Agent-Based Simulation of Kernel P Systems with Division Rules Using FLAME

Raluca Lefticaru[1,2], Luis F. Macías-Ramos[3],
Ionuţ Mihai Niculescu[4], Laurenţiu Mierlă[2,4]

[1] CENTRIC, Sheffield Hallam University,
153 Arundel Street, Sheffield, S1 2NU, UK
[2] Department of Computer Science, University of Bucharest,
Str. Academiei nr. 14, 010014, Bucharest, Romania
`raluca.lefticaru@fmi.unibuc.ro`
[3] Research Group on Natural Computing, Dept. Computer Science and Artificial Intelligence, University of Seville Avda. Reina Mercedes S/N, 41012, Sevilla, Spain
`lfmaciasr@us.es`
[4] Department of Mathematics and Computer Science, University of Pitesti,
Str. Targu din Vale 1, 110040, Pitesti, Romania
{`ionutmihainiculescu,laurentiu.mierla`}`@gmail.com`

Abstract. Kernel P systems (or *kP systems*) bring together relevant features from several P systems flavours into a unified *kernel* model which allows solving complex problems using a straightforward *code programming approach*. `kPWorkbench` is a software suite enabling specification, parsing and simulation of kP systems models defined in the kernel P–Lingua (or `kP-Lingua`) programming language. It has been shown that any computation of a kP system involving only rewriting and communication rules can be simulated by a family of Communicating Stream X-Machines (or *CSXM*), which are the core of `FLAME` *agent based simulation environment*. Following this, kPWorkbench enables translating kP-Lingua specifications into FLAME models, which can be simulated in a sequential or parallel (MPI based) way by using the FLAME framework. Moreover, `FLAME GPU` framework enables efficient simulation of CSXM on CUDA enabled GPGPU devices. In this paper we present an extension of kPWorkbench framework to generate FLAME models from kP–Lingua specifications including structural rules; and consider translation of FLAME specifications into FLAME GPU models. Also, we conduct a performance evaluation regarding simulation of equivalent kP systems and CSXM models in kPWorkbench and FLAME respectively.
Keywords: Membrane computing; kernel P systems; communicating stream X-machines; agent-based simulation.

1 Introduction

Membrane computing is, to date, the youngest Natural Computing discipline. It was introduced in 1998 by *Gheorghe Păun*, as a paradigm which addresses models taking inspiration form the structure and functioning of cells present in

living beings, considering such cells as living entities themselves able to process and generate information.

Computing devices of membrane computing are called membrane systems or P systems [22]. Basic ingredients of a P system are (i) *a membrane structure*, consisting in a set of *regions* delimited by *membranes*; and (ii) *multisets of objects* placed within the regions. Objects may be transformed according to some *evolution rules*, which are applied in a non-deterministic maximally parallel way (emulating how chemical reactions take place among compounds). To emulate *cell membrane permeability*, *evolution rules* can transform existing objects within a region and, additionally, *transfer* them among *adjacent* regions – objects pass through the membrane separating the regions.

Basically, there are three ways to categorize membrane systems: *cell-like* P systems, *tissue–like* P systems and *neural-like* P systems. In cell-like P systems, membranes are arranged in a hierarchical way, inspired by the inner structure of the biological cells. In tissue-like P systems, cells are set in nodes of a directed graph, inspired from the cell inter-communication in tissues. Similarly, in neural-like P systems, cells are arranged in nodes of a directed graph, taking inspiration from the way in which neurons exchange information by the transmission of electrical impulses (spikes) along axons. Neither tissue-like nor neural-like consider the possibility of cells containing inner compartments, that is, in such variants cells are elemental compartments.

Kernel P systems (or *kP systems*) [6] are a novel variant of membrane systems aiming to bring together relevant features from several P systems flavours into a unified *kernel* model which allows solving complex problems using a straightforward *code programming approach.* In particular, a kP system model is defined by placing *compartment type instances* in the nodes of a *dynamic graph.* Each type represents a kind of *elemental compartment* which is associated with a sequence of *rule blocks.* Following this, each rule block is defined by both a *set of guarded evolution rules* and an *execution strategy.* A guarded rule associated to a compartment type is an extension of a classic evolution rule where a new syntactical element, the *guard*, is added. A guard is a logical condition over the multiplicity of objects belonging to the multiset associated to any instance of the corresponding type. Evolution rules may be either *rewriting and communication* rules or *structure changing rules* (cell division, cell dissolution, link creation or link destruction rules). A given compartment instance executes its rule blocks sequentially, with applicable rules to be executed for each block according to the its own execution strategy.

P–Lingua framework [27], possibly the most widely known simulation software for membrane computing, provides support for a reduced version of kP systems, known as *simple kernel P systems.* As a separate effort from that of P–Lingua, a brand new software project, known as `kPWorkbench` [7, 28], was created aiming to provide full support for kP systems as well as advanced model checking features. kPWorkbench allows the specification of kP systems in the kernel P–Lingua (or `kP-Lingua)` programming language, which share some similarities with the original P–Lingua one. kPWorkbench framework features a native sim-

ulator, allowing the simulation of kP system models written in kP–Lingua. On the other hand, kPWorkbench's model checking environment permits the formal verification of kernel P system models.

Regarding parallel simulation of kP systems, in [20] it was shown that any computation of a kP system involving only rewriting and communication evolution rules can be simulated by a family of *Communicating Stream X-Machines* (or *CSXM*), which are extended forms of state machines, having memory and processing functions. CSXM can be efficiently simulated in a parallel way by means of two template-based software frameworks called `FLAME` (*Flexible Large-Scale Agent Modelling Environment*) [29] and `FLAME GPU` [30]. FLAME allows MPI [31] based efficient simulation of CSXM models written in the `FLAME` agent-based specification language, while FLAME GPU allows *CUDA (Compute Unified Device Architecture)* [12, 19, 21, 32] based efficient simulation of CSXM models written in the `FLAME GPU` specification language, an extension/variant of the FLAME one. Both FLAME and FLAME GPU have been used in several experiments, which were performed on *High Performance Computing (HPC)* platforms due to the scale of the associated models and, subsequently, resource-intensive simulation tasks. Some examples include modelling oxygen-responsive transcription factors in *Escherichia coli* [1] or the complex cellular tissue simulation [24].

Following this, in [8] kPWorkbench was extended to provide automated translation from kP systems models written in kP–Lingua into CSXM models written in FLAME specification language. This translation addressed kP systems involving only rewriting and communication evolution rules, with the transformation of systems involving structural rules left as an open issue. Moreover, no support for automated translation to FLAME GPU was provided either.

In this work, we take a step forward regarding the aforementioned results tackling new challenges. Firstly, we address an extension of the kPWorkbench framework to generate FLAME models from kP–Lingua specifications including structural rules such as division and dissolution rules. Secondly, we address the translation of FLAME specifications into FLAME GPU models. Finally, we conduct a performance study regarding the simulation of equivalent kP systems and CSXM models in kPWorkbench and FLAME (serial and parallel mode) respectively. This is conducted following the trail of [2] and [13]. In particular, [13] presents a first approach of implementing the pulse generator model in FLAME GPU [13], conducting a performance comparison with FLAME. Remarkably, the FLAME GPU model used was manually translated, since there was no public available tool to automate the conversion.

This paper is structured as follows. Section 2 outlines previous related work. Section 3 introduces the theoretical background. Section 4 presents our modelling approach in FLAME, while Section 5 presents the possible ways of extending it to FLAME GPU. A case study illustrating the cited approach is discussed in Section 6. Finally, conclusions and further work are drawn in Section 7.

2 Related Work

In this Section, we briefly outline the state-of-the art of parallel simulation of P systems on High Performance Computing (HPC) platforms.

Both P–Lingua and kPWorkbench, as a vast majority of software tools for membrane computing, implement the simulation algorithms in a *sequential* way. This effectively neglects the inherent parallelism of P systems, and leads to non-efficient simulations from the computational complexity point of view. Fortunately, an increasing variety of simulators specially intended to run on *massively-parallel platforms* have been developed along the years. Such HPC platforms include *Field Programmable Gate Array circuits (FPGAs)* [23], *microcontrollers* [9], *computer clusters* [3, 4, 26] and *General–Purpose Graphic Processing Unit (GPGPU)* devices.

In particular, GPGPU hardware comprises a very affordable technology, providing in a *single* device *hundreds* of *massively parallel processors* supporting several *thousand* of concurrent *threads*. To date, many general purpose applications have been successfully migrated to GPGPU platforms, showing good *speed-ups* compared to their corresponding sequential versions. Two are the main programming models enabling software development oriented to GPGPUs. On the one hand, *CUDA (Compute Unified Device Architecture)* programming model, [12, 19, 21, 32] and on the other hand, *Open Computing Language (OpenCL)* framework [18, 25, 33].

An updated exhaustive list of parallel simulators for P systems regarding the aforementioned approaches can be consulted in [14], with the reader also encouraged to check [5] and [15], from where an encyclopaedic knowledge can be obtained. Finally, a couple of surveys summarising the topic can be consulted in [16, 17].

With respect to parallel simulation of kernel P systems, to the best knowledge of the authors, there are only a few related software applications due to the novelty of the model. On the one hand, a parallel implementation on GPGPU architectures using CUDA is reported in [11]. On the other hand, regarding the simulation of kP systems with agents, FLAME and FLAME GPU simulation platforms are detailed in [2, 13, 20], as we discussed above.

3 Background

This Section gives the basic definitions and major results regarding kernel P systems and communicating stream X-machines, following largely from [6, 20]. For this, we will assume that the reader is familiar with usual notations from formal languages, membrane computing and finite automata domains and refer to [6, 10, 20] for further technical details and examples.

We first begin recalling the formal definition of kernel P systems (or kP systems).

Definition 1. *A* kP system *of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where*

- *A is a finite set of elements called objects;*
- *μ defines the membrane structure, which is a graph, (V, E), where V are vertices representing components (compartments), and E edges, i. e., links between components;*
- *$C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, is a compartment of the system consisting of a* compartment type, *t_i, from a set T and an initial multiset, $w_{i,0}$ over A; the type $t_i = (R_i, \rho_i)$ consists of a set of evolution rules, R_i, and an execution strategy, ρ_i;*
- *i_0 is the output compartment where the result is obtained.*

A kernel P system can have several compartment instances of the same type: while they share the same set of rules and execution strategies, they may have different multiset of objects at different computation steps and different neighbours according to the graph relation specified by (V, E). Within the kernel P systems framework, the following kinds of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \longrightarrow y\{g\}$, where $x \in A^+$ and y represents a multiset of objects over A^* with potential different compartment type targets (each symbol from the right side can be sent to a different compartment, specified by its type; if more compartments of the same type are linked to the current compartment, then one is randomly chosen).
  Compared to cell-like P systems, the targets in kP systems are the type of compartments to which the objects will be sent, not particular instances. Also, for kP systems, complex guards can be represented, using multisets over A with relational and Boolean operators.
  For example, rule $r : ab \longrightarrow bc\{\geq a^3 \wedge < b^2\}$ can be applied if and only if the current multiset includes the left hand side of r, i. e., ab and the guard holds: the current multiset has at least 3 a's and less then 2 b's.
- *structure changing* rules: membrane division, membrane dissolution, link creation and link destruction rules, which all may also incorporate complex guards and that are covered in detail in [6].

As we stated above, besides of a set of evolution rules, each compartment type in a kP system has an associated execution strategy. Execution strategies offer a lot of flexibility to the kP system designer, as the rules corresponding to a compartment can be grouped in several blocks, every block having one of the following strategies:

- *sequential*: if the current rule is applicable, then it is executed, advancing towards the next rule/block of rules; otherwise, the execution terminates;
- *choice*: a non-deterministic choice within a set of rules, one and only one applicable rule will be executed if such a rule exists, otherwise the whole block is simply skipped;
- *arbitrary*: the rules from the block can be executed zero or more times by nondeterministically choosing any of the applicable rules;
- *maximal parallel*: the classic execution mode used in membrane computing.

On the other hand, a stream X-machine is an extended form of finite state machine in which the transitions are labelled by (partial) functions (called *processing functions*) instead of simple symbols. Remarkably, the machine has a memory M, that can be imagined as the domain of the variables of the system to be modelled. The input received by the machine is processed in order: depending on the current state of the machine and the input symbol to be processed, one of the processing functions will read the current input symbol, discard it from the input sequence and produce an output symbol while (possibly) changing the value of the memory and taking the machine to a different state. Finally, if several processing functions can compute the same input, then one of them is randomly chosen (non-determinism). Formal definition of stream X-machines follows:

Definition 2. *A* Stream X-Machine *(SXM for short) is a tuple* $Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$*, where:*

- Σ *and* Γ *are finite sets called the input alphabet and output alphabet respectively;*
- Q *is the finite set of states;*
- M *is a (possibly) infinite set called memory;*
- Φ *is the type of* Z*, a finite set of function symbols. A basic processing function* $\phi : M \times \Sigma \longrightarrow \Gamma \times M$ *is associated with each function symbol* ϕ*.*
- F *is the (partial) next state function,* $F : Q \longrightarrow 2^Q$*. As for finite automata,* F *is usually described by a state-transition diagram.*
- I *and* T *are the sets of initial and terminal states respectively,* $I \subseteq Q$*,* $T \subseteq Q$*;*
- m_0 *is the initial memory value, where* $m_0 \in M$*;*
- *all the above sets, i. e.,* $\Sigma, \Gamma, Q, M, \Phi, F, I, T$ *, are non-empty.*

Several theoretical frameworks have been developed addressing communicating stream X-machines, that is, concurrent systems where different stream X-machines work in parallel exchanging data via communication channels. In what follows, we recall the one from [10], which is the closest to the the implementation of FLAME, according to [20].

Definition 3. *A Communicating* Stream X-Machine System *(CSXMS for short) with* n *components is a tuple* $S_n = ((Z_i)_{1 \le i \le n}, E)$*, where:*

- $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i,0})$ *is the SXM with number* i*,* $1 \le i \le n$*.*
- $E = (e_{ij})_{1 \le i,j \le n}$ *is a matrix of order* $n \times n$ *with* $e_{ij} \in \{0, 1\}$ *for* $1 \le i, j \le n$*,* $i \ne j$ *and* $e_{ii} = 0$ *for* $1 \le i \le n$*.*

The CSXMS works as follows:

- Each individual Communicating SXM (CSXM for short) is a SXM plus an infinite input queue (FIFO structure); the CSXM consumes inputs from its queue.
- An input symbol received from the external environment (also a FIFO structure) will move to the input queue of one CSXM, if it is contained in its input alphabet. If more than one CSXM satisfies such condition, then the symbol will enter the input queue of one of these in a non-deterministic fashion.

- Each pair of CSXMs, say Z_i and Z_j, have two unidirectional communication channels. The communication channel from Z_i to Z_j is enabled if $e_{ij} = 1$ and disabled otherwise
- There exists the possibility for an output symbol produced by a CSXM, say Z_i, to pass to the input queue of another CSXM, say Z_j, providing that the communication channel between them is enabled, and if the symbol is included in the input alphabet of Z_j. If several CSXMs satisfy these conditions one of them is non-deterministic chosen, whereas if none exists the symbol goes to the output environment (also a FIFO structure).

One important result proving the possibility of simulating kP systems with CSXMs is given in the following theorem from [20].

Theorem 1. *For any kP system, $k\Pi$, of degree n and using only rewriting and communication rules, there is a communicating stream X-machine system, S_{n+1}, with $n+1$ components such that for any multiset w computed by $k\Pi$ there is a complete sequence of transitions in S_{n+1} leading to $s(w)$.*

In this Theorem, w is the final configuration of the kP system, $w = (w_1, \ldots, w_n)$, where each w_i represents the final multiset occurring in compartment i. On the other hand, $s(w)$ corresponds to any of the strings obtained by concatenating the symbols occurring in w. Remarkably, the Proof of this Theorem, as shown in [20], suggests the manner in which a FLAME model for a given kP system can be constructed. We will briefly examine this in the next Section.

4 Modelling Kernel P Systems with Structure Changing Rules in FLAME

In this Section, we describe how kernel P systems incorporating structure changing rules can be mapped into FLAME specification language. We start recalling the way in which Communicating Stream X-Machines Systems are defined in that language.

FLAME framework provides an environment for defining communicating agents, specified in an XML format, which contains information regarding their memory variables, name of processing functions, message structures that can be exchanged for communication, etc. FLAME uses an implementation of CSXMs in which: (a) the associated automaton of each CSXM has no loops (this ensures that the execution will end after a finite number of processing functions calls); (b) the CSXMs receive no inputs from the environment – inputs are usually those produced in the previous computation step or the ones defined in the initial configuration file; and (c) the processing function scripts are written in C files.

The input Communicating Stream X-Machines System is defined in XMML format (X-Machine Mark-up Language), which is translated by FLAME into simulation program source code, either in its serial or parallel (MPI) version.

Next, this program can be compiled together with the agent processing functions script files (written in C) by any C/C++ compiler, giving place to simulation executable code can be run, in the case of the parallel version, on High Performance Computing (HPC) platforms.

To take advantage of the aforementioned efficient simulation capability of FLAME, in [8] kPWorkbench was extended to provide automated translation from kP systems models written in kP–Lingua into CSXM models written in FLAME specification language. This translation addressed kP systems involving only rewriting and communication evolution rules, with the transformation of systems involving structural rules left as an open issue. In what follows, we outline the main ideas in which such transformation process relies on (additional details can be found in [8, 13, 20]):

- Each compartment type in the kP system is associated an agent type in FLAME, while each compartment type instance is associated an agent of the corresponding agent type.
- Multisets of objects from each compartment type instance are stored in the corresponding agent memory using, in general, dynamic arrays of complex data types.
- Execution strategies (sequential, choice, arbitrary, maximal parallel) of the compartment types are encapsulated in C functions.
- Communication between compartments is materialized by using FLAME's agents message passing mechanism. In particular, communication among linked compartments is ensured by means of message filtering, while the non-deterministic choice among one of the possible target compartments is ensured by means of non-deterministic message processing.
- Finally, the graph structure of the kP system, which maps the links among compartments, can be stored in a distributed way among each agent memory as a dynamic array containing the identifiers of the agents representing the compartments sharing an active link with the compartment represented by the current agent.

Next, we address how kP systems incorporating structural rules (membrane division, membrane dissolution, link creation, link destruction) can be translated into FLAME. Let us notice that previous experiments regarding kP systems to FLAME translation, such as the ones references above, did not considered kP systems having structural rules. We start describing the process for membrane dissolution, link creation and link destruction, which is quite straightforward compared to that of membrane division:

- Membrane dissolution can be implemented by either (1) extending the agent memory with a flag-type data value storing whether the corresponding compartment is active or has been dissolved; or (2) removing the corresponding FLAME agent, when the membrane dissolution takes place. The first approach should be used when keeping trace of kP system evolution is required and, subsequently, agent deletion is not aimed. In both approaches,

each time that a dissolution takes place, all the agents having connections to the "dissolved" agent have to be notified via messages, in order to update their connection arrays.
- Link creation and link destruction rules can be implemented by adding or removing elements in the connection arrays accordingly.

In what follows we address transformation of kP systems involving division rules. Translation of such rules is the most challenging to implement and, consequently, we devote a specific part of this paper to its study.

4.1 Implementing kP Systems Division Rules in FLAME

Let us recall that, in general, P systems operate by applying rewriting rules defined over multisets of objects associated to the different membranes, in a synchronized non-deterministic maximally parallel way. P systems show a double level of parallelism: a first level comprises parallel application of rules within individual membranes, while a second level comprises all the membranes working simultaneously, that is, in parallel. These features make P systems powerful computing devices. In particular, the double level of parallelism allows a space-time trade-off enabling *the generation of an exponential workspace in polynomial time*. This is usually accomplished by applying iteratively membrane creation rules, such as division rules.

As such, P systems are suitable to tackle relevant real-life problems, usually involving **NP**-complete problems. Moreover, P systems are excellent tools to investigate on the computational complexity boundaries, in particular tackling the **P versus NP** problem. In this way, by studying how the ingredients relative to their syntax and semantics affect to their ability to solve **NP**–complete problems in a feasible way, computational properties, sharper frontiers between efficiency and non-efficiency can be discovered.

In order to take advantage of the full power of P systems, it is required to simulate them on HPC platforms, which can suitably manage the demanding resource requirements of their inherent double parallelism. In the particular case of kernel P systems, this involves efficiently simulating division rules on massively parallel devices. This can be accomplished by transforming the corresponding kP systems models into FLAME specification language.

Regarding this, FLAME inherently favours the possibility of simulating membrane division, since it supports adding new agents during the simulation execution in such a way that all the newly created agents are introduced at the beginning of the next iteration. Nevertheless, several tasks have to be performed to properly simulate membrane division of a given compartment:

- The newly created agent memory has to be initialized with the data corresponding to the multiset of the underlying newly created compartment.
- The connection arrays of both the newly created agent and each agent representing a compartment linked to the original dividing one have to be updated. As such, it is required to implement a mechanism for the newly created agents to be associated unique identifiers.

– This is accomplished by creating in the system a single instance of a new agent type, called the *instance manager*. This agent will receive and process requests of new identifiers from agents representing dividing compartments via message passing. The instance manager will then provide – again via message passing – a new identifier that will be used by the "dividing" agent to initialize the newly created agents and to send a connection array update signal to all its linked agents.

Fig. 1 shows the visual representation of a FLAME model incorporating different kind of translated evolution rules. In particular, this state machine visualisation is automatically drawn by the FLAME editor, based on the kPWorkbench generated model of a kP system solving `SubSetSum`. States are represented with ellipses, processing functions with rectangles, and messages exchanged between agents with green parallelograms.

Comparing the Main and Output compartments in Fig.1, one can check that the Main compartment has additional states and transitions corresponding to the application of division rules, preliminaries for the creation of new membranes, request identifiers or receive identifiers messages.

It is worth pointing out that a FLAME agent structure will vary depending on the execution strategies and blocks from its corresponding kP system compartment. For example, comparing the kP–Lingua specification from Fig. 2 and its corresponding FLAME model in Fig. 1, one can easily identify that each *choice* block has a corresponding processing function and next state in the CSXM. Similarly, the execution order respect of several strategies is reflected in the agent structure. In addition, ramifications may appear for the cases when a structural rule is chosen to be applied.

Following the process outlined above, the existing kP–Lingua to FLAME kPWorkbench translator module has been extended to support translation of division and dissolution rules with the corresponding algorithm included as an Appendix at the end of this paper. The new module has been successfully used to generate FLAME models for instances of `SubSetSum`. Such instances have then been used in the experiments detailed in Section 6.

5 Adapting the Modelling Approach to FLAME GPU

In this Section we briefly discuss some design constraints when translating FLAME models to FLAME GPU and recommended workarounds.

Although FLAME GPU framework is an extension of FLAME, models designed for FLAME are not supported by FLAME GPU. Apart from small differences that could be easily tackled, e.g. using a slightly different XML schema, with different namespace or tag names, there are also important differences in the data types which can be used by each environment.

In order to address these issues, some design guidelines have to be taken into account, as noted in [13], where authors manually translated a model of a pulse generator from FLAME to FLAME GPU, in order to have it implemented in both frameworks and compare their performance with kPWorkbench.

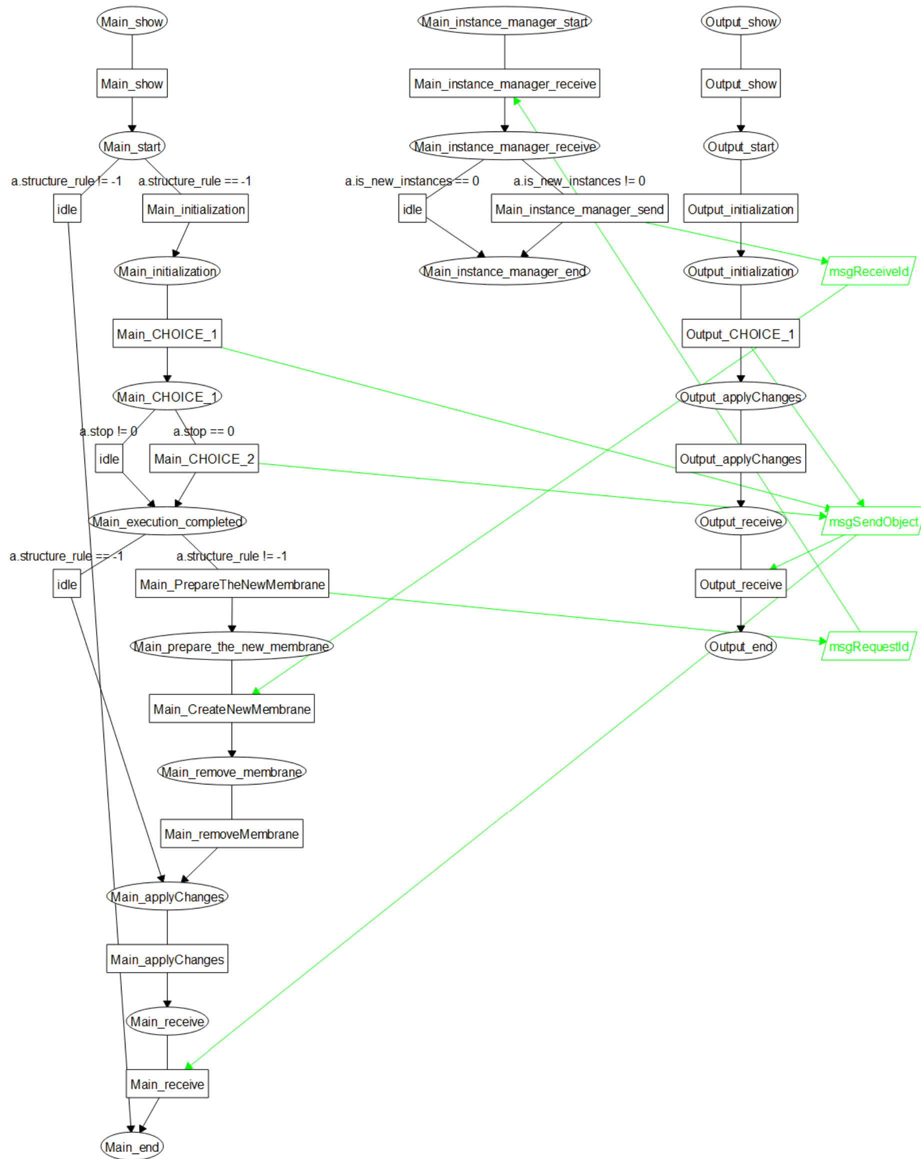

Fig. 1: Graph of the states, functions and messages between agents, corresponding to the FLAME model for Subset Sum problem, having two agent types for the compartments types Main and Output, plus and an additional Main Instance Manager agent in charge of the Main compartment division process (for allocation of new identifiers)

Firstly, memory in FLAME GPU is pre-allocated due to CUDA programming model constraints. As such, agents memory neither support dynamic arrays nor

fixed arrays with complex types. The recommender workaround is the serialization of dynamic arrays (with/without composed objects) which appear in the FLAME models into static arrays of basic types, recommended to have fixed length equal to a power of 2.

Secondly, although in FLAME it is possible to add several new agents in one step, which is useful for example when a membrane is divided into 3 new compartments, in FLAME GPU only one agent is possible to be added per function. Consequently, the recommended workaround implies creating additional functions in the X-machine structure, to add the remaining membranes to be created.

Finally, contrary to FLAME, in FLAME GPU each agent can only create a single message, which clash with communication rules semantics, where multiple compartments may receive different objects. The recommended workaround is to expand the memory space for each message, allowing it to contain data for multiple targets.

6 Case Study: Subset Sum Problem

As previous work on modelling kP systems with FLAME [2, 20] and FLAME GPU [13] addressed models with communication and rewriting rules, in this Section we will illustrate the case of a kP–Lingua model consisting in a kP system family solving `SubSetSum`, which involves division rules as well as other kernel P systems specific features, such as presence of guards plus sequential and choice execution strategies.

The `SubSetSum` problem can be roughly described as: "Given the set $S_n = \{a_1, a_2, \ldots, a_n\}$, is there a subset of S_n, having the sum of elements equal to x?" Regarding to our model, in order to ensure that the computation will continue until all the possible membrane divisions have been applied, we have considered $S_n = \{1, 2, \ldots, n\}$ and the expected sum $x = n(n+1)/2$, which will return the *yes* answer in this case. The kP-Lingua specification considered for $n = 4$ is given in Fig. 2, however corresponding files have been generated for each $n \in \{2, 3, \ldots, 20\}$, more details and a complete folder containing all the files and scripts needed to run the experiments are provided for download on the kPWorkbench website [5].

The model shown in Fig. 2 is a slightly adapted version from that of [8], and was chosen because of its programming-like structure (sequential blocks) and its simplicity regarding beforehand computation of the number of expected execution steps ($n+1$) and number of membranes in the last configuration ($2^n + 1$). Two compartment types are used, named *Main* and *Output*, respectively. The *Main* compartment type contains two choice blocks to (1) generate the *positive answer* when required; and (2) generate the subsets to be checked by applying division rules. The *Output* compartment type take care of (1) controlling the execution by means of a *counter* object; and (2) generating the *negative answer* when required.

[5] `http://kpworkbench.org/index.php/case-studies`

Simplicity of the model eased the experimentation process, since the goal was to conduct non-deterministic experiments but assuring that the computation would end after the same number of steps, with a maximum number of membranes created by division, but different execution traces each time.

```
type Main {
  choice {
    = 10x: a -> {yes} (Output) .
    > 10x: a -> halt .
  }
  choice {
    !r1: a -> [a, r1][x, a, r1] .
    !r2: a -> [a, r2][2x, a, r2] .
    !r3: a -> [a, r3][3x, a, r3] .
    !r4: a -> [a, r4][4x, a, r4] .
  }
}
type Output {
  choice {
    start -> step .
    <5step : step -> 2step .
    <yes & =5step: step -> 2step, no, halt .
  }
}
main {a} (Main) - output {start} (Output) .
```

Fig. 2: kP–Lingua specification for `SubSetSum` (n=4)

In order to assess the performance of the different modelling approaches and implementations for kPWorkbench and FLAME, we conducted several experiments involving different instances of the presented kP system solution to `SubSetSum`, and their translated FLAME counterparts, respectively. With respect to FLAME, besides the serial simulation of the models, which was addressed in previous works like [2, 13, 20], also the parallel MPI based simulation provided by FLAME was considered. Since FLAME does not support MPI simulation in Windows environments, the Sevilla HPC Server [34] `mulhacen` was configured with FLAME and Open MPI [35] to conduct the experiments.

Comparisons were realised between average execution times for: kPWorkbench, FLAME in serial mode and FLAME in parallel version, run with different number of processors, $NP \in \{2, 3, 4\}$, running all on the same machine, a Xeon Server, having 4 cores, Intel i5 Xeon E3-1230V3 @ 3.30GHz, main memory 32 GBytes DDR3 @ 2400Mhz.

For each instance of `SubSetSum`, $n \in \{2, 3, \ldots, 20\}$ and each tool configuration, 3 runs were considered. In each case the user time + system time was recorded and the average total time for these three runs was considered.

n	Compart.	Flame	kPWorkbench	Flame NP 2	Flame NP 3	Flame NP 4
2	4	0.00	0.09	0.00	0.00	0.18
3	8	0.00	0.09	0.00	0.00	0.22
4	16	0.00	0.09	0.00	0.00	0.18
5	32	0.00	0.10	0.00	0.00	0.23
6	64	0.01	0.11	0.19	0.19	0.21
7	128	0.02	0.14	0.21	0.25	0.28
8	256	0.05	0.15	0.18	0.27	0.43
9	512	0.10	0.24	0.27	0.43	0.51
10	1024	0.15	0.36	0.47	0.67	0.85
11	2048	0.26	0.65	0.82	1.10	1.74
12	4096	0.53	1.29	1.52	2.19	3.42
13	8192	1.04	2.78	3.03	4.74	6.33
14	16384	2.33	6.73	6.47	9.59	14.91
15	32768	5.30	17.69	15.03	22.73	30.10
16	65536	13.41	52.56	35.56	53.58	70.36
17	131072	36.17	202.01	89.81	138.66	187.16
18	262144	106.16	832.31	250.45	387.24	507.62
19	524288	352.28	3388.19	793.75	1211.27	1609.34
20	1048576	1088.26	13605.23	3575.90	5787.97	7885.79

Table 1: Average times for solving Subset sum problem for different n values

Table 1 shows the statistics after 3 runs. Columns have the following meaning: n value; the number of compartments resulted from division at the end of computation (2^n of type Main); average time needed by Flame serial version, by kPWorkbench, by Flame parallel version using 2, 3 or 4 processors. All the times are given in seconds and were measured by `/usr/bin/time -v`, in order to have the same metric for all the tools.

Tools configuration. Because all the execution times would have been increased by FLAME saving to disk all the intermediary configurations, we have chosen to output the same amount of information with each tool (chosen rules and new configurations) and save only the last configuration file in XML format for the FLAME simulation.

Comparing our case study with previous experiments assessing FLAME and kPWorkbench performance [2, 13], this is the first time when FLAME is not saving large XML files after each iteration. This modification, realised in order to have equal conditions, is explaining why the FLAME simulator is obtaining better times in this case study, compared to kPWorkbench, although in previous experiments it has been different. Also, another important difference is the type of the model, previous experiments did not use division and this could result in different execution times.

As in the previous experiments only the serial version of FLAME was employed, we lack of other results to compare with, in order to address the question: why are the times obtained in the parallel version much higher than the serial

version? One explanation could be that model chosen does not have very complex processing functions, so the time needed for communication between processors is increasing the total time needed, without benefiting from the parallelism. Another explanation could be some tweaking settings of FLAME or Open MPI, which could be better configured, in order to get the best performance for the parallel implementation.

Also, in Table 1 we have provided the number of compartments for each n as a *rough* estimator of the space used: with a larger n the number of rules and also of object types for the compartment increases. So, a larger number of compartments comes also with more rules and more object types to be stored.

In order to have a better visualisation of the experimental results, the data from Table 1 has been represented in Fig. 3. The left plot shows the average time versus the number of compartments, the right plot displays for the horizontal axis the logarithmic scale of its left counterpart.

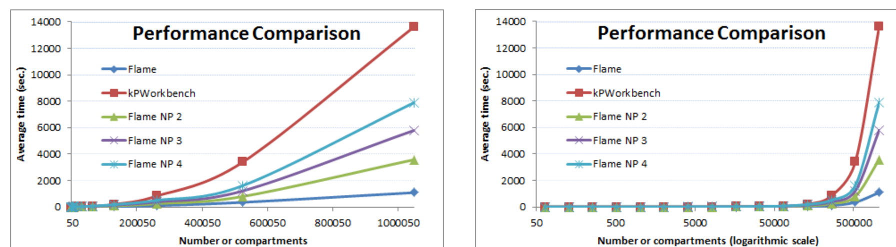

Fig. 3: Comparative simulation results for kPWorkbench, FLAME serial and parallel versions, with NP 2, 3 and 4

Unfortunately, at the time of writing this paper, there are neither public available tools for conversion from kP-Lingua specifications to FLAME GPU, nor from FLAME models to FLAME GPU. This is the reason why FLAME GPU models were not considered in the evaluation, although we have studied this possibility. However, as reported in a previous article [13], where a pulse generator model was translated manually to FLAME GPU, better execution times are expected for GPU version, but the construction of the model is much more tedious compared to the FLAME version.

7 Conclusions and Further Work

This paper presents recent efforts towards modelling of kernel P systems with structural rules in two agent based simulation environments, known as FLAME and FLAME GPU. In this context, we have extended the module of kPWorkbench for automated generation of FLAME models from kP–Lingua specifications to consider kP systems with division and dissolution rules. We also provided an overview of the differences between FLAME and its GPU version, and

outlined the main issues that should be taken into account for a model transformation.

Finally, we have conducted experiments to compare the performances of FLAME, serial and parallel versions, with respect to kPWorkbench, for a kernel P system with division rules.

As future work, we will tackle the automated translation from either kP-Lingua specifications or their FLAME counterparts models to FLAME GPU specifications, based on the main ideas presented here. Also we will address alternative MPI implementations and realise performance comparisons about the application of division and dissolution rules on more complex examples.

Acknowledgements

The work of RL, IN and LM was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PNII-ID-PCE-2011-3-0688).

References

[1] H. Bai, M. D. Rolfe, W. Jia, S. Coakley, R. K. Poole, J. Green, and M. Holcombe. Agent-based modeling of oxygen-responsive transcription factors in *Escherichia coli. Plos computational biology*, 10(4), 2014.

[2] M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High Performance Simulations of Kernel P Systems. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*. IEEE, 2014, pp. 409–412.

[3] G. Ciobanu and G. Wenyuan. P Systems Running on a Cluster of Computers. In C. Martín–Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*. Vol. 2933, in Lecture Notes in Computer Science, pp. 123–139. Springer Berlin Heidelberg, 2004.

[4] L. Diez Dolinski, R. Núñez Hervás, M. Cruz Echeandía, and A. Ortega. Distributed Simulation of P Systems by Means of Map-Reduce: First Steps with Hadoop and P-Lingua. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*. Vol. 6691, in Lecture Notes in Computer Science, pp. 457–464. Springer Berlin Heidelberg, 2011.

[5] M. García-Quismondo. Modelling and simulation of real-life phenomena in Membrane Computing. PhD thesis. University of Seville, Nov. 2013.

[6] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierla, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez. Kernel P Systems - Version I. *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*:97–124, Aug. 2013.

[7] M. Gheorghe, F. Ipate, L. Mierla, and S. Konur. kPWorkbench: A Software Framework for Kernel P Systems. In *Thirteenth Brainstorming Week on Membrane Computing, WBMC 2015, Sevilla, Spain, February 2-6, 2015.* L. F. Macías-Ramos, G. Păun, A. Riscos-Núñez, and L. Valencia–Cabrera, editors. Fenix Editora, 2015, pp. 179–194.

[8] M. Gheorghe, S. Konur, F. Ipate, L. Mierla, M. E. Bakir, and M. Stannett. An Integrated Model Checking Toolset for Kernel P Systems. In *Membrane Computing - 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers.* G. Rozenberg, A. Salomaa, J. M. Sempere, and C. Zandron, editors. Vol. 9504. In Lecture Notes in Computer Science. Springer, 2015, pp. 153–170.

[9] A. Gutiérrez, L. Fernández, F. Arroyo, and V. Martínez. Design of a Hardware Architecture Based on Microcontrollers for the Implementation of Membrane Systems. In *Symbolic and Numeric Algorithms for Scientific Computing, 2006. SYNASC '06. Eighth International Symposium on*, 2006, pp. 350–353.

[10] F. Ipate, T. Bălănescu, P. Kefalas, M. Holcombe, and G. Eleftherakis. A new model of communicating stream X-machine systems. *Romanian Journal of Information Science and Technology*, 6(1):165–184, 2003.

[11] F. Ipate, R. Lefticaru, L. Mierlă, L. V. Cabrera, H. Han, G. Zhang, C. Dragomir, M. J. P. Jiménez, and M. Gheorghe. *Kernel P Systems: Applications and Implementations.* In. *Proceedings of The Eighth International Conference on Bio–Inspired Computing: Theories and Applications (BIC-TA), 2013.* Z. Yin, L. Pan, and X. Fang, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1081–1089.

[12] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2010.

[13] S. Konur, M. Kiran, M. Gheorghe, M. Burkitt, and F. Ipate. Agent-Based High-Performance Simulation of Biological Systems on the GPU. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015.* IEEE, 2015, pp. 84–89.

[14] L. F. Macías-Ramos. Developing efficient simulators for cell machines. PhD thesis. University of Seville, Feb. 2016.

[15] M. Á. Martínez-del-Amor. Accelerating Membrane Systems Simulators using High Performance Computing with GPU. PhD thesis. University of Seville, May 2013.

[16] M. A. Martínez-del-Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núòez, and M. J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. *Fundam. Inf.*, 136(3):269–284, July 2015.

[17] M. A. Martínez-del-Amor, L. F. Macías-Ramos, L. Valencia-Cabrera, and A. R. a. Mario J. Pérez-Jiménez. Accelerated simulation of P systems on the GPU: A survey. In *Bio-inspired computing - theories and applications - 9th international conference, BIC-TA 2014, wuhan, china, october 16-19, 2014. proceedings.* L. Pan, G. Paun, M. J. Pérez-Jiménez, and T. Song, editors. Vol. 472. In Communications in Computer and Information Science. Springer, 2014, pp. 308–312.

[18] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley, 1st ed., 2011.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[20] I. Niculescu, M. Gheorghe, F. Ipate, and A. Stefanescu. From Kernel P Systems to X-Machines and FLAME. *Journal of Automata, Languages and Combinatorics*, 19(1-4):239–250, 2014.

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[22] G. Păun. Computing with Membranes. *J. Comput. Syst. Sci.*, 61(1):108–143, 2000.

[23] B. Petreska and C. Teuscher. A Reconfigurable Hardware Membrane System. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*. Vol. 2933, in Lecture Notes in Computer Science, pp. 269–285. Springer Berlin Heidelberg, 2004.

[24] P. Richmond, D. C. Walker, S. Coakley, and D. M. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[25] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 864–876.

[26] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739–750, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[27] The P-Lingua Website.
URL: `http://www.p-lingua.org/`.

[28] kPWorkbench Home Page.
URL: `http://kpworkbench.org/`.

[29] FLAME website.
URL: `http://www.flame.ac.uk/`.

[30] FLAME GPU website.
URL: `http://www.flamegpu.com/`.

[31] The Message Passing Interface (MPI) standard.
URL: `http://www.mcs.anl.gov/research/projects/mpi/`.

[32] The NVIDIA Website.
URL: http://www.nvidia.com/content/global/global.php.

[33] OpenCL standard webpage.
URL: http://www.khronos.org/opencl.

[34] The Sevilla HPC Server.
URL: http://www.gcn.us.es/gpucomputing/.

[35] The Open MPI project.
URL: https://www.open-mpi.org/.

Algorithm 1 Transforming a kP Systems into Flame algorithm

```
1: procedure AddTransition(startState, stopState, strategy, guard)
   ▷ procedure adding the appropriate transition strategy to the current agent stack given as parameter and FLAME function applying rules conforming to execution strategy
   ▷ guard is an optional parameter that represents the transition guard
2:    if strategy is Sequence then
3:       agentTransitions.Push(startState, stopState, SequenceFunction, guard)
         ▷ FLAME function SequenceFunction applies rules in sequentially mode
4:    else if strategy is Choice then
5:       agentTransitions.Push(startState, stopState, ChoiceFunction, guard)
         ▷ FLAME function ChoiceFunction applies rules in choice mode
6:    else if strategy is ArbitraryParallel then
7:       agentTransitions.Push(startState, stopState, ArbitraryParallelFunction, guard)
         ▷ FLAME function ArbitraryParallelFunction applies rules in arbitrary parallel mode
8:    else if strategy is MaximalParallel then
9:       agentTransitions.Push(startState, stopState, MaximalParallelFunction, guard)
         ▷ FLAME function MaximalParallelFunction applies rules in maximal parallel mode
10:   end if
11: end procedure
12:
   ▷ main algorithm for traforming a kP system into Flame
13:
14: agentsStates.Clear()
15: agentsTransitions.Clear()
   ▷ empty state and transition stacks of agents
16: foreach membrane in kPSystem do
   ▷ for each membrane of kP system build corresponding agent, consisting of states and transitions
17:    agentStates.Clear()
18:    agentTransitions.Clear()
       ▷ empty state and transition stacks of agent that is built for the current membrane
19:    agentStates.Push(startState)
       ▷ adding the initial state of the X machine
20:    agentStates.Push(initializationState)
       ▷ adding initialization state
21:    agentTransitions.Push(startState, initializationState, IsNotPreviousApplyStructureRule)
       ▷ adding transition between the initial and initialization states; this transition performs objects allocation on rules and other initializations; if the agent is active, no rule of structure has been applied in the previous iteration
22:    agentTransitions.Push(startState, endState, IsPreviousApplyStructureRule)
       ▷ adding transition between the initial and end state; if the agent is inactive, a rule of structure has been applied in the previous iteration
23:    foreach strategy in membrane do
       ▷ for each strategy of the current membrane the corresponding states and transitions are built
24:       previousState = agentStates.Top()
          ▷ the last state is stored in a temporary variable
25:       if is first strategy and strategy.hasNext() then
          ▷ when the strategy is the first of several, state and transition corresponding to the execution strategy are added
26:          agentStates.Push(strategy.Name)
27:          AddTransition(previousState, strategy.Name, strategy)
28:       else
29:          if not strategy.hasNext() then
             ▷ if it is the last strategy, the transition corresponding to the execution strategy is added
30:             AddTransition(previousState, completedExecutionState, strategy)
31:          else
```

Algorithm 1 Transforming a kP Systems into Flame algorithm (continued)

```
32:             agentStates.Push(strategy.Name)
            ▷ add corresponding state of the current strategy
33:             if strategy.Previous() is Sequence then
            ▷ verify that previous strategy is of sequence type
34:                 AddTransition(previousState, strategy.Name, strategy, IsApplyAllRules)
                ▷ add transition from preceding strategy state to the current strategy state. The guard is
                  active if all the rules have been applied in the previous strategy transition.
35:                 agentTransitions.Push(previousState, completedExecutionState, IsNotApplyAllRules)
                ▷ add transition from preceding strategy state to state in which all strategies were finalized.
                  The guard is active if not all rules have been applied in the previous strategy transition
36:             else
37:                 AddTransition(previousState, strategy.Name, strategy)
                ▷ add transition from preceding strategy state to the current strategy state
38:                 agentTransitions.Push(previousState, completedExecutionState, IsApplyStructureRule)
                ▷ add transition from preceding state strategy to state in which all strategies were finalized.
                  The guard is active when the structural rule has been applied on the previous strategy
                  transition
39:             end if
40:         end if
41:     end if
42:   end for
43:   agentStates.Push(completedExecutionState)
    ▷ adding state in which all strategies were finalized
44:   agentStates.Push(PrepareTheNewMembranes)
    ▷ adding state in which id(s) is required for newly created membrane(s)
45:   agentTransitions.Push(completedExecutionState, PrepareTheNewMembranes, IsApplyStructureRule)
    ▷ add the transition to the prepare the new membranes state on which id(s) is required for newly created
      membrane(s), if the structure rule has been applied. The request is made through messages to the
      instance manager, agent that allocate new IDs for new agents of current type.
46:   agentStates.Push(CreateNewMembrane)
    ▷ adding state in which IDs are received through messages from instance manager agent and the new
      agents are created
47:   agentTransitions.Push(PrepareTheNewMembranes, CreateNewMembrane)
    ▷ on this transition the new agents are created with the new received ids
48:   agentStates.Push(applyChangesState)
    ▷ adding state in which changes produced by the applied rules are committed
49:   agentTransitions.Push(completedExecutionState, applyChangesState, IsNotApplyStructureRule)
    ▷ add transition to the apply changes state where changes produced by rules are applied, if has not been
      applied any structure rule
50:   agentTransitions.Push(applyChangesState, receiveState)
    ▷ adding transition on which changes produced by the applied rules are committed
51:   agentStates.Push(receiveState)
    ▷ add state that receives objects sent by applying the communication rules in other membranes
52:   agentTransitions.Push(receiveState, endState)
    ▷ add transition to the end state that receives objects sent by applying the communication rules in other
      membranes
53:   agentStates.Push(endState)
    ▷ add the final state
54:   agentsStates.PushAll(agentStates.Content())
    ▷ add the contents of the stack that holds the current agent states to the stack that holds the states of all
      agents
55:   agentsTransitions.PushAll(agentStates.Content())
    ▷ add the contents of the stack that holds the current agent transitions to the stack that holds the transitions
      of all agents
56: end for
```