Implementing in Prolog an Effective Cellular Solution to the Knapsack Problem

Andrés Cordón-Franco, Miguel A. Gutiérrez-Naranjo, Mario J. Pérez-Jiménez, Agustin Riscos-Núñez, and Fernando Sancho-Caparrini

Abstract. In this paper we present an implementation in Prolog of an effective solution to the Knapsack problem via a family of deterministic P systems with active membranes using 2-division.

1 Introduction

The aim of this work is to present an effective solution to the Knapsack problem through a simulator, written in Prolog, implementing deterministic P systems with active membranes using 2-division.

The different variants of P systems found in the literature are generally thought as generating devices, and many of them have been proved to be computationally complete. Nevertheless, it is not usual to find in these variants effective solutions to hard numerical problems. The model we study here, P systems with active membranes (see [4], section 7.2.), works with symbol–objects, and one of its most relevant features is that it provides rules for division of membranes. These rules enable the creation of an exponential workspace in linear time, allowing thus the design of efficient cellular solutions.

The paper is organized as follows: Section 2 briefly presents a linear-time solution to the decision Knapsack problem using a family of deterministic P systems with active membranes; Section 3 gives some ideas about the simulator developed in Prolog used to implement this solution; a standard work session with the interface provided with the simulator is included in Section 4; finally, in Section 5 some conclusions and future work about the subject of this paper are presented.

2 Solving the Decision Knapsack Problem in Linear Time

The decision Knapsack problem can be stated as follows:

> *Given a knapsack of capacity $k \in \mathbf{N}$, a set A of n elements, where each element has a "weight" $w_i \in \mathbf{N}$ and a "value" $v_i \in \mathbf{N}$, and a constant $c \in \mathbf{N}$, decide whether or not there exists a subset of A such that its weight does not exceed k and its value is greater than or equal to c.*

We will represent the instances of the problem using tuples of the form $(n, (w_1, \ldots, w_n), (v_1, \ldots v_n), k, c)$, where n is the size of the set A; $(w_1, \ldots, w_n)$ and $(v_1, \ldots v_n)$ are the weights and values, respectively, of the elements of A; moreover, k and c are the two bound constants. We can define in a natural way additive functions w and v that correspond to the data in the instance.

A family of language recognizer P systems with active membranes using 2-division, without cooperation nor dissolution nor priority among rules solving this problem is presented in [5]. We refer the reader to this paper for a complete understanding of the process followed by this family, and for the formal verification that this family in fact solves the Knapsack problem.

The family presented there is

$$\Pi = \{(\Pi(\langle n, c, k\rangle), \Sigma(n, c, k), i(n, c, k)) : (n, c, k) \in \mathbb{N}^3\},$$

where $\langle n, c, k\rangle$ is a bijection from $\mathbb{N}^3$ to $\mathbb{N}$ induced by the pair function $\langle x, y\rangle = [(x + y)(x + y + 1)/2] + x$ namely, $\langle n, c, k\rangle = \langle\langle n, c\rangle, k\rangle$; the input alphabet is $\Sigma(n, c, k) = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$; the input membrane is $i(n, c, k) = e$ and the P system

$$\Pi(\langle n, c, k\rangle) = (\Gamma(n, c, k), \{e, s\}, \mu, \mathcal{M}_s, \mathcal{M}_e, R)$$

is defined as follows:

- Working alphabet:
  $\Gamma(n, c, k) = \{a_0, a, \bar{a}_0, \bar{a}, b_0, b, \bar{b}_0, \bar{b}, \hat{b}_0, \hat{b}, d_+, d_-, e_0, \ldots, e_n, q_0, \ldots, q_{2k+1},$
  $q, \bar{q}, \bar{q}_0, \ldots, \bar{q}_{2c+1}, x_0, \ldots, x_n, y_0, \ldots, y_n, yes, no, z_0, \ldots, z_{2n+2k+2c+6}, \#\}$.
- Membrane structure: $\mu = [_s\ [_e\ ]_e\ ]_s$.
- Initial multisets: $\mathcal{M}_s = z_0$; $\mathcal{M}_e = e_0\, \bar{a}^k\, \bar{b}^c$.
- The set of evolution rules, R, consists of the following rules:

(a) $[_e e_i]^0_e \to [_e q]^-_e [_e e_i]^+_e$, for $i = 0, \ldots, n$.
$[_e e_i]^+_e \to [_e e_{i+1}]^0_e [_e e_{i+1}]^+_e$, for $i = 0, \ldots, n-1$.

(b) $[_e x_0 \to \bar{a}_0]^0_e$; $[_e x_0 \to \epsilon]^+_e$; $[_e x_i \to x_{i-1}]^+_e$, for $i = 1, \ldots, n$.
$[_e y_0 \to \bar{b}_0]^0_e$; $[_e y_0 \to \epsilon]^+_e$; $[_e y_i \to y_{i-1}]^+_e$, for $i = 1, \ldots, n$.

(c) $[_e q \to \bar{q} q_0]^-_e$; $[_e \bar{a}_0 \to a_0]^-_e$; $[_e \bar{a} \to a]^-_e$; $[_e \bar{b}_0 \to \hat{b}_0]^-_e$; $[_e \bar{b} \to \hat{b}]^-_e$.

(d) $[_e a_0]^-_e \to [_e]^0_e \#$; $[_e a]^0_e \to [_e]^-_e \#$.

(e) $[_e q_{2j} \to q_{2j+1}]^-_e$, for $j = 0, \ldots, k$; $[_e q_{2j+1} \to q_{2j+2}]^0_e$, for $j = 0, \ldots, k-1$.

(f) $[_e q_{2j+1}]^-_e \to [_e]^+_e \#$, for $j = 0, \ldots, k$.

(g) $[_e \bar{q} \to \bar{q}_0]^+_e$; $[_e \hat{b}_0 \to b_0]^+_e$; $[_e \hat{b} \to b]^+_e$; $[_e a \to \epsilon]^+_e$.

(h) $[_e b_0]^+_e \to [_e]^0_e \#$; $[_e b]^0_e \to [_e]^+_e \#$.

(i) $[_e \bar{q}_{2j} \to \bar{q}_{2j+1}]^+_e$, for $j = 0, \ldots, c$; $[_e \bar{q}_{2j+1} \to \bar{q}_{2j+2}]^0_e$, for $j = 0, \ldots, c-1$.

(j) $[_e \bar{q}_{2c+1}]^+_e \to [_e]^0_e yes$; $[_e \bar{q}_{2c+1}]^0_e \to [_e]^0_e yes$.

(k) $[_s z_i \to z_{i+1}]^0_s$, for $i = 0, \ldots, 2n + 2k + 2c + 5$; $[_s z_{2n+2k+2c+6} \to d_+ d_-]^0_s$.

(l) $[_s d_+]^0_s \to [_s]^+_s d_+$; $[_s d_- \to no]^+_s$; $[_s yes]^+_s \to [_s]^0_s yes$; $[_s no]^+_s \to [_s]^0_s no$.

Notice that the membrane labelled with e is the input membrane where the weights and values are provided. This is done via a multiset over the alphabet Σ that is introduced into membrane e before starting the computation. Therefore, at the beginning, the multiplicities of objects x_j and y_j (with $1 \leq j \leq n$) encode the weights and the values, respectively, of the corresponding elements of A. The multiplicities of $\bar{a}$ and $\bar{b}$ encode the constants k and c, respectively, while the weight and value of the subset associated with each membrane will be encoded as the multiplicity of objects $\bar{a}_0$ and $\bar{b}_0$, respectively. The other elements of the working alphabet are used due to technical reasons.

3 The Simulator

We use a new version of the P system simulator presented in [2]. This simulator is written in Prolog[1]. This language is expressive enough to handle symbolic knowledge in a natural way and to deal with complex structures, in our case, configurations of P systems.

On one hand, the tree-based data structure and the use of infix operators defined *ad hoc* by the programmer allow us to *imitate* the natural language and the user can follow the evolution of the system without any knowledge of Prolog. On the other hand, the structure of membranes and the rules of the systems are Prolog facts and the design of the inference engine that performs the evolutions of the system has a natural treatment from a programmer point of view.

In the current version[2], the simulator has two different parts. First, we can consider the inference engine. It is a Prolog program which takes as input an initial configuration of a P system and a set of rules, and carries out the evolution. The simulator only explores one branch of the computation tree; therefore, a necessary condition to ensure a faithfully evolution of the system is dealing with a *confluent* P system (i.e., all the computations with the same initial configuration must give the same output). Let us emphasize that the inference engine is completely general, that is, it does not depend on the P system considered at all. The second part of the simulator provides a generator tool to automatically build the initial configuration and the set of the rules for concrete instances of some well known **NP**–complete problems (e.g., SAT, Validity, Subset Sum and, of course, the Knapsack problem). The design of this generator tool allows future additions of specific modules for new problems. These modules could be easily designed by any Prolog user for carrying out his own experiments.

The new version of the simulator improves the previous one. The main improvements are related to programming techniques. From an user point of view, the new version provides an easier way of introducing the parameters and offers a broader set of type of problems to solve. Nevertheless, the formal representation

[1] A good starting point for Prolog can be [1] or [8].

[2] Available from the website `http://www.cs.us.es/gcn`

in Prolog of the basic structures of P systems is the same as the one introduced in [2].

In order to clarify the format used for representing the configurations and rules of the system, let us remember that a given configuration for the membrane structure will be expressed by means of a labelled tree. In short, [ ] is the *position* to denote the root of the tree and it will be associated with the skin; then [i] will denote the *position* of the i-th inner membrane to the skin.

In a general form, to denote that in the t-th step of its evolution, a P system P has a membrane at position [pos] with label h, polarity α, and m as multiset, we shall write

$$P \texttt{ :: } h \texttt{ ec } \alpha \texttt{ at } [pos] \texttt{ with } m \texttt{ at\_time } t\texttt{.}$$

The rules are also represented as literals, according to the following format:

- $[_h x \rightarrow y]_h^\alpha$
  P `rule` x `evolves_to` $[y]$ `in` h `ec` α`.`
- $x[_h\ ]_h^{\alpha_1} \rightarrow [_h y]_h^{\alpha_2}$
  P `rule` x `out_of` h `ec` α_1 `sends_in` y `of` h `ec` α_2`.`
- $[_h x]_h^{\alpha_1} \rightarrow [_h\ ]_h^{\alpha_2} y$
  P `rule` x `inside_of` h `ec` α_1 `sends_out` y `of` h `ec` α_2`.`
- $[_h x]_h^{\alpha_1} \rightarrow [_h y]_h^{\alpha_2} [_h z]_h^{\alpha_3}$
  P `rule` x `inside_of` h `ec` α_1 `divides_into` y `inside_of` h `ec` α_2
  `and` z `inside_of` h `ec` α_3`.`

It is worth pointing out that each rule is indeed a Prolog fact. Several operators (such as `inside_of`, `divides_into`, ...) have been defined *ad hoc* in order to obtain a natural-language-like appearance. The simulator also deals with dissolution rules although they are not used in the presented solution to solve the Knapsack problem.

4 A Prolog Session

In this section we show a session for one instance of the problem. We consider a set $A = \{a_1, a_2, a_3, a_4\}$ with four elements ($n = 4$), with weights $w(a_1) = 3$, $w(a_2) = 2$, $w(a_3) = 3$, $w(a_4) = 1$, and values $v(a_1) = 1$, $v(a_2) = 3$, $v(a_3) = 3$, $v(a_4) = 2$. The question is to decide whether or not there exists $B \subseteq A$ such that the weights of the elements in B do not exceed 3 ($k = 3$) and their values are greater than 4 ($c = 4$).

According to the design presented in [5], the P system that solves this instance is $\Pi(\langle 4, 3, 4\rangle)$ with input $x_1^3\, x_2^2\, x_3^3\, x_4\, y_1\, y_2^3\, y_3^3\, y_4^2$. The initial configuration associated with the previous instance of the problem is shown in Fig. 1. The following two Prolog facts allow us to represent[3] this initial configuration (we have chosen the name `p1` to denote the P system that solves the above instance of the problem).

[3] Note that we use ASCII symbols to represent the objects of the alphabet; e.g., `a_` stands for $\bar{a}$, `b0g` stands for $\hat{b}_0$, and so on.

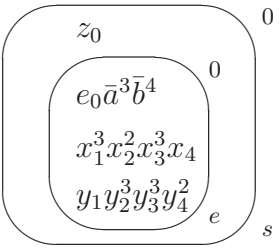

Fig. 1. Initial configuration

```
p1 :: s ec 0 at [] with [z0] at_time 0.
p1 :: e ec 0 at[1] with [e0, a_, a_, a_, b_, b_, b_, b_,
                         x1, x1, x1, x2, x2, x3, x3, x3,
                         x4, y1, y2, y2, y2, y3, y3, y3,
                         y4, y4] at_time 0.
```

The simulator automatically generates them from the data typed by the user and stores them in a text file.

Observe that the multiplicities of the objects x_j and y_j correspond to the weights and values of the elements a_j, respectively. Also, there are three copies of $\bar{a}$ ($k = 3$) and four copies of $\bar{b}$ ($c = 4$).

In the same way as above, the simulator automatically generates the set of rules associated with the instance of the problem. This generation is done by instantiating several schemes of rules to the concrete values of the parameters. This produces a text file containing the rules that can be easily edited, modified or reloaded by the user. We would like to remark that the set of rules only depends on the parameters n, k, and c. Consequently, if we want to solve several instances with the same parameters, we only need to generate the set of rules once.

In this example, we have obtained 89 rules. Some of them are listed in the Appendix.

To start with the simulation of the evolution of the P system `p1` from the time 0, we type the following command:

```
?- evolve(p1,0).
```

The simulator returns the configuration at time 1 and the set of used rules indicating how many times they have been used. Moreover, if the skin sends out any object, then this will be reported to the user. The multisets are represented as lists of pairs `obj`–`n`, where `obj` is an object and `n` is the multiplicity of `obj` in the multiset.

```
?- evolve(p1,0).

p1 :: s ec 0 at [] with [z1-1] at_time 1
```

```
p1 :: e ec -1 at [1] with [a_-3, b_-4, q-1, x1-3, x2-2, x3-3,
                           x4-1, y1-1, y2-3, y3-3, y4-2] at_time 1
p1 :: e ec 1 at [2] with [a_-3, b_-4, e0-1, x1-3, x2-2, x3-3,
                           x4-1, y1-1, y2-3, y3-3, y4-2] at_time 1

Used rules in the step 1:
  * The rule 1 has been used only once
  * The rule 57 has been used only once
```

In this step only rules 1 and 57 have been applied. Rule 1 is a division rule, the membrane labelled by e at position `[1]` divides into two membranes that are placed at positions `[1]` and `[2]`; rule 57 is an evolution rule, z_0 evolves to z_1 in the skin membrane. To obtain the next configuration in the evolution of `p1`, now we type:

```
?- evolve(p1,1).

p1 :: s ec 0 at [] with [z2-1] at_time 2
p1 :: e ec -1 at [1] with [a-3, bg-4, q0-1, q_-1, x1-3,
           x2-2, x3-3, x4-1, y1-1, y2-3, y3-3, y4-2] at_time 2
p1 :: e ec 0 at [2] with [a_-3, b_-4, e1-1, x0-3, x1-2,
           x2-3, x3-1, y0-1, y1-3, y2-3, y3-2] at_time 2
p1 :: e ec 1 at [3] with [a_-3, b_-4, e1-1, x0-3, x1-2,
           x2-3, x3-1, y0-1, y1-3, y2-3, y3-2] at_time 2

Used rules in the step 2:
  * The rule 6 has been used only once
  * The rule 14 has been used 3 times
  * The rule 15 has been used 2 times
             ...
       evolution rules from set (b) (see Appendix)
             ...
  * The rule 22 has been used only once
  * The rule 25 has been used 4 times
  * The rule 26 has been used 3 times
  * The rule 58 has been used only once
```

In this step, the first relevant membrane (that is, one of the membranes having negative charge and containing an object `q0`) appears at position `[1]`. This membrane is associated with the empty set. The membrane at position `[2]` will continue dividing to generate new membranes. All the subsets associated with these descendent membranes will contain the object a_1. On the other hand, the membrane at position `[3]` is responsible of the membranes corresponding to all the nonempty subsets that do not contain object a_1.

Notice that rules from (b) have been applied. That is, the weight and values calculation stage has already begun. This stage takes place in parallel with the generation one. Immediately after an element a_j is added to the associated subset

(i.e., when an object ej appears in a membrane with neutral charge), $w(a_j)$ new copies of a0_ and $v(a_j)$ new copies of b0_ are generated in the membrane. This process is controlled by the rules in (b), according to the polarity changes produced by the division rules in (a).

The purpose of the system's first stage is to generate a single relevant membrane for each subset of A, i.e., $2^n = 2^4 = 16$ relevant membranes in all. Other non relevant membranes are generated as well due to technical reasons. This is done in the first $2n + 2 = 10$ steps of the computation.

The simulator also allows us to ask for a configuration at time t without showing the previous steps by typing the following command.

```
?- configuration(p1,4).
       ...
p1 :: e ec -1 at [2] with [a-3, a0-3, b0g-1, bg-4, q0-1, q_-1,
                x1-2, x2-3, x3-1, y1-3, y2-3, y3-2] at_time 4
       ...
```

In this step the second relevant membrane is obtained. Note that the electric charge is negative and the object q0 belongs to it. The subset associated with this membrane is the unitary subset $\{a_1\}$. Let us observe that there are three copies of a0 ($w(a_1) = 3$) and one copy of b0g ($v(a_1) = 1$).

```
?- evolve(p1,4).
       ...
p1 :: e ec  0 at [2] with [a-3, a0-2, b0g-1, bg-4, q1-1,
            q_-1, x1-2, x2-3, x3-1, y1-3, y2-3, y3-2] at_time 5
p1 :: e ec -1 at [3] with [a-3, a0-2, b0g-3, bg-4, q0-1,
            q_-1, x1-3, x2-1, y1-3, y2-2] at_time 5
       ...

Used rules in the step 5:
       ...
  * The rule 27 has been used only once
       ...
```

In this step, the membrane at position [2] starts the checking stage for the weight function w (note that rule 27 from the set (d) has been applied, see Appendix). A new relevant membrane appears at position [3]. The subset associated with this membrane is the unitary subset $\{a_2\}$. In the following steps new relevant membranes appear and their corresponding checking stages start.

```
?- configuration(p1,10).

        ...
p1 :: e ec -1 at [12] with [a-2, a0-2, b0g-5, bg-4, q2-1, q_-1]
                                                    at_time 10
        ...
```

```
p1 :: e ec -1 at [24] with [a-3, a0-9, b0g-9, bg-4, q0-1, q_-1]
                                                    at_time 10
        ...
```

In this step the relevant membrane associated with the total subset appears at position `[24]`. It is the last relevant membrane to be generated, that is, no more division will take place in the rest of the computation and no new relevant membranes will appear.

Let us focus on the membrane at position `[12]`. This membrane encodes the subset $\{a_2, a_4\}$, which is the only solution for the instance considered in our example. There have already been carried out two steps of the checking stage for w (note the counter q2).

```
?- configuration(p1,16).
            ...
p1 :: e ec 1 at [12] with [b0g-5, bg-4, q_-1] at_time 16
            ...
```

The checking stage is finished in this membrane. There was the same amount of objects `a` and `a0` so the checking has been successful. Next, a transition step is carried out that leads to the checking stage for the values. This stage mainly consists in applying rules 44 and 45 (see Appendix) alternatively until we run out of objects `b0` or `b`.

```
?- configuration(p1,25).
      ...
p1 :: e ec 1 at [12] with [b0-1, q_8-1] at_time 25
      ...
```

Notice that the checking for the values has been finished and it has been successful because the number of objects `b0` was greater than the number of objects `b`. Let us pay attention now to the *answer stage*.

```
?- evolve(p1,25).

p1 :: s ec 0 at [] with [# -127, z26-1] at_time 26
           ...
p1 :: e ec 0 at [12] with [q_9-1] at_time 26
           ...
```

The inner membrane at position `[12]` is now ready to send to the skin an object `yes` (see rule 56 in the Appendix).

```
?- evolve(p1,26).

p1 :: s ec 0 at [] with [# -127, yes-1, z27-1] at_time 27
           ...
```

```
p1 :: e ec 0 at [12] with [] at_time 27
           ...

Used rules in the step 27:
  * The rule 56 has been used only once
  * The rule 83 has been used only once
```

Due to technical reasons (cf. [5]) the counter in the skin will wait two more steps before releasing the special objects d_+ and d_-.

```
?- configuration(p1,29).

p1 :: s ec 0 at [] with [# -127, d1-1, d2-1, yes-1] at_time 29
       ...
```

In this step all inner processes are over. The only object that evolves is `z28` (see the set of rules (k) in Section 2 and note that $28 = 2n + 2k + 2c + 6$).

```
?- evolve(p1,29).

p1 :: s ec 1 at [] with [# -127, d2-1, yes-1] at_time 30
      ...

Used rules in the step 30:
  * The rule 86 has been used only once
The P-system has sent out d1 at step 30
```

In this step the object `d1` leaves the system and the skin gets positive charge.

```
?- evolve(p1,30).

p1 :: s ec 0 at [] with [# -127, no-1] at_time 31
        ...

Used rules in the step 31:
  * The rule 87 has been used only once
  * The rule 88 has been used only once
The P-system has sent out d1 at step 30
The P-system has sent out yes at step 31
```

In this step the object `yes` is sent out of the system. To check the system we try to evolve one more time, though this is a halting configuration.

```
?- evolve(p1,31).
No more evolution!
The P-system p1 has already reached a halting
configuration at step 31
```

5 Conclusions and Future Work

In this paper we have pointed out that P systems are suitable to solve efficiently numerical **NP**–complete problems. We have presented a family of deterministic P systems with active membranes using 2–division that solves the Knapsack problem in an *effective* and *efficient* way (in fact, in linear time). Furthermore, we have presented a simulation of a concrete instance of the problem using a new version of the Prolog simulator introduced in [2].

It is not easy to find examples in the literature related to effective solutions for **NP**–problems using P systems and a lot of work remains to be done in this field. Looking for general patrons and recycling subroutines in the algorithms are open problems.

Following this idea, an effective tool for simulation allows us to carry out faithful studies of the evolution of the system and to check, step by step, how the rules are applied.

Besides, the scope of the Prolog program that we have presented goes beyond the simulation tool itself. Both the chosen way to represent the knowledge about the system and the design of the inference engine lead us towards putting in touch P systems with other well known problems and techniques of computation and Artificial Intelligence which have a natural representation in Prolog. We think that this connection deserves to be deeply studied in the next future.

Acknowledgement. Support this research through the project TIC2002-04220-C03-01 of the Ministerio de Ciencia y Tecnología of Spain, cofinanced by FEDER funds, is gratefully acknowledged.

References

1. Bratko, I.: *PROLOG Programming for Artificial Intelligence*, Third edition. Addison-Wesley, 2001.
2. Cordón-Franco, A., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Sancho-Caparrini, F.: A Prolog simulator for deterministic P systems with active membranes, *New Generation Computing*. In Press.
3. Păun, G.: Computing with membranes, *Journal of Computer and System Sciences*, **61**(1), 2000, 108–143.
4. Păun, G.: *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
5. Pérez-Jiménez, M.J.; Riscos-Núñez, A.: A linear solution to the Knapsack problem by P systems with active membranes. In this volume.
6. Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini F.: A polynomial complexity class in P systems using membrane division. *5th Workshop on Descriptional Complexity of Formal Systems*, July 12–14, 2003, Budapest, Hungary.
7. The P Systems Web Page: `http://psystems.disco.unimib.it/`
8. Logic Programming: `http://www.afm.sbu.ac.uk/logic-prog/`

Appendix

In what follows, we show the rules generated by the simulator for the instance of the problem considered in section 4. Note that the number after ** is the ordinal of the corresponding rule.

```
% Set (a)
p1 rule e0 inside_of e ec 0 divides_into q inside_of e ec -1
   and e0 inside_of e ec 1 ** 1.
            ...
p1 rule e4 inside_of e ec 0 divides_into q inside_of e ec -1
   and e4 inside_of e ec 1 ** 5.

p1 rule e0 inside_of e ec 1 divides_into e1 inside_of e ec 0
   and e1 inside_of e ec 1 ** 6.
            ...
p1 rule e3 inside_of e ec 1 divides_into e4 inside_of e ec 0
   and e4 inside_of e ec 1 ** 9.

% Set (b)
p1 rule x0 evolves_to [a0_] in e ec 0 ** 10.

p1 rule y0 evolves_to [b0_] in e ec 0 ** 11.

p1 rule x0 evolves_to [] in e ec 1 ** 12.

p1 rule y0 evolves_to [] in e ec 1 ** 13.

p1 rule x1 evolves_to [x0] in e ec 1 ** 14.
             ...
p1 rule x4 evolves_to [x3] in e ec 1 ** 17.

p1 rule y1 evolves_to [y0] in e ec 1 ** 18.
            ...
p1 rule y4 evolves_to [y3] in e ec 1 ** 21.

% Set (c)
p1 rule q evolves_to [q_, q0] in e ec -1 ** 22.

p1 rule b0_ evolves_to [b0g] in e ec -1 ** 23.

p1 rule a0_ evolves_to [a0] in e ec -1 ** 24.

p1 rule b_ evolves_to [bg] in e ec -1 ** 25.

p1 rule a_ evolves_to [a] in e ec -1 ** 26.
```

```
% Set (d)
p1 rule a0 inside_of e ec -1 sends_out # of e ec 0 ** 27.

p1 rule a inside_of e ec 0 sends_out # of e ec -1 ** 28.

% Set (e)
p1 rule q0 evolves_to [q1] in e ec -1 ** 29.

p1 rule q2 evolves_to [q3] in e ec -1 ** 30.

p1 rule q4 evolves_to [q5] in e ec -1 ** 31.

p1 rule q6 evolves_to [q7] in e ec -1 ** 32.

p1 rule q1 evolves_to [q2] in e ec 0 ** 33.

p1 rule q3 evolves_to [q4] in e ec 0 ** 34.

p1 rule q5 evolves_to [q6] in e ec 0 ** 35.

% Set (f)
p1 rule q1 inside_of e ec -1 sends_out # of e ec 1 ** 36.

p1 rule q3 inside_of e ec -1 sends_out # of e ec 1 ** 37.

p1 rule q5 inside_of e ec -1 sends_out # of e ec 1 ** 38.

p1 rule q7 inside_of e ec -1 sends_out # of e ec 1 ** 39.

% Set (g)
p1 rule q_ evolves_to [q_0] in e ec 1 ** 40.

p1 rule b0g evolves_to [b0] in e ec 1 ** 41.

p1 rule bg evolves_to [b] in e ec 1 ** 42.

p1 rule a evolves_to [] in e ec 1 ** 43.

% Set (h)
p1 rule b0 inside_of e ec 1 sends_out # of e ec 0 ** 44.

p1 rule b inside_of e ec 0 sends_out # of e ec 1 ** 45.

% Set (i)
```

```
p1 rule q_0 evolves_to [q_1] in e ec 1 ** 46.

p1 rule q_2 evolves_to [q_3] in e ec 1 ** 47.

p1 rule q_4 evolves_to [q_5] in e ec 1 ** 48.

p1 rule q_6 evolves_to [q_7] in e ec 1 ** 49.

p1 rule q_8 evolves_to [q_9] in e ec 1 ** 50.

p1 rule q_1 evolves_to [q_2] in e ec 0 ** 51.

p1 rule q_3 evolves_to [q_4] in e ec 0 ** 52.

p1 rule q_5 evolves_to [q_6] in e ec 0 ** 53.

p1 rule q_7 evolves_to [q_8] in e ec 0 ** 54.

% Set (j)
p1 rule q_9 inside_of e ec 1 sends_out yes of e ec 0 ** 55.

p1 rule q_9 inside_of e ec 0 sends_out yes of e ec 0 ** 56.

% Set (k)
p1 rule z0 evolves_to [z1] in s ec 0 ** 57.

p1 rule z1 evolves_to [z2] in s ec 0 ** 58.
            ...
p1 rule z26 evolves_to [z27] in s ec 0 ** 83.

p1 rule z27 evolves_to [z28] in s ec 0 ** 84.

p1 rule z28 evolves_to [d1, d2] in s ec 0 ** 85.

% Set (l)
p1 rule d1 inside_of s ec 0 sends_out d1 of s ec 1 ** 86.

p1 rule d2 evolves_to [no] in s ec 1 ** 87.

p1 rule yes inside_of s ec 1 sends_out yes of s ec 0 ** 88.

p1 rule no inside_of s ec 1 sends_out no of s ec 0 ** 89.
```