Simulations of Self Replicating Loops

Frédéric Boussinot
EMP/INRIA, Mimosa Project
2004 route des Lucioles
F-06902 Sophia-Antipolis
http://www.inria.fr/mimosa/Frederic.Boussinot

August 13, 2004

Abstract

One describes how to create and simulate self-replicating loops in the context of cellular automata spaces and of reactive programming. Cell behaviors are programs instead of standard look-up tables. A destruction mechanism is implemented and walls are introduced in order to contain loops in closed areas.

1 Introduction

One considers self-replicating loops (SR Loops) in cellular automata spaces. These loops are basically the ones of Langton[3] except that they don't have a sheath. They are made of sequences of genes that are duplicated and interpreted during the replication process. Replication is informally described on Figure 1; pictures from 1 to 12 show the main steps of the process:

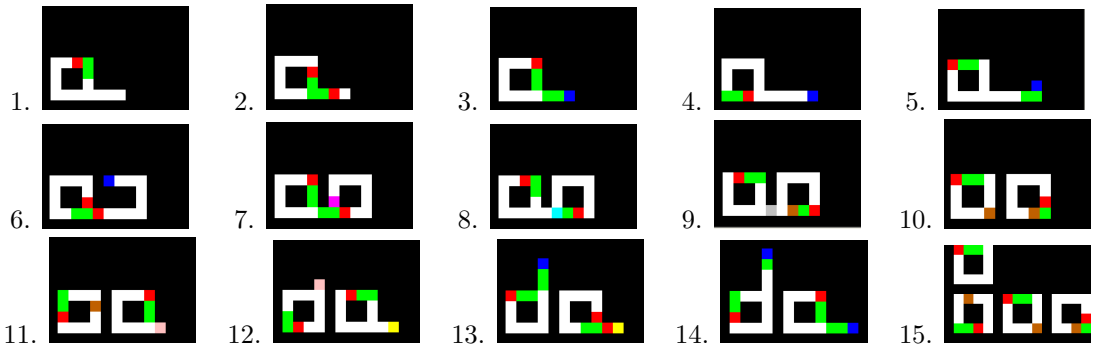

Figure 1: Main Steps of Loop Replication

1. This is the initial loop. It is made of a square of white, red (or dark grey) and green (or light grey) cells, and of an arm starting from the right/bottom corner of the square. The red cell is interpreted as a gene which makes the arm turn one step in the counter-clockwise direction. The two green cells are interpreted as genes which make the arm grow one step. The genes are moving in the counter-clockwise direction along the square and the arm.
2. The genes are duplicated at the beginning of the arm (instant 11).
3. A translator (in blue) appears when a turning gene (a red one) reaches the end of the arm (instant 12). The purpose of the translator is to interpret genes.
4. The arm has grown as the two green genes have been interpreted (instant 20) .

5. A red gene has just being interpreted and the arm turns (instant 27).
6. A second turn has been made by interpreting an other red gene (instant 46).
7. The arm has turned back on itself and a collide cell (in magenta) is created on top of the arm cell (instant 60).
8. A barrier (in cyan) is created at bottom of the previous collide cell (instant 61). It replaces a green gene.
9. A sprout cell (in brown) is created on the right of the previous barrier (instant 62). The cell on the left of the previous barrier turns to a stop cell (in grey).
10. The arm has been retracted up to the initial loop, and a sprout has been created in it (instant 63).
11. In the new loop, a new arm has been created (pre-waiter cell, in pink) when the sprout has reached the right/bottom corner (instant 65).
12. In the new loop, the pre-waiter cell has turned to a waiter cell (in yellow; instant 67). Now, a turn gene is awaited, to trigger the arm construction. In the initial loop, a pre-waiter cell is created by the sprout.
13. A new translator is created in the initial loop (instant 75). The replication process can now continue from the new arm of the initial loop.
14. A new translator is created in the new loop (instant 77). The replication process can continue from the new arm in the copy.
15. Both loops have created new loops which will themselves produce new loops (instant 128).

In [4, 5], Sayama introduces a destruction mechanism in order to manifest evolution-like global behaviors. In this text, one considers a different destruction mechanism. In the new mechanism, in case of collision, the touched loop is destroyed and the touching loop starts to retract its arm, just like during self replication. Moreover, retraction of the arm can be stopped by the arrival of new genes. As in Sayama's work, an evolution-like process can appear using this new destruction mechanism.

Moreover, one introduces a new kind of cells, called *walls*. Walls have no specific behavior and lead to arm retraction when touched. Walls can thus be used to create borders for loops. Using walls, it becomes possible to confine loops to closed areas.

This text describes an environment for simulating SR Loops. This environment is implemented with reactive programming[1], but no deep knowledge of it is needed for creating new simulations. The SR Loops and the code to simulate them are available on the Web. A first version of SR Loops implementation is described in [2], which mainly considers reactive programming to implement cellular automata.

The structure of the paper is as follows: section 2 describes the cells; construction of SR Loops and configuration of simulations are considered in section 3; finally, several experiments are described in section 4. The C code of the main functions introduced in the text is given in annex.

2 Cells

The implementation of cellular automata spaces considered in this text is described in details in [2]. Here, one just considers points that are specific to SR Loops. Information used for cells is twofold: a *status* and a *state*.

1. The status of a cell defines it as a quiescent cell, as a wall, or as a *directed cell*. The notion of a quiescent cell is standard in the context of cellular automata. A wall does not have any proper behavior and is used to delimit borders for loops. A directed cell basically has a direction associated to it[1].

[1] Cells have four neighbors identified by their direction: top, right, bottom, left (von Neumann neighborhood).

2. The state of a cell can be one of the following:

   - `BASIC`: the cell has no specific behavior.
   - `COLLIDE`: the state when the arm gets back to itself.
   - `BARRIER`: the state used for producing sprouts.
   - `GROW_GENE`, `CLK_GENE`, and `INVCLK_GENE`: the genes.
   - `TRANSLATOR`: the cell is ready to interpret genes.
   - `SPROUT`: the state which starts a new arm when reaching a corner.
   - `PRE_WAITER` and `WAITER`: the states to deal with sprouts.
   - `STOP`: for arm retraction, until the initial loop is reached. Stopped cells turn to quiescent at the next instant.
   - `ERASE`: for cell destruction. Erased cells turn to quiescent at the next instant.
   - `NONE`: the state of quiescent cells (not drawn on screen).

A cell is said to be *dead* (macro `DEAD_NEIGHBOR`) if it is quiescent, or if it is a wall, or if its state is `STOP` or `ERASE`.

To *fire* a neighbor n of a cell c means to give n an order to change its state (see [2] for details). If n is not dead, then c is erased. Otherwise, a firing order is transmitted to n. The firing will become effective only if it is unique (determinism): then, n changes its state according to the firing order.

The behavior of a cell is defined by the function `cell_behavior` considered in 2.4. Four auxiliary functions are first defined: `translator`, `pre_waiter`, `waiter`, and `step`. All four functions take as argument a directed cell. Let d be the direction of the cell (actually, its status). The neighbor in direction d is called the *source cell*. The neighbor in the counter-clockwise direction is called the *invclk cell* (macro `INVCLK`). The neighbor in the clockwise direction is called the *clk cell* (macro `CLK`). The neighbor in the opposite direction of d is called the *opposite cell* (macro `OPPOSITE`).

2.1 Translator Function

The function `translator` translates genes. The considered gene is the one of the source. The gene value is tested using the macro `NEIGHBOR_STATE`.

The cell pointed to by the gene (opposite for `GROW_GENE`, clk for `CLK_GENE`, and invclk for `INVCLK_GENE`) is tested for death. If it is dead, then the current cell state changes to `COLLIDE`. Otherwise, it changes to `BASIC` and a new translator cell is fired in place of the dead cell.

2.2 Pre-Waiter and Waiter Functions

The two functions `pre_waiter` and `waiter` are used to create new sprouts.

The `pre_waiter` function has for unique task to fire a waiter cell at next instant. The current cell state is then changed to `BASIC`.

The `waiter` function tests for a turning gene (`INVCLK` or `CLK`) in the source and when it is the case, it changes the current cell state to `BASIC` and fires the opposite cell as a translator.

2.3 Step Function

The function `step` is called for directed cells which are neither translators, nor waiters, nor pre-waiters.

1. A cell changes to `STOP` in front of a wall. `STOP` cells are used for arm retraction.
2. The current cell becomes a translator when the source cell holds a turning gene while the 3 others positions are dead. Translators are thus issued from turning genes reaching the end of the arm.

3. When the invclk cell is a collide cell (produced when a translator cannot progress forward), then the current cell becomes a barrier with the same direction as the collide cell. This is the situation encountered when the arm loops back on itself. The change of direction is a way to close the new loop construction.

4. When the clk cell is a collide cell, then the current cell becomes a barrier with the same direction as the collide cell. This is the dual of previous rule, for the clk gene.

5. In front of a barrier with a different direction, the cell turns to `STOP`. This starts arm retraction.

6. The cell becomes a sprout when the source cell is a barrier. This is the case in the copy of the loop, when the arm is cut.

7. In front of a stopped cell, the current cell turns to `STOP` when both invclk and clk cells are dead. Otherwise, the cell becomes a sprout. This is the case when the destruction of the arm reaches the initial loop.

8. A pre-waiter is fired in the opposite cell if it is dead and if the source cell is a sprout. This occurs in the initial cell when a sprout reaches a corner. In this case, construction of a new arm can start.

9. If none of the previous rules apply, then state of the source cell is copied into the current cell.

2.4 Cell Behavior

The function `cell_behavior` processes the following actions in order:

1. Nothing is done if the current cell is a wall (it is then simply drawn).

2. Cell firing is processed if needed and possible. If the cell has been fired more than once, a non-deterministic situation is detected, and the cell is erased. The cell is also erased if it is not quiescent. Otherwise, the cell is assigned the status and state issued from the unique firing.

3. Processing is over if the cell is quiescent.

4. If the cell is stopped or erased, it is changed to quiescent (function `eliminate_stopped`).

5. The cell is erased if the source is dead or if one of its neighbors is erased (function `propagate_erase`).

6. Otherwise, functions `pre_waiter`, `waiter`, `translator`, or `step` are called according to the state of the current cell.

7. Finally, the cell is drawn according to its state (function `draw_cell`).

Three points are important to note:

- In function `step`, the gene of the current cell can be replaced by a barrier in rule 3 an 4, or by a sprout in rules 6 and 7. This is the way new shapes are created.

- When function `translator` cannot interpret a gene because the target cell is not dead, then cell state is changed to collide. At the next instant, a barrier is created (rules 3 and 4 of `step`) which will lead to the creation of a stopped cell (rule 5). In function `propagate_erase`, a cell is erased if its source is dead; this is the case when the source is stopped. There is then an a-symetry: the cells issued from the barrier are erased, while those leading to it are stopped. This is the basic difference with the destruction mechanism of Sayama.

- Stopped cells are considered as dead cells. Thus, arm retraction can be canceled by the arrival of a turning gene, when rule 1 of function `step` applies. In this case, a new translator is created, arm retraction stops, and building of the arm can restart.

3 Loops and Simulations

One now considers ways to build loops and to configure the global simulation.

3.1 Loop Construction

Loops are built using the `build_loop` function. The first parameter is the number of lines of the second parameter which is an array of strings defining the constructed loop. The third and fourth parameters define the top/left position at which the construction will start.

Each line of the loop is parsed to extract the status (function `get_status`) and state (function `get_state`) of the built cells (by default, all other cells are quiescent). The parsing is based on the following encoding:

- 0,1,2,3 define the basic directed cells (0 corresponds to top, 1 to right, 2 to bottom, and 3 to left).
- 9 defines a wall.
- a,b,c,d are the directed variants of the grow gene (a corresponds to top, b to right, c to bottom, and d to left).
- e,f,g,h are the directed variants of the invclk gene.
- i,j,k,l are the directed variants of the clk gene.

One adopts the convention that a loop named X should be described in a file named `X.c` in which the function `build_simulation` should be defined. For example, consider the following description:

```
char *X[] = {
"11fc",
"0  c",
"0  2",
"0333333",
};

void build_simulation (void)
{
   build_loop (4,X,MAXX/2,MAXY/2);
}
```

This description defines the loop shown in the first picture of Figure 1; it has one right-directed invclk gene (f, in red) and two bottom-directed grow genes (c, in green). The loop is placed at the center of the simulation (`MAXX` and `MAXY` are dimensions of the simulation; see next section).

3.2 Simulation Configuration

Some configuration variables in file `config.h` can be used to configure simulations:

- `MAXX` and `MAXY` are the dimensions of the CA space (typically 200x200).
- The larger `ZOOM` is from 1, the larger the graphics is.
- Cells are created on need when variable `CELL_ON_NEED` is defined. When not defined, all cells are created before execution.
- One (and only one!) of the following variables must be defined:
  1. `SR` corresponds to the standard self-replicating loops, without destruction.
  2. `SDSR` introduces the destruction mechanism of Sayama. It is needed for evo-loops.

- When defined, `FREEZE` is the number of instants before freezing the execution; useful for taking snapshots.
- When defined, `EXIT` is the number of instants before exiting; useful to measure execution time.
- When defined, `SKIP_UPDATE` is the number of graphical updates that are skipped between two actual updates; useful to speed-up the graphics.
- When `WALL_BORDER` is defined, the whole simulation is enclosed by walls (see 4.3).

To simulate a loop described in file `X.c`, one has simply to define in the makefile a new target `X` which sets the variable `PROG`:

```
X:
    make PROG=X
```

The simulation is run by the command `make X`.

4 Experiments

Several experiments are described in this section. In 4.1 is defined a SR Loop that manifests an evolution-like process. The way to build loops from a linear description is considered in 4.2. Finally, loops enclosed by walls are considered in 4.3.

4.1 Evo Loops

Let us consider the evo-loop of Figure 2:

```
char *evoLoop[] = {
" 111g",
" a  c",
" a  c",
" a  c",
" a  c",
" a  c",
" e3333333",
};
```

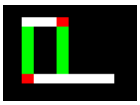

Figure 2: Evo Loop

On the left of Figure 3 is the initial situation of the evo loop in a simulation made of 200x200 cells; on the right is the situation after 40,000 instants. The only loops that remain present are small ones with only one sequence of 3 genes. All other loops have been eliminated. Note that walls (light-blue colored) have been created all along the applet borders.

4.2 Linear Loops

It is possible to define a loop as a unique line of genes whose interpretation first builds the loop. For example, consider the definitions:

```
#define Start "h3"
#define End   "333333333333"
#define IHead "h3"
#define CHead "13"
```

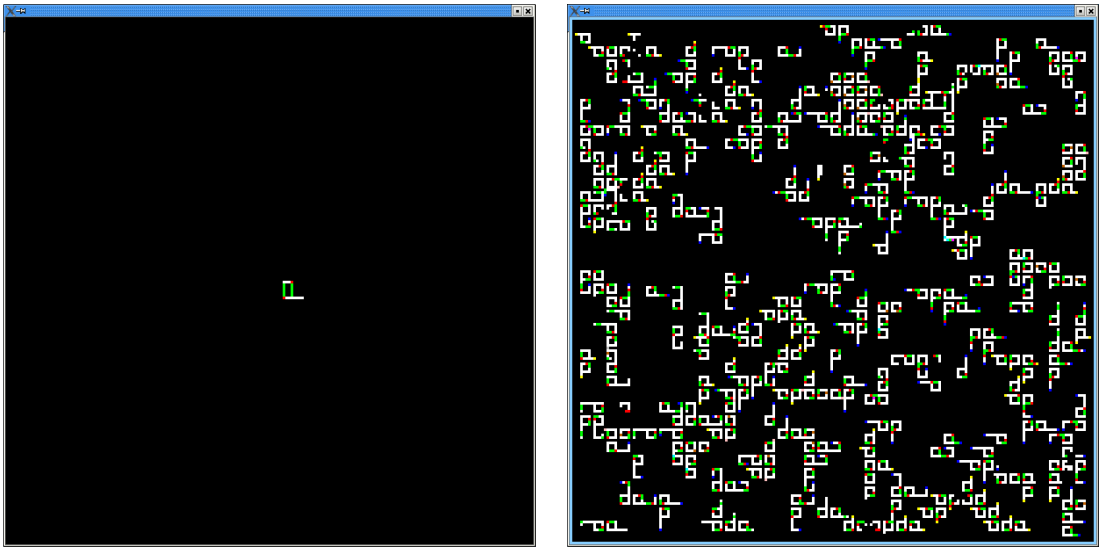

Figure 3: Evo Loop Behavior

```
#define Grow2  "dd"
#define C2 Grow2 CHead
#define I2 Grow2 IHead
#define Side2 Grow2 IHead

#define SmallI End I2 Side2 Side2 Side2 Start

char *lineLoop[] = {SmallI};
```

This defines a line shown on first picture of Figure 4. After 16 instants, the line is shown on second picture. At instant 47, the loop of third picture is formed and starts replication.

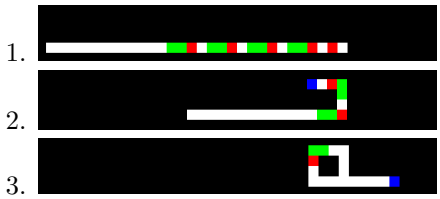

Figure 4: Linear Loop

The macro `Start` creates a translator to interpret the genes. The macro `End` produces a trailing list of left directed cells. The first three sides of the loop are created by three instances of `Side2`. They are made of 2 grow genes following one counter-clockwise turning gene. The loop embeds the final `I2` part which is actually similar to `Side2`.

One can easily create variants of previous loop. For example, adding one more `I2` would create a loop that turns more quickly than the initial one. As another example, consider the loop defined by:

```
#define FastC End C2 C2 C2 Side2 Side2 Side2 Start
```

A fast small loop is created that turns in the clockwise direction. The simulation after 400 instants is shown on left picture of Figure 5. On the right, is the situation after the same number of instant for a loop

embedding only one `C2` sequence; one can see that the number of created loop is smaller than with previous loop.

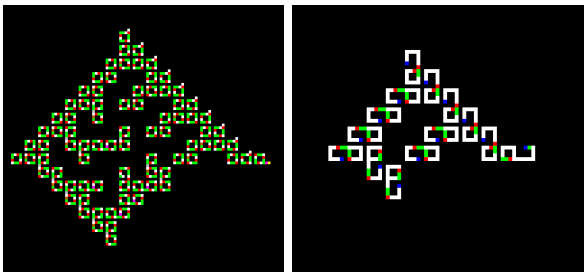

Figure 5: Fast and Slow Clk Loops after 400 instants

Linear loops can be useful for building examples of evolution process. For example, consider the loop defined by:

```
#define ShapeEvo End2 C6 C4 C6 I12 Side12 Side12 Side12 Start
```

`Side12` is similar to `Side2` except that 12 `d` cells are used, instead of 2. In the same way, `I12`, `C6`, `C4`, and `End2` extend the corresponding previous macros.

Two kinds of loops are created: bigger ones with the `INVCLK` gene and smaller ones with the `CLK` gene. The situation at instant 1,600 is shown on the left part of Figure 6. At instant 8,000, the smaller loops have won; this is shown on the right picture.

Figure 6: Shape Evolution

Evolution can also result from loops with the same shape but running at different speeds. For example, consider:

```
#define SpeedEvo End3 C2 C2 C2 C2 C2 C2 C2 Side12 Side12 Side12 Start
```

Loops with 1,2, or 3 sequences of genes appear during the simulation. However, loops with 3 sequences run more quickly and thus self-reproduce more often than other loops. As some new loops with 1 or 2 sequences are created from ones with 3 sequences, they never actually disappear (at least during the first 200,000 tested instants).

4.3 Enclosed Loops

Using walls, one can enclose loops in a room in order to confine them to it. Note that this is actually the case of the whole simulation when the macro `WALL_BORDER` is defined. In this case, walls are created all along the borders of the simulation.

Let us consider a loop confined to a closed room, the description of which appears in Figure 7. The loop arm has a cyclic behavior, growing, reaching a border, retracting, and turning in the counter-clockwise direction. Figure 8 shows several steps of the behavior:

```
"99999999999999999999",
"9                  9",
"9                  9",
"9                  9",
"9      1111g       9",
"9      a   c       9",
"9      a   c       9",
"9      a   c       9",
"9      a   c       9",
"9      a333333     9",
"9                  9",
"9                  9",
"9                  9",
"99999999999999999999"
```

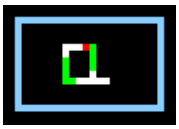

Figure 7: Loop in a Closed Room

1. The loop at instant 1.
2. The arm touches the right border (instant 34).
3. The arm retracts (instant 40).
4. A new sprout is created (instant 49).
5. The arm touches the top border (instant 58).
6. A new sprout is created (instant 66).
7. The arm touches the left border (instant 86)
8. The arm goes toward the bottom border (instant 100).
9. The arm goes toward the right border (instant 133).

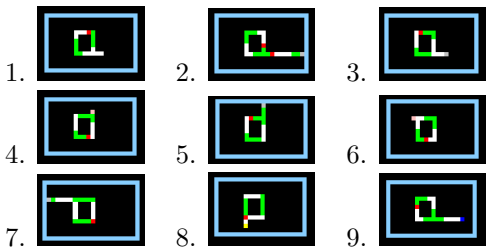

Figure 8: Confined Loop

Holes in the walls can be used by loops to escape. Such an escape is shown on Figure 9. A loop is placed in a room with a hole at the bottom side on the left part of the figure (instant 1). On the right is the situation after 36,000 instants where the loop has escaped and self-replicated. Note that the position of the hole is of course crucial for the arm to be able to pass through it.

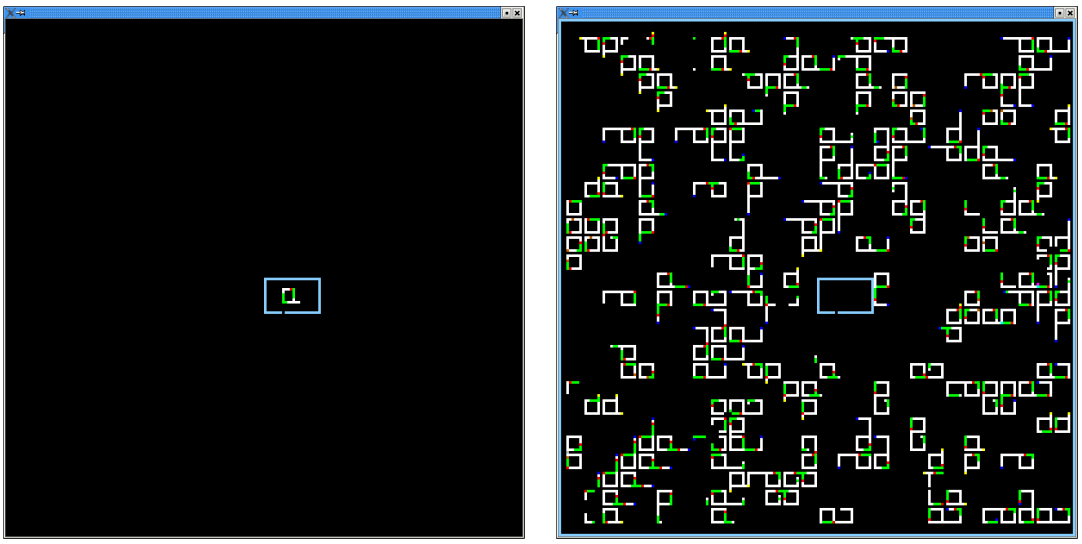

Figure 9: Escaped Loop

The arm retracts when it touches a wall or another loop. In some situations, this leads to a kind of "creeping" behavior, in which the arm moves along a wall. For example, the first steps of the exiting loop of Figure 9 are described on Figure 10:

1. The initial situation.
2. The loop has found the hole and the arm is passing through it.
3. The arm turns because of a invclk (red) gene.
4. After a new turn, the arm touches the wall again.
5. The arm starts to retract.
6. The arm continues to progress as a new translator is created.
7. The arm turns another time and touches the wall again.
8. The progression continues.
9. After a new turn, the arm reaches the top of the room and touches it.
10. After progression, the arm touches the top again.
11. Now, the arm can begin the descent along the left side.
12. The arm returns to itself.
13. A new sprout is created in the enclosing loop, and the inner loop starts to retract.
14. A new middle-sized loop is created and the arm of the inner loop has restarted to grow.

Observing the arm, one can have the feeling that the loop explores the space and tries to enclose the room. This resembles an "emergent" behavior.

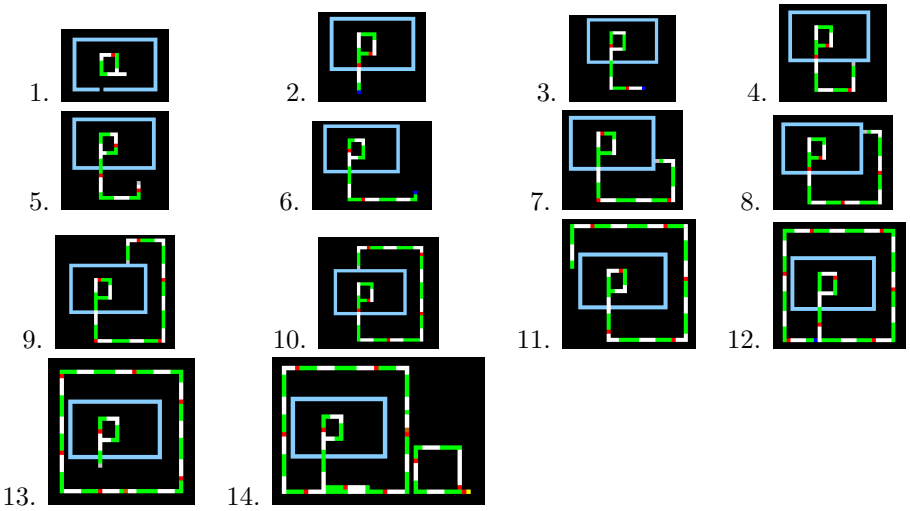

Figure 10: Creeping Arm

5 Conclusion

One has described an environment to simulate self-replicating loops and the way to use it. Loops behaviors are programs, and are not, as traditionally, coded as lookup tables. A new destruction mechanism is defined in which a loop is not systematically destroyed when it touches another loop. Special cells, called walls, are introduced in order to confine loops to closed areas. One of the basic features of self-replication is the potential for exponential growth. This can also be seen as a basic problem to which walls could give a solution. Several experiments are considered which show how loops behave in various contexts including closed areas. The implementation described in this text is available on the Web.

References

[1] Reactive programming web site, http://www-sop.inria.fr/mimosa/rp.

[2] F. Boussinot. *Reactive Programming of Cellular Automata*. Inria research report, RR-5183, May 2004.

[3] C. G. Langton. Self-reproduction in cellular automata. *Physica D*, 10:135–144, 1984.

[4] H. Sayama. *Constructing evolutionary systems on a simple deterministic cellular automata space*. Phd, University of Tokyo, Department of Information Science, 1998.

[5] H. Sayama. Introduction of structural dissolution into Langton's self-reproducing loop. In *C. Adami, R.K. Belew, H. Kitano, C.E. Taylor (Eds), Artificial Life, Proc. of the Sixth International Conference on Artificial Life*, pages 114–122, 1998.

Annex

Step Function

```
static void step (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int source = d->status;
   int opposite = OPPOSITE (source);
```

```
   int invclk = INVCLK (source);
   int clk = CLK (source);
   // 1. stop in front of a wall
   if (NEIGHBOR_STATUS (opposite,WALL)) {
      d->state = STOP;
      return;
   }
   // 2. translator produced by terminal turning genes
   if ( TURNING_GENE_IN (source) && DEAD_NEIGHBOR (opposite)
        && DEAD_NEIGHBOR (invclk) && DEAD_NEIGHBOR (clk) ) {
      d->state = TRANSLATOR;
      return;
   }
   // 3. when collide detected in invclk, change to barrier state
   if (NEIGHBOR_STATE (invclk,COLLIDE)) {
      d->status = invclk; // get the collide state. to close the new loop
      d->state = BARRIER;
      return;
   }
   // 4. when collide detected in clk, change to barrier state
   if (NEIGHBOR_STATE (clk,COLLIDE)) {
      d->status = clk;
      d->state = BARRIER;
      return;
   }
   // 5. stop if barrier forward and state different. starts arm destruction
   if (NEIGHBOR_STATE (opposite,BARRIER) && STATUS_OF_NEIGHBOR(opposite) != source) {
      d->state = STOP;
      return;
   }
   // 6. barrier produces sprout in the copy
   if (NEIGHBOR_STATE (source,BARRIER)) {
      d->state = SPROUT;
      return;
   }
   // 7. reverse progression of stop along the arm; finishes at first corner
   if (NEIGHBOR_STATE (opposite,STOP)) {
      if (!DEAD_NEIGHBOR (invclk) || !DEAD_NEIGHBOR (clk)) {
         d->state = SPROUT;
      } else {
         d->state = STOP;
      }
      return;
   }
   // 8. new waiter created in the copy when sprout reaches a corner
   if (NEIGHBOR_STATE (source,SPROUT) && DEAD_NEIGHBOR (opposite)) {
      fire (cell,opposite,PRE_WAITER);
      // state remains the same
      return;
   }
   // 9. copy the source state
   if (d->neighborhood[source]) d->state = STATE_OF_NEIGHBOR(source);
}
```

Translator Function

```
static void translator (thread_t cell)
{
```

```
   cell_data_t d = local_data (cell);
   int source = d->status;
   int opposite = OPPOSITE (source);
   int invclk = INVCLK (source);
   int clk = CLK (source);
   int dir;
   // genes translation
   if (NEIGHBOR_STATE (source,GROW_GENE)) dir = opposite;
   else if (NEIGHBOR_STATE (source,INVCLK_GENE)) dir = invclk;
   else if (NEIGHBOR_STATE (source,CLK_GENE)) dir = clk;
   else return; // not a gene

   if (!DEAD_NEIGHBOR (dir)) { d->state = COLLIDE; return; }
   fire_and_change (cell,dir,TRANSLATOR,BASIC);
}
```

Waiter Functions

```
static void pre_waiter (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int source = d->status;
   fire_and_change (cell,OPPOSITE (source),WAITER,BASIC);
}

static void waiter (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int source = d->status;
   if (TURNING_GENE_IN (source)) {
      fire_and_change (cell,OPPOSITE (source),TRANSLATOR,BASIC);
   }
}
```

Erase Propagation

```
static int propagate_erase (thread_t cell)
{
   cell_data_t d = local_data (cell);
   int source = d->status;
   if (DEAD_NEIGHBOR (source) ||
       NEIGHBOR_STATE (OPPOSITE (source),ERASE) ||
       NEIGHBOR_STATE (INVCLK (source),ERASE) ||
       NEIGHBOR_STATE (CLK (source),ERASE))
   {
      d->state = ERASE;
      return 1;
   }
   return 0;
}
```