21st Brainstorming Week on Membrane Computing (BWMC 2025)

From WHILE programs to Spiking Neural P systems

Alberto Leporati and **Lorenzo Rovida**

University of Milano-Bicocca

Department of Informatics, Systems, and Communication

alberto.leporati@unimib.it, lorenzo.rovida@unimib.it

Seville – January 22-24, 2025

A "high-level" programming language for building spiking neural P systems

Imagine you want to build (say) a spiking neural (SN) P system that computes (say) the square function: $f(n) = n^2$

- You may work directly with SN P systems
- You may first write a program for a register machine, and then build the SN P system by composing ADD and SUB modules
  - This substitution can be performed *automatically*
  - It works for many universal models of P systems (and not only)
- However, working directly with register machines is uncomfortable
  - So what about writing a program in a « high-level » programming language, which is then compiled to an equivalent program for register machines?

A “high-level” programming language for building spiking neural P systems

- We propose to make both translations automatically, that is:

- The first compiler would be fixed, the others would depend upon the model of P systems considered
- The output could be given in P-Lingua
- To start with, the high-level language should be very easy
  - A possible candidate: the WHILE language
  - Of course, the WHILE language is Turing-complete

A "high-level" programming language for building spiking neural P systems

- The WHILE language (used in Davies et al.'s book):
  - Variables x_j, for $j \in \mathbb{N}$, each containing a non-negative integer value
  - Assignment commands:

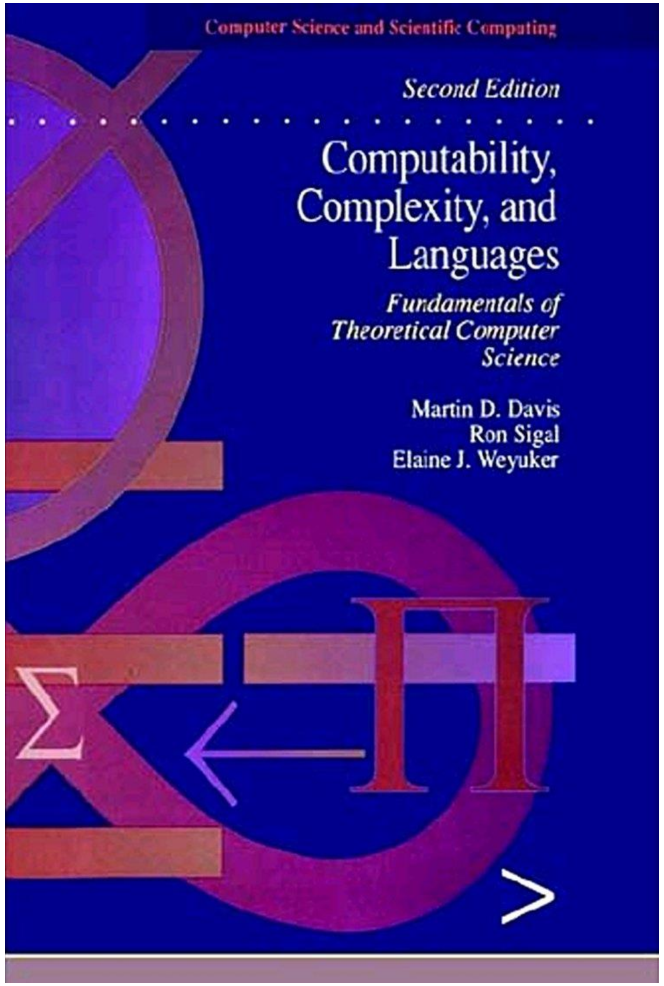
    $x_k := 0 \qquad x_k := x_j + 1 \qquad x_k := x_j \dot{-} 1$ (truncated decrement)
  - While commands:

    while $x_k \neq 0$ do C

    where C is an arbitrary command
  - Compound commands:

    begin C_1; C_2; … C_m; end $\qquad (m > 0)$

    where C_1; C_2; … C_m are arbitrary commands
  - A program is a compound command



A "high-level" programming language for building spiking neural P systems

- The WHILE language can be extended through macros of the kind

$$x_i = Op(x_j, x_k)$$

  For example, *Op* can be *Sum*, *Product*, *TruncatedSum*, *IntegerDivision*, *Mod*, *CantorPairingFunction*, …

- Other possible natural extensions/alternatives:
  - Using a more sophisticated/expressive language
    - Programs would be easier to write, but the compiler would be harder to write
  - What about a concurrent programming language?
    - Inspired from Occam?

- For now, we have worked with the WHILE language

An extended WHILE grammar

$$
\begin{aligned}
\langle \textit{program} \rangle & \rightarrow [\langle \textit{include} \rangle] \\
& \quad \text{`}\texttt{Program}\text{'}\ \texttt{[a-zA-Z]}\,\texttt{[a-zA-Z0-9]}\ast\ \text{`}\texttt{;}\text{'} \\
& \quad [\langle \textit{description} \rangle] \\
& \quad [\langle \textit{input} \rangle] \\
& \quad \langle \textit{statements-list} \rangle \\
& \quad \langle \textit{output} \rangle \\
\langle \textit{include} \rangle & \rightarrow \text{`}\texttt{include}\text{'} \rightarrow \texttt{[a-zA-Z0-9\_.-]}\ast\ \text{`}\texttt{;}\text{'}\ [\langle \textit{include} \rangle] \\
\langle \textit{description} \rangle & \rightarrow \text{`}\texttt{/*}\text{'}\ \texttt{[a-zA-Z0-9\_.-\textbackslash s]}\ast\ \text{`}\texttt{*/}\text{'} \\
\langle \textit{input} \rangle & \rightarrow \text{`}\texttt{input}\text{'}\ \langle \textit{registers} \rangle \\
\langle \textit{composed-statement} \rangle & \rightarrow \text{`}\texttt{begin}\text{'}\ \langle \textit{statements} \rangle\ \text{`}\texttt{end}\text{'} \\
\langle \textit{output} \rangle & \rightarrow \text{`}\texttt{output}\text{'}\ \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle
\end{aligned}
$$

An extended WHILE grammar

$$
\begin{array}{lll}
\langle \textit{statements} \rangle & \rightarrow & \langle \textit{statement} \rangle\ [\langle \textit{statement} \rangle] \\
\langle \textit{statement} \rangle & \rightarrow & \langle \textit{assignment} \rangle \\
 & | & \langle \textit{inc} \rangle \\
 & | & \langle \textit{dec} \rangle \\
 & | & \langle \textit{while} \rangle \\
 & | & \langle \textit{macro} \rangle \\
 & | & \langle \textit{comment} \rangle \\
\langle \textit{assignment} \rangle & \rightarrow & \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{=}\text{'}\ \texttt{[0-9]}*\ \text{`}\texttt{;}\text{'} \\
\langle \textit{inc} \rangle & \rightarrow & \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{=}\text{'}\ \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{+}\text{'}\ \text{`}\texttt{1}\text{'}\ \text{`}\texttt{;}\text{'} \\
\langle \textit{dec} \rangle & \rightarrow & \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{=}\text{'}\ \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{-}\text{'}\ \text{`}\texttt{1}\text{'}\ \text{`}\texttt{;}\text{'} \\
\langle \textit{while} \rangle & \rightarrow & \text{`}\texttt{while}\text{'}\ \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{!=}\text{'}\ \text{`}\texttt{0}\text{'}\ \text{`}\texttt{do}\text{'}\ \langle \textit{composed-statement} \rangle \\
 & | & \text{`}\texttt{while}\text{'}\ \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{!=}\text{'}\ \text{`}\texttt{0}\text{'}\ \text{`}\texttt{do}\text{'}\ \langle \textit{statements} \rangle\ \text{`}\texttt{end while}\text{'} \\
\langle \textit{comment} \rangle & \rightarrow & \text{`}\texttt{//}\text{'}\ \texttt{.*} \\
\langle \textit{registers} \rangle & \rightarrow & \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle \\
 & | & \text{`}\texttt{x\_}\text{'}\ \langle \textit{id} \rangle\ \text{`}\texttt{,}\text{'}\ \langle \textit{registers} \rangle \\
\hline
\langle \textit{id} \rangle & \rightarrow & \texttt{[0-9]\{1, 2\}}
\end{array}
$$

Translating WHILE programs to Register Machines programs

- Assignment statement:

$$T_{W \to RM}(x_i := 0) = \begin{cases} l_0 : & \texttt{SUB}(r_i), l_0, l_1 \\ l_1 : & \dots \end{cases}$$

- Increment statement ($x_i = x_j + 1$, case $i = j$):

$$T_{W \to RM}(x_i := x_j + 1) = \begin{cases} l_0 : & \texttt{ADD}(r_i), l_1, l_1 \\ l_1 : & \dots \end{cases}$$

Translating WHILE programs to Register Machines programs

- Increment statement ($x_i = x_j + 1$, case $i \neq j$):

$$T_{W \to RM}(x_i := x_j + 1) = \begin{cases} l_0 : & \mathtt{SUB}(r_i), l_0, l_1 \\ l_1 : & \mathtt{ADD}(r_{22}), l_2, l_2 \\ l_2 : & \mathtt{ADD}(r_i), l_3, l_3 \\ l_3 : & \mathtt{SUB}(r_j), l_1, l_4 \\ l_4 : & \mathtt{SUB}(r_{22}), l_4, l_6 \\ l_5 : & \mathtt{ADD}(r_j), l_4, l_4 \\ l_5 : & \mathtt{SUB}(r_{22}), l_5, l_6 \\ l_6 : & \ldots \end{cases}$$

Reset r_i to 0, then move the value of r_j to both r_i and r_{22} (an auxiliary register, since moving destroys the origin), then increment r_i, then move the value of r_{22} to r_j

Translating WHILE programs to Register Machines programs

- Decrement statement ($x_i = x_j \dot{-} 1$, case $i = j$):

$$T_{W \to RM}(x_i := x_j \dot{-} 1) = \begin{cases} l_0 : & \mathtt{SUB}(r_i \\ l_1 : & \ldots \end{cases}$$

- Decrement statement ($x_i = x_j \dot{-} 1$, case $i \neq j$):

$$T_{W \to RM}(x_i := x_j \dot{-} 1) = \begin{cases} l_0 : & \mathtt{SUB}(r_i), l_0, l_1 \\ l_1 : & \mathtt{ADD}(r_{22}), l_2, l_2 \\ l_2 : & \mathtt{ADD}(r_i), l_3, l_3 \\ l_3 : & \mathtt{SUB}(r_j), l_1, l_4 \\ l_4 : & \mathtt{SUB}(r_i), l_5, l_5 \\ l_5 : & \mathtt{SUB}(r_i), l_6, l_6 \\ l_6 : & \mathtt{SUB}(r_{22}), l_7, l_8 \\ l_7 : & \mathtt{ADD}(r_j), l_6, l_6 \\ l_8 : & \mathtt{SUB}(r_j), l_9, l_9 \\ l_9 : & \mathtt{SUB}(r_j), l_{10}, l_{10} \\ l_{10} : & \ldots \end{cases}$$

Reset r_i to 0, then move the value of r_j to both r_i and r_{22} (an auxiliary register), then decrement r_i, then move the value of r_{22} to r_j

Translating WHILE programs to Register Machines programs

- While statement:

$$T_{W\to RM}(\texttt{while}\ \ x_i \neq 0\ \ \texttt{do}\ \ C) = \begin{cases} l_0: & \texttt{SUB}(r_i), l_1, l_{n+3} \\ l_1: & \texttt{ADD}(r_i), l_2, l_2 \\ l_2: & C_1 \\ l_3: & C_2 \\ \dots & \\ l_{n+1}: & C_n \\ l_{n+2}: & \texttt{SUB}(r_{22}), l_0, l_0 \\ l_{n+3}: & \dots \end{cases}$$

From WHILE programs to SN P systems

Examples

Grammars can be checked out at `whilecompiler/grammars`

1) While → Registers Machine language

Using the library it is straightforward to translate a WHILE program into a registers machine program, as the example shows:

```
from whilecompiler import translator

program = """
begin
    //This is a comment
    x_0 = 1;
    x_1 = 3;
    x_0 = x_0 + 1;
end
"""

compiled = translator.while_to_rm(program, as_string = True)
print(compiled)
```

Gives as output:

```
SUB(0), 0, 1
ADD(0), 2, 2
SUB(1), 2, 3
ADD(1), 4, 4
ADD(1), 5, 5
ADD(1), 6, 6
ADD(0), 7, 7
```

From WHILE programs to SN P systems

2) While → Spiking Neural P-System

It is really easy and straightforward to build SN P-System from a WHILE language code.

```
rm_model = translator.while_to_rm(while_program)

psystem = translator.rm_to_psystem(rm_model)

print(psystem)
```

The output of this code is

```
P-System containing 36 neurons, 78 synapses and 23 registers.
```

From WHILE programs to SN P systems

3) Simulating the SN P-System

There are two ways to simulate the generated P-System. The first one is done using this library and it is about calling the `simulate()` function available in the `SNPSystem` class. An example is given:

```
rm_model = translator.while_to_rm(while_program)

psystem = translator.rm_to_psystem(rm_model)

psystem.simulate()
```

From WHILE programs to SN P systems

Another way to simulate the SN P-System is through the P-Lingua software using a simulator described in [2]. The library is able to convert a `SNPSystem` object into a `.pli` file (which is a description of the system, consisting of neurons, synapses and rules) that is executable by P-Lingua.

```
rm_model = translator.while_to_rm(while_program)

psystem = translator.rm_to_psystem(rm_model)

psystem.export_to_pli('exported.pli')
```

Then, we are able to simulate the system using the following command:

```
!java -jar plinguacore4.jar plingua_sim -PLI exported.pli -o output_report.txt;
```

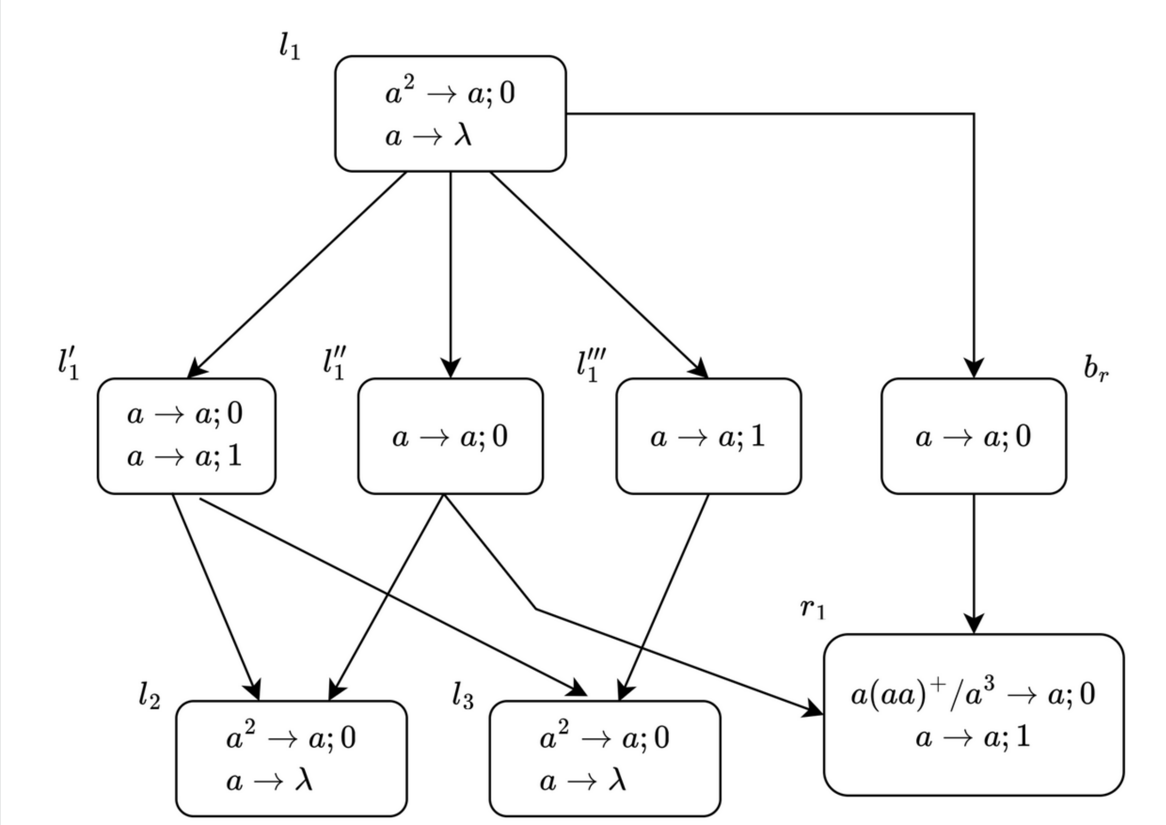
Notice that we need Java installed and `plinguacore4.jar`, which is available here (optionally, it is possible to find it in `utils` folder of this repository).

In this case, it is useful to check the output report and see the contents of the registers neurons.

From WHILE programs to SN P systems

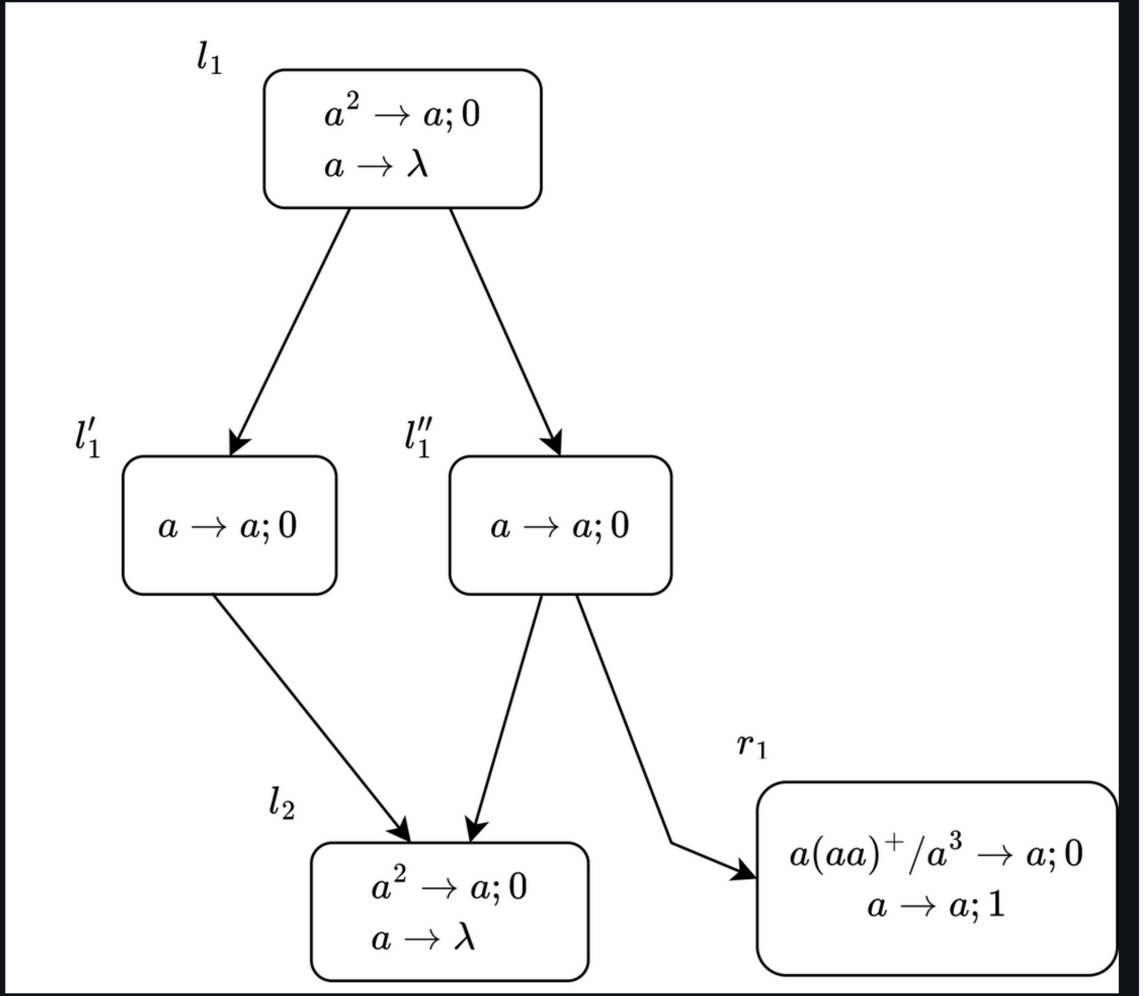
The translation from a registers machine program to a SN P-System has been made according to the following rules (refer to [2] for a in-depth explaination)

- ADD(1), 2, 3: add one to register 1 and jump non deterministically to label l_2 or l_3 (this behavior has been simulated using a random jump). Notice that the two labels are not equal.



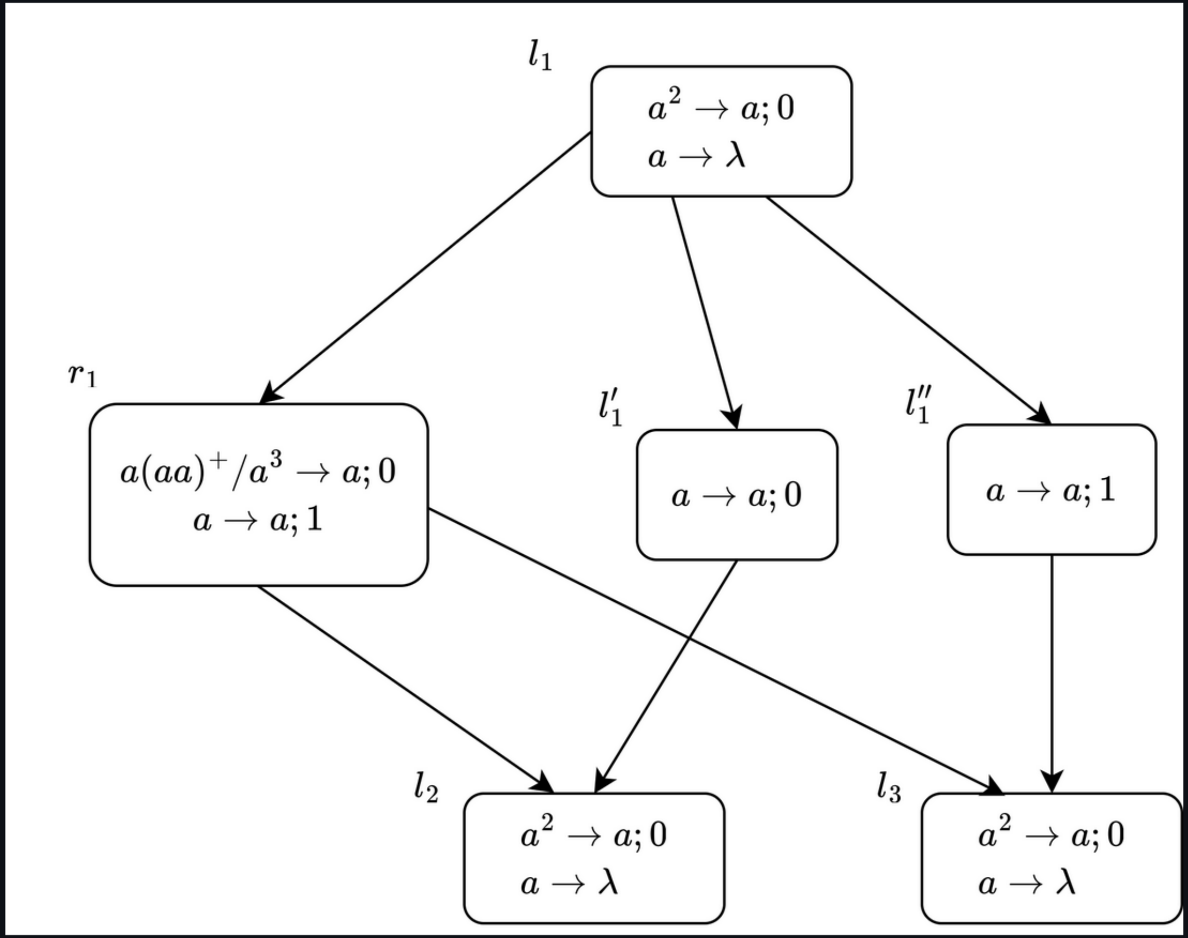
From WHILE programs to SN P systems

- ADD(1), 2, 2: add one to register 1 and jump deterministically to l_2.



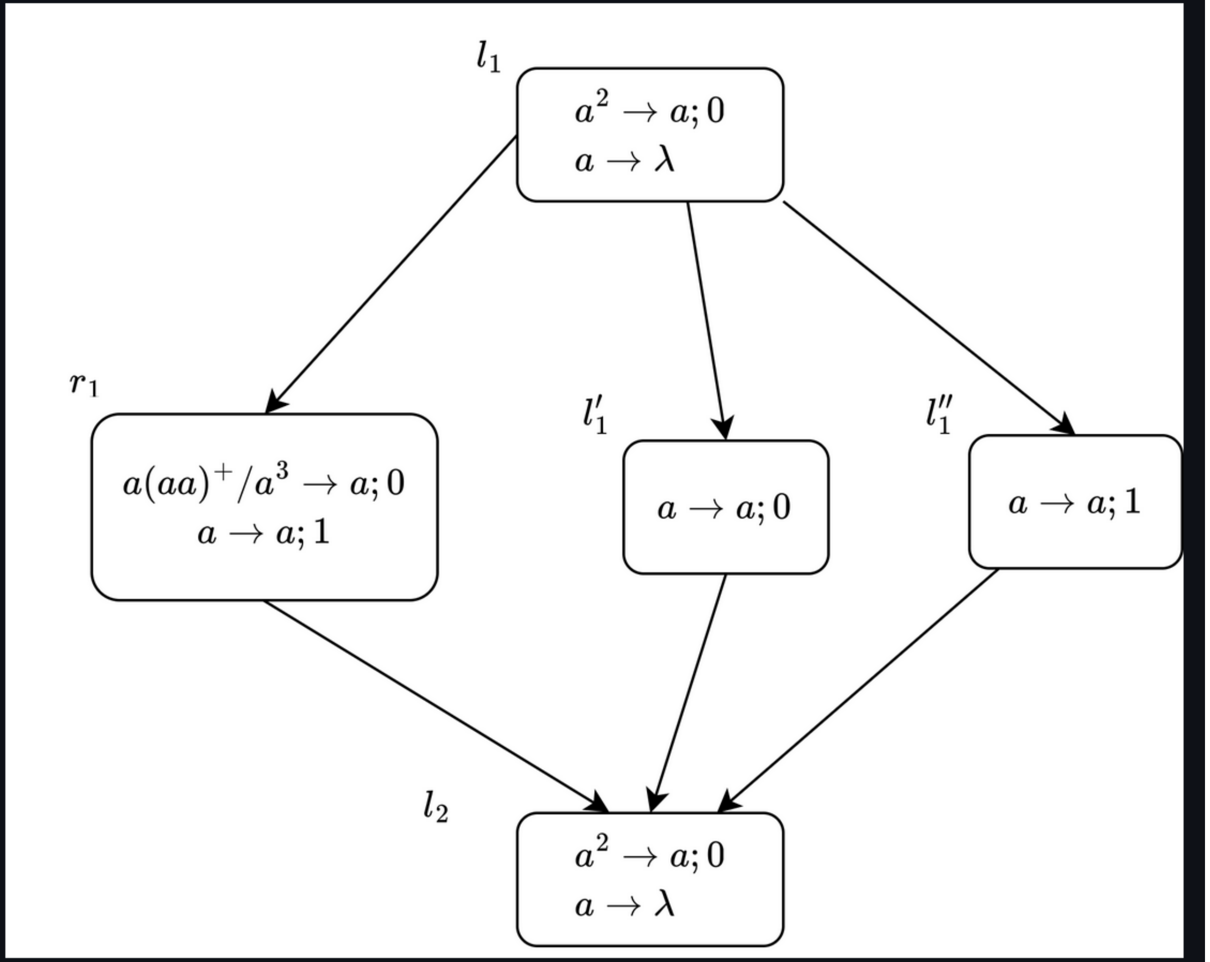
From WHILE programs to SN P systems

- SUB(1), 2, 3: subtracts one from register 1 and jump to l_2 if r_1 was not empty. Jump to l_3 otherwise



From WHILE programs to SN P systems

- SUB(1), 2, 2: subtracts one from register 1 and jump to l_2 in any case.



Using existing macros

Existent macros are defined in `whilecompiler/macros/std.wp`, and it is possible to use them by importing the file, just write `import macros/std.wp` on top of your WHILE program.

For instance, the following program uses the '+' operator:

```
include "macros/std.wp";
Program example;
input x_0, x_1;
begin
    x_2 = x_0 + x_1;
end
output x_2;
```

The next example, on the other hand, uses a function:

```
include "macros/std.wp";
Program example;
input x_0;
begin
    assign(x_2, x_0);
end
output x_2;
```

Creating new macros

It is also possible to define new macros. In order to do that, create a `.wp` file in the same folder of your program. For instance, `macroexamples.wp`. The compiler will look for custom macros in the local path.

A macro can be defined as a function (witn n arguments):

```
def macro function assign (x_a x_b) {
    x_a = x_b + 1;
    x_a = x_a - 1;
}
```

or as a binary operator (with three arguments)

```
def macro operator '+' (x_a, x_b, x_c) {
    x_22 = x_b + 1;
    x_23 = x_c + 1;

    while x_23 != 0 do
        begin
            x_22 = x_22 + 1;
            x_23 = x_23 - 1;
        end

    x_22 = x_22 - 1;
    x_a = x_22 - 1;

    x_22 = 0;
    x_23 = 0;
}
```

Creating new macros

You can check the grammar in `whilecompiler/grammars/while_macro.tx`. The body of a macro will be then parsed as a standard WHILE program.

The compiler will replace 'x_a', 'x_b' etc. with the contents of the program. For instance, by writing `assign(x_1, x_2)`, the macro defined above will replace `x_a` with `x_1`, and `x_b` with `x_2`.

It is a good practice to reset the temporary registers used in macros and to put the result in the first argument.

Future work

- Improve macro management
- Implement input and output spike trains
- Extend the technique to other computational models
- Other possible natural extensions/alternatives:
  - Using a more sophisticated/expressive language
    - Programs would be easier to write, but the compiler would be harder to write
  - What about a concurrent programming language?
    - Inspired from Occam?

Thank you
for your attention !

Alberto Leporati and **Lorenzo Rovida**

alberto.leporati@unimib.it, lorenzo.rovida@unimib.it