Defining and Executing P-systems with Structured Data in K

Traian Şerbănuţă, Gheorghe Ştefănescu, and Grigore Roşu

Department of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801
Email: `{tserban2,stefanes,grosu}@cs.uiuc.edu`

Abstract. K is a rewrite-based framework proposed for giving formal executable semantics to programming languages and/or calculi. K departs from other rewrite-based frameworks in two respects: (1) it assumes multisets and lists as builtin, the former modeling parallel features, while the latter sequential ones; and (2) the parallel application of rewriting rules is extended from non-overlapping rules to rules which may overlap, but on parts which are not changed by these rules (may overlap on "read only" parts). This paper shows how P-systems and variants can be defined as K (rewrite) systems. This is the first representation of P-systems into a rewrite-based framework that captures the behavior (reaction steps) of the original P-system step-for-step. In addition to providing a formal executable semantic framework for P-systems, the embedding of P-systems as K systems also serves as a basis for experimenting with and developing new extensions of P-systems, for example with structured data. A Maude-based application for executing P-systems defined in K has been implemented and experimented with; initial results show computational advantages of using structured objects in P-systems.

1 Introduction

K [23] (see also [14]) is a rewrite-based framework which has been proposed and developed (starting with 2003) as an alternative (to structural operational semantics) formal executable semantic framework for defining programming languages, language-related features such as type systems, computation calculi, etc. K's strength can be best reflected when defining concurrent languages or calculi, because it gives those a truly concurrent semantics, that is, one in which concurrent steps take place concurrently also in the semantics (instead of interleaving them, as conventional operational semantics do). K assumes as builtin and is optimized for multisets and lists, the former modeling parallel features, while the latter sequential ones. More importantly, rewriting rules in K can be applied concurrently even when they overlap, assuming that they do not change the overlapped portion of the term (may overlap on "read only" parts). Core parts of many programming languages or computation models are already defined in K, including Scheme [16], KOOL [13], Milner's EXP language [17], Turing machines, CCS [18], as well as type systems for these, etc. - see [23].

A fast developing class of computation models was introduced by Paun in 1998 [20] exploiting ideas from chemistry and biology (see [22] for a recent survey). They are called *membrane systems* (or *P-systems*) and combine nested membrane structures with computation mechanisms inspired by the activity of living cells. There is a large variety of P-systems studied in the literature and a few toy implementations for developing applications. The original motivation was to link this research to formal language theory studies, but the model is more general, coming with important suggestions in many fields, for instance in the design of new parallel programming languages.

Both K definitions and P-systems use potentially nested membranes as a spatial modularization mechanism to encapsulate behaviors and to structure the systems (see Fig. 1(a) for an example of nested membranes). P-systems are inspired by chemistry and biology, using "objects" (abstract representations of chemical molecules) which interact; all communication is at the object level. Objects move from region to region (in the basic model, these are neighboring regions) either directly, or by using symport/antiport mechanisms. When far reaching regions are targeted, special tags are to be added to objects to reach the destination and, in most of the models, the objects have to travel through membranes from region to region to reach the final destination.

The K-framework uses a similar membrane structure (called "cell"), but for a different goal, leading to important differences. The main objective of K is to model high-level programming languages and calculi, for instance allowing OO- or multi-threading programming. To this end, the objects within the membranes are structured. This structure is both in space and in time. The spatial aspect refers to the use of algebraic terms to describe the objects floating in the membrane soups (regions). Due to this algebraic structure, one has more power to express object interaction. However, there is another equally important mechanism, which is not explicitly present in P-systems or in CHAMs [6, 7], namely the use of computation tasks, as control structures evolving in time. To this end, a new data type is builtin in the K framework, the *list* structure,[1] used to capture sequential orders on tasks' execution. The "communication" in K is at a high level, data being "moved" from a place to another place in a single step. It is this combination of structured data and their use in a mixture of nested soups (for parallel processes) and lists (for sequential tasks) which makes it possible to effectively define various high-level languages in K.

P-systems may be described without membranes, too. Indeed, using the tree associated to the membrane structure, one can tag each object with the path in the tree from the root to the membrane where the object is in. The objects may be used in a unique huge soup and the evolution rules, previously used in specific regions, become global rules, but applied to similar path-tagged objects. This representation highlights the advantages of using membrane systems: the matching and the interaction between objects are localized and may be more efficiently implemented. The price to be paid is that communication between

[1] Lists can be encoded with (multi)sets, but allowing them as "first-class citizens" gives the K user the capability to efficiently match fragments of lists.

arbitrary membranes is not straightforward and has to be implemented with small steps of passing objects from region to region to reach the final destination. In K one has a somehow mixed setting: K uses membranes to enforce local rewriting, but the matching rules are global.

Three main classes of P-systems have been extensively studied:

- P-systems as transition systems [“classical” P-systems]
- P-systems as communicating systems [symport/antiport P-systems]
- P-systems as structure-evolving systems [P-systems with active membranes]

We present formalizations in K of these three basic types of P-systems and of many key elements used in the plethora of P-systems found in the literature. We believe that this is enough evidence that K can serve as a suitable semantic framework for defining P-systems in particular, and, in general, for experimenting with new parallel programming languages based on paradigms from natural science, like P-systems, for which efficient implementations are notoriously difficult to develop. We make two additional contributions:

- We extend P-systems from a setting with unstructured objects (or using a simple monoid structure on objects, i.e., strings) to one where objects are given by arbitrary equational specifications. Most data types may be represented by algebraic specification techniques, hence one can use this line to incorporate complex data types into P-systems.
- We have developed a running environment for P-systems using the embedding techniques discussed in this paper and the implementation of K in Maude. The paper includes a few experiments with both unstructured and structured objects which demonstrate the large increase in expressivity and performance due to adding structure to objects.

The paper is organized as follows. Sections 2 and 3 briefly introduce K and P-systems, respectively, referring the reader to the literature for more detail. Sections 4, 5 and 6 show how the three types of P-systems above are defined in K. Section 7 discusses our implementation and Section 8 concludes the paper.

2 K

K [23] is a framework for defining programming languages based on rewriting logic RWL [15], in the sense that it has two types of sentences: *equations* for structural identities and *rules* for computational steps. It has an implementation in Maude and it allows for a fast development of efficient interpretors. We briefly describe the basic concepts and notations used in K (see [23] and the referenced website for a detailed presentation of K; ask the author of [23] for the current version of the draft book) using as an example a simple concurrent language. We start with the simple imperative language IMP defined in Tables 1 and 2. Then, we extend IMP with threads and call the resulting language CIMP.

$Int ::=$	$\ldots$ all integer numbers		
$Bool ::=$	$\mathsf{true} \mid \mathsf{false}$		
$Name ::=$	all identifiers; to be used as names of variables		
$Val ::=$	Int		
$AExp ::=$	$Val \mid Name$		
$\mid$	$AExp + AExp$	$[strict,\ extends\ +_{Int\times Int\to Int}]$	(r_1)
$BExp ::=$	$Bool$		
$\mid$	$AExp \leq AExp$	$[seqstrict,\ extends\ \leq_{Int\times Int\to Bool}]$	(r_2)
$\mid$	$\mathsf{not}\ BExp$	$[strict,\ extends\ \neg_{Bool\to Bool}]$	(r_3)
$\mid$	$BExp\ \mathsf{and}\ BExp$	$[strict(1)]$	(r_4)
$Stmt ::=$	$Stmt; Stmt$	$[s_1; s_2 = s_1 \curvearrowright s_2]$	(r_5)
$\mid$	$Name := AExp$	$[strict(2)]$	(r_6)
$\mid$	$\mathsf{if}\ BExp\ \mathsf{then}\ Stmt\ \mathsf{else}\ Stmt$	$[strict(1)]$	(r_7)
$\mid$	$\mathsf{while}\ BExp\ \mathsf{do}\ Stmt$		(r_8)
$\mid$	$\mathsf{halt}\ AExp$	$[strict]$	(r_9)
$Pgm ::=$	$Stmt; AExp$		

Table 1. K-annotated syntax of IMP.

Annotating syntax. The plus operation (r_1) is said to be "strict" in both arguments/subexpressions (i.e., they may be evaluated in any order), while the conditional (r_7) is strict only in its first argument (i.e., the boolean expression has to be evaluated first); finally, in "less-than" (r_2) the evaluation is "seqstrict" (i.e., the arguments are evaluated in the sequential order). Some operations "extend" other operations on primitive data types (which may be computed with external libraries, for example). All attributes can be desugared with equations or rules (if they are not already equations or rules); they are nothing but notational convenience. For example, the "extends" attribute in (r_1) desugars to rule "$i_1 + i_2 \to i_1 +_{Int\times Int\to Int} i_2$". The desugaring of "strictness" is explained shortly.

Semantics. The K semantics is defined with equations and rules that apply on a nested *cell* (or *membrane*) structure, each cell containing either multisets or lists of elements. A K semantics of IMP is described in Table 2. K *configurations* are specified using cells $\llparenthesis\ S\ \rrparenthesis_h$, where h is a cell index and S is any term, in

$KResult ::= Val$	$\llparenthesis \underset{\cdot}{\underline{x := v}} \rrparenthesis_k\ \langle\!\llparenthesis (x, \underset{v}{\underline{\_}}) \rrparenthesis_{state}$
$K ::= KResult \mid \mathsf{List}^{\cdot}_{\curvearrowright}[K]$	$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \to s_1$
$Config ::= \llparenthesis K \rrparenthesis_k \mid \llparenthesis \mathsf{Set}[(Name, Val)] \rrparenthesis_{state}$	$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \to s_2$
$\mid\ \llparenthesis \mathsf{Set}[Config] \rrparenthesis_{\top}$	$\llparenthesis \underset{\mathsf{if}\ b\ \mathsf{then}\ (s; \mathsf{while}\ b\ \mathsf{do}\ s)\ \mathsf{else}\ \cdot}{\underline{\mathsf{while}\ b\ \mathsf{do}\ s}} \rrparenthesis_k$
$\llparenthesis \underset{v}{\underline{x}} \rrparenthesis_k\ \langle\!\llparenthesis (x, v) \rrparenthesis_{state}$	$\llparenthesis \mathsf{halt}\ i \rrparenthesis_k \to \llparenthesis i \rrparenthesis_k$
$\mathsf{true\ and}\ b \to b, \quad \mathsf{false\ and}\ b \to \mathsf{false}$	

Table 2. K configuration and semantics of IMP.

particular a multiset or a list of possibly other cells. $\mathsf{Set}[S]$ and $\mathsf{List}[S]$ denote multisets and lists of terms of type S; by default, the empty set or list is denoted by a dot "$\cdot$", and multiset elements are separated by space while list elements are separated by comma. If one wants a different separator or unit, or wants to emphasize a default one, then one can specify it as sub- and/or super-script; for example, $\mathsf{List}^{\cdot}_{\curvearrowright}[S]$ denotes "$\curvearrowright$"-separated lists of elements of type S of unit "$\cdot$". The syntactic category K stays for *computations* and typically has a "$\curvearrowright$"-separated list structure of *computational tasks*, with the intuition that these are processed sequentially. What "processed" means depends upon the particular definition. Strictness attributes are syntactic sugar for special equations allowing subcomputations to be "scheduled for processing". The strictness attributes in Table 1 correspond to ($k, k_1, k_2 \in K$, $r_1 \in \mathit{KResult}$, $x \in \mathit{Name}$):

$$
\begin{array}{ll}
k_1 + k_2 = k_1 \curvearrowright \square + k_2 & k_1 \ \mathsf{and}\ k_2 = k_1 \curvearrowright \square \ \mathsf{and}\ k_2 \\
k_1 + k_2 = k_2 \curvearrowright k_1 + \square & x := k = k \curvearrowright x := \square \\
k_1 \leq k_2 = k_1 \curvearrowright \square \leq k_2 & \mathsf{if}\ k\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 = k \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2 \\
r_1 \leq k_2 = k_2 \curvearrowright r_1 \leq \square & \mathsf{halt}\ k = k \curvearrowright \mathsf{halt}\ \square \\
\mathsf{not}\ k = k \curvearrowright \mathsf{not}\ \square &
\end{array}
$$

The square "$\square$" is part of auxiliary operator names called *computation freezers*; for example, "$\square + \_$" freezes the computation k_2 in the first equation above.

In K, the following derived notations are used to indicate an occurrence of an element in a cell: $(\!| \ S \ \rangle_h$ - at the top; $\langle \ S \ |\!)_h$ - at the very end; $\langle \ S \ \rangle_h$ - anywhere. The first two notations are useful when the configuration is a list, but they are not used in the representation of P-systems in K described in this paper. Rules can also be written in *contextual form* in K, where subterms to be replaced are underlined and the terms they are replaced by are written underneath the line. For example, the assignment rule

$$(\!| \underset{\cdot}{\underline{x := v}} \rangle_k \ \langle (x, \underset{v}{\underline{\_}}) \rangle_{state}$$

reads as follows (underscore "_" matches anything): if an assignment "$x := v$" is at the top of the computation, then replace the current value of x in the state by v and dissolve the assignment statement. We prefer to use the more conventional notation "$l \rightarrow r$" instead of "$\frac{l}{r}$" when the entire term changes in a rule.

Extending IMP with threads. We extend IMP with threads and call the resulting language CIMP (from concurrent IMP). We only add a spawning statement to the syntax of the language, without any explicit mechanisms for synchronization.

$\mathit{Stmt} ::= \ldots \mid \mathsf{spawn}\ \mathit{Stmt}$

Interestingly, all we have to do is to add a K rule for thread creation and nothing from the existing definition of the K semantics of IMP has to be changed. Here is the rule for spawning:

$$(\!| \ \frac{\mathsf{spawn}(s)}{\cdot} \ \rangle_k \quad \frac{\cdot}{(\!| \ s \ \rangle_k}$$

Therefore, multiple $\llparenthesis\ \_\ \rrparenthesis_k$ cells can live at the same time in the top cell, one per thread. Thanks to the concurrent rewriting semantics of K, different threads can access (read or write) different variables in the state at the same time. Moreover, two or more threads can concurrently read the same variable. This is precisely the intended semantics of multi-threading, which is, unfortunately, not captured by SOS definitions of CIMP, which enforce an interleaving semantics based on nondeterministic linearizations of the concurrent actions. While K allows a concurrent semantics to a language, note that it does *not* enforce any particular "amount of concurrency"; in particular, K's concurrent rewriting relation, denoted "$\Rrightarrow$", includes any number of rewrites that can be executed concurrently, from one rewrite to a maximal set of rewrites; thus, an interleaved execution or a maximally parallel one are both valid rewrite sequences in K.

Once a thread is terminated, its empty cell (recall that statements are processed into empty computations) will never be involved into any matching, so it plays no role in the future of the program, except, perhaps, that it can overflow the memory in actual implementations of the K definition. It is therefore natural to cleanup the useless cells with an equation of the form:

$$\llparenthesis \cdot \rrparenthesis_k = \cdot$$

Synchronization mechanisms through lock acquire and release, as well as through rendez-vous barriers, are discussed in detail in [23].

3 Membrane systems

Membrane systems (or P-systems) are computing devices abstracted from the structure and the functioning of the living cell [1].

In classical *transition P-systems* [20], the main ingredients of such a system are the *membrane structure* (a hierarchical cell-like arrangement of membranes[2]), in the compartments of which *multisets* of symbol-objects evolve according to given *evolution rules.* The rules are localized, associated with the membranes (hence, with the compartments), and they are used in a *nondeterministic maximally parallel* manner (a unique clock is assumed, the same for all compartments). A computation consists of a sequence of transitions between system configurations leading to a halting configuration, where no rule can be applied. With a halting computation one associates a result, usually in the form of the number of objects present in a distinguished membrane. Thus, such a system works with numbers inside (multiplicities of objects in compartments) and provides a number as the result of a computation.

From a different perspective, P-systems may be seen as communicating systems. In this view, a P-system, better known as *symport/antiport P-system* [19], computes by moving objects through membranes, in a way inspired by biology. The rules are of the forms (x, in) and (x, out) (*symport* rules, with the meaning that the objects specified by x enter, respectively exit, the membrane with which

[2] See [8], for a similar structure.

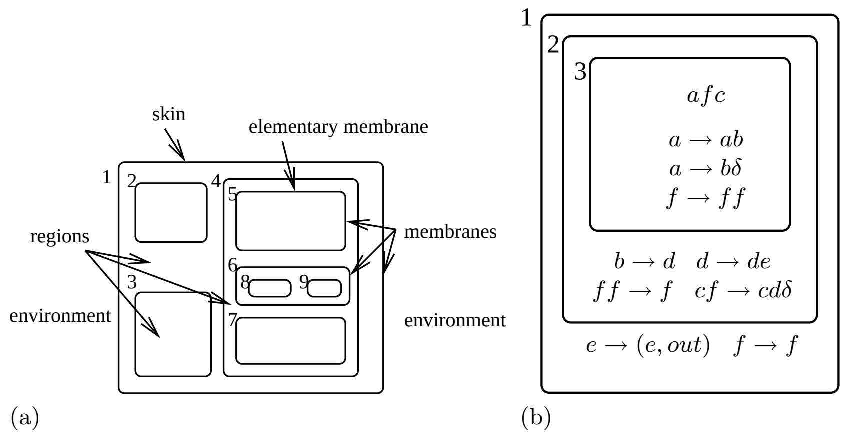


Fig. 1. A membrane system (a) and a "classical" P system (b).

the rule is associated), and $(x, out; y, in)$ (*antiport* rules: the objects specified by x exit and those specified by y enter the membrane at the same time). By rules of these types associated with the skin membrane of the system, objects from the environment can enter the system and, conversely, objects from the system can be sent out into the environment. One can also use *promoters* (*inhibitors*) associated with the symport/antiport rules, in the form of objects which must be present in (resp. absent from) a compartment in order to allow the associated rule to be used.

Finally, a feature which may be added to any of the previous types of P-systems is the possibility to dynamically change the membrane structure. The resulting P-systems are called *P-systems with active membranes* [21].

4 P-systems as transition systems (classical P-systems)

This is the classical type of P-systems, originally introduced in [20]. In this model, each membrane has an associated set of rules. The objects can travel through membranes and they may be transformed by the rules (the rules can create or destroy objects). A membrane may be dissolved and its objects flood into the parent region, while the rules vanish.

4.1 Basic transition P-systems

A *transition P-system*, of degree $m \geq 1$, is formally defined by a tuple

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o),$$

where: (1) O is an alphabet of *objects*; (2) $C \subseteq O$ is the set of catalysts; (3) μ is a membrane structure (with the membranes bijectively labeled by natural numbers $1, \ldots, m$); (4) $w_1, \ldots, w_m$ are multisets over O associated with the regions

$1, \ldots, m$ of μ, represented by strings from O^* unique up to permutations; (5) $R_1, \ldots, R_m$ are finite sets of rules associated with the membranes $1, \ldots, m$; the rules are of the form

$u \to \overline{v}$ or $u \to \overline{v}\delta$
with $u \in O^+$ and $\overline{v} \in (O \times \mathit{Tar})^*$, where $\mathit{Tar} = \{\mathit{here}, \mathit{in}, \mathit{out}\}$

(6) i_o is the label of the output membrane, an elementary one in μ; alternatively, i_o may be 0 indicating that the collecting region is the environment. When δ is present in the rule, its application leads to the dissolution of the membrane and to the abolishment of the rules associated with the membrane just dissolved.

A membrane is denoted by $[_h\ ]_h$. By convention, $[_h u]_h$ denotes a membrane with u present in the solution (among other objects). Starting from the *initial configuration*, which consists of μ and $w_1, \ldots, w_m$, the system passes from one configuration to another by applying a transition, i.e., the rules from each set R_i in a *non-deterministic and maximally parallel* way. A sequence of transitions is called a *computation*; a computation is *successful* if and only if it halts. With a successful computation one associates a *result*, in the form of the number of objects present in membrane i_o in the halting configuration.

An example is presented in Fig. 1(b) – it computes/generates the square numbers $(k+1)^2, k \geq 0$. When a rule with δ is applied, the corresponding membrane and its rules are dissolved and its current objects are flooded into the parent region. A typical, terminating evolution of this system is as follows: It starts in the region of membrane 3 (the other regions have no objects), then membrane 3 is dissolved and the evolution continues in the region of membrane 2, and when membrane 2 is dissolved, the final stage of the execution is in the region of membrane 1 (the skin membrane). In the region of membrane 3, the first two rules have a conflict on a: when the 2nd rule is applied, object a, as well as membrane 3, disappear; the rule for f is independent and has to be used each time another rule is applied; to conclude, when membrane 3 disappears, we are left with b^{k+1} and $f^{2^{k+1}}$ objects which are flooded into the region of membrane 2. In the region of membrane 2, at each cycle the f's are divided by 2 and, in parallel, a copy of each b (now rewritten in d) is created. Finally one gets $(k+1)^2$ copies of object e when membrane 2 is dissolved, which are passed to the region of membrane 1 and then into the external environment.

4.2 Basic transition P-systems in K

Given a P-system $\mathit{\Pi}$, we define a K-system $K(\mathit{\Pi})$, as follows:

– A membrane with a contents $[_h S]_h$ (S is the full contents of $[_h\ ]_h$) is represented in K as[3]

$$(\!|\ S\ |\!)_h$$

The top configuration is $(\!|\ (\!|\ \ |\!)_{skin}\ \ (\!|\ \ |\!)_{env}\ |\!)_\top$, including a representation of the objects in the environment.

[3] Another representation may be $(\!|\ (\!|\ h\ |\!)_{id}\ S\ |\!)_{cell}$, if one prefers a single cell type. This can also be an implementation optimization, to avoid structured cell labels.

– The rules in Π are represented as global rules in $K(\Pi)$, their localization being a side-effect of membrane name matching. The evolution rules reduce to two basic rules used in a membrane $[_h\ ]_h$ represented in K as follows:
  - $u \to \overline{v}$, with $u \in O^+, \overline{v} \in (O \times Tar)^*$, where $Tar = \{here, in, out\}$
    For a $\overline{v} \in (O \times Tar)^*$, let v be its restriction to O. Next (recall that composition is commutative), let v be written as $v_h v_i v_o$, where v_h contains the objects that remain in the region of membrane $[_h\ ]_h$, v_i those that enter into other regions through internal membranes, and v_o those that go out through the membrane. The associated rule in K is the following: for any $k \geq 1$ and $v_i = v_1 \ldots v_k$ (with nonempty v_j's) we have a rule

$$\llparenthesis\ \underset{v_h}{\underline{u}}\ \ \llparenthesis\underset{v_1}{\underline{\ \cdot\ }}\rrparenthesis_{\_} \ldots \llparenthesis\underset{v_k}{\underline{\ \cdot\ }}\rrparenthesis_{\_}\ \ \rrparenthesis_h\ \ \underset{v_o}{\underline{\ \cdot\ }}$$

  - $u \to u\delta$ (δ indicates that the membrane is dissolved)

$$\llparenthesis\ u\ z\ \rrparenthesis_h \to u\ z$$

For the first rule $u \to \overline{v}$, a second possibility would be to perform two steps: move v_h, v_o first, then for all remaining tagged objects (v_r, in) use a matching rule $\llparenthesis\ (v_r, in)\ \llparenthesis\ \underset{v_r}{\underline{\ \cdot\ }}\ \rrparenthesis_{\_}\ \rrparenthesis_{\_}$.

The rules for object movement and membrane dissolution may be combined. For instance, the rule $u \to \overline{v}\delta$, with $v = v_h v_i v_o$ as above, may be represented (in the simple case when all objects which enter through an internal membrane use the same membrane i) by

$$\llparenthesis\ u\ \llparenthesis w \rrparenthesis_i\ z\ \rrparenthesis_h \to v_h\ \llparenthesis w\ v_i \rrparenthesis_i\ z\ v_o$$

In this interpretation, first the objects are moved out through the membrane, then the membrane is dissolved. One can go the other way round, first to dissolve the membrane and then to move the objects out; the K rule is

$$\llparenthesis\ \underset{v_h\ \llparenthesis\ wv_i\ \rrparenthesis_i\ z}{\underline{\llparenthesis\ u\ \llparenthesis\ w\ \rrparenthesis_i\ z\ \rrparenthesis_h}}\ \rrparenthesis_{\_}\ \underset{v_o}{\underline{\ \cdot\ }}$$

Parallel rewriting in K. The rewriting logic in K extends the one in RWL by allowing for overlapping rules which overlap on parts that are not changed by these rule (on "read-only" parts). Such an extension is needed, e.g., when two threads read the store at the same time.

As an extension of the rewriting mechanism in RWL, the rewriting in K allows for the application of an arbitrary number of rewriting rules in a step. However, it does not constrain the user to use a "maximal parallel" rewriting like in the case of P-systems. Such an option may be handled at the meta-level by selecting an appropriate strategy for applying the rules. Actually, there is little evidence that a "maximal parallel" strategy is present in the living cells - it is a working hypothesis to keep the evolution simpler and to find results in the theoretical model.

4.3 Variations of transition P-systems

Object movement. We start with a few variations of the object movement rule (presented in [22]), then we describe their associated rules in K.

- (deterministic *in*) In this variant, in_j is used instead of a simple *in* to indicate that an object goes into the region of the internal membrane j; its K-translation is

$$\langle\!|\ \underset{v_h}{\underline{u}}\ \ \langle\!|\ \underset{v_1}{\underline{\cdot}}\ |\!\rangle_{j_1}\cdots\langle\!|\ \underset{v_k}{\underline{\cdot}}\ |\!\rangle_{j_k}\ |\!\rangle_h\ \ \underset{v_o}{\underline{\cdot}}$$

  where v_h, v_i, v_o are as above and $v_1, \ldots, v_k$ are the objects in v that go into the internal regions of membranes $j_1, \ldots, j_k$, respectively.

- (polarities) One can use classes of membranes with polarities, say $+/0/-$. The newly created objects may have $+/0/-$ polarities, as well. The objects with $+/-$ polarities go into the internal regions of membranes with opposite polarities, while those with 0 polarity may stay or goes outside. This is a case in between the above two extreme alternatives: complete freedom to go in any internal membrane and precise target for each object. Therefore, it is easy to provide K definitions for these situations. There are various ways to add algebraic structure for polarities. For example, one way pair each datum and each membrane label with a polarity; in the case of data we write the polarity as a superscript, e.g., a^+ or a^-, while for the membranes we write it as a superscript of the membrane, e.g., $\langle\!|\ u\ |\!\rangle_h^+$ or $\langle\!|\ u\ |\!\rangle_h^-$. The membrane polarity may be changed, as well. For example, to describe that in the presence of objects u the polarity is changed from $+$ to $-$ one can use the K rule

$$\langle\!|\ u\ |\!\rangle_h^+ \to \langle\!|\ u\ |\!\rangle_h^- \quad \text{or, equivalently,} \quad \langle\!|\ u\ |\!\rangle_h^{\underset{-}{\underline{\pm}}}$$

- (arbitrary jumps) In this variant, one can directly move an object from a region to another region.
  The P-systems rule is $[_h u]_h \to [_{h'} v]_{h'}$. To represent this rule in K we need an explicit rule for matching membranes at different levels, denoted $\langle\!|^*\ \ ^*|\!\rangle$ ($\langle\!|^*\ \ ^*|\!\rangle$ means a match in any recurrently included cell, not only in the current one):
  - if the membranes are not contained one into the other, then

$$\langle\!|\ \langle\!|^*\ \underset{\cdot}{\underline{u}}\ ^*|\!\rangle_h\ \ \langle\!|^*\ \underset{v}{\underline{\cdot}}\ ^*|\!\rangle_{h'}\ |\!\rangle$$

  - if the jump is into the region of an enclosed membrane, then

$$\langle\!|\ \underset{\cdot}{\underline{u}}\ \ \langle\!|^*\ \underset{v}{\underline{\cdot}}\ ^*|\!\rangle_{h'}\ |\!\rangle_h$$

  - if the jump is into the region of an outside membrane, then

$$\langle\!|\ \underset{v}{\underline{\cdot}}\ \ \langle\!|^*\ \underset{\cdot}{\underline{u}}\ ^*|\!\rangle_h\ |\!\rangle_{h'}$$

  Instances of this rule capture the particular cases of in^*/out^*, notation used in P-systems to indicate a movement into an elementary/skin membrane.

Membrane permeability. In the standard setting, a membrane is passive (label 1) and the specified objects can pass through it. The membrane can be dissolved (label 0), as well. One can add impenetrable membranes (label 2), as well. The status of the membranes may be dynamically changed as a side-effect of applying reaction rules.

This case is similar to the case of polarities: One can use pairs (h, i) to represent the membrane *id* and its permeability level. The rules are simple variations of the basic P-systems evolution rules and may be easily handled in K.

Rules with priorities. Capturing various control mechanisms on applying the rules is a matter of strategies. Strategies are commonly held separate of rules in many rewriting approaches. One important way to restrict the maximal parallelism convention is to apply the rules according to their priorities. In a strong version, only the rule with the highest priority applies; in a weak version, when all possible applications of the highest priority rule have been resolved, a next rule with less priority is chosen, and so on.

In the current implementation of P-systems over K and Maude, priorities are handled at Maude "meta-level" and using the corresponding priority algorithm to capture a P-system maximal parallel step. (K has not developed particular strategies and uses the strategies inherited from Maude.)

Promoters/inhibitors. The presence of promoters/inhibitors may be seen as additional context for rules to apply. In the case of promoters, a reaction rule $l \to r$ in membrane h applies only if the objects in a promoter set z are present in the solution (they should be different from those in l):

$$\langle\!|\ \underset{r}{\underline{l}}\ z\ |\!\rangle_h$$

The case of inhibitors is opposite: in the presence of the objects in z the rule cannot apply. The case of inhibitors requires a K rule with a side condition

$$\langle\!|\ \underset{r}{\underline{l}}\ z\ |\!\rangle \quad \text{where } z \not\subseteq x$$

Complex side condition like the above are handled by means of conventional conditional rewrite rules in our Maude implementation of K.

Catalysts. The role of catalysts may be viewed in two opposite ways: (1) either they may be seen as a sine-qua-non ingredient for a reaction $a \to b$ to take place; or (2) they may be seen as a way to control the parallelism, by restricting a free reaction $a \to b$ to the number of occurrences of the catalyst. In other extensions, catalysts may move from a membrane to another, or may change their state (each catalyst is supposed to have a finite number of isotopic forms).

Catalysts are just objects, hence representing their behavior in K is straightforward. However, notice that c is required for a to become b in a catalyst $\langle\!|\underset{bc}{\underline{ac}}|\!\rangle_h$, while for a promoter a becomes b if c is present $\langle\!|\, \frac{a}{b}\ c \,|\!\rangle_h$. In RWL, the two rules above would be identical, while in K, c actively participates to first rule, passively to the second (in the latter case, c may be shared by more rules applied in parallel).

Border rules. Border rules are particular object evolution rules of the following type $xu[_i vy \rightarrow xu'[_i v'y$. They allow to test and modify the objects from two neighboring regions. Such a rule may be represented in K by

$$x\ \frac{u}{u'}\ \langle\!|\ \frac{v}{v'}\ y\ |\!\rangle_{-}$$

5 P-systems as communicating systems (symport/antiport P-systems)

This type of P-systems was introduced in [19]. In this variant, the environment is considered to be an inexhaustible source of objects of any type. The evolution rules are called symport/antiport rules and only move the objects through the membranes (they do not create or destroy objects).

Basic symport/antiport P-systems. A *symport/antiport P system*, of degree $m \geq 1$, is formally defined by a tuple

$$\Pi = (V, T, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m, i_o),$$

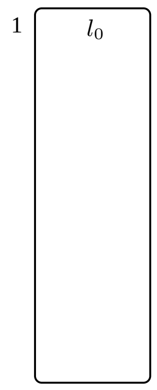
where: (1) V is an alphabet of *objects*; (2) $T \subseteq V$ is the terminal alphabet; (3) μ is a membrane structure (with the membranes bijectively labeled by natural numbers $1, \ldots, m$); (4) $w_1, \ldots, w_m$ are multisets over V associated with the regions $1, \ldots, m$ of μ, represented by strings from V^*; (5) $E \subseteq V$ is the set of objects which are supposed to appear in an arbitrarily large number of copies in the environment; (6) $R_1, \ldots, R_m$ are finite sets of symport and antiport rules associated with the membranes $1, \ldots, m$:

– a symport rule is of the form (x, in) or (x, out), where $x \in V^+$, with the meaning that the objects specified by x enter, respectively exit, the membrane, and
– an antiport rule is of the form $(x, in; y, out)$, where $x, y \in V^+$, which means that x is taken into the membrane region from the surrounding region and y is sent out of the membrane;

(7) i_o is the label of the output membrane, an elementary one in μ.

With the symport/antiport rules one can associate *promoters* $(x, in)|_z$, $(x, out)|_z$, $(x, out; y, in)|_z$, or *inhibitors* $(x, in)|_{\neg z}$, $(x, out)|_{\neg z}$, $(x, out; y, in)|_{\neg z}$, where z is a multiset of objects; such a rule is applied only if z is present, respectively not present.

Starting from the *initial configuration*, which consists of μ and $w_1, \dots, w_m, E$, the system passes from one configuration to another by applying the rules from each set R_i in a *non-deterministic and maximally parallel* way.[4] A sequence of transitions is called a *computation*; a computation is *successful* if and only if it halts. With a successful computation we associate a *result*, in the form of the number of objects from T present in membrane i_o in the halting configuration.



$E = \{a_r | 1 \leq r \leq m\} \cup \{l, l', l'', l''', l^{iv} | l \in B\}$
for l_1 : $(\texttt{add}(r), l_2, l_3)$:
- $(l_1, out; a_r l_2, in)$
- $(l_1, out; a_r l_3, in)$
for l_1 : $(\texttt{sub}(r), l_2, l_3)$:
- $(l_1, out; l_1' l_1'', in)$
- $(l_1' a_r, out; l_1''', in)$
- $(l_1'', out; l_1^{iv}, in)$
- $(l_1^{iv} l_1''', out; l_2, in)$
- $(l_1^{iv} l_1', out; l_3, in)$
for l_h : $\texttt{halt}$:
- (l_h, out)

Fig. 2. A "symport/antiport" P system.

Example. An example is presented in Fig. 2. It describes how symport/antiport P-systems may simulate counter machines (it is well known that counter machines, with at least 2 counters, are computationally universal, see [11]). A counter machine uses a finite number of counters and a program consisting of labeled statements. Except for the begin and halt statements, the other statements perform the following actions: (1) increase a counter by one, (2) decrease a counter by one, or (3) test if a counter is zero. For the simulation in Fig. 2, we use an equivalent definition of counter machines where the statements are of the following two types:

(a) $l_1 : (add(r), l_2, l_3)$ (add 1 to r and nondeterministically go to l_2 or l_3)
(b) $l_1 : (sub(r), l_2, l_3)$ (if r is not 0, subtract 1 and go to l_2, else go to l_3)

The simulation works as follows: The statement to be executed next is in the (unique) cell, while the others stay outside. At each step, the current statement leave the cell and the one to be next executed goes inside the cell. During this process, the counter associated to the statement goes updated. The processing of a type (b) statement is slightly more complicate as a trick is to be used to check if the counter is zero, see [22].

[4] We recall that the environment is supposed inexhaustible, at each moment all objects from E are available in any number of copies we need.

Basic symport/antiport P-systems in K. The previous representation in K of transition P-systems and their variations almost completely covers this new type of P-systems. What is not covered is the behavior of the environment. The K rule is actually an equation

$$\langle\!\!\mid\; x \;\mid\!\!\rangle_{env} = \langle\!\!\mid\; x \;\mid\!\!\rangle_{env}\; x$$

Variations of symport/antiport P-systems. Most of the variations used for transition P-systems can be used here, as well.

6 P-systems with active membranes

The third main class of P-systems brings an important additional feature: the possibility to dynamically change the membrane structure. The membranes can evolve themselves, either changing their characteristics or getting divided.

6.1 Basic P-systems with active membranes

A *P-system with active membranes* [21] is formally defined by a tuple

$$\Pi = (O, H, \mu, w_1, ..., w_m, R)$$

where: (1) $m \geq 1$ (the initial degree of the system); (2) O is the alphabet of objects; (3) H is a finite set of labels for membranes; (4) μ is a membrane structure, consisting of m membranes having initially neutral polarizations, labeled (not bijectively) with elements of H; (5) $w_1, \ldots, w_m$ are strings over O, describing the multisets of objects placed in the m regions of μ; (6) R is a finite set of developmental rules, of the following forms:

– (a) object evolution rules: for $h \in H, e \in \{+, -, 0\}, a \in O, v \in O^*$,

$$[_h a \to v]_h^e$$

– (b) "in" communication rules: for $h \in H, e_1, e_2 \in \{+, -, 0\}, a, b \in O$,

$$a[_h\ ]_h^{e_1} \to [_h b]_h^{e_2}$$

– (c) "out" communication rules: for $h \in H, e_1, e_2 \in \{+, -, 0\}, a, b \in O$,

$$[_h a]_h^{e_1} \to [_h\ ]_h^{e_2} b$$

– (d) dissolving rules: for $h \in H, e \in \{+, -, 0\}, a, b \in O$,

$$[_h a]_h^e \to b$$

– (e) division rules (elementary membranes, only): for $h \in H, e1, e2, e3 \in \{+, -, 0\}, a, b, c \in O$

$$[_h a]_h^{e_1} \to [_h b]_h^{e_2} [_h c]_h^{e_3}$$

The objects evolve in the maximally parallel manner, while each membrane can be involved in only one rule of types (b)-(e). More precisely, first the rules of type (a) are used, and then the other rules.

The label set H has been specified because it is allowed to change the membrane labels. Notice that one uses a dictionary of rules, each label in H coming with its own set of rules. For instance, a division rule can be of the more general form

– (e') general division: for $h_1, h_2, h_3 \in H, e_1, e_2, e_3 \in \{+, -, 0\}, a, b, c \in O$

$$[_{h_1} a]^{e_1}_{h_1} \to [_{h_2} b]^{e_2}_{h_2} [_{h_3} c]^{e_3}_{h_3}$$

One can consider other variations as the possibility of dividing membranes in more than two copies, or even of dividing non-elementary membranes. Therefore, in P-systems with active membranes the membrane structure evolves during the computation not only by decreasing the number of membranes by dissolution, but also increasing it by division.

6.2 Basic P-systems with active membranes in K

Except for membrane division, P-systems with active membranes are similar to transition P-systems, hence we can borrow the previous translation in K. However, for the sake of clarity, we prefer to give the K-representation for the full set of rules (a)-(e). A membrane with polarity is denoted by pairs (h, e) with $h \in H$ and $e \in \{+, -, 0\}$.

– object evolution rules: $[_h a \to v]^e_h$ $(h \in H, e \in \{+, -, 0\}, a \in O, v \in O^*)$

$$\llparenthesis\ \frac{a}{v}\ \rrparenthesis^e_h$$

– "in" communication rules: $a[_h]^{e_1}_h \to [_h b]^{e_2}_h$ $(h \in H, e_1, e_2 \in \{+, -, 0\}, a, b \in O)$

$$\frac{a}{\cdot} \quad \llparenthesis\ \frac{\cdot}{b}\ \rrparenthesis^{\frac{e_1}{e_2}}_h$$

– "out" communication rules: $[_h a]^{e_1}_h \to [_h]^{e_2}_h b$ $(h \in H, e_1, e_2 \in \{+, -, 0\}, a, b \in O)$

$$\llparenthesis\ \frac{a}{\cdot}\ \rrparenthesis^{\frac{e_1}{e_2}}_h \quad \frac{\cdot}{b}$$

– dissolving rules: $[_h a]^e_h \to b$ $(h \in H, e \in \{+, -, 0\}, a, b \in O)$

$$\llparenthesis\ a \quad x\ \rrparenthesis^e_h \to b \quad x$$

– division rules for elementary membranes: $[_h a]^{e_1}_h \to [_h b]^{e_2}_h [_h c]^{e_3}_h$ $(h \in H, e1, e2, e3 \in \{+, -, 0\}, a, b, c \in O)$

$$\llparenthesis\ a \quad x\ \rrparenthesis^{e_1}_h \to \llparenthesis\ b \quad x\ \rrparenthesis^{e_2}_h \quad \llparenthesis\ c \quad x\ \rrparenthesis^{e_3}_h$$

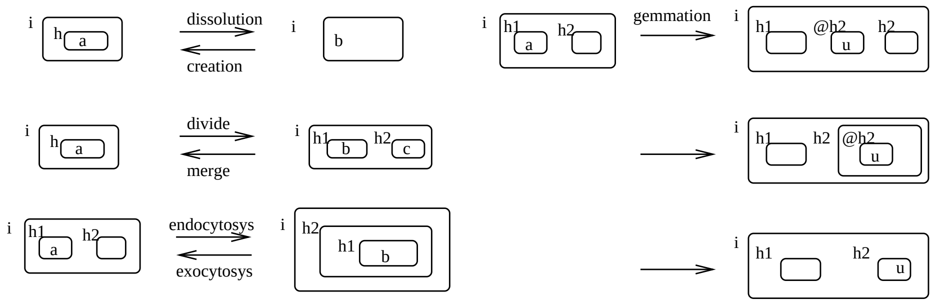


Fig. 3. Membrane handling operations.

6.3 Variations of P-systems with active membranes

Membrane creation. This rule is $a \to [_h v]_h$, i.e., after its application a new membrane is inserted into the system. The rules in the new membrane depend on h and they are taken from a dictionary. The K rule is

$$a \to (\!|\ v\ |\!)_h$$

Merging of membranes. This rule is $[_h x]_h [_{h'} y]_{h'} \to [_{h''} z]_{h''}$, allowing to merge the contents of two neighboring membranes (the rules for $[_{h''}..]_{h''}$ are taken from a dictionary). The K rule is

$$(\!|\ x\ |\!)_h \quad (\!|\ y\ |\!)_{h'} \to (\!|\ z\ |\!)_{h''}$$

Split of membranes. This operation is opposite to merge. Its format is $[_{h''} z]_{h''} \to [_h x]_h [_{h'} y]_{h'}$ and the K rule is

$$(\!|\ z\ |\!)_{h''} \to (\!|\ x\ |\!)_h \quad (\!|\ y\ |\!)_{h'}$$

One appealing version is to put into a membrane the objects of a given type and in the other the remaining ones.

Endocytosys and exocytoses. Endocytosys is a rule $[_h x]_h [_{h'}]_{h'} \to [_{h'} [_h y]_h]_{h'}$, i.e., in one step a membrane and its contents enter into a neighboring membrane. The K rule is

$$\frac{(\!|\ x\ |\!)_h}{\cdot} \quad (\!|\ \frac{\cdot}{(\!|\ y\ |\!)_h}\ |\!)_{h'}$$

Gemmation. One can encapsulate into a new membrane $[_{@h}\ ]_{@h}$ a part u of the solution to be carried to membrane $[_h\ ]_h$. The new membrane $[_{@h} u]_{@h}$ travels through the system, being safe for its contents. By convention, it travels with the speed of one membrane per clock cycle following the shortest path towards the destination.

The final result may be described as in the case of general object jumps. What is different here is the step-by-step journey of $[_{@h} u]_{@h}$. This can be done using the shortest path from h' to h in the tree associated to this membrane system.[5] The details are left to the reader.

7 Implementation, experiments

The intractability of P-systems with plain objects. While P-systems is a model with massive parallelism, its huge potential is not fully exploited by current approaches due to the lack of object structure.[6] The illustrating example of computing n^2 with the P-system in Fig. 1(b) is not a fortunate one. For instance, to represent 999 one needs 999 objects in the membrane and the computation ends up with 998001 objects in the final region; however, during computation an exponential mechanism to record the number of steps is used and in an intermediary soup there are more than 2^{999} objects, significantly more than the atoms in the Universe.[7] There are other sophisticated representation mechanisms in cells which may be used to achieve fast and reliable computations of interest for cells themselves. As shown below, with structured objects we break ground and achieve fast computations for P-systems.

A K/Maude implementation. We have developed an application for running P-systems using our embedding of P-systems in K and the implementation of K in Maude (Maude [10] can be downloaded at http://maude.cs.uiuc.edu/). The application can be accessed online at

`http://fsl.cs.uiuc.edu/index.php/Special:MaudeStepperOnline`

One can choose the examples in the p-system directory. The application allows to run a P-system blindly, or in an interactive way; another option is to ask for displaying the transition graph. (The options for interactive running or graph display make sense for small examples, only.)

Results: structured vs. unstructured objects. We have performed a few experiments, both with plain, unstructured objects and with structured ones.

For the former case, we implemented the P-system in Fig. 1(b). (We have already commented on its intractability above.) We run experiments on our server with the following constraints: up to 2 minutes and using no more than 500 MB of RAM. With these constraints, we were able to compute n^2, but *only up to* $n = 18$. The results are presented in Table 3(a).

For the latter case (structured objects) we limited ourselves to natural numbers and run two experiments with P-systems for computing factorial function and for looking for prime numbers using Eratosthenes sieve. In both cases, we

[5] One does not consider the complicated case when the delivery gets lost into the system due to a reconfiguration of the membrane structure.

[6] A few P-systems with particular structured objects (strings, conformons) are presented in [20, 12].

[7] The Universe is estimated to 4×10^{79} hydrogen atoms, while 2^{999} is roughly 10^{300}.

were able to run large experiments: in the first case, we were able to compute 3000! (a number with 12149 digits) within the given constraints; in the second case, all prime numbers less than 1500 where found in no more than 1 minute. The results are collected in Table 3(b),(c). The transition graph for the example with prime numbers up to $n = 10$ is displayed in Fig. 4.

square	time(ms)	rewritings	parallel steps
6	11	2316	13
10	123	20012	21
15	3882	562055	31
18	32294	4463244	37
19	failure		

(a)

factorial	time(ms)	no rew.	parallel steps
10	0	93	4
100	8	744	7
1000	545	7065	10
3000	6308	21079	12
3500	failure		

(b)

primes	time(ms)	no rew.	parallel steps
10	1	156	2
100	33	16007	6
1000	19007	2250684	14
1500	50208	5402600	15
2000	failure		

(c)

Table 3. Runs for P-systems with and without data structure on objects.

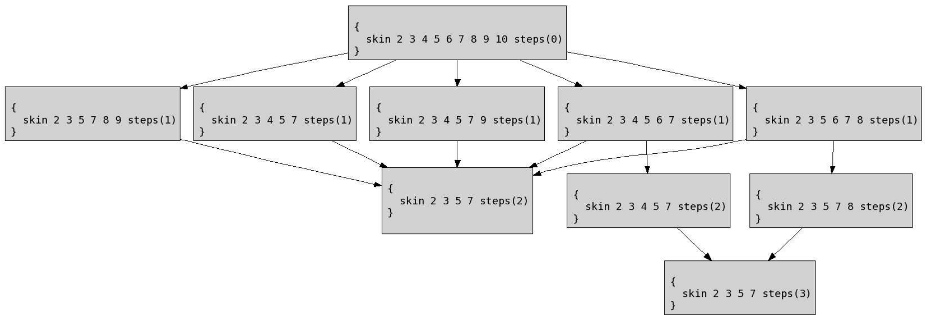

{ skin 2 3 4 5 6 7 8 9 10 steps(0) }
{ skin 2 3 5 7 8 9 steps(1) }
{ skin 2 3 4 5 7 steps(1) }
{ skin 2 3 4 5 7 9 steps(1) }
{ skin 2 3 4 5 6 7 steps(1) }
{ skin 2 3 5 6 7 8 steps(1) }
{ skin 2 3 5 7 steps(2) }
{ skin 2 3 4 5 7 steps(2) }
{ skin 2 3 5 7 8 steps(2) }
{ skin 2 3 5 7 steps(3) }

Fig. 4. The graph of computing the prime numbers up to 10.

The P-systems for the last two examples are flat, the computation being similar to that used in Γ-programs. It is possible to describe P-systems with more membranes for this problem and to find the resulting speedup, but this is out of the scope of the current paper.

8 Related and future work, conclusions

A similar approach to membrane computing via rewriting logic was developed by Lucanu and his collaborators in a series of papers, including [2–5]. The focus in the cited papers was to use rewriting logic to study the existing P-systems, while our approach is more on exploiting the relationship between P-systems and the K framework for enriching each with the strong features of the other.

K was designed as a framework for defining programming languages and has powerful mechanisms to represent programs via its list structures. Our embedding of P-systems in K suggests to include a *control nucleus* in each membrane. The role of this structure it to take care of the rules which are to be used in the membrane. A nucleus generates a set of rules for the next (nondeterministic, parallel maximal) step. When the computation step is finished the rules are deleted and the nucleus produces a new set of rules to be used in the next computation step, and so on. This way, one gets a powerful mechanism for controlling the evolution of P-systems, narrowing their inherent nondeterminism and opening a way to a better understanding of their behavior.

The classical model of P-systems uses a fixed set of rules for each membrane, so a simple nucleus program may be used to generate this set of rules at each step. One can think at multiple possibilities for these nucleus programs – thanks to the structured objects, any program written in a usual programming language may be used. The embedding of P-systems in K described in this paper naturally extends to this new type of P-systems and may be used to get a running environment for them.

References

1. B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J.D. Watson: *Molecular Biology of the Cell*, 3rd ed. Garland Publishing, New York, 1994.
2. O. Andrei, G. Ciobanu, and D. Lucanu: Structural Operational Semantics of P Systems. In: *Proc. Workshop on Membrane Computing 2005*, LNCS 3850, Springer 2006, 31–48.
3. O. Andrei, G. Ciobanu, and D. Lucanu: Expressing Control Mechanisms of Membranes by Rewriting Strategies. In: *Proc. Workshop on Membrane Computing 2006*, LNCS 4361, Springer 2006: 154–169
4. O. Andrei, G. Ciobanu, and D. Lucanu: Operational Semantics and Rewriting Logic in Membrane Computing. *Electr. Notes Theor. Comput. Sci.* 156 (2006), 57–78.
5. O. Andrei, G. Ciobanu, and D. Lucanu: A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science*, 373 (2007), 163–181.
6. J.-P. Banatre, A. Coutant, and D. Le Metayer: A Parallel Machine for Multiset Transformation and Its Programming Style. *Future Generation Computer Systems*, 4 (1988), 133–144.
7. G. Berry and G. Boudol: The Chemical Abstract Machine. *Theoretical Computer Science*, 96 (1992), 217–248.

8. L. Cardelli: Brane Calculi. In: *Proc. Computational Methods in Systems Biology*, LNCS 3082, Springer 2005, 257–278.
9. G. Ciobanu, G. Paun, G. Stefanescu: P Transducers. *New Generation Computing*, 24 (2006), 1–28.
10. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J. F. Quesada: Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285 (2002), 187–243.
11. M. Davis, R. Sigal, and E.J. Weyuker. *Computability, Complexity, and Languages.* Second Edition: Fundamentals of Theoretical Computer Science. Morgan Kaufmann, 1994.
12. P. Frisco. The Conformon-P System: A Molecular and Cell Biology-Inspired Computability Model. *Theoretical Computer Science*, 312 (2004), 295–319.
13. M. Hills and G. Rosu: KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In: *Proc. RTA 2007*, LNCS 4533 Springer 2007, 246–256.
14. M. Hills, T. Serbanuta, and G. Rosu: A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters. *Electr. Notes Theor. Comput. Sci*, 176, 4 (2007), 215–231.
15. J. Meseguer: Conditioned Rewriting Logic as a United Model of Concurrency. *Theoretical Computer Science* 96 (1992), 73–155.
16. P. Meredith, M. Hills, and G. Rosu: *A K Definition of Scheme*. Technical report UIUCDCS-R-2007-2907, October 2007.
17. R. Milner: A Theory of Type Polymorphism in Programming. *J. Computer System Sciences* 17(3) (1978), 348–375
18. R. Milner: *Communication and concurrency.* Prentice-Hall, 1989.
19. A. Paun and G. Paun: The power of communication: P-systems with symport/antiport. *New Generation Computing*, 20 (2002), 295–306.
20. G. Paun: Computing with membranes. *Journal of Computer and System Sciences*, 61 (2000), 108–143.
21. G. Paun: P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal of Automata, Languages and Combinatorics*, 6 (2001), 75–90.
22. G. Paun: *Introduction to Membrane Computing.* 12th Estonian Winter School in Computer Science, 2007.
23. G. Rosu: *K: A Rewriting-Based Framework for Computations – Preliminary version.* Technical Report UIUCDCS-R-2007-2926, Department of Computer Science, University of Illinois, 2007. Previous versions published as technical reports UIUCDCS-R-2006-2802 in 2006, UIUCDCS-R-2005-2672 in 2005. K was first introduced in the context of Maude in Fall 2003 as part of a programming language design course (report UIUCDCS-R-2003-2897). `http://fsl.cs.uiuc.edu/k`.
24. URL: The Web Page of Membrane Computing: http://ppage.psystems.eu/