Towards a Programming Language in Cellular Computing

Miguel Angel GUTIÉRREZ-NARANJO
Mario J. PÉREZ-JIMÉNEZ
Agustín RISCOS-NÚÑEZ

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: {magutier, marper, ariscosn}@us.es

Abstract. Several solutions to hard numerical problems using P systems have been presented recently, and strong similarities in their designs have been noticed. In this paper we present a new solution, an effective one to the Partition problem via a family of deterministic P systems with active membranes using 2-division. We intend to show that the idea of a *cellular programming language* is possible, indicating some "subroutines" that can be used in a variety of situations and therefore could be useful for attacking new problems in the future.

1 Introduction

Cellular Computing is an emergent branch in the field of Natural Computing. Since Gh. Păun introduced it (see [4]) much work has been done, from various points of view. Computer scientists, biologists, formal linguists and complexity theoreticians have contributed enriching the field with their different approaches.

The present paper is focused on the design of a family of P systems that solves a numerical NP-complete problem, namely, the Partition problem. The design of this solution is inspired in several previous works on other problems, mainly the Subset-Sum and the Knapsack problems, but also the VALIDITY and SAT. The similarities between the design introduced here and the solutions presented in [6], [7], [9] and [11] will be highlighted and some conclusions will be extracted from them.

The paper is organized as follows: first some preliminary ideas about the framework of *Complexity classes* are introduced in the next section; then, in section 3 a cellular solution for the Partition problem is presented, and some comments about the possibility of generalizing the design are given in section 4; and the conclusions and future work lines are given in section 5.

2 Preliminaries

We assume that the reader is familiar with the general notion of P systems (a detailed description can be found at [3]). However, we will comment some details about the specific variant that we are using.

Recall that a decision problem, X, is a pair (I_X, θ_X) such that I_X is a language over a finite alphabet (whose elements are called *instances*) and θ_X is a total boolean function over I_X. That is, the answer to each instance of the problem will be either TRUE or FALSE. This is why we are interested in using a computing device that is able to receive an input, process it, and deliver a boolean answer.

In our case, we have chosen the class of P systems with input and with external output (special objects Yes and No will be used to implement the boolean answer). In order to obtain a significant speed-up, we will work in the active membrane model, and so we are allowed to use membrane division to obtain an exponential workspace in polynomial time. We impose also some restrictions, for instance, we want the systems to be *confluent* (all computations with the same input lead to the same output), also every computation must be finite and, furthermore, we want that the answer is delivered in the last step of the computation. These conditions, together with other considerations, are deeply studied in [12], where a slightly different design (improving the efficiency) of the family of P systems that solves the Partition problem is presented, in the framework of *complexity classes* in P systems.

We will not get into details here, but please note that we will say that the family of P systems presented *solves* the Partition problem according to the formal definition presented in [12].

3 Solving the Partition Problem in Linear Time

The *Partition* problem can be stated as follows:

> *Given a set A of n elements, where each element has a "weight" $w_i \in \mathbf{N}$, decide whether or not there exists a partition of A into two subsets such that they have the same weight.*

We will represent the instances of the problem using tuples of the kind $(n, (w_1, \dots, w_n))$, where n is the size of the set A and $(w_1, \dots, w_n)$ is the list of weights of the elements from A. We can define in a natural way an additive function w that corresponds to the data in the instance.

We will address the resolution of the problem via a brute force algorithm. The strategy can be roughly split into the following subgoals:

- *Generation stage*: use membrane division to get one membrane for each subset.

- *Calculation stage*: compute in each membrane the weight of its associated subset and the weight of its complementary.

- *Checking stage*: compare in each membrane $w(B)$ with $w(B^c)$, where B is the associated subset.

- *Output stage*: the answer is delivered according to the results of the checkings.

The family presented here is

$$\mathbf{\Pi} = \{(\Pi(n), \Sigma(n), i(n)) : n \in \mathbf{N}\}.$$

For each element of the family, the input alphabet is $\Sigma(n) = \{x_1, \ldots, x_n\}$, the input membrane is always the same, $i(n) = e$, and the P system $\Pi(n) = (\Gamma(n), \{e, r, s\}, \mu, \mathcal{M}_e, \mathcal{M}_r, \mathcal{M}_s, R)$ is defined as follows:

- Working alphabet:

$$\Gamma(n) = \{a_0, a, b_0, b, c, d_0, d_1, d_2, e_1, \ldots, e_n, g, \bar{g}, \hat{g}, \check{g}, h_0, h_1, i_1, i_2, i_4, i_5,$$
$$p, \bar{p}, q, x_1, \ldots, x_n, Yes, No, No_0, z_1, \ldots, z_{2n+1}, \#\}.$$

- Membrane structure: $\mu = [\ [\ ]_e\ [\ ]_r\ ]_s$.

- Initial multisets: $\mathcal{M}_e = e_0 g$; $\mathcal{M}_r = b_0 h_0$ and $\mathcal{M}_s = z_1$.

- The set of evolution rules, R, consists of the following rules:

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

The goal of these rules is to generate one membrane for each subset of A. Indeed, exactly the same two schemes of rules are used for the generation stages in the Subset-Sum and the Knapsack case. Here is how these rules work: in each step (according to the index of e_i), we consider an element of A and either we add it to the subset associated with the membrane, B, or we put it in the complementary subset, B^c. Note that a membrane can proceed to the checking stage only after it gets a negative charge; a positively charged membrane where the object e_n appears will get blocked (it will be dissolved, see rules in (i)).

(b) $[x_0 \to a_0]_e^0$, $[x_0 \to \bar{p}]_e^+$,
$[x_i \to x_{i-1}]_e^+$, for $i = 1, \ldots, n$,
$[x_i \to \bar{p}]_e^-$, for $i = 1, \ldots, n$.

At the beginning, the multiplicities of the objects x_j (with $1 \leq j \leq n$) encode the weights of the corresponding elements of A. They are not present in the definition of the system, but they are inserted as input in the membrane labelled by e before starting the computation: for each $a_j \in A$, w_j copies of x_j have to be added to the input membrane. During the computation, at the same time as elements are added to the subset associated with a membrane, objects a_0 and $\bar{p}$ are generated to store the weight of such subset and of its complementary. Again, these schemes of rules are almost identical to the ones used for the calculation stages of the Subset-Sum and Knapsack, the only difference is that here the weight of the complementary is kept, and there it was just removed.

It is worth noticing that the index-rotation technique was already used in [9] and [11] to deal with the set of variables in an ordered manner, even though there was no weight calculation there.

(c) $[q \to i_1]_e^-$, $[\bar{p} \to p]_e^-$, $[a_0 \to a]_e^-$.

When a membrane gets negatively charged, the two first stages (i.e. generation and calculation stages) end, and then some transition rules are applied. Objects a_0 and $\bar{p}$, whose multiplicities encode the weights of the associated subset, $w(B)$, and of its complementary, $w(B^c)$, are renamed for the next stage, when their multiplicities are compared. Similar renaming rules can be found in the designs of the Subset-Sum and Knapsack solutions; they are useful to avoid conflicts between rules from the checking stage and rules from the generation and calculation stages.

(d) $[a]_e^- \to [\ ]_e^0\#, \quad [p]_e^0 \to [\ ]_e^-\#.$

These rules implement the comparison above mentioned (that is, they check whether $w(B) = w(B^c)$ holds or not). They work as a loop that erases objects a and p one by one alternatively, changing the charge of the membrane in each step. Exactly the same method can be used to compare the multiplicities of whatever two objects of the working alphabet, so again we find rules that might be re-used when attacking other numerical problems.

(e) $[i_1 \to i_2]_e^-, \quad [i_2 \to i_1]_e^0.$

A marker that controls the previous loop is described here. The index of i_j and the electric charge of the membrane give enough information to point out if the number of objects a is greater than (less than or equal to) the number of objects p.

Here we find the first important difference in the design with respect to the ones for Subset-Sum and Knapsack. There counters were used, and the schemes of rules depended on the number of steps that the checking was going to last. But now the number of steps depends on the total weight of the set A, and we cannot use this information if we want an uniform design. However, there are good news: the rules used here can be used also in general, so new versions of the solutions to Subset-Sum and Knapsack using this subroutine can be given.

(f) $[i_1]_e^0 \to [\ ]_e^+ No.$

This rule, together with the ones in the next item, take care of the result of the checking. If a subset $B \subseteq A$ verifies that $w(B) > w(B^c)$, then at the end of the calculation stage there will be less objects p than a inside the membrane associated with it. This forces the loop described in (e) to halt: the moment will come when there are no objects p left, and then the rule $[i_2 \to i_1]_e^-$ will be applied but it will not be possible to apply the rule $[p]_e^0 \to [\ ]_e^-\#$ at the same time. Thus, an object i_1 will be present in the membrane and the latter will be neutrally charged, so the rule (f) will be applied ending the checking stage with a negative result.

(g) $[i_2 \to i_4 c]_e^-,$
$[c]_e^- \to [\ ]_e^0\#, \quad [i_4 \to i_5]_e^0,$
$[i_5]_e^0 \to [\ ]_e^+ Yes, \quad [i_5]_e^- \to [\ ]_e^+ No.$

If, on the contrary, $w(B) \leq w(B^c)$ holds, then the objects a will be exhausted before the objects p. It is important to distinguish between the cases where the multiplicity of p is strictly greater than the multiplicity of a and the cases where both multiplicities coincide. This is why object c gives again neutral charge to the membrane and then object i_5 checks if a rule $[p]_e^0 \to [\ ]_e^-\#$ is applied or not.

(h) $[p \to \#]_e^+, \quad [a \to \#]_e^+.$

If after the checking loop of rules in (d) has finished there are still some objects p or a in the membrane, then they can be erased (just for "cleaning" purposes).

(i) $[e_n]_e^+ \to \#,$
$[a_0 \to \#]_s^0, \quad [\bar{p} \to \#]_s^0, \quad [g \to \#]_s^0.$

These rules also perform a "cleaning" task, dissolving the membranes that are not meaningful and erasing the contents that these dissolutions spill in the skin membrane. This is not essential in the design, but it is helpful.

(j) $[z_i \rightarrow z_{i+1}]_s^0$, for $i = 1, \ldots, 2n$,
$[z_i \rightarrow d_0 d_1]_s^0$,
$[d_1]_s^0 \rightarrow [\ ]_s^+ d_1$.

Before the answer is sent out, the system has to make sure that all the relevant membranes have finished their checking stages. To do this, first we wait for $2n+1$ steps and then we activate the process. This needs to be done in order to make sure that the division process is over, and thus we know that from this moment on, the membranes that finish their checking stage will have a positive charge, and only them (see the rules in (f) and (g) for the end of the checking and note that we get rid of the spare membranes via the rules in (i)).

(k) $[g]_e^- \rightarrow [\ ]_e^- \bar{g}$,
$[\bar{g} \rightarrow \hat{g}]_s^+$,
$\hat{g}[\ ]_e^+ \rightarrow [\hat{g}]_e^0$.

As we said before, we need to check if all the relevant membranes have finished their checking stages. This is done using the objects g that are present in the skin and the auxiliary membrane labelled by r (see the next set of rules). There must be 2^n copies of g, because each relevant membrane sends one, and there is one relevant membrane for each subset of A, that is 2^n in all.

(l) $d_0[\ ]_r^0 \rightarrow [d_0]_r^-$,
$[h_0 \rightarrow h_1]_r^-$, $[h_1 \rightarrow h_0]_r^+$,
$[b_0]_r^- \rightarrow [_r]_r^+ b$, $\hat{g}[\ ]_r^+ \rightarrow [\hat{g}]_r^-$,
$b[\ ]_r^- \rightarrow [_r b_0]_r^+$, $[\hat{g}]_r^+ \rightarrow [\ ]_r^- \hat{g}$,
$[h_0]_r^+ \rightarrow [\ ]_r^+ d_2$, $[d_2]_s^+ \rightarrow [\ ]_s^0 d_2$.

The membrane labelled by r is present in the initial configuration, but remains inactive until an object d_0 "wakes it up". The purpose of the membrane is to perform a loop where the objects $\hat{g}$ are involved, in such a way that if there are no objects $\hat{g}$ available in the skin, then the loop will halt. Thus, we can detect if there are no objects $\hat{g}$ present in the skin region. This fact will mean that all the relevant membranes have finished their checking stage, and that the system is ready to send out the answer (Yes or No).

(m) $[No \rightarrow No_0]_s^-$,
$[Yes]_s^- \rightarrow [\ ]_s^0 Yes$,
$[No_0]_s^- \rightarrow [\ ]_s^0 No$.

Finally, the output process is activated. The skin membrane needs to be negatively charged before the answer is sent out. Object d_2 takes care of this (see the previous set of rules) and then, if the answer is affirmative, an object Yes will be sent out recovering the neutral charge for the skin. Note that the answer Yes has some priority over the negative answer, in the sense that we first check if there is any object Yes and then, if it is not the case, the answer No will be sent out. This little trick of changing the electrical charge of the skin membrane and using the auxiliary object No_0 is also used in the other two designs, so hopefully this feature can be also saved for future designs.

4 Generalizing the Rules

In this section we will present an overview of the computation, commenting how the rules work and sketching the first instructions that could be added to the library of subroutines that we intend to create.

In the first step of the computation, the rule $[e_i]_e^0 \to [q]_e^- [e_i]_e^+$ is applied, for $i = 0$. From this moment on, the rest of division rules will be applied in turns, in such a way that whenever a negatively charged membrane is created, it will not divide anymore. The concept of subset *associated* with an internal membrane is an abstraction, because there are no witness-objects in the membrane to encode it, but we can agree to "associate" subsets with membranes following this definition:

- The subset associated with the initial membrane is the empty one.
- When an object e_j appears in a neutrally charged membrane (with $j < n$), then the j-th element of A is selected and added up to the previous associated subset. Once the stage is over, the associated subset will not be modified anymore.
- When a division rule is applied, the two newborn membranes inherit the associated subset from the original membrane.

As we already said, the rules in (a) are exactly repeated in the designs for Subset-Sum and Knapsack. Thus, we could create a new instruction, valid to use it in the designs of P systems, called for example

$$gen\_subsets(n)$$

This is nothing but a notation, whenever we find this in a design we should replace it by the set of rules described in (a). We can also make use, if needed, of the semantic notion of associated subset in further stages of the computation.

For instance, this is done in the calculation stage. The weights of the elements are added only if the element is selected for the associated subset.

$$calc\_weight(n) \equiv \left(\begin{array}{ll} [x_0 \to subs]_e^0, & \\ [x_0 \to compl]_e^+, & \\ [x_i \to x_{i-1}]_e^+, & \text{for} \;\; i = 1, \dots, n \\ [x_i \to compl]_e^-, & \text{for} \;\; i = 1, \dots, n \end{array} \right)$$

The object *compl* can be substituted by λ if we do not want information about the complementary, or by other objects, depending on the concrete problem that we are addressing (maybe we could add a second variable that says if the weight of the complementary should be computed or not). The object *subs* encodes with its multiplicity the weight of the associated subset; we are free to use any object instead, it depends on our specific notation. For instance, in the Knapsack problem, two functions have to be computed: the weight and the value of the subset. Thus, we have to include twice the rules, using two different sets of indexed objects, one for each function (for instance, in [7], x_j and y_j were used, for $j = 0, \dots, n$).

The generation and calculation stages end in a membrane when it gets a negative charge for the first time, and we have at our disposal a witness-object q that appears in the membrane exactly in that moment. If we want to perform now the comparison between the multiplicities of two objects, we need to rename all the objects in the membrane, to make sure that there does not exist overlapping, i.e., we want to avoid nondeterminism.

The renaming step depends strongly on the problem, because the new objects that are needed depend on how many stages we want to perform later on.

The next set of rules is (d). When these two rules are applied iteratively, a loop is created. The charge of the membrane changes from negative to neutral and back to negative in every loop, until one of the two objects that are being used is exhausted and the loop halts.

$$check\_weight \equiv \begin{pmatrix} [obj1]_e^- \rightarrow [\ ]_e^0\#, \\ [obj2]_e^0 \rightarrow [\ ]_e^-\# \end{pmatrix}$$

Observe that this time the scheme of rules does not depend on n, we just compare the number of occurrences of two objects, *obj*1 and *obj*2. The names of these objects can be customized, as well as the two charges that are used. We can again recall the design of the Knapsack as an example, because two checking stages were carried out there, with different objects and different charges, but the same changing-charge-loop design.

Next, let us pay attention to the rules that take care of the result of the checking. As we said before, instead of using a counter that increases its index in each step we use two objects as markers, and this suffices to detect when the loop has halted. The evolution of these markers is as rules from item (e) show.

$$[i_{same} \rightarrow i_{diff}]_e^-, \quad [i_{diff} \rightarrow i_{same}]_e^0$$

Let us concentrate on the termination of the checking stage. First of all, if the multiplicity of *obj*1 is greater than the multiplicity of *obj*2, then the moment will come when the rule $[obj1]_e^- \rightarrow [\ ]_e^0\#$ will be applied but its counterpart in the loop ($[obj2]_e^0 \rightarrow [\ ]_e^-\#$) will not be applied in the next step, and so the membrane will keep a neutral charge for two consecutive evolution steps. This fact is detected by the marker and in the following step the rule

$$[i_{same}]_e^0 \rightarrow [\ ]_e^+ z_{more}$$

will be applied, bringing the checking stage of this membrane to its end. In the case of the Partition problem, the fact that there are more copies of object *obj*1 than of *obj*2 means a negative answer, so we replace z_{more} by No, but in other problems it could be replaced by Yes or by other special objects in order to activate further stages.

If, on the contrary, the multiplicity of *obj*1 is smaller than or equal to the multiplicity of *obj*2, then the membrane will have a negative charge for two consecutive steps, but we need to check if any object *obj*2 is still present in the membrane. This can be checked using the rules

$$[i_{diff} \rightarrow aux_1 c]_e^-, \quad [c]_e^- \rightarrow [\ ]_e^0\#$$

then, in the next step the rule $[aux_1 \rightarrow aux_2]_e^0$ will be applied, and the charge will change if and only if there are some objects *obj*2 left (via the rule $[obj2]_e^0 \rightarrow [\ ]_e^-\#$). Finally, we differentiate the results depending on the electrical charge:

$$[aux_2]_e^0 \rightarrow [\ ]_e^+ z_{equal}, \quad [aux_2]_e^- \rightarrow [\ ]_e^+ z_{less}$$

Again, in our problem we have customized z_{equal} to be Yes and z_{less} to be No, but this depends on the condition that we are checking. Maybe in some problems instead of using objects No we are interested in blocking the membrane, and this can be done simply removing the corresponding rule.

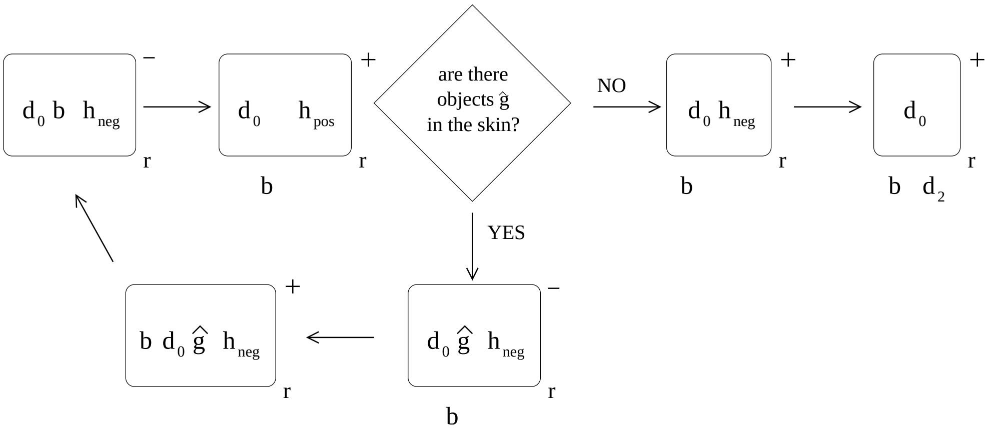

Figure 1: Membrane r detection loop

$$marker\_eq \equiv \left(\begin{array}{l} [i_{same} \rightarrow i_{diff}]_e^-, \quad [i_{diff} \rightarrow i_{same}]_e^0, \\ [i_{same}]_e^0 \rightarrow [\ ]_e^+ No, \\ [i_{diff} \rightarrow aux_1 c]_e^-, \quad [c]_e^- \rightarrow [\ ]_e^0 \#, \\ [aux_1 \rightarrow aux_2]_e^0, \\ [aux_2]_e^0 \rightarrow [\ ]_e^+ Yes, \quad [aux_2]_e^- \rightarrow [\ ]_e^+ No \end{array} \right)$$

The corresponding variants *marker_leq* and *marker_geq* can be defined if we consider that the successful result of the checking is to detect that the number of objects $obj1$ is less than (or greater than, respectively) the number of $obj2$.

It is time now to comment how the answer process is managed. It was already hinted in the previous section that there is a membrane, labelled by r, that plays a central role. Let us explain the process step by step (see Figure 1 for a graphical description of the detection loop).

We need to check if 2^n membranes have finished their checking stages, and to do this it seems a good idea to use 2^n objects, in order to make use of the parallelism of the model. To generate these objects we include an object g in the initial configuration and we use the rule $[g]_e^- \rightarrow [\ ]_e^- \bar{g}$. The object g is replicated every time the membrane divides, and this rule is applied when the membrane ends the generation and calculation stages and before starting the checking. We know that 2^n copies of $\bar{g}$ will be sent to the skin in some moment of the computation, but not simultaneously.

The idea is to make these objects return back to their membranes when the checkings are over. In order to avoid interferences with some checking stages that last longer than others, we add a counter and a renaming rule, and the process of counting how many membranes have finished their checking stage does not start until an object d_1 is sent out to the environment and the skin gets a negative charge.

In the same step, an object d_0 enters the membrane r and activates the loop described by the rules in (l) and depicted in Figure 1. The idea is that membrane r tries to *fish* any object $\hat{g}$ present in the skin: the object b plays the role of *bait*, because it gives positive charge to the membrane allowing thus the rule $\hat{g}[\ ]_r^+ \rightarrow [\hat{g}]_r^-$ to apply (it is clear that if there are no objects $\hat{g}$ in the skin, then the rule cannot be applied). There is a marker inside the membrane that controls if any object actually entered the membrane, and in negative case, an object d_2 will be sent to the skin in order to finish the answer stage.

$$detector \equiv \left(\begin{array}{l} d_0[\,]_r^0 \to [d_0]_r^-, \\ [h_{neg} \to h_{pos}]_r^-, \quad [h_{pos} \to h_{neg}]_r^+, \\ [b]_r^- \to [_r]_r^+ b, \quad \hat{g}[\,]_r^+ \to [\hat{g}]_r^-, \quad b[\,]_r^- \to [_r b_0]_r^+, \quad [\hat{g}]_r^+ \to [\,]_r^- \hat{g}, \\ [h_{neg}]_r^+ \to [\,]_r^+ d_2, \\ [d_2]_s^+ \to [\,]_s^0 d_2 \end{array} \right)$$

After that, we just let the system output the answer, giving one step of advantage to object Yes, in such a way that when we obtain in the environment an object Yes or an object No we know that the system has halted (the computation has finished) and that the object answers correctly the instance of the problem that we were considering.

$$answer \equiv \left(\begin{array}{l} [No \to No_0]_s^- \\ [Yes]_s^- \to [\,]_s^0 Yes \\ [No_0]_s^- \to [\,]_s^0 No \end{array} \right)$$

5 Final Remarks

Up to now, the idea of a programming language has not been deeply discussed in the community of researchers in Membrane Computing, but actually it is not hard to find some similarities between different designs conceived for different purposes: the use of the changes in the polarization, the technique of working with indexed objects and making a rotation on the indexes, the use of renaming rules in order to inhibit the evolution of an object until a specific instant in the computation, and of course the use of counters (an indexed object that increases its index up to a certain value and then transforms into something different), among others. It is worth mentioning as an example of applying these strategies the design for the multidimensional Knapsack problem presented in [2].

In this paper a first informal approach is made to state some “macro-rules” that may be used in a variety of situations. Of course, it is possible to define other subroutines for other variants of P systems, and also much more work can be done in this field, increasing the list of instructions of this, so to say, programming language.

As an example of the usefulness of the subroutines outlined in the previous section, let us see how the design of the solutions for the Subset-Sum, Knapsack and Partition would look like:

SUBSET-SUM
gen_ subsets (n)
calc_ weight (n)
rename
check_ weight
marker_ eq
counter (n)
clean_ dissolve
detector
answer

KNAPSACK
gen_ subsets (n)
calc_ weight1 (n)
calc_ weight2 (n)
rename
check_ weight1
marker_ leq
rename
check_ weight2
marker_ geq
counter (n)
clean_ dissolve
detector
answer

PARTITION
gen_ subsets (n)
calc_ weight$_{compl}$ *(n)*
rename
check_ weight
marker_ eq
counter (n)
clean_ dissolve
detector
answer

Also the solutions for SAT and VALIDITY problems could be rewritten in this form (we refer to [10] for the exhaustive description of the rules and of the similarities between the two designs).

$$\begin{array}{c} gen\_assignments(n) \\ calc\_satisfied\_clauses(n,m) \\ synchronization \\ check\_truth\_value(n,m) \\ counter(n,m) \\ answer \end{array}$$

In this case, a division process is carried out at the beginning of the process to generate one membrane for each possible truth assignment of the n variables appearing in the formula. There, the electrical charges of the membranes in each step are meaningful, because they determine whether a variable will be assigned a TRUE value or a FALSE value. This strategy is very close to the one used in our generation stage.

In parallel with the division process, inside each membrane some objects keep track of which of the m clauses is (are) satisfied whenever we assign a truth value to a variable. This is somehow a weight calculation process, if the clause i gets a value of TRUE with the current assignment of variable j, then we add the "witness" of the clause, otherwise, we skip to the next variable. The technique of rotating the indexes is used here.

Concerning the checking process, the situation is different, because we need to check that all the clauses are satisfied, instead of comparing the multiplicities of two objects. However, the method that is used to control when the checking stage ends is a counter in the skin, and this is also used for the numerical problems. Also the answering process is very similar, the object Yes gets some priority over the object No by means of a counter and of the electric charge of the skin membrane.

Acknowledgements. The support of 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] 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*, to appear.

[2] Pan, L., Martín-Vide, C.: Solving multidimensional 0-1 Knapsack problem by P systems with input and active membranes, in the present volume.

[3] Păun, Gh.: *Membrane Computing. An introduction*, Springer-Verlag, Berlin, 2002.

[4] Păun, Gh.: Computing with membranes, *Journal of Computer and System Sciences*, **61**(1), 2000, 108–143.

[5] Păun, Gh., Rozenberg, G.: A guide to membrane computing, *Theoretical Computer Sciences*, 287, 2002, 73–100.

[6] Pérez-Jiménez, M.J., Riscos-Núñez, A.: Solving the Subset-Sum problem by active membranes, *New Generation Computing*, to appear.

[7] Pérez-Jiménez, M.J., Riscos-Núñez, A.: A linear solution for the Knapsack problem using active membranes, in C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg and A. Salomaa (eds.), *Membrane Computing. Lecture Notes in Computer Science*, vol. 2933, 2004, 250–268.

[8] Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: *Teoría de la Complejidad en modelos de computatión celular con membranes*, Editorial Kronos, Sevilla, 2002.

[9] Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: A polynomial complexity class in P systems using membrane division, *Proceedings of the 5th Workshop on Descriptional Complexity of Formal Systems*, Budapest, Hungary.

[10] Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: Complexity classes in P systems, *Meeting of the European Molecular Computing Consortium. Volume of abstracts*, Turku, Finland, May 15-17, 2003, p. 18.

[11] Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: Solving VALIDITY problem by active membranes with input, *Proceedings of the Brainstorming Week on Membrane Computing*, M. Cavalieri, C. Martín-Vide, Gh. Păun (eds.), Report GRLMC 26/03, 2003, 279–290.

[12] Riscos-Núñez, A., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: An efficient cellular solution for the Partition problem, in this volume.