power plants. This may result from various causes, including hysteresis phenomena concerning analog signals, or other physical or technological reasons; but there has been evidence of such oscillations due to the functional specification level itself. In these cases, formal analysis would have detected the problem before it actually occured.

The second implication of instability concerns distributed command: as many industrial systems are implemented on several processors (and this is obviously the case in power plants), stability often appears as a necessary condition for the global validity of the command system. Therefore, it may be useful to predict both stability itself, and the maximum delay within which it is reached.

Being aware of these reasons amongst others, the French nuclear safety authority (IPSN) is keeping a close look on formal methods, even if it has not *formally* recommended them up to now. Anticipating the recommendation, the French utility Electricité de France (EDF) has already undertaken an R&D program on the subject; it is interesting to point out that Boolean stability was the first item to be studied.

The techniques proposed in this paper have been experimented on actual systems (control software for nuclear plants). All the presented examples come from this actual software, and were considered to be problematic before being analyzed. These experiments show that the proposed heuristics give precise results, in practice: almost all the cases which were not found stable presented actual unstability problems. These experiments were performed by hand, but the feasibility of such a manual application shows that its cost has nothing to do with that of exact verification, a point which is highly confirmed by the first experiments with the implemented prototype.

These results are so encouraging that an actual industrial tool for checking stability is under implementation. The tool takes system descriptions written in LUSTRE [HCRP91], or by means of "logical functional diagrams" used at EDF. All the Boolean conditions are dealt with using BDDs [Ake78, Bry86].

Stability should be a very interesting concept in synchronous programming [Hal93]. In fact, the stable state graph is an abstraction very similar — on a macroscopic level — to the synchronous abstraction, that considers as atomic all the "micro-steps" performed in the same reaction. From this point of view, stability is the macroscopic counterpart of the "causality" [SBT96, HM95] property, considered in synchronous programs. An interesting perspective would be to adapt the techniques proposed here to obtain a cheap causality checker for synchronous programs and circuits.

The notion of stability can also be very fruitful in verification. For the kind of systems we consider, critical properties are very likely to be required to hold only in stable states. In this case, only the reduced graph of stable states has to be examined, and this can dramatically reduce the cost of verification by model-checking. For instance, in a machine with n inputs, if all the inputs are sampled in a memory before being used, the number of states is multiplied by 2^n with respect to the number of stable states.

References

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HL97] J.-F. Héry and J.-C. Laleuf. Stabilité de la réalisation des DFL. Technical Report Electricité de France, 1997.

[HM95] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), September 1995.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Symp. on the Foundations of Computer Science*, Providence R.I., 1977. IEEE.

[SBT96] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science*, June 1986.