Modelling Dynamically Organised Colonies of Bio-entities

Marian Gheorghe[1], Ioanna Stamatopoulou[2], Mike Holcombe[1], and Petros Kefalas[3]

[1] Department of Computer Science, University of Sheffield, UK
{M.Gheorghe, M.Holcombe}@dcs.shef.ac.uk
[2] South-East European Research Center, Thessaloniki, Greece
istamatopoulou@seerc.info
[3] Department of Computer Science, CITY College, Thessaloniki, Greece
kefalas@city.academic.gr

Abstract. The dynamic nature of biological systems' structure, and the continuous evolution of their components require new modelling approaches. In this paper it will be investigated how these systems composed of many dynamic components can be formally modelled as well as how their configurations can be altered, thus affecting the communication between parts. We use two different formal methods, communicating X-machines and population P systems, both with dynamic structures. It will be shown that new modelling approaches are required in order to capture the complex and dynamic nature of these systems.

1 Introduction

Biological systems are modelled in different ways depending on the aim of the model. There are models trying to exhibit the general behaviour of the system based mainly on continuous approaches. In this way a generic description of the system's behaviour is defined in terms of mathematical functions evolving in time. Another perspective is based on individual components interacting toward achieving certain goals. In this latter case an emergent property of the system, not obvious from the components' behaviour, is mostly envisaged. For example the behaviours of the social insects are directed towards the benefit of the colony as a whole, and this is done through self-organisation and specialisation. Local interactions with other insects, and with the environment produce solutions to problems that colonies face. No one insect in the colony can give a picture of the whole environment, but information can be learnt through interaction. Bees, for example, can determine how busy a colony is when they bring nectar to hive. Instead of passing all this onto one bee (who will distribute it), small portions of nectar will be passed onto many bees. The bee can determine how busy the hive is by calculating how long it has to wait to pass nectar onto another bee [11].

This perspective on modelling biological systems is investigated in this paper and mainly relies on describing components as agents. An agent is a fairly

J.-P. Banâtre et al. (Eds.): UPP 2004, LNCS 3566, pp. 207–224, 2005.
© Springer-Verlag Berlin Heidelberg 2005

complex computer system that is situated in some environment and is capable of flexible, autonomous actions in order to meet its design objectives [16]. The extreme complexity of agent systems is due to substantial differences between the attributes of their components, high computational power required by the processes running within these components, huge volume of data manipulated by these processes and finally possibly extensive amount of communication in order to achieve coordination and collaboration. The use of a computational framework that is capable of modelling both the dynamic aspects (i.e. the continuous change of agents states together with their communication) and the static aspects (i.e. the amount of knowledge and information available), will facilitate modelling and simulation of such complex systems.

Many biological processes seem to behave like multi-agent systems, as for example a colony of ants or bees, a flock of birds, cell tissues etc. [6]. The vast majority of computational biological models based on an assumed, fixed system structure is not realistic. The concept of growth, division and differentiation of individual components (agents) and the communication between them should be addressed in order to create a complete biological system which is based on rules that are linked to the underlying biological mechanisms allowing a dynamic evolution.

For example, consider the case of an ant colony. Each ant has its own evolution rules that allow it to grow, reproduce and die over time or under other specific circumstances; other rules define the movement behaviour of the ants. The ants are arranged in some two- or three-dimensional space, and this layout implies the way ants interact with others in the local neighbourhood. The structure of the colony, changes over time, thus imposing a change in their interactions.

In the last years attempts have been made to devise biology inspired computational models in the form of generative devices [25], [26], unconventional programming paradigms [2], bio-engines solving NP hard problems [1], adequate mechanisms to specify complex systems [13]. In this paper we have selected two formal methods, X-machines and population P systems, in order to model biological systems with dynamic organisation as multi-agent systems. Each of these methods possesses different characteristics which will be examined through the modelling process. These modelling paradigms take their inspiration from biology and are used to specify problems occurring in nature.

The structure of this paper is as follows: Section 2 describes the biological system modelled in this paper. Sections 3 and 4 present the theory regarding communicating X-machines and population P systems, respectively. Section 5 presents the actual models developed for an ant colony behaviour. Section 6 discusses some issues concerning the experiments conducted. Finally, Section 7 concludes the paper.

2 Pharaoh's Ants

Monomorium pharaonis, the Pharaoh's ants, are species of small ants that originated from North Africa. They measure up to two millimetres in length. The

small size of the ants make them ideal for studying in a laboratory as their living environment requires little room. Colonies have a rapid reproductive cycle, around five and half weeks from egg to an adult, which is another useful trait for a study colony.

Typically a Pharaoh's ant colony will contain anywhere between 100 and 5000 ants. The smallest natural colonies of around 100 ants usually contain: one queen, 35 workers, 12 pupae and some brood. The largest colonies tend to have over 100 queens. 200 ants are usually used for experimental purposes to keep the colony manageable.

The ants spend much of their time doing nothing; this redundancy in the colony allows them to respond rapidly to large food finds. This allows them to efficiently transport the food to the nest before their competitors. Ants doing nothing can be referred as inactive. An ant can become active in many different ways: spontaneously by hunger, being recruited to forage by another ant, another ant soliciting food or another ant offering food. These interactions tend to happen within the nest.

The problem that will be modelled further on in this study presents the behaviour of a simple colony of ants in a nest. The Pharaoh's ants behaviour takes into account a very simplified situation where the colony is sitting in an rectangular environment and consists only of workers. The ants are either inactive or move around looking for food and when this is not found then they go outside the hive to forage for food. When two ants come across they might exchange food if one is hungry and the other one is not - it was in an inactive state. The ants go out to forage when they are hungry, no source food is identified (i.e. no other ant that might provide some food) and a trail pheromone leading to an exit point from the hive is discovered.

This simple problem is of interest for a number of reasons:

- it is a simple and realistic enough case study
- it shows a combination of both independent behaviour of ants inside of the environment as well as synchronised behaviour, e.g. when two ants come across to exchange food
- it has an important degree of repetitiveness using the same type of ant in a number of instances but also slightly small variations between them through the food distribution across the ant colony and their different position in the environment
- it uses different activities that requires distinct execution time periods.

There are a number of thresholds associated to the level of food that is exchanged between two ants, the level of food defining the hungry state, the time to forage for food.

This case study will be modelled by using two approaches, the communicating X machine paradigm and the population P system approach. The two methods have complementary appealing characteristics. X-machines being a state-based formalism appear to be more suitable for representing their internal data and knowledge of each of the participating entities (ants), and how the stimuli received from the environment can change their internal state. They have also been

extended so as to facilitate communication among components and this allows the modelling of a collection of units in an incremental manner that distinguishes between the individual components definitions and the communicating issues. Though work is being done towards this direction, the way X-machines are defined does not accommodate a straightforward way of dealing with systems dynamically reconfigured. In an attempt to find alternative ways towards this end, in the form of other computing devices that may exhibit this characteristic, effort is being dedicated to exploring the modelling prospects of Population P Systems, which naturally (by definition) employ the quality of reconstructing themselves. Finally, work has also been done on finding a formal relationship among the two formalisms [20] whereby simple rules are established for the transformation of P systems into X-machines.

3 Communicating X-Machines

The X-machines formal method [7], [12] forms the basis for a specification language with a great potential to software engineers. It is rather intuitive while at the same time formal descriptions of data types and functions can be written in any known mathematical notation.

For modelling systems containing more than one agent, the X-machine components need to be extended with new features, such as hierarchical decomposition and communication. A communicating X-machine model consists of several X-machines that are able to exchange messages. This involves the modelling of the participating agents and the definition of the rules of their communication.

The complete model is a *communicating X-machine system* Z defined as a tuple:

$$Z = ((C_i)_{i=1,\ldots,n}, CR)$$

where:

- C_i is the i-th communicating X-machine component, and
- CR is a relation defining the communication among the components, $CR \subseteq C \times C$ and $C = \{C_1, \ldots, C_n\}$. A tuple $(C_i, C_k) \in CR$ denotes that the X-machine component C_i can output a message to a corresponding input stream of the X-machine component C_k for any $i, k \in \{1, \ldots, n\}$, $i \neq k$.

A *communicating X-machine component* C_i is defined as a tuple [22]:

$$C_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi C_i, F_i, q_{0_i}, m_{0_i})$$

where:

- Σ_i and Γ_i are the input and output alphabets respectively.
- Q_i is the finite set of states.
- M_i is the (possibly) infinite set called memory.
- ΦC_i is a set of partial functions φ_i that map an input and a memory value to an output and a possibly different memory value, $\varphi_i : \Sigma_i \times M_i \rightarrow \Gamma_i \times M_i$.

There are four different types of functions in ΦC_i (in all of the following it is $\sigma \in \Sigma_i$, $\gamma \in \Gamma_i$, m, $m' \in M_i$; $(\sigma)_j$ means that input is provided by machine C_j whereas $(\gamma)_k$ denotes an outgoing message to machine C_k):

- the functions that read input from the standard input stream and write their output to the standard output stream:
$$\varphi_i\ (\sigma, m) = (\gamma, m')$$
- the functions that read input from a communication input stream and write their output to the standard output stream:
$$\varphi_i\ ((\sigma)_j, m) = (\gamma, m')$$
- the functions that read input from the standard input stream and write their output to a communication output stream:
$$\varphi_i\ (\sigma, m) = ((\gamma)_k, m')$$
- the functions that read input from a communication input stream and write their output to a communication output stream:
$$\varphi_i\ ((\sigma)_j, m) = ((\gamma)_k, m')$$

– F_i is the next state partial function, $F_i : Q_i \times \Phi C_i \rightarrow Q_i$, which given a state and a function from the type ΦC_i determines the next state. F_i is often described as a state transition diagram.
– q_{0_i} and m_{0_i} the initial state and initial memory respectively.

Graphically on the state transition diagram we denote the acceptance of input by a stream other than the standard by a solid circle along with the name C_j of the communicating X-machine component that sends it. Similarly, a solid diamond with the name C_k denotes that output is sent to the C_k communicating X-machine component. An abstract example of a Communicating X-machine component is depicted in Fig. 1.

The above allows the definition of systems of a static configuration. However, most multi-agent systems are highly dynamic and this requires that their structure and the communication among the agents is constantly changing. For this to happen in a communicating X-machine model, control has to be taken over by another system acting on a higher level. This controlling device can be modelled as a set of meta-rules that refer to the configuration of the system or as a meta-X-machine that will be able to apply a number of operators which will be affecting the structure of the communicating system [21]. These operators are defined below.

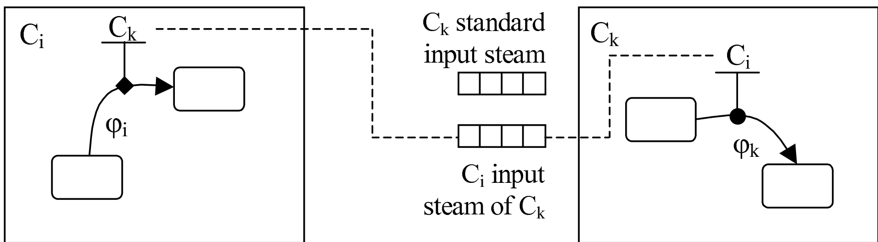

Fig. 1. An example of communication between two X-machine functions

Attachment Operator. This operator is responsible for establishing communication between an existing communicating X-machine component and a set of other existing components. Its definition is:

$$\mathbf{ATT}: \ \mathcal{C} \times \mathcal{Z} \rightarrow \mathcal{Z}$$

where $\mathcal{C}$ is the set of communicating X-machine components, and $\mathcal{Z}$ is the set of communicating X-machine systems. For an existing component $C \in \mathcal{C}$ and a communicating X-machine system Z (to which C belongs to) a new communicating X-machine system Z' will be built that has different communication channels. The components remain the same except that for each function φ of the component machine C the streams of the other components, if any, it receives inputs from or sends outputs to are specified. Similarly, the communicating functions of the other components, with which C establishes communication, become related to the streams of the component C so that input can be received or output can be sent to it. It is this kind of relationships between the component C and the other components that define how the whole system is to communicate as a collection of units cooperating through streams of data.

Detachment Operator. This operator is used in order to remove communication channels between an existing communicating X-machine component and a set of other existing components with which it currently communicates. Its definition is:

$$\mathbf{DET}: \ \mathcal{C} \times \mathcal{Z} \rightarrow \mathcal{Z}$$

where $\mathcal{C}$, $\mathcal{Z}$ are defined as previously. In this case all the relationships between the component C and its streams and the other components and their streams are broken down.

Generation Operator. A new communicating X-machine component is created and introduced into the system. If communication is required, according to the underlying communication rules, between the new and existing component(s), then communication channels are established. The definition of the operator is:

$$\mathbf{GEN}: \ \mathcal{C} \times \mathcal{Z} \rightarrow \mathcal{Z}$$

where $\mathcal{C}$, $\mathcal{Z}$ are defined as previously.

Destruction Operator. This operator removes the component from the system along with all the communication channels that relate it to other components. This means that the corresponding streams that were used so that other components could send/receive messages to/from the removed component are also removed. The operator is defined as follows:

$$\mathbf{DES}: \ \mathcal{C} \times \mathcal{Z} \rightarrow \mathcal{Z}$$

where $\mathcal{C}$, $\mathcal{Z}$ are defined as previously.

Conceptually, the meta-system could be considered to play the role of the environment to the actual communicating system. Because the meta-machine should be able to control the reconfiguration of the communicating system through the application of the above operators, it should possess the following information at all times:

- The communicating system $Z = ((C_1, \ldots, C_i, \ldots, C_n), CR)$,
- The current system state SZ of Z. SZ is defined as a set of tuples $SZ = \{sz \mid \exists C_i, 1 \leq i \leq n,\ sz = (q_c, M_c, \varphi_c)_i$, where q_c is the current state in which C_i is in, M_c is the current memory of C_i and φ_c is the last function that was applied in $C_i\}$,
- Definitions of all components that exist or may be added to the system. These definitions act as genetic codes (GC) for the system. GC is a set of tuples, $GC = \{\ldots(\Sigma, \Gamma, Q, M, \Phi, F, \Phi_\mathrm{R}, \Phi_\mathrm{W})_j, \ldots\}$ where the first six elements are as in the definition of the X-machine given in the previous section and the last two the set of functions that may be involved in communication with other components (i.e. Φ_R includes the functions that may read from communicating streams and Φ_W the ones that may write to communicating streams). In other words, only the types of components that may appear in the system at any point are a priori fixed.

Using the above information, the control device can generate a new component and attach it to the communicating machine Z, through the operator GEN, destruct an existing component of Z and rearrange the communication of the other components appropriately, through the operator DES, and add or remove channels of communication between a component and a communicating machine due to some system reconfigurations, through the operators ATT and DET.

The communicating X-machine system provides a modelling tool, where a complex system can be decomposed in small components that can be modelled as simple X-machine models. The communication side of all these components can be specified separately in order to form the complete system as a communicating X-machine model. This implies a modular bottom-up approach and supports an iterative gradual development. It also facilitates the reusability of existing components, making the management of the whole project more flexible and efficient, achieving its completion with lower cost and less development time.

The communicating X-machine method supports a disciplined modular development, allowing the developers to decompose the system under development and model large scale systems. Since the communicating X-machine model is viewed as the composition of X-machine type components with their initial memory and initial state as well as with a set of input/output streams and associations of these streams to functions, the development of a general model of a complex system can be mapped into the following distinct actions: (a) Develop X-machine type components independently of the target system, or use existing models as they are. (b) Code the X-machine model into XMDL. With the use of tools that are built around the XMDL language it is possible to syntactically check the model and then automatically animate it [23]. Through this simulation it is possible for the developers to informally verify that the model corresponds

to the actual system under development, and then also to demonstrate the model to the end-users aiding them to identify any misconceptions regarding the user requirements. (c) Use the formal verification technique (model checking) for X-machine models in order to increase the confidence that the proposed model has the desired characteristics. This technique enables the designer to verify the developed model against temporal logic formulas that express the properties that the system should have. (d) Test the implementation against the model. X-machines support not only static but also dynamic analysis. It is possible to use the formal testing strategy to test the implementation and prove its correctness with respect to the X-machine model. (e) Create X-machine instances of the original types and determine the way in which the independent instance models communicate. (f) Extend the model to a communicating system in order to provide additional functionality by defining the interaction between components. (g) Define appropriate meta-rules that describe the reconfiguration of the system.

With the continuous verification and testing from the early stages risks are reduced and the developer is confident of the correctness of the system under development throughout the whole process. It is worth noticing that components that have been verified and tested can be reused without any other quality check.

X-machine modelling is based on a mathematical notation, which, however, implies a certain degree of freedom, especially as far as the definition of functions are concerned. In order to make the approach practical and suitable for the development of tools around X-machines, a standard notation is devised and its semantics fully defined [19]. The aim is to use this notation, namely X-Machine Description Language (XMDL), as an interchange language between developers who could share models written in XMDL for different purposes. To avoid complex mathematical notation, the language symbols are completely defined in ASCII.

Briefly, an XMDL model is a list of definitions corresponding to the construct tuple of the X-machine definition. The language also provides syntax for (a) use of built-in types such as integers, Booleans, sets, sequences, bags, etc., (b) use of operations on these types, such as arithmetic, Boolean, set operations etc., (c) definition of new types, and (d) definition of functions and the conditions under which they are applicable. In Table 1 basic keywords used in XMDL to describe a stream X-machine are presented and briefly explained. In XMDL, the functions take two parameter tuples, i.e. an input symbol and a memory value, and return two new parameter tuples, i.e. an output and a new memory value. A function may be applicable under conditions (*if-then*) or unconditionally. Variables are denoted by a preceding `?`. The informative `where` in combination with the operator `<-` is used to describe operations on memory values. A function has the following general syntax:

```
#fun <function name> ( <input tuple> , <memory tuple> ) =
if <condition expression> then
   ( <output tuple>, <memory tuple> )
where   <informative expression>.
```

Table 1. XMDL keywords

X-machine element	XMDL syntax	Informal semantics
$\mathcal{M}$	$\#model\ < modelname >$	Assigns a name to a model
Σ	$\#input\ < setofinputs >$	Describes the input set
Γ	$\#output\ < setofoutputs >$	Describes the output set
Q	$\#states\ < setofstates >$	Defines the set of states
M	$\#memory\ < memorytuple >$	Defines the memory tuple
q_0	$\#init_state\ < state >$	Sets the initial state
m_0	$\#init_memory\ < memory >$	Sets the initial memory
F	$\#transition\ (q, \phi) = q$	Defines each transition in F
Φ	$\#fun\ < functiondefinition >$	Defines a function in Φ

XMDL has also been extended (XMDL-c) in order to code communicating components. XMDL-c is used to define instances of models by providing a new initial state and a new initial memory instance:

```
#model <model_instance> instance_of <model_type>
with:
#init_state <initial_state>;
#init_memory <initial_memory>.
```

In addition, XMDL-c provides syntax that facilitates the definition of the communicating functions. The general syntax is the following:

```
#communication of function <function_name>:
#reads from <model instance>;
#writes <message tuple> to <model_instance>
    using <variable> from output <output tuple> and
    using <variable> from input <input tuple> and
    using <variable> from memory <memory tuple>
where   <informative expression>.
```

A function can either read or write or both from other components (model instances). It is not necessary to specify the incoming message because it is of the same type as the input defined in the original component. However, it is necessary to specify the outgoing message as a tuple which may contain values that exist in either output or input tuples of the function or even in the memory tuple of the component. The informative expression is used to perform various operations on these values before they become part of the outgoing message tuple.

Based on XMDL and XMDL-c, various tools have been developed [18] such as translators to other notations or executable code (e.g. Z, Prolog), an animator that simulates the computation occurring in an X-machine or communicating X-machine system, a model checker for X-machines etc. It should be worthwhile to investigate towards expanding model-checking and testing techniques for the Communicating X-machine formalism though this should be attempted after formally establishing the theoretical framework.

4 Population P System Model

Membrane computing represents a new and rapidly growing research area which is part of the natural computing paradigm. Already a monograph has been dedicated to this subject [26] and some fairly recent results can be found in [27], [24]. Membrane computing has been introduced with the aim of defining a computing device, called *P system*, which abstracts from the structure and the functioning of living cells [25]. Membranes are among the main elements of the living cells which separate the cell from its environment and split the content of the cell into small compartments by means of internal membranes. Each compartment contains its own enzymes and their specialized molecules. Therefore, a membrane structure has been identified as the main characteristic of every P system that is defined as a hierarchical arrangement of different membranes embedded in a unique main membrane that identify several distinct regions inside the system. Each region contains a finite multiset of objects and a finite set of rules either modifying the objects or moving them from a place to another one. Formally we have the following:

Definition 1. *A P system is a construct*

$$\Pi = (V, \mu, w_1, w_2, \ldots, w_n, R_1, R_2, \ldots, R_m, i_O),$$

where:

1. *V is a finite set of symbols called objects;*
2. *μ is a membrane structure consisting of m membranes, with the membranes (and hence the regions) injectively labeled by $1, 2, .., m$;*
3. *for each $1 \leq i \leq n$, R_i is a finite set of evolution and communication rules; the evolution rules rewrite different objects with others and the objects of the result may stay in the same region or may go into another one; pure communication rules, called also symport/antiport rules exchange objects between two regions (for details see [26]);*
4. *$i_O \in \{1, 2, .., m\}$ is the label of an elementary membrane that identifies the output membrane.*

The basic feature of a P system is the membrane structure μ that consists of a hierarchical arrangement of m distinct membranes embedded in a unique main membrane called the skin membrane. This membrane structure is usually represented as a string of pairs of matching square brackets, which are labeled in an one-to-one manner by $1, 2, .., m$. Each pair of square brackets represents a membrane (membrane i) with its corresponding region (the region delimited by membrane i, or region i). Moreover, this representation makes possible to point out the relationships of inclusions among membranes and regions: we say a region i contains a membrane j if and only if, the pair of square brackets labeled by i embraces the pair of square brackets labeled by j.

Then, each region i contains a finite multiset of objects w_i, which defines the initial content of the region i, and a finite set of rules R_i.

As usual, by starting from the initial configuration, a computation is obtained by applying to the objects contained in the various regions the corresponding set of rules in a maximal parallel manner. A computation is said to be successful if it reaches a configuration where no more rules can be applied to the objects in the system.

A natural generalisation of the P system model can be obtained by considering P systems where its structure is defined as an arbitrary graph. Each node in the graph represents a membrane and contains a multiset of objects and a set of rules modifying these objects and communicating them between membrane components. The communication between two components is possible if they are connected by an edge of the graph [26]. These networks of communicating membranes are also known as *tissue P systems* because, from a biological point of view, they can be interpreted as an abstract model of multicellular organisms. If the components are regarded not only as simple cells surrounded by membranes but as more general bio-entities then this model may be considered for more complex organisms, or colonies of simple or more complex components.

These populations of individuals are usually far from being stable; mechanisms enabling new components to be added or removed, links between them to be dynamically updated, play a fundamental role in the evolution of a biological system as a population of interacting/cooperating elements.

We introduce here a notion of population P systems as a finite collection of different componets that are free of forming/removing bonds according to a finite set of bond making rules in a given environment.

Definition 2. *A population P system is a construct [3]*

$$\mathcal{P} = (V, \gamma, \alpha, w_E, C_1, C_2, \ldots, C_n, c_O)$$

where:

1. *V is a finite alphabet of symbols called objects;*
2. *$\gamma = (\{1, 2, \ldots n\}, E)$, with $E_n \subseteq \{\{i, j\} \mid 1 \leq i \neq j \leq n\}$, is a finite undirected graph;*
3. *α is a finite set of bond making rules $(i, x_1; x_2, j)$, with $x_1, x_2 \in V^*$, and $1 \leq i \neq j \leq n$;*
4. *$w_E \in V^*$ is a finite multiset of objects initially assigned to the environment;*
5. *$C_i = (w_i, S_i, R_i)$, for each $1 \leq i \leq n$, is a component of the system with:*
   (a) $w_i \in V^*$ a finite multiset of objects,
   (b) S_i is a finite set of communication rules;
   (c) R_i is a finite set of transformation rules;
6. *c_O is the output component.*

A population P system $\mathcal{P}$ is defined as a collection of n components where each component C_i corresponds in an one-to-one manner to a node i in a finite undirected graph γ, which defines the initial structure of the system. Components are allowed to communicate alongside the edges of the graph γ, which are unordered pairs of the form $\{i, j\}$, with $1 \leq i \neq j \leq n$. The components C_i, $1 \leq i \leq n$, are

associated in an one-to-one manner with the set of nodes $\{1, 2, \ldots n\}$. For this reason, each component C_i will be subsequently identified by its label i from the aforementioned set.

Each component C_i contains a finite multiset of objects w_i, a finite set of communication rules S_i, and a finite set of transformation rules R_i.

Component capability of moving objects alongside the edges of the graph is then influenced by particular bond making rules in α that allow these components to form new bonds. In fact, a bond making rule $(i, x_1; x_2, j)$ specifies that, in the presence of a multiset x_1 in i and a multiset x_2 inside j, a new bond can be created between these two components. This means a new edge $\{i, j\}$ can be added to the graph that currently defines the structure of the system.

The model introduced will be further enriched with a concept of type which enables us to instantiate components. Each type component apart from objects and rules may also contain variables. The model enriched with these new features will be used from now on using a notation that is closer to a programming paradigm. An example of a component type is defined as follows:

```
component type a;
  element x, y, z; mult = 3;
  var t = 5, u = 10 : int;
  rule  x y --> x x z; -- (r1)
  rule  z   --> z z;   -- (r2)
  rule  t   --> t+u;   -- (r3)
  end;
```

In this example a component type a is defined with object elements x, y, z, two variables t, u and three rewriting rules $r1, r2, r3$.

The objects x, y have one instance each, whereas z occurs in three copies. The integer variables t, u are introduced with initial values 5 and 10, respectively. The rules $r1$ and $r2$ rewrite xy and z, respectively, whereas $r3$ rewrites variable t with the sum of the values contained in t and u.

The rules may also be preceded by some Boolean conditions which allow the corresponding rules to be applied when these guards are true. Apart from rewriting rules, communication rules, division rules, and death rules are also provided. All the rules have an execution time associated with. By default this is 1, but may be greater than 1 as well and this means the rule needs more than 1 evolution steps in order to be performed.

From each component type various instances may be created. During the instantiation process the implicit values associated with various objects may be changed.

```
instance a1, a2: a;
  element x; mult 100;
instance a3: a;
```

In the example above two components $a1, a2$ are instantiated from a with 100 occurrences of x; y, z occur with the values mentioned in the definition of a. The component $a3$ has the same objects as a defined.

Apart from component types it is also possible to define the environment with objects of different types, but also bond making rules that create links between various components of the system.

5 Case Study: An Ant Colony

The first modelling approach will use an X-machine method. The ant is modelled so that it accepts a tuple (pos, stimuli) as input to its functions. The first element of the input tuple is a tuple representing the coordinates in which stimuli is perceived whereas the second element is the description of the stimuli. The latter can be *pheromone* or *space* describing the space denoted by the coordinates, a *hungry* or a *non-hungry* ant describing whether an ant that is perceived in the given coordinates carries food or not or, finally, a number greater than zero representing the quantity of food that is received by another ant that carries food. The memory of the ant holds (a) its current *position*, (b) the amount of *food* it carries, (c) a number denoting the *food quantity threshold* beneath which the ant becomes hungry, (d) the *food decay rate*, a number denoting the quantity of food that is consumed by the ant in each time unit and (e) the *food portion* that is to be given by an ant that is carrying food to another which is hungry. The behaviour of the systems is given by different functions processing input stimuli.

All these types are defined using XMDL declarations.

The *becomeHungry* function reduces the amount of food that the ant carries according to the food decay rate but is only applied when the updated value of the food quantity becomes less or equal to the hunger threshold $?ft$ in order to bring the ant to the *hungry* state.

```
#fun becomeHungry ((?p, ?in), (?pos, ?f, ?ft, ?fdr, ?mfp)) =
    if ?nf =< ?ft then
    ((gotHungry), (?pos, ?nf, ?ft, ?fdr, ?mfp))
    where ?nf <- ?f - ?fdr.
```

The $giveFood$ function is applied when an ant gives $?mfp$ amount of food to the hungry ant it met. The updated food quantity that the ant will carry afterwards is reduced by the donated food portion $?mfp$ as well as by the food decay rate $?fdr$. All possible input is ignored by the ant which returns to the *inactive* state.

```
#fun giveFood ((?p, ?in), (?pos, ?f, ?ft, ?fdr, ?mfp)) =
    ((givingFood), (?pos, ?nf, ?ft, ?fdr, ?mfp))
    where ?food_reduction <- ?fdr + ?mfp
    and ?nf <- ?f - ?food_reduction.
```

The *die* function ignores all possible input and is only applied when the quantity of food in an ant's memory (the amount of food it is carrying) is equal to zero. It outputs a "dying" message and leaves the memory structure unaltered.

```
#fun die ((?p, ?in), (?pos, ?f, ?ft, ?fdr, ?mfp)) =
    if ?what_is_left =< 0 then
    ((dying), (?pos, 0, ?ft, ?fdr, ?mfp)).
    where ?what_is_left <- ?f - ?fdr.
```

To demonstrate how the reconfiguration operators are used we consider the case that after a food transaction, communication between the two ants needs to halt. The corresponding rule that will apply the detachment operator which will remove the communication channels between the two ant instances is:

$$\begin{aligned} &(q_c,\ M_c,\ takeEnoughFood)_i \in SZ \\ \vee &(q_c,\ M_c,\ takeNotEnoughFood)_i \in SZ \\ &\rightarrow \mathbf{DET}\ (i,\ Z) \end{aligned}$$

Using the P system approach we may describe some of the rules applied to simulate the behaviour of the ant colony.

Each ant has a specific amount of found which will decrease as the time goes by. This is captured by the rule

```
foodL --> foodL-FoodDecayRate
```

where $foodL$ is a variable pointing to the current level of food; after applying this rule the updated value of this is obtained. When the level of food is under a threshold the ant will become hungry; this is shown by a rule

```
inactive and foodL < HungryLevel: inactive --> hungry
```

which will change *inactive* to *hungry* when the Boolean condition preceding the rule is true.

When an ant is hungry it is moving around looking for food. This is simulated by a communication rule which will put in the environment the current position and is getting a new position nearby.

```
Neighbour(pos,pos'): (out pos; in pos')
```

This rule will be read as "if the two positions pos and pos' are next to each other - predicate $Neighbour(pos, pos')$ is true, then the current position pos is sent out in the environment and a new position pos' from the environment enters the component".

When two ants are next to each other a bond making rule will create a link.

```
Neighbour(ant.pos, ant.pos'): <ant,ant>
```

In this case a bond will be created between the two ants if their positions are close enough.

A food transfer may take place between two ants that are linked.

```
transfer and foodL > HungryLevel:
(out FoodTransfer from foodL; outComponent ant) time=10
```

The ant that is not hungry and is in state transfer will provide $FoodTransfer$ units from its current amount of food. Correspondingly the ant that is at the other end of the link will receive the same $FoodTransfer$ that will be added to its amount of food. The transfer will take 10 units of simulation time.

When an ant cannot find food and its amount of food becomes 0 then it will die.

```
foodL <= 0:  (component_death)
```

6 Experiments

A variety of experiments were performed to examine how the Pharao's ant models behave [5]. Each worker ant logged a history of all food related actions it performed. These log files can then be analysed once the simulation has completed. The environment consisted of a nest with four entrances or exits, each of which was situated on a compass point (north, south, east, west). The colony consisted of 100 workers in a nest of 3cm by 3cm. The experiments show that the colony manages to distribute the food among the colony members before a mass forage occurs; the process has a cyclic nature and a smooth gradient.

In nature worker specialisation occurs. This is shown by a low proportion of worker ants focusing on feeding other colony members or foraging. The first case might be illustrated by some ants missing the forage cycle although no specific constraints were imposed in this respect and consequently a sort of emergent specialisation may be noticed. Although in our experiments in almost all the steps of the simulation there were ants missing the forage process is too early to identify a specialisation due to the reduced scale of the simulation time considered. It is likely that a more specific analysis of parameters involved in a long run simulation may lead to some conclusions regarding this phenomenon.

7 Conclusions

This work has been an attempt to model a simple biological system by using two different methods, namely population P systems and communicating X-machines. This simple case study shows the need to approach the dynamic structure and organisation of biological systems with models exhibiting such properties. Bond making rules in the context of P systems and operators **ATT**, **DET**, **GEN**, and **DES** in the case of communicating X-machines represent the key elements introduced in order to cope with the systems' dynamicity.

There are advantages to both methods, though at different modelling levels. The X-machine approach appears to be a natural model to express the internal behaviour of the components because they can naturally describe the internal states, transitions between them caused by stimuli and represent the data structures. However a communicating X-machine model cannot by itself manage the required reconfiguration, which is a prominent property of these biological systems. As a result, an external device, in the form of a meta-X-machine, is

necessary; this device possesses global control over the structure of the overall system. Control is achieved through meta-operations which change the way that components interact or function. Population P systems, on the other hand, possess a natural trait for capturing the behaviour of a community of entities and how the structure of such a community may change over time. The new characteristics introduced in this paper, guards to rules, variables, arithmetic operations, improve the potential of this model to specify the internal behaviour of the componets.

Both methods have sound theoretical foundations and act as formal specification languages. Towards this end, the X-machine Description Language (XMDL) [19] has been defined offering the ability of formally describing X-machine models and acting as an interchange tool for software engineers. XMDL also serves as a common basis for the development of tools, such as the X-System [23], that allow the syntactical check and automatic animation of the models. In this paper the elements (component type, environment definition, component instantiation etc) of a specification language based on P systems were introduced for the first time.

In addition to this practical aspect, X-machines have further techniques supporting the modelling activity such as formal verification of desired system properties [8] and complete testing [14]. Towards practical modelling, appropriate XML notation in order to define population P systems is currently under development and soon expected to be made available. Formal properties of some classes of population P systems are also under investigation. Effort has been put into modelling a P system as a communicating X-machine [20]. Further investigations regarding possible transformations between communicating X-machine models and population P system models would be useful in order to support both a formal theoretical comparison as well as the modelling activity.

This case study shows not only the benefits of approaching systems with a dynamic structure by models exhibiting naturally these properties and the need to further develop these models, but it also suggests that this way of modelling may be reused in other contexts where multi-agent paradigm has to be considered. Communities of bacteria, cells in tissues, or more complex organisms composed of simpler components may be modelled in a similar way. In the next future we aim to use this approach in certain biological systems where to identify their emergent behaviour as well as potentially new computational paradigms inspired by these systems.

Acknowledgement. We are grateful to dr. George Eleftherakis who has helped us in developing the communicating X-machine methodology and to our students that have built various tools and experimented different case studies. We especially would like to thank James Clarke, Peter Langton, Liancheng Lu, Taihong Wu, Yang Yang who have built a tool helping experimenting with the X-machine model and Fei Lu and Ming-Hsin An who have worked on building a tool allowing to simulate P system specifications. The research of Marian Gheorghe was supported by the Engineering and Physical Science Research Council (EPSRC) of United Kingdom, Grant GR/R84221/01.

Bibliography

[1] Adleman, L.M. 1994. Molecular computation of solutions to combinatorial problems, *Science*, 226, 1021-1024.

[2] Banatre, J.P., Le Metayer, D. 1990. The gamma model and its discipline of programming, *Science of Computer Programming*, 15, 55-77.

[3] Bernardini, F., Gheorghe, M. 2004. Population P Systems, *Journal of Universal Computer Science*, 10, 509-539.

[4] Bianco, L., Fontana, F., Franco, G., Manca, V. 2004. P systems in Bio Systems. In Păun, Gh., ed. 2004. *Application of P systems*, submitted.

[5] Clarke, J., Langton, P., Lu, L., Wu, T., Yang, Y. 2002. Computational models of Pharao's ants using X-machines. Department of Computer Science, University of Sheffield, MSc final report.

[6] Dorigo, M., Maniezzo, V., Colorni, A. 1996. The Ant System: Optimisation by a colony of co-operating agents, *IEEE Transactions on Systems, Man and Cybernetics*, 26, 1-13.

[7] Eilenberg, S. 1974. Automata, Languages and Machines, Academic Press.

[8] Eleftherakis, G. 2003. *Formal Verification of X-Machine Models: Towards Formal Development of Computer-based Systems*, PhD Thesis, Department of Computer Science, University of Sheffield.

[9] Eleftherakis, G., Kefalas, P. 2000. Model Checking Safety Critical Systems specified as X-Machines, *Analele Universitatii Bucharest, Matematica-Informatica series*, 49, 59-70.

[10] Eleftherakis, G., Kefalas, P., Sotiriadou, A. 2003. Formal Modelling and Verification of Reactive Agents for Intelligent Control. In *Proceedings of the 12th Intelligent System Applications to Power Systems Conference* (ISAP).

[11] Gregson, A. M., Hart, A. G., Holcombe, M., Ratnieks, F. L. W. 2003. Partial nectar loads as a cause of multiple nectar transfer in the honey bee (Apis mellifera): a simulation model. *Journal of Theoretical Biology*, 222, 1-8.

[12] Holcombe, M. 1988. X-machines as a Basis for Dynamic System Configuration, *Software Engineering Journal*, 3, 69-76.

[13] Holcombe, M. 2001. Computational models of cells and tissues: Machines, agents and fungal infection, *Briefings in Bioinformatics*, 2, 271-278.

[14] Holcombe, M., Ipate, F. 1998. *Correct Systems: Building a Business Process Solution*, Springer-Verlag, London, 1998.

[15] Ipate, F., Holcombe, M. 1997. An Integration Testing Method that is proved to find all faults, *International Journal of Computer Mathematics*, 63, 159-178.

[16] Jennings, N.R. 2000. On agent-based software engineering, *Artificial Intelligence*, 117, 277-296.

[17] Kapeti, E., Kefalas, P. 2000. A Design Language and Tool for X-Machines Specification, In Fotiadis, D.I., Spyropoulos, S.D., eds. 2000. *Advances in Informatics*, World Scientific Publishing Company, 134-145.

[18] Kefalas P. 2000. Automatic translation from X-machines to Prolog. TR-CS01/00, Dept. of Computer Science, CITY Liberal Studies.

[19] Kefalas, P. 2000. XMDL user manual: version 1.6. TR-CS07/00, Dept. of Computer Science, CITY Liberal Studies.

[20] Kefalas, P., Eleftherakis, G., Holcombe, M., Gheorghe, M. 2003. Simulation and Verification of P Systems through Communicating X-Machines, *BioSystems*, 70, 135-148.

[21] Kefalas, P., Eleftherakis, G., Holcombe, M., Stamatopoulou, I. 2004. Formal Modelling of the Dynamic Behaviour of Biology-Inspired Agent-based Systems, In Gheorghe, M. ed. 2004. *Molecular Computational Models: Unconventional Approaches*, Idea Publishing, Inc, accepted.

[22] Kefalas, P., Eleftherakis, G., Kehris, E. 2003. Communicating X-Machines: A Practical Approach for Formal and Modular Specification of Large Systems, *Journal of Information and Software Technology*, 45, 269-280.

[23] Kefalas, P., Eleftherakis, G., Sotiriadou, A. 2003. Developing Tools for Formal Methods. *Proceedings of the 9th Panhellenic Conference in Informatics*. 625-639.

[24] Martin-Vide, C., Mauri, G., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) 2004. *Membrane Computing*. International Workshop, WMC 2003, Tarragona. Revised Papers in Lecture Notes in Computer Science 2933, Springer-Verlag, Berlin / Heidelberg / New York.

[25] Păun, G. 2000. Computing with membranes, *Journal of Computer and System Sciences*, 61, 1, 108-143.

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

[27] Păun, Gh., Rozenberg, G., Salomaa, A., Zandron, C.(eds.) 2002. *Membrane Computing*. International Workshop, WMC-CdeA 02, Curtea de Arges, Romania. Revised Papers in Lecture Notes in Computer Science 2597, Springer-Verlag, Berlin / Heidelberg / New York.