

L I F C

LABORATOIRE D'INFORMATIQUE DE L'UNIVERSITE DE FRANCHE-COMTE

EA 4269

Formal convergence proof for discrete dynamical systems

Jean-François Couchot

Rapport de Recherche no RR 2010–3

THÈME 3 – 4 May 2010



Formal convergence proof for discrete dynamical systems

Jean-François Couchot

Thème 3

AND

4 May 2010

Abstract: This work shows how to automatically prove convergence properties of discrete dynamical network for any asynchronous iterations with bounded delays. The approach is based on temporal properties verification through model checking. Both theoretical and practical aspects are addressed: for the former, the approach is proved to be correct and complete; for the latter, complexity issues and experiments on non trivial examples show the efficiency of the approach. The whole approach is applied on a running example.

Key-words: Discrete dynamical systems, Asynchronous iterations, Pure parallel iterations, Connection graph, Stability proof, Model Checking

Preuve formelle de convergence de systèmes dynamiques discrets

Résumé : Ce travail montre comment obtenir automatiquement la preuve qu'un système dynamique discret converge lorsqu'il est itéré de manière asynchrone avec des délais bornés. L'approche est basée sur la vérification de propriétés temporelles à l'aide de model checker. Les aspects théoriques et pratiques sont étudiés : en premier lieu, l'approche est prouvée comme étant correcte et complète; en second lieu, la complexité est étudiée; des expérimentations sur des exemples non triviaux sont menées, montrant ainsi l'efficacité du tout. Les points clés du travail sont illustrés à l'aide d'un exemple fil rouge.

Mots-clés : systèmes dynamiques discrets, iterations (a)synchrones, preuve de convergence graphe de connexion, model checking

Formal convergence proof for discrete dynamical systems

Jean-François COUCHOT

LIFC, EA 4269, University of Franche-Comté
IUT Belfort-Montbéliard, France

Abstract. This work shows how to automatically prove convergence properties of discrete dynamical network for any asynchronous iterations with bounded delays. The approach is based on temporal properties verification through model checking. Both theoretical and practical aspects are addressed: for the former, the approach is proved to be correct and complete; for the latter, complexity issues and experiments on non trivial examples show the efficiency of the approach. The whole approach is applied on a running example.

1 Introduction

A dynamical network is a system whose evolution only depends on its history. A global state of that kind of network is called a *configuration*. This work focuses on *discrete* dynamical network (later denoted as DDN): configurations are measured at discrete times ($t = 0, 1, 2, \dots$) and each configuration takes its values into a product of finite domains. DDNs have many applications, for instance, in genetic networks [1,2], in neural networks [3,4].

There are basically two ways of making DDNs evolve: synchronous and asynchronous modes. If a DDN is iterated synchronously, the successor of a configuration is computed by applying a map on a partition of the elements (following a given strategy) keeping the others unchanged. Asynchronous iterations are a generalization of synchronous ones that allows the existence of a delay from the instant where a component has a specific value and the moment where another component can use this value.

In that context, one should ensure that systems have some basic properties. For instance, a system that computes a result has to self stabilize in a finite time. This is identified as a *convergence*, contrary to the *divergence* in which the system may enter in a cycle or have chaotic behavior.

Establishing a convergence proof of DDNs may be quite difficult: in the better case, one only has to ensure that DDN fulfil hypotheses of a previously established convergence theorem [3,5]. In the worst case, one has to find a fine metric and to prove that it strictly decreases, what is an error prone task due to the number of (sub) cases that may have to be studied.

However, a tool that exhaustively computes all the configurations following a given iteration mode is an answer to this proof need. Provided correction and

completeness of the translation to that tool are addressed, convergence proof only rely on the answer returned by the tool. Nevertheless, even for DDNs with a few number of elements, the configuration number associated with delays handling and choosing elements to iterate quickly becomes to huge to be memorized with usual data structures.

Model-Checkers is a class of tools that address the problem of automatically checking whether a model meets a given property. To address the combinatorial blow up of the state-space, state of the art model checkers apply sound methods like partial order reduction, abstraction techniques, when this is not sufficient. They have been already applied on genetic networks [6,7,8] expressed as specific DDN, but none of these studies deals with delays of asynchronous mode.

This work presents how to efficiently simulate a DDN on a model checker taking delays into account with the objective to formally establish convergence proofs. The first contribution of this work is the translation of DDNs (system and iteration modes) into PROMELA which is the model language of the state of the art SPIN [9,10] model checker (Sect. 4). The second contribution is the correction and completeness proof of the translation for convergence property (Sect. 5). The third contribution is the practical issues of this work where complexity problems and experiments are addressed (Sect. 6). Finally, to make the document self-contained, the Section 2 is dedicated to the formalisation of DDN and introduces the running example, and basis of PROMELA are recalled in Sect. 3. Concluding remarks and perspectives are given in Sect. 7

2 Formalisation

This section recalls definitions of DDNs and presents a running example.

2.1 Discrete Time Discrete States dynamical Networks

This section formalizes discrete dynamical networks sketched in introduction.

A DDN is a collection of n components. Each component i , $1 \leq i \leq n$, takes its value X_i among a finite domain E_i . Let E be the space product $E = \prod_{i=1}^n E_i$.

A configuration of the network at discrete time t (also called at iteration t) is the vector

$$X^{(t)} = (X_1^{(t)}, \dots, X_n^{(t)}) \in E.$$

The dynamic of the system is described according to a function $F : E \rightarrow E$ such that

$$F(X) = (F_1(X), \dots, F_n(X)).$$

In the sequel, the *strategy* $(J^{(t)})^{t \in \mathbb{N}}$ is the sequence of characteristic function of components that may be updated at time t .

Practically, it is represented as a $n \times n$ diagonal matrix such that $J_{ii}^{(t)} = 1$ if and only if it is allowed to modify $X_i \in E_i$ at time t . Moreover, the strategy $(J^{(t)})^{t \in \mathbb{N}}$ is *pseudo-periodic* if for each component i , the set $\{t \mid J_{ii}^{(t)} = 1\}$ is infinite.

LIFC

Let I be the identity matrix of size n , and $X^0 = (X_1^0, \dots, X_n^0)$ an initial configuration. The synchronous iterations modes (including pure parallel mode, sequential mode, chaotic modes) are defined for times $t = 0, 1, 2, \dots$ by:

$$X^{(t+1)} = (I - J^{(t)})X^{(t)} + J^{(t)}F(X^{(t)}) \quad (1)$$

Indeed:

- pure parallel iterations constrain $J^{(t)}$ to be equal to the identity matrix for any t ;
- sequential iterations constrain $J_{ii}^{(t)}$ to be null, except for i equal to $1 + t \bmod n$, where it is 1;
- chaotic iterations do not constrain $(J^{(t)})^{t \in \mathbb{N}}$.

We then formalize the fully asynchronous mode with overlapping updates [11,3]:

- components may be updated in a random order, and even not updated at all, according to a strategy $(J^{(t)})^{t \in \mathbb{N}}$, as in the synchronous chaotic mode;
- at each iteration t , each component may update its own state according to the last values it has received from the other components at the current time. These values depend on computation and communication delays between these components.

Let $(S^{(t)})^{t \in \mathbb{N}}$ be the sequence of $n \times n$ matrices whose element $S_{ij}^{(t)}$ represents the iteration numbers (less or equal to t) at which the value $X_j \in E_j$, available at time t on component i , has been produced on component j . This sequence is further referred as sequence of last available dates. We suppose that $(S_{ij}^{(t)})$ is increasing for each i and j . Hence, delay between the data sending from j and the data reception on i is defined by $\delta_{ij}^t = t - S_{ij}^{(t)} \geq 0$. Notice that delays are supposed to be bounded by a constant δ_0 . More formally, we have

$$\exists_{\mathbb{N}} \delta_0 . (\forall_{\mathbb{N}} t . (\forall_{[1, \dots, n]} i, j . \delta_{ij}^t \leq \delta_0)) \quad (2)$$

Similarly to equation (1), asynchronous iteration mode is defined by:

$$X^{(t+1)} = (I - J^{(t)})X^{(t)} + J^{(t)} \begin{pmatrix} F_1 \left(X_1^{(S_{11}^{(t)})}, \dots, X_n^{(S_{1n}^{(t)})} \right) \\ \vdots \\ F_n \left(X_1^{(S_{n1}^{(t)})}, \dots, X_n^{(S_{nn}^{(t)})} \right) \end{pmatrix} \quad (3)$$

Notice that if $S^{(t)}$ is the matrix identically equal to t ($S_{ij}^{(t)} = t$, for any $1 \leq i \leq j \leq n$, i.e. all the delays are null), asynchronous iterations are equivalent to chaotic iterations.

2.2 Running Example

We consider the same running example as [12], i.e. five elements taking their value in the set $\{0, 1\}$ following the map:

$$F(X) = \begin{cases} f_1(X_1, X_2, X_3, X_4, X_5) = X_1 \bar{X}_2 + \bar{X}_1 X_2 \\ f_2(X_1, X_2, X_3, X_4, X_5) = \bar{X}_1 + X_2 \\ f_3(X_1, X_2, X_3, X_4, X_5) = X_3 \bar{X}_1 \\ f_4(X_1, X_2, X_3, X_4, X_5) = X_5 \\ f_5(X_1, X_2, X_3, X_4, X_5) = \bar{X}_3 + X_4 \end{cases}$$

where \bar{a} , sum and product are the usual boolean operators.

The Figure 1 gives the induced incidence graph i.e. the graph whose vertices are elements and whose edges from i to j represent that f_j depends on i .

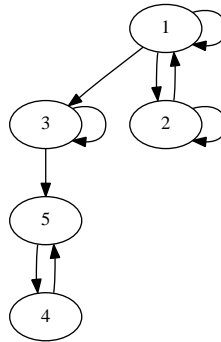


Fig. 1. Incidence graph of running example

The prototype developed for [12] have shown that pure parallel iterations converge. It has helped us to find pseudo-periodic strategies making chaotic iterations (and then asynchronous iterations) diverging.

3 Process Meta Language (PROMELA)

This section restricts to fundamentals of the Process Meta Language (PROMELA) that are in our concern. More details can be found in [10,9,13].

Built-in types considered by PROMELA are bool, byte, short and int. As in the C language, we can declare one-dimensional constant length arrays or user defined data types (introduced by the `typedef` keyword). These latter types are widely used to define two-dimensional arrays.

Running example. The listing given in Fig. 2 corresponds to variable declarations that result from the translation of the running example. It defines:

LIFC

- the constant `N` and `d_0` respectively standing for the number n of elements and the delay bound δ_0 ;
- two arrays (`X` and `Xp`) of five boolean variables; the cells `X[i]` and `Xp[i]` are associated to the variable X_i of the DDDN; they memorize the value of X_i respectively before and after an update;
- the array `sync` indexed by elements and whose values `sync[i]` and `sync[j]` are equal if there is no delay between i and j ;
- the array `mods` of elements that have to be modified at the current iteration; intuitively, it corresponds to the modified elements identified by the strategy.
- the structured data type `vals` and the array of arrays `Xs[j - 1].v[k - 1]` aims at storing $X_k^{S_{jk}^t}$ at iteration of time t ;
- the two-dimensional array named `array_of_channels` of $N*N$ elements of type `chan` (see below).

```
#define N 5
#define d_0 5

bool X [N]; bool Xp [N]; int sync [N]; int mods [N];
typedef vals{bool v [N]};
vals Xs [N];

typedef a_send{chan sent [N] = [d_0] of {bool}};
a_send array_of_channels [N];

chan unlock_elements_update = [1] of {bool};
chan sync_mutex = [1] of {bool};
```

Fig. 2. Example of PROMELA arrays and channels in use in the DDNs translation

A channel allows for transferring messages between processes in first-in first-out order. It is declared by the keyword `chan` following by its capacity (constant), its name and the structure of the messages stored in the channel. In the previous example, we successively declare:

- a channel `sent` that may store `d_0` messages of type `bool`;
- the two channels `unlock_elements_update` and `sync_mutex` of one `bool` message further used as semaphores.

PROMELA embeds the *process* notion that allows for modelling both concurrency and system distribution. A process is declared with the `proctype` keyword and is instantiated either immediately (when its declaration is prefixed by the `active` keyword) or whilst executing the run operator. Among processes, the `init` one is the initial process that aims at initializing variables, launching other processes, ...

Notice that variables may be local or global. The scope is the process in the former case, whereas it is the whole program in the latter case.

Assignment statements are interpreted as in the C language, starting by evaluating expression and next by modifying the variable value. Two statements concern any channel `ch`: receiving and sending a message `m`; this is respectively represented by `ch ? m` and `ch ! m`. The receive statement consumes the value in the head of the channel `ch`, and assigns it to the variable `m` (provided `ch` is initialized and not empty). Similarly the send statement appends the value of `m` in the tail of `ch`, provided it is initialized and not full. In both cases, the process is locked in this statement until the two both conditions are established.

The `if` (resp. the `do`) control-flow construct is a non-deterministic guarded choice (respectively a non-deterministic guarded loop). For both choice and loop, if more than one guard is enabled, one of them will be selected non-deterministically and executed.

In the `init` process detailed in Fig. 3, we first initialize the `sync` array in the `define_sync()` function not detailed here: intuitively, there is no delay in value transmission for elements i and j s.t. `sync[i]` and `sync[j]` are equal. This constant array allows to simulate synchronous iterations (when `sync[i]` is equal to 0 for all elements i) asynchronous iterations (when `sync[i]` is equal to i for all elements i) or mixed one.

Next a loop of length N non-deterministically initializes the global array variable `Xp`. For each element i , if iterations are asynchronous,

- we first memorize the value of `Xp[i]` into each `Xs[j].v[i]` since the matrix S^0 is identically equal to (0)
- next, the value of i (represented by `Xp[i]`) should be transmitted to j if there is an edge from i to j in the incidence graph. In that case, the function `has_next` (not precised here) memorizes this graph and sets to true the variable `is_succ`. It allows to send the value of i . into the channel `array_of_channels[i].sent[j]`

4 Translating a Discrete Dynamical Network into a PROMELA Model

As seen above, variable types considered by PROMELA are `bool`, `byte`, `short` or `int`. They are convenient to memorize element values of DDNs that range over finite domains.

4.1 Strategy

We first show how to translate a finite strategy, and then we present the extension to periodic strategy and pseudo-periodic strategy.

A finite strategy is composed of a finite sequence of matrices $J = [J^0; \dots; J^l]$ representing elements updated at a given time. Scheduling elements according to J is achieved by the `scheduler` process as shown in Fig. 4. This process is a

LIFC

```

init{
  atomic{
    define_sync ();
    int i; i=0;
    do
      :: i == N -> break;
      :: i < N -> {
        if
          :: Xp[i] =0;
          :: Xp[i] =1;
        fi;
        j=0;
        do
          :: j == N -> break;
          :: j < N -> Xs[j].v[i] = Xp[i];
            j++;
        od;
        if
          :: iter_mode == ASYNC ->
            int j; j=0;
            bool is_succ=0;
            do
              :: j == N -> break;
              :: j < N ->{
                hasNext(i,j);
                if
                  :: (sync[i] != sync[j] && is_succ ==1) ->
                    array_of_channels[i].sent[j] ! Xp[i];
                  :: (sync[i] == sync[j] || is_succ ==0) ->
                    skip
                fi;
                j++;
              }
            od;
          :: iter_mode != ASYNC -> skip
        fi;
        i++;
      }
    od;
    sync_mutex ! 1 ;
  }
  run scheduler() ;
}

```

Fig. 3. PROMELA init process

sequence of unlocks of the `elements_updates` process through the channel. To

achieve this we first set into `ar_len` the number of elements to be modified and next we set into `mods` the index of elements that are going to evolve. The mutual exclusion of `elements_updates` execution is addressed by the semaphores `unlock_elements_updates` and `sync_mutex`. In Fig. 4, we first ask for the update of the two elements 3 and 4 and next ask for the update of the three elements 0, 1 and 2.

```

proctype scheduler(){
  sync_mutex ? 1 -> atomic{
    ar_len=2;
    mods[0] = 3;mods[1] = 4;
    unlock_elements_update ! 1
  };
  sync_mutex ? 1 -> atomic{
    ar_len=3;
    mods[0] = 0;mods[1] = 1;mods[2] = 2;
    unlock_elements_update ! 1
  }
}

```

Fig. 4. Scheduler process for finite strategy $[\{3, 4\}, \{0, 1, 2\}]$

We are left to translate other strategies. A periodic strategy is obviously translated into a `do ... od` loop whose single entry is the translation of the inner strategy as above.

Finally, an execution with a pseudo-periodic strategy is translated into a `do ... od` as before but with 2^{n-1} entries. Each of them corresponds to the translation of the parallel update of a non empty set of elements in $\{1, \dots, n\}$ as above.

4.2 Update of a Set of Elements

Updating a set $J^t = \{j_1, \dots, j_m\}$ of elements that occur in the strategy $(J^t)^{t \in \mathbb{N}}$ is implemented by the `elements_updates` process given in Fig. 5. This active process waits until it is unlocked by the `scheduler` process through the semaphore `unlock_elements_update`. The implementation is then fivefold:

1. it starts with updating the variable `X` with the values of `Xp` thanks to the `update_X` function (not detailed here);
2. it memorizes into `Xs` the current known values of elements thanks to the `retrieve_elements_values` function (see Sect. 4.3);
3. a loop over the number `ar_len` of elements that have to evolve iteratively updates the value of `j` (through the execution of the `F()` function) provided this has to evolve, i.e. it is referenced by `mods[count]`; source code of `F` is given in Fig. 6 and is a direct translation of the map F ;

LIFC

4. new values Xp are symbolically sent to other elements for future access thanks to the `broadcast(Xp)` function (see Sect. 4.3);
5. finally, this process informs the scheduler about the end of the task (through the semaphore `sync_mutex`).

```

active proctype elements_update(){
  do
    :: unlock_elements_update ? 1 -> {
      atomic{
        bool is_succ=0;
        update_X ();
        retrieve_elements_values ();
        int count = 0;
        int j = 0;
        do
          :: count == ar_len -> break ;
          :: count < ar_len ->
            j = mods[count];
            F();
            count++;
        od;
        broadcast(Xp);
        sync_mutex ! 1
      }
    }
  od
}

```

Fig. 5. Translation of update of element set

4.3 Values Transmission

Synchronous and asynchronous modes differ in the value transition delays: in the former case, all delays between elements are null whereas in the latter case, they are bounded by a finite constant. Furthermore, the mixed iteration mode [12] allows to constraint some of the elements to evolve synchronously without constraining the other ones.

Functions `retrieve_elements_values` and `broadcast` given in Fig. 7 and Fig. 8 respectively memorize and transmit the element values. They are developed to handle both synchronous and asynchronous iterations.

The former possibly updates the variable Xs needed by elements that have to be modified. For each one (i.e. for each element in `mods`) that is represented by the variable j , it retrieves the values of other elements (labeled by i). There are two cases:

```

inline F(){
  if
    :: j==0 -> Xp[0] = ( Xs[j].v[0] & !Xs[j].v[1]) |
                    (!Xs[j].v[0] & Xs[j].v[1])
    :: j==1 -> Xp[1] = !(Xs[j].v[0] | Xs[j].v[1])
    :: j==2 -> Xp[2] = Xs[j].v[2] & ! Xs[j].v[0]
    :: j==3 -> Xp[3] = Xs[j].v[4]
    :: j==4 -> Xp[4] = (!Xs[j].v[2] | Xs[j].v[3])
  fi
}

```

Fig. 6. Translation of the function F

```

inline retrieve_elements_values(){
  int countv = 0 ;
  do
    :: countv == ar_len -> break ;
    :: countv < ar_len ->
      j = mods[countv];
      i = 0;
      do
        :: (i == N) -> break;
        :: (i < N && sync[i] == sync[j] ) -> {
          Xs[j].v[i]= Xp[i] ;
          i++}
        :: (i < N && sync[i] != sync[j] ) -> {
          hasnext(i,j);
          if
            :: skip
            :: is_succ==1 && nempty(array_of_channels[i].sent[j]) ->
              array_of_channels[i].sent[j] ? Xs[j].v[i] ;
          fi ;
          i++}
      od;
      countv++
    od
}

```

Fig. 7. The `retrieve_elements_values` function

- when iterations concerning the two nodes i and j have to be synchronized, (represented by `sync[i] == sync[j]`), the value of the element i that is available to j is directly the last computed value of X_i , i.e. `Xp[i]`;
- otherwise, there are two subcases that possibly update the value that j knows about i (that may be chosen in a non-deterministic way):
 - from the perspective of j the value of i may not change (the `skip` statement) or is not relevant; this latter case arises when there is no edge from i to j in the incidence graph, i.e. when the value of `is_succ` that is computed by `hasnext(i, j)` is 0; then the value of `Xs[j].v[i]` is not modified;
 - otherwise, `Xs[j].v[i]` is assigned with the value stored into the channel `array_of_channels[i].sent[j]` (provided this one is not empty). Element values are added into this channel during the `broadcast` function as follows.

The `broadcast` function aims at memorizing the values of X represented by `Xp` into the `array_of_channels`. It allows the SPIN model checker to execute the PROMELA model as if it allowed delays between processes. For that reason, when an iteration has to synchronize the elements i and j (i.e. when `sync[j] == sync[i]` is true), no delay emulation is performed. In the asynchronous mode, there are two cases concerning the value of X_i :

- either it is abstracted away to allow i not to take into account all the values of j ; this case occurs either through the `skip` statement or again when there is no edge from i to j in the incidence graph;
- or it is memorized into the channel `array_of_channels[j].sent[i]` (provided it is not full).

4.4 Convergence Property

We are left to show how to formalize into the SPIN model checker that a DDN with n elements has converged. We then have to translate that, starting from any configuration, the following equation is established :

$$\exists t_0. (\forall t. t > t_0 \Rightarrow \bigwedge_{1 \leq i \leq n} X_i^{t+1} = X_i^t).$$

Thanks to the variables `X` and `Xp` that respectively memorize the value of X before and after an update, it is necessary and sufficient to establish the following Linear Temporal Logic (LTL) formula:

$$\diamond(\square Xp = X) \tag{4}$$

where \diamond and \square have the usual meaning i.e. respectively eventually and always in the subsequent path.

```

inline broadcast(values){
  int countb; countb=0;
  do
    :: countb == ar_len -> break ;
    :: countb < ar_len ->
      j = mods[countb];
      i = 0 ;
      do
        :: (i == N) -> break;
        :: i < N && sync[i] == sync[j] -> {
          i++;}
        :: (i < N && sync[i] != sync[j]) -> {
          hasnext(j,i);
          if
            :: skip
            :: is_succ==1 && nfull(array_of_channels[j].sent[i]) ->
              array_of_channels[j].sent[i] ! values[j]
          fi ;
          i++;}
      od;
      countb++
    od
  }

```

Fig. 8. The broadcast function

4.5 Discussion

A coarse approach could consist in providing one process for each element. However, the distance with the mathematical model (equ. 3) of such a translation would be larger than the method presented along these lines. It induces it would be harder to prove the soundness and completeness of such a translation. For that reasons we have developed a PROMELA model that is as close as possible to the mathematical one.

5 Proof of Translation Correction

This section establishes the soundness and completeness of the approach (Theorems 1 and 2). Four technical lemmas are first shown to ease the proof of the two theorems.

Lemma 1 (Absence of deadlock) *Let ϕ be a DDN model and ψ be its translation. There is no deadlock in any execution of ψ .*

PROOF. In current translation, deadlocks of PROMELA may only be introduced through sending or receiving messages in channels. Sending (resp. receiving) a message in the `broadcast` (resp. `retrieve_elements_values`) function is executed only if the channel is not full (resp. is not empty). In the `elements_update` and `scheduler` processes, each time one adds a value in any semaphores channels (`unlock_elements_update` and `sync_mutex`), the corresponding value is read; avoiding deadlocks by the way.

Lemma 2 (Strategy Equivalence) *Let ϕ be a DDN model whose strategy is $(J^t)^{t \in \mathbb{N}}$ and ψ be its translation. There exists a SPIN execution with weak fairness s.t. the scheduler asks `elements_updates` to modify elements of J^t at iteration t .*

PROOF. The proof is obvious for $t = 0$. Now, let us suppose that it is established until t is some t_0 . If the strategy is finished or periodic, since the iteration t_0 does not lock SPIN (lemma 1), the result is established. If the strategy is only supposed to be infinite but pseudo-periodic, thanks to the weak fairness equity property, the `elements_updates` will be asked to modify elements of J^t at iteration t .

In what follows, let Xs_{ij}^t be the value of `Xs[j - 1].v[i - 1]` after the t th call to the function `retrieve_elements_value`. Furthermore, let Y_{ij}^k be the element at index k in the channel `array_of_channels[i].sent[j]` of size m , $m \leq \delta_0$; Y_{ij}^0 and Y_{ij}^{m-1} are respectively the head and the tail of the channel. Secondly, let $(M_{ij}^t)^{t \in \frac{\mathbb{N} - \{0\}}{2}}$ be a sequence such that M_{ij}^t is the partial function that associates to each k , $0 \leq k \leq m - 1$, the tuple $(Y_{ij}^k, a_{ij}^k, c_{ij}^k)$ while entering into the `elements_updates` a iteration t where:

- Y_{ij}^k is the value of the channel `array_of_channels[i].sent[j]` at index k ;

- a_{ij}^k is the date (previous to t) when Y_{ij}^k has been added;
- c_{ij}^k is the date (after t) it is consumed.

M_{ij}^t has the following signature:

$$M_{ij}^t : \{0, \dots, max - 1\} \rightarrow E_i \times \mathbb{N} \times \mathbb{N}$$

$$k \in \{0, \dots, m - 1\} \mapsto M_{ij}^t(k) = (Y_{ij}^k, a_{ij}^k, c_{ij}^k).$$

Intuitively, M_{ij}^t is the memory of `array_of_channels[i].sent[j]` while starting the iteration t . Notice that the domain of any M_{ij}^1 is $\{0\}$ and $M_{ij}^1(0) = (Xp[i], 0, 0)$: the `init` process initializes indeed `array_of_channels[i].sent[j]` with `Xp[i]`.

Let us show how to make the non-deterministic inside the two functions `retrieve_elements_values` and `broadcast` compliant with equation (3). Intuitively, the function M_{ij}^{t+1} is obtained as successive updates of M_{ij}^t through the two functions `retrieve_elements_values` and `broadcast`. Abusively, let $M_{ij}^{t+1/2}$ be the value of M_{ij}^t after the former function.

In what follows, we consider elements i , $1 \leq i \leq n$ and elements j , $1 \leq j \leq n$ that are updated. At iteration t , $t \geq 1$, let $(Y_{ij}^0, a_{ij}^0, c_{ij}^0)$ be the value of $M_{ij}^t(0)$ at the beginning of `retrieve_elements_values`. If t is equal to $c_{ij}^0 + 1$ then we execute the instruction that assigns Y_{ij}^0 (i.e. the head value of `array_of_channels[i].sent[j]`) to Xs_{ij}^t . In that case, the function M_{ij}^t is updated as follows: $M_{ij}^{t+1/2}(k) = M_{ij}^t(k)$ for each k , $0 \leq k \leq m - 2$ and $m - 1$ is removed from the domain of $M_{ij}^{t+1/2}$. Otherwise (i.e. when $t < c_{ij}^0 + 1$ or when the domain of M_{ij} is empty) the `skip` statement is executed and $M_{ij}^{t+1/2} = M_{ij}^t$.

In the function `broadcast`, if there exists some k , $k \geq t$ such that $S_{ij}^k = t$, let c_{ij} be defined by $\min\{k \mid S_{ij}^k = t\}$. In that case, we execute the instruction that adds the value `Xp[j]` to the tail of `array_of_channels[j].sent[i]`. M_{ij}^{t+1} is defined as an extension of $M_{ij}^{t+1/2}$ in m such that $M_{ij}^{t+1}(m)$ is $(Xp[j], t, c_{ij})$. Otherwise (i.e. when $\forall k. k \geq t \Rightarrow S_{ij}^k \neq t$ is established) the `skip` statement is executed and $M_{ij}^{t+1} = M_{ij}^{t+1/2}$.

Lemma 3 (Existence of SPIN Execution) *For any sequences $(S^t)^{t \in \mathbb{N}}$, $(J^t)^{t \in \mathbb{N}}$, for any map F there exists a SPIN execution such that for any iteration t , $t \geq 1$, for any i, j , $1 \leq i \leq n$, $1 \leq j \leq n$ we have the following properties:*
If the domain of M_{ij}^t is not empty, then

$$\begin{cases} M_{ij}^1(0) = (X_j^{S_{ij}^0}, 0, 0) \\ \text{if } t \geq 2 \text{ then } M_{ij}^t(0) = (X_j^{S_{ij}^c}, S_{ij}^c, c), c = \min\{k \mid S_{ij}^k > S_{ij}^{t-2}\} \end{cases} \quad (5)$$

Secondly we have:

$$\forall t'. 1 \leq t' \leq t \Rightarrow Xs_{ij}^{t'} = X_i^{S_{ij}^{t'-1}} \quad (6)$$

LIFC

Thirdly, for any $k \in J^t$, let k' be $k - 1$. Then, the value of the computed variable $\mathbf{Xp}[k']$ at the end of the `elements_updates` process is equal to X_k^{t+1} i.e. $F_k \left(X_1^{S_{k1}^t}, \dots, X_n^{S_{kn}^t} \right)$ at the end of the t -th iteration.

PROOF. The proof is done by induction on the number of iterations.

Initial case For the first item, by definition of M_{ij}^t , we have $M_{ij}^1(0) = (\mathbf{Xp}[j], 0, 0)$ that is obviously equal to $\left(X_j^{S_{ij}^0}, 0, 0 \right)$.

Next, the first call to the function `retrieve_elements_value` either assigns the head of `array_of_channels[i].sent[j]` to `Xs[j][i]` or does not modify `Xs[j][i]`. Thanks to the `init` process, both cases are equal to $\mathbf{Xp}[i]$, i.e. X_i^0 . The equation (6) is then established.

For the last item, let k , $1 \leq k \leq n$ and $k' = k - 1$. At the end of the first execution of the `elements_updates` process, the value of $\mathbf{Xp}[k']$ is $F(k', \mathbf{Xs}[k'] [0], \dots, \mathbf{Xs}[k'] [n - 1])$. Thus, by definition of Xs , it is equal to $F(k', Xs_{k1}^1, \dots, Xs_{kn}^1)$. Thanks to equation (6), we can conclude the proof.

Inductive case Suppose now the lemma 3 to be established until iteration l .

First, if domain of definition of the function M_{ij}^l is not empty, by induction hypothesis $M_{ij}^l(0)$ is $\left(X_j^{S_{ij}^c}, S_{ij}^c, c \right)$ where c is $\min\{k \mid S_{ij}^k > S_{ij}^{l-2}\}$.

At iteration l , if $l < c + 1$ then the `skip` statement is executed in the `retrieve_elements_values` function. Thus, $M_{ij}^{l+1}(0)$ is equal to $M_{ij}^l(0)$. Since $c > l - 1$ then $S_{ij}^c > S_{ij}^{l-1}$ and hence, c is $\min\{k \mid S_{ij}^k > S_{ij}^{l-1}\}$.

We now consider that at iteration l , l is $c + 1$. In other words, M_{ij} is modified depending on the domain $\text{dom}(M_{ij}^l)$ of M_{ij}^l :

- if $\text{dom}(M_{ij}^l) = \{0\}$ and $\forall k. k \geq l \Rightarrow S_{ij}^k \neq l$ is established then $\text{dom}(M_{ij}^{l+1})$ is empty and the lemma is established;
- if $\text{dom}(M_{ij}^l) = \{0\}$ and $\exists k. k \geq l \wedge S_{ij}^k = l$ is established then $M_{ij}^{l+1}(0)$ is $(\mathbf{Xp}[j], l, c_{ij})$ that is added in the `broadcast` function s.t. $c_{ij} = \min\{k \mid S_{ij}^k = l\}$. Let us prove that we can express $M_{ij}^{l+1}(0)$ as $\left(X_j^{S_{ij}^{c'}}, S_{ij}^{c'}, c' \right)$ where c' is $\min\{k \mid S_{ij}^k > S_{ij}^{l-1}\}$. First, it is not hard to establish that $S_{ij}^{c_{ij}} = l \geq S_{ij}^l > S_{ij}^{l-1}$ and thus $c_{ij} \geq c'$. Next, since $\text{dom}(M_{ij}^l) = \{0\}$, then between iterations $S_{ij}^c + 1$ and $l - 1$, the broadcast function has not updated M_{ij} . Formally we have

$$\forall t, k. S_{ij}^c < t < l \wedge k \geq t \Rightarrow S_{ij}^k \neq t.$$

Particularly, $S_{ij}^{c'} \notin \{S_{ij}^c + 1, \dots, l - 1\}$. We can apply the third item of the induction hypothesis to deduce $\mathbf{Xp}[j] = X_j^{S_{ij}^{c'}}$ and we can conclude.

- if $\{0, 1\} \subseteq \text{dom}(M_{ij}^l)$ then $M_{ij}^{l+1}(0)$ is $M_{ij}^l(1)$. Let $M_{ij}^l(1) = (\mathbf{Xp}[j], a_{ij}, c_{ij})$. By construction a_{ij} is $\min\{t' \mid t' > S_{ij}^c \wedge (\exists k. k \geq t' \wedge S_{ij}^k = t')\}$ and c_{ij} is

$\min\{k \mid S_{ij}^k = a_{ij}\}$. Let us show c_{ij} is equal to $\min\{k \mid S_{ij}^k > S_{ij}^{l-1}\}$ further referred as c' . First we have $S_{ij}^{c_{ij}} = a_{ij} > S_{ij}^c$. Since c by definition is greater or equal to $l-1$, then $S_{ij}^{c_{ij}} > S_{ij}^{l-1}$ and then $c_{ij} \geq c'$. Next, since c is $l-1$, c' is $\min\{k \mid S_{ij}^k > S_{ij}^c\}$ and then $a_{ij} \leq S_{ij}^{c'}$. Thus, $c_{ij} \leq c'$ and we can conclude as in the previous part.

The case where the domain $\text{dom}(M_{ij}^l)$ is empty but the formula $\exists k. k \geq l \wedge S_{ij}^k = l$ is established is equivalent to the second case given above and then is omitted.

Secondly, let us focus on the formula (6). At iteration $l+1$, let c' be defined as $\min\{k \mid S_{ij}^k > S_{ij}^{l-1}\}$. Two cases have to be considered depending on whether S_{ij}^l and S_{ij}^{l-1} are equal or not.

- If $S_{ij}^l = S_{ij}^{l-1}$, since $S_{ij}^{c'} > S_{ij}^{l-1}$, then $S_{ij}^{c'} > S_{ij}^l$ and then c' is distinct from l . Thus, the SPIN execution detailed above does not modify Xs_{ij}^{l+1} . It is obvious to establish that $Xs_{ij}^{l+1} = Xs_{ij}^l = X_i^{S_{ij}^{l-1}} = X_i^{S_{ij}^l}$.
- Otherwise S_{ij}^l is greater than S_{ij}^{l-1} and c is thus l . According to equations (5) we have proved, we have $M_{ij}^{l+1}(0) = (X_i^{S_{ij}^l}, S_{ij}^l, l)$. Then the SPIN execution detailed above assigns $X_i^{S_{ij}^l}$ to Xs_{ij}^{l+1} , which ends the proof of (6).

We are left to prove the induction of the third part of the lemma. Let $k, k \in J^{l+1}$ and $k' = k - 1$. At the end of the first execution of the `elements_updates` process, we have $Xp[k'] = F(k', Xs[k'] [0], \dots, Xs[k'] [n-1])$. By definition of Xs , it is equal to $F(k', Xs_{k_1}^{l+1}, \dots, Xs_{k_n}^{l+1})$. Thanks to equation(6) we have proved, we can conclude the proof.

Lemma 4 *Bounding the size of channels to $\max = \delta_0$ is sufficient when simulating a DDN where delays are bounded by δ_0 (cf. equation (2)).*

PROOF. For any i, j , at each iteration t , thanks to equation (2) element i has to know at worst δ_0 values that are $X_j^{S_{ij}^{t-1}}, \dots, X_j^{S_{ij}^{t-\delta_0}}$. They can be stored into any channel of size δ_0 .

Theorem 1 (Soundness wrt convergence property) *Let ϕ be a DDN model and ψ be its translation. If any execution of ψ converges, then any iteration of ϕ converges.*

PROOF. It is easier to show the contraposition of the theorem. The preceding lemmas have shown that for any execution of the DDN, there exists an execution of the PROMELA model that simulates it. If some iterations make the DDN diverge, then they make the PROMELA model diverge too.

Theorem 2 (Completeness) *Let ϕ be a DDN model and ψ be its translation. If an execution of ψ diverges under weak fairness property, then some iterations of ϕ diverge.*

LIFC

PROOF. For a diverging execution of ψ , it is easy to construct corresponding iterations of the DDN. Executions are performed under weak fairness property; we then detail what are continuously enabled:

- elements $1, \dots, n$: they are infinitely often updated leading to pseudo-periodic strategy;
- `skip` receiving and sending messages via channel: S_{ij} is increasing

Computed DDN has then the convenient property.

Next section presents practical keynotes about the approach.

6 Practical Issues

This section first gives some notes about complexity and later presents experiments.

6.1 Complexity Calculus

Theorem 3 (Number of states) *Let ϕ be a DDN model with n elements, m edges in the incidence graph and ψ be its translation into PROMELA. The configuration number of ψ SPIN execution is bounded by $2^{m \times (\delta_0 + 1) + 2n}$.*

PROOF. A configuration is a valuation of global variables. Their number only depends on those that are not constant.

The variables `Xp X` leads to 2^{2n} states. Each channel of `array_of_channels` may yield $1 + 2^1 + \dots + 2^{\delta_0} = 2^{\delta_0 + 1} - 1$ states. Since the number of edges in the incidence graph is m , there are m non-constant channels, leading to approximately $2^{m \times (\delta_0 + 1)}$ states. The number of configurations is then bounded by $2^{m \times (\delta_0 + 1) + 2n}$.

Notice we have not considered the global variable `Xs` since it should transformed into a local variable of the `elements_update` process.

Running example. Let δ_0 be 5, we then have at worst 2^{46} configurations.

6.2 Experiments

The method detailed along the line of this article have been applied on some examples to formally prove their convergence. The Figures 9 and 10 summarize convergence status of some examples respectively for synchronous and asynchronous iterations. Iterations are considered to be either parallel or pseudo-periodic. In asynchronous mode, delays are supposed to be bounded by δ_0 set to 10. We furthermore distinguish cases when delays are only restricted to be bounded and with cases when delays are bounded and even null between some elements as detailed in the mixed mode [12]. For each mode, we ask SPIN to verify the LTL convergence property (4). In these arrays, P is true (\top) provided

the property is established, false (\perp) otherwise, M is the amount of memory usage (in MB) and T is the time needed on a Intel Centrino Dual Core 2 Duo @1.8GHz with 2GB of memory, both to establish or refute the property.

The example [2] deals with a network composed of two genes taking their values into $\{0, 1, 2\}$. Since parallel iterations is already diverging, the same behavior is observed for all other modes.

In the example extracted from [14], we have 10 processors computing a binary value. Due to the huge number of dependencies between these calculus, δ_0 is reduced to 1. It nevertheless leads to about 2^{100} configurations in asynchronous iterations. Notice the absence of cycle in the incidence graph allows to conclude that any pseudo-periodic iteration mode is converging.

Execution times and memory usage show that translation is efficient enough even on a small computer to provide an answer for simple models. It then meets objectives of establishing formal proof of convergence. Applied on bigger models, experiments shows the limits of the approach.

		Parallel			P. Periodic		
		P	M	T	P	M	T
Running [12]	\top	2.7	0.01s	\perp	369.371	0.509s	
[2] example	\perp	2.5	0.001s	\perp	2.5	0.01s	
[14] example	\top	36.7	12s	\top			

Fig. 9. Experimentations with Synchronous Iterations

		Mixed Mode						Only Bounded					
		Parallel			P.Periodic			Parallel			P.Periodic		
		P	M	T	P	M	T	P	M	T	P	M	T
Running [12]	\top	409	1m11s	\perp^1	370	0.54	\perp	374	7.7s	\perp^2	370	0.51s	
[2] example	\perp	2.5	0.001s	\perp	2.5	0.01s	\perp	2.5	0.01s	\perp	2.5	0.01s	
[14] example	\top			\top			\perp			\perp			

Fig. 10. Experimentations with Asynchronous Iterations

7 Conclusion and Future Work

We first present related works. Stochastic based approaches have been implemented to generate both boolean neural networks, strategies and delays for asynchronous iterations first in Discrete System Evolution (DSE) research software [14,15] and in [3,4]. Next an exhaustive computation analyzes only the convergence for DSE and furthermore the existence of attraction basins in the latter case.

As far as we know, the closest work is SMBioNet [7] that takes as input a interaction graph G , a finite set of states E , and a temporal property expressed in Computational Tree Logic (CTL). It then generates all the maps F from

E to itself whose interaction graph is G . It outputs only the maps F whose chaotic iterations reduced to modify only one element satisfy the given temporal formula. The verification step is performed by the NuSMV model checker [16].

In this work, we have shown how convergence proof for any asynchronous iterations of discrete dynamical networks with bounded delays can be automatically achieved. The key idea is to translate the network (map, strategy) into PROMELA and to leave the SPIN model checker establishing the validity of the temporal property corresponding to the convergence. The correction and completeness of the approach have been proved, notably by computing a SPIN execution of the PROMELA model that have the same behaviors than initial network. The complexity of the problem is addressed. It shows that non trivial example may be addressed by this technique. This fact is illustrated on three examples of distinct origins.

Among drawbacks of the method, one can argue that bounded delays is only realistic in practice for close systems. However, in real large scale distributed systems where bandwidth is weak, this restriction is too strong. In that case, one should only consider that matrix S^t follows the iterations of the system, i.e. for all i, j , $1 \leq i \leq j \leq n$, we have $\lim_{t \rightarrow \infty} S_{ij}^t = +\infty$. One challenge of this work should consist in weakening this constraint. We plan as future work to take into account other automatic approaches to discharge proofs notably by proof based model checking [17].

References

1. R. Thomas, Boolean formalization of genetic control circuits, *Journal of Theoretical Biology* 42 (1973) 563–585.
2. A. Richard, J.-P. Comet, Necessary conditions for multistationarity in discrete dynamical systems, *Discrete Applied Mathematics* 155 (18) (2007) 2403–2413.
3. J. M. Bahi, S. Contassot-Vivier, Stability of fully asynchronous discrete-time discrete-state dynamic networks, *IEEE Transactions on Neural Networks* 13 (6) (2002) 1353–1363.
4. J. M. Bahi, S. Contassot-Vivier, Basins of attraction in fully asynchronous discrete-time discrete-state dynamic networks, *IEEE Transactions on Neural Networks* 17 (2) (2006) 397–408.
5. F. Robert, *Les systèmes dynamiques discrets*, Vol. 19 of *Mathématiques et Applications*, Springer, 1995.
6. H. De Jong, J. Geiselman, C. Hernandez, M. Page, Genetic network analyzer: qualitative simulation of genetic regulatory networks., *Bioinformatics* 19 (3) (2003) 336–44.
7. G. Bernot, J.-P. Comet, A. Richard, J. Guespin, A fruitful application of formal methods to biological regulatory networks: Extending Thomas' asynchronous logical approach with temporal logic, *J. Theor. Biol.* 229 (3) (2004) 339–347.
8. A. Gonzales, A. Naldi, L. Sánchez, D. Thieffry, C. Chaouiya, GINsim: a software suite for the qualitative modelling, simulation and analysis of regulatory networks, *Biosystems* 84 (2) (2006) 91–100.
9. G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, Pearson Education, 2003.

10. M. Ben-Ari, *Principles of the Spin Model Checker*, 2008.
11. D. P. Bertsekas, J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*, Prentice-Hall, Inc., 1989.
12. J. M. Bahi, S. Contassot-Vivier, J.-F. Couchot, Convergence results of combining synchronism and asynchronism for discrete-state discrete-time dynamic network (2009).
13. C. Weise, An incremental formal semantics for PROMELA, in: SPIN97, the Third SPIN Workshop, 1997.
14. J. M. Bahi, C. Michel, Simulations of asynchronous evolution of discrete systems, *Simulation Practice and Theory* 7 (1999) 309–324.
15. J. M. Bahi, C. Michel, Convergence of discrete asynchronous iterations, *International Journal Computer Mathematics* 74 (2000) 113–125.
16. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: An opensource tool for symbolic model checking, in: E. Brinksma, K. G. Larsen (Eds.), *CAV*, Vol. 2404 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 359–364.
17. J.-F. Couchot, A. Giorgetti, N. Kosmatov, A uniform deductive approach for parameterized protocol safety, in: D. F. Redmiles, T. Ellman, A. Zisman (Eds.), *ASE*, ACM, 2005, pp. 364–367.



L I F C

Laboratoire d'Informatique de l'université de Franche-Comté
UFR Sciences et Techniques, 16, route de Gray - 25030 Besançon Cedex (France)

LIFC - Antenne de Belfort : IUT Belfort-Montbéliard, rue Engel Gros, BP 527 - 90016 Belfort Cedex (France)
LIFC - Antenne de Montbéliard : UFR STGI, Pôle universitaire du Pays de Montbéliard - 25200 Montbéliard Cedex (France)

<http://lifc.univ-fcomte.fr>