Π Machines: Virtual Machines Realizing Graph Structured Transition P Systems

Apostolos Syropoulos
GREEK MOLECULAR COMPUTING GROUP
366, 28th October Str.
GR-671 00 Xanthi, GREECE
e-mail: asyropoulos@yahoo.com

Abstract

P systems is a model of computation inspired by the way cells live and function. A typical P system consists of nested compartments surrounded by *porous* membranes, which contain data that are transformed by transformation rules. P systems can be simulated by a distributed computing system, where each compartment of a given system is simulated by a (remote) process, each running on a different node. By adopting a more "liberal" membrane structure where compartments do not necessarily form a tree-structure but a graph-structure, we get a more general model of computation, which we call graph structured P systems. Any instance of the new model can be implemented by a network of virtual machines, called Π machines, where each machine is able to implement the functionality of any simple compartment.

Keywords: virtual machines, graph structured P systems, distributed computing.

1 Introduction

P systems [3, 4] is a model of computation that is based on an abstraction of the living cell. In its most simple form, a P system consists of nested compartments each surrounded by a *porous* membrane. This structure of nested compartments is called a *membrane* structure. Figure 1 depicts a rather complex membrane structure. Each compartment is populated by a multiset, which draws from a set of objects. Also, each compartment is associated with a number of multiset rewriting rules. Computation starts by simultaneously and non-deterministically applying rules to the contents of each compartment while actions take place synchronously. When it is impossible to apply any rule, then the result of the computation is equal to the cardinality (i.e., the total number of elements) of the multiset contained in a designated compartment called the *output* compartment. Since one can introduce various other features into a P system and, thus, define other forms of P system, the simple systems just described are known in the literature as *transition* P systems.

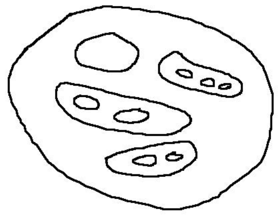

Figure 1: A floor plan of a rather complex membrane structure.

Because rules are applied in parallel, one may think that each compartment is an independent machine, which is just part of a network. However, since rules are applied synchronously, the network looks more like a distributed system. This observation has led this author and his colleagues to explore the connection between P systems and distributed computing in [8]. This connection was later elaborated in [6]. The authors of [8] have demonstrated that one can very naturally simulate the functionality of transition P systems in a distributed environment. In particular, based on Java's *Remote Method Invocation* these authors designed and implemented a Java class realizing the basic functionality of a typical compartment of a P system. Objects of this class can be used to simulate any transition P system. More specifically, one object plays the rôle of the "top" compartment. In the beginning, this object reads a description of a P system (its membrane structure, the rewriting rules associated with each compartment, and the multisets that populate each compartment), checks whether there are enough objects running on possibly different machines so as to start the simulation and if there are, the simulation begins.

The simulator described in [8] formed the basis for an effort to design a virtual machine to implement the functionality of a simple compartment. Quite naturally this effort led to the idea of defining graph-structured P systems and, then, to define virtual machines capable of implementing each node of such a network. The resulting virtual machines are called Π machines. Clearly, graph-structured P systems are a generalization of ordinary P systems and as such they may be of interest to people working in the area of natural computing. However, it is quite possible that someone may object the usefulness of these systems by claiming that these systems have nothing to do with cells. In response, let us first observe that a graph-structured system can be used to describe networks of trees, which may "describe" cells. More generally, my reply to comments like this is that one should use Nature as a source or inspiration, not as a framework that delimits one's imagination.

Relation to Similar Projects There are at least two programming "systems" that are similar to our graph-structured P systems and the virtual machines capable of implementing them. However, there are fundamental differences between them and our proposal. In particular, the Cell Programming Language, which was designed and implemented by Pankaj Agarwal [1], is a tool that allows its user to write programs that mimic the life of a biological cell. In other words, CPL is a tool for the creation of computer simulations of living cells. Our virtual machines are not a simulation tool (at least of this kind), but machines designed to realize any graph structured P system. On the other hand, the chemical abstract machine, which was designed by Gérard Berry and Gérard Boudol [2], is built around the notion of a chemical solution consisting of molecules and being (magically) stirred continuously. Molecules may interact with other molecules to create new molecules. Also, molecules may dissolve to create new, simpler, molecules, etc. Indeed, this model of computation seems more akin to our proposal, nevertheless, the cham is an abstract machine while Π machines are virtual machines. But one can compare the expressive power of chams with those of graph-structured P systems, although the later have nothing to do with the so-called chemical metaphor.

Plan of the paper First, I formally define graph structured P systems. Next, there is a rather informal description of the structure of Π machines, their instruction set, and how they do operate. Right after, there is a precise description of the instruction set that is followed by a few remarks concerning this work.

2 Transitional Graph Structured P Systems

As was noted above, the membrane structure of a typical P system forms a hierarchical tree structure, which, in my opinion, is not general enough. Since each tree is a graph, but not vice versa, it it clear that graph-structured P system should be more expressive and, possibly, more powerful than tree-structured P systems. This simple idea motivated the definition of graph structured P systems:

Definition 2.1 A graph-structured P system is a collection of compartments, each surrounded by porous membrane, multisets, drawn from the same set of objects, and multiset rewriting rules that is specified as follows:

$$\Pi = (O, g_{mn}, (w)_{ij}, (R)_{ij}, (\mathcal{I})_{ij}),$$

where

1. O is a set of objects or, simply, an alphabet that denotes types of information.

2. g_{mn} is a relation in $\{1,\ldots,m\}\times\{1,\ldots,n\}$, that describes a *network* of nodes that define the graph structure of the system. Here $n,m \in \mathbb{N}$, $\mathbb{N}=\{0,1,2,\ldots\}$, are fixed numbers for each system definition. Also, in what follows, the letters p and q with two subscripts will, in general, denote some compartment.

3. $(w)_{ij}$, $1 \leq i \leq m$ and $1 \leq j \leq n$, are strings representing multisets of objects, which are elements of O, and denote conglomerations of information. Each string is associated with compartment p_{ij}.

4. $(R)_{ij}$ are finite sets of *evolution rules* over O; R_{ij} is associated with compartment p_{ij}. An evolution rule is of the form $u \to v$, where u is a string over O and v is a string over O_{NTAR}, where $O_{\text{NTAR}} = O \times \text{NTAR}$, and $\text{NTAR} = \{\text{here}, \text{out}\} \cup \{\text{to}_{ij} | (1 \leq i \leq m) \wedge (1 \leq j \leq n)\}$.

5. $(\mathcal{I})_{ij}$ is a collection of designated compartments that are called the *output* compartments.

The rules that belong to some set of evolution rules R_{ij} are applied to the objects of the compartment p_{ij} simultaneously, synchronously, and non-deterministically. Given a rule $u \to v$ the effect of applying this rule to a compartment p_{ij} is to remove the submultiset of objects specified by u and to insert the objects o_i specified by v in the regions designated by the target commands ℓ_i associated with each object. In particular,

1. If $(a, \text{here}) \in v$, the object a will be placed into compartment p_{ij} (i.e., the compartment where the action takes place). In other words, this command introduces a new object into the current compartment.

2. When $(a, \text{out}) \in v$, the object a will be placed into the compartment that is "nearer" to the current compartment or to any compartment that lies in its proximity, in case more than one compartment have this property (the choice is always nondeterministic). Observe that we can imagine that all compartments are laid on the nodes of a square grid. Thus, it is easy to compute the relative distance between any two compartments.

3. If $(a, \text{to}_{ij}) \in v$, the object a will be transported into compartment p_{ij}.

At any given moment, the sum of the cardinalities of the multisets contained in the compartments i_{kl} is equal to the natural number that has been computed so far.

Clearly, it is quite possible to add some input nodes, which will get input from the environment and, possibly, be used as a way to dump information to the environment. Although I do not plan to elaborate on this idea, nevertheless, I will discuss it again in passing in the next section.

3 The Virtual Π Machine

In this section I will present the Π machine and its operation. Also, I will present the structure of programs that these machines accept. Here the term *program* will refer to the assembly program analog of ordinary machines. Although it is not difficult to define a binary file format for our programs, there is no real reason to invent yet another binary file format since programs are very simple.

Each Π machine consists of a memory, a processor and communication channels. The memory of a machine is divided into two distinguished areas: the *text* area and the *data* area. The text area is used to store commands, while the data area is used to store objects in some compact form. A processor executes the commands stored in the text area and manipulates the data stored in the data area. Also, a processor can execute commands for other external processors, implementing in this way a form of remote code execution. The communication channels are used to send and receive information from/to other machines. Each Π machine is associated with a unique address so as to be able to distinguish machines being parts of a network. This number is similar to the MAC address (Media Access Control address) of a NIC (network interface card). Information travel in the form of packets containing a single object plus the sender's and the receiver's unique address. Packets that cannot be delivered are simply disregarded. A processor neither can write data to a full memory, nor it can delete data from an empty memory.

The text area is actually a command pool from which nondeterministically the machine picks a command to execute. The machine stops only when it cannot execute a command. In other words, a machine's operation is equivalent to the following loop:

```
while (1) {
  randomly select a command;
```

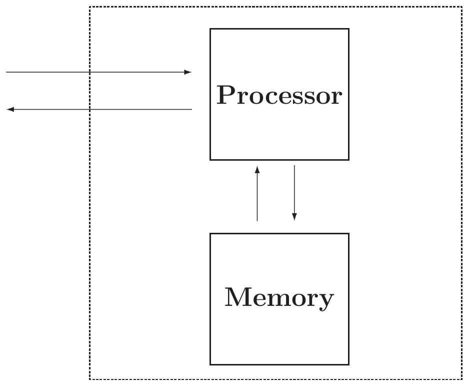

Figure 2: A diagram depicting the general structure of a Π machine. The arrows between the processor and the memory depict the internal communication channels, while the other pair of arrows depict the outgoing and incoming communication channels.

```
     if (command is not meaningless) {
       execute command;
       wait for sufficient time;
    }
    else {
       abort program execution;
    }
}
```

A command is called *meaningless* when it is impossible to execute it. For example, when a command tries to add data to a full memory it becomes meaningless and, thus, causes the machine to abort the execution of the program. Another situation where a command becomes meaningless is when there are no data to alter or transport. Since I do not want to impose any implementation technique or algorithm, I will not specify how the machine `randomly selects a command`. For example, if the commands are stored in some indexed linear structure and a pseudo-random number generator is used to randomly pick a command, then this would be a reasonable implementation of this requirement. When the execution of a particular command has finished, the machine has to wait for sufficient time so as to fulfill requests from other machines (e.g., to delete a bunch of objects from its memory).

The programs a Π machine accepts reflect its structure. In particular, programs consist of two distinguished sections: the *data* section and the *text* section. The data section is used to specify the initial data, that is, the multiset which will be stored in the data area of the memory of a particular machine, and the text section contains the code that the machine will store in the text area of its memory. Later on, the machine will execute this code. Since a machine does operate in a nondeterministic way, it does not make any sense to pay any attention to the order by which the commands are specified. Below, I give the general structure of a generic Π machine program:

$$\begin{array}{ll}
\texttt{data}: & \\
& n_1, o_1 \\
& n_2, o_2 \\
& \vdots \\
& n_s, o_s \\
\texttt{text}: & \\
& \mathrm{com}_1 \\
& \mathrm{com}_2 \\
& \vdots
\end{array}$$

$$\text{com}_r$$

$$\texttt{end}:$$

Here $n_i \in \mathbb{N}$ and o_i is an element of an alphabet. The notation n_i, o_i means that the machine's memory contains n_i copies of o_i. Also, com_i denotes any command from the following list:

dispatch d, i Remove all d's from the machine that executes this command and send them to the designated machine i.

replace d, d', i Replace all occurrences of d in the memory of the designated machine i with d'. If $i = -1$, then this operation takes place for the machine that is actually executing this command.

delete n, d, i Delete n occurrences of d from the memory of the designated machine i. If n is not a natural number but the symbol $*$, then it deletes all occurrences of the symbol d. In addition, if $i = -1$, this operation takes place locally.

create n, d, i Create n copies of object d and store them in the memory of machine i. As in all previous case, if $i = -1$, then the operation affects the memory of the machine that actually executes this command.

isempty i, j This command checks whether the data area of the memory of the designated machine i is empty or not. In case it is empty, then the designated machine j halts and clears the text area of its memory and locks itself up prohibiting any further alteration to the contents of the data area of its memory. Notice that this command was introduced mainly to solve problems that have been encountered in the representation of recursive functions as P systems (see [7]).

It is possible to add an *input* command that will get an unspecified number of copies of some unspecified object to the memory of the machine that executes this command. Note that we do not demand that the object belong to the alphabet of the system. Thus, they might be some new symbols that either do not affect the computation or they do affect it in an unexpected way.

Let us now give a precise description of what exactly each command is doing. In different words, I will state the semantics of Π machines by explaining what each command is doing. At any given moment, each machine is characterized by the command it executes, the totality of the commands stored in the text area of its memory, the contents of its memory data area, and the size of the free data area memory. In different words, assuming that time is discrete:

Definition 3.1 At any given moment a Π machine is characterized by a quintuple

$$\bigl(c, i, (E)_i, M, s\bigr),$$

where c is the command being executed, $(E)_i$ is a collection of indexed commands, which represents the collection of commands stored in the text area of its memory, i is the index of the next command to be executed, m is a multiset representing the contents of the data area of its memory, and s is the total size of the data area of its memory (allocated plus free memory).

Obviously, at any moment the free data area memory is equal to $s - \text{card}\, M$. In the beginning, a machine is characterized by the following quintuple:

$$\bigl(\varepsilon, i, (E)_i, M, s\bigr),$$

where ε denotes the empty command. When a command becomes meaningless, the machine is characterized by the following quintuple:

$$\bigl(s_j, -1, \emptyset, M'^{\cdots\prime}, s\bigr),$$

where -1 in place of the index of the next command denotes that the there is no next command to be executed (i.e., the machine is about to halt or has already halted). Also, the collection of commands to be executed has become empty. More compactly, the operation of a machine can be described by a transition relation like the following one:

$$\begin{array}{rl} \bigl(\varepsilon, i_0, (E)_i, M, s\bigr) & \rightarrow \\ \bigl(s_{i_0}, i_1, (E)_i, M', s\bigr) & \rightarrow \\ \cdots\cdots\cdots\cdots\cdots\cdots & \cdots \\ \bigl(s_j, -1, \emptyset, M'^{\cdots\prime}, s\bigr). & \end{array}$$

As note above, each machine can modify the data area of the memory of any other machine with which it coexists in some network. Since, it has been explicitly stated that the machine operate nondeterministically, this implies the memory should be modified accordingly. However, as stated above, it is a natural requirement to demand that first, at each tick of the clock, the memory is modified by the processor of the machine to which belongs the memory and, then, by all other machines in a truly nondeterministic way.

In order to proceed it is necessary to recall a few definition from the formal theory of multisets. As usual, readers familiar with the various notions and the corresponding symbols employed to define them can safely skip the next paragraph.

A multiset A is characterized by a function $A : X \to \mathbb{N}$, where X is some universe set. In general, $A(a) = n$ means that the multiset A contains n copies of a. To simply the discussion that follows, a multiset will be described, and not just characterized, by some function.

Definition 3.2 Let $A, B : X \to \mathbb{N}$ be two multisets. Their sum $A \uplus B$ is a new multiset such that

$$(A \uplus B)(x) = A(x) + B(x), \forall x \in X.$$

Definition 3.3 Suppose that $A : X \to \mathbb{N}$ is a multiset and $B \subseteq X$ is a set whose characteristic function is $\chi_B : X \to \{0, 1\}$, then the set-removal of B from A is a new multiset $A \circledast B$ such that

$$(A \circledast B)(x) = A(x) \cdot (1 - \chi_B(x)), \forall x \in X.$$

Definition 3.4 Let $A, B : X \to \mathbb{N}$ be two multisets, then the removal of B from A is a new multiset $A \ominus B$ such that

$$(A \ominus B)(x) = \max\{A(x) - B(x), 0\}, \forall x \in X.$$

We are now going to give a formal description of the effects of the commands presented above. For simplicity, in all cases I will assume that no command is meaningless.

The dispatch command First let us see how the machine's own memory is altered:

$$\begin{aligned} \Big((\text{dispatch } d, n), i_k, (E)_i, M, s\Big) &\to \\ \Big((\text{dispatch } d, n), i_k, (E)_i, M \circledast \{d\}, s\Big) &\to \\ \cdots\cdots\cdots& \\ \big(c_{i_k}, i_{k+1}, (E)_i, M', s\big) &\to \\ \cdots\cdots\cdots& \end{aligned}$$

Next, we need to see how is modified the memory of machine n:

$$\begin{aligned} \big(c'_{j_{l-1}}, j_l, (E)_j, M_n, s_n\big) &\to \\ \big(c'_{j_{l-1}}, j_l, (E)_j, M'_n, s_n\big) &\to \\ \cdots\cdots\cdots& \\ \Big(c'_{j_l}, j_{l+1}, (E)_i, M' \uplus \big(M \ominus (M \circledast \{d\})\big), s\Big) &\to \\ &\to \\ \cdots\cdots\cdots& \end{aligned}$$

The replace command Since it is not known which memory this command affects, let us see how it operates, in general:

$$\begin{aligned} \big(c'_{j_{l-1}}, j_l, (E)_j, M_n, s_n\big) &\to \\ \big(c'_{j_{l-1}}, j_l, (E)_j, M'_n, s_n\big) &\to \\ \cdots\cdots\cdots& \\ \Big(c'_{j_l}, j_{l+1}, (E)_i, \big(M''_n \ominus (Mi \circledast \{d\})\big) \uplus \underbrace{\{d', \ldots, d'\}}_{x \text{ copies}}, s\Big) &\to \\ \cdots\cdots\cdots& \end{aligned}$$

where $x = \mathrm{card}(M''_n \ominus (M''_n \circledast \{d\}))$.

The delete command This command behaves like the replace command, so, again, let us see how it operates, in general:

$$\begin{aligned}
\big(c', j_l, (E)_j, M_i, s_n\big) &\quad\rightarrow\\
\big(c', j_l, (E)_j, M'_i, s_n\big) &\quad\rightarrow\\
\cdots\cdots\cdots &\\
\Big(c_k, i_{k+1}, (E)_i, M''_i \ominus \underbrace{\{d, \ldots, d\}}_{n \text{ copies}}, s\Big) &\quad\rightarrow\\
\cdots\cdots\cdots &
\end{aligned}$$

When n is replaced by $*$, the data area of its memory contains the multiset $M''_i \circledast \{d\}$.

The create command For simplicity let us describe the command when $i = -1$:

$$\begin{aligned}
\Big((\text{create } d, n, -1), i_k, (E)_i, M, s\Big) &\quad\rightarrow\\
\Big((\text{create } d, n, -1), i_k, (E)_i, M \uplus \underbrace{\{d, \ldots, d\}}_{n \text{ copies}}, s\Big) &\quad\rightarrow\\
\cdots\cdots\cdots &\\
\big(c_k, i_{k+1}, (E)_i, M', s\big) &\quad\rightarrow\\
\cdots\cdots\cdots &
\end{aligned}$$

The isempty command First let us see what happens to the machine that executes this command:

$$\begin{aligned}
\big((\text{isempty } n, m), i_k, (E)_i, M, s\big) &\quad\rightarrow\\
\big(c_{i_k}, i_{k+1}, (E)_i, M, s\big) &\quad\rightarrow\\
\cdots\cdots\cdots &
\end{aligned}$$

Clearly, when the memory of node n is empty, then node m should stop, in other words:

$$\begin{aligned}
\big(c'', j_l, (E)_j, M, s\big) &\quad\rightarrow\\
\cdots\cdots\cdots &\\
\big(c'', -1, \emptyset, M, s\big) &\quad\rightarrow\\
\cdots\cdots\cdots &
\end{aligned}$$

4 Conclusions

Graph-structured transition P systems are a generalization of transition P system, where the tree-structured membrane structure is replaced by the more general graph-structured membrane structure. In addition, data can be transported from any node (a compartment surrounded by a membrane) to any other node. Thus, achieving maximum flexibility. Also, there are more than one output nodes, while there is provision for input nodes that will facilitate the communication with the external world. With a few words, the Π machine is a virtual machine that can be used to implement, and not just simulate, any graph-structured transition P system. By finding a suitable way to encode integer functions as graph-structured P system (see [5, 7] for two proposals on how to encode recursive functions as P systems), it will be possible to use them to compute concrete functions. In addition, they might be used as a simple tool for the design and implementation of complex distributed systems.

Acknowledgments

I would like to thank Christophe Poucet for his many comments and suggestions on a draft of this paper.

References

[1] Agarwal, P. The Cell Programming Language. *Artificial Life 2*, 1 (1994), 37–77.

[2] Berry, G., and Boudol, G. The chemical abstract machine. *Theoretical Computer Science 96* (1992), 217–248.

[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] Romero-Jiménez, A., and J.Pérez-Jiménez, M. Computing Partial Recursive Functions by Transition P Systems. In *Membrane Computing*, C. Martín-Vide and G. Păun, Eds., no. 2933 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004, pp. 320–340.

[6] Syropoulos, A. On P Systems and Distributed Computing. In *Pre-proceedings of the Fifth Workshop On Membrane Computing (WMC5).* 2004, pp. 405–413. Milan, Italy.

[7] Syropoulos, A., Doumanis, S., and Sotiriades, K. T. Computing Recursive Functions with P Systems. In *Pre-proceedings of the Fifth Workshop On Membrane Computing (WMC5).* 2004, pp. 414–421. Milan, Italy.

[8] Syropoulos, A., Mamatas, E. G., Allilomes, P. C., and Sotiriades, K. T. A Distributed Simulation of Transition P Systems. In *Membrane Computing*, C. Martín-Vide and G. Păun, Eds., no. 2933 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2004, pp. 357–368.