Self assembling graphs

Vincent Danos[1] and Fabien Tarissan[2]

[1] Équipe PPS, CNRS & Université Paris VII
[2] Équipe PPS, Université Paris VII

Abstract. A self-assembly algorithm for synchronising agents and have them arrange according to a particular graph is given. This algorithm, expressed using an ad hoc rule-based process algebra, extends Klavins' original proposal [1], in that it relies only on point-to-point communication, and can deal with any assembly graph whereas Klavins' method dealt only with trees.

1 Introduction

In a number of different subject areas, nanotechnologies [2], amorphous computations [3], molecular biology [4], one commonly finds a debate about whether and how complex shapes, structures and functions can be generated by local interactions between simple components. Klavins addressed this question in the field of robotics [1]. The problem is that of synchronising a population of autonomous agents and have them achieve a particular disposition in space specified as a tree. The aim of the present paper is to extend the solution given by Klavins to the case of arbitrary graphs, and to provide a formalization of the self-assembly algorithm that takes complete care of the subtler part of building a distributed consensus among agents.

The idea of the algorithm is to circulate between agents belonging to a same connected component a single copy of a mapping of their component. Whoever possesses this mapping can either pass it over to a neighbour, or decide to create a new connection, based on a successful point-to-point communication with another agent. Note that since agents are building a potentially cyclic graph, they may have to create edges to their own component. Necessary updates after a growth decision are shipped along a tree spanning the current component. Both the component and the tree are dynamically created. Interestingly, the algorithm is parameterized by the choice of a *growth scenario* specifying when an edge can be created. Thus, the solution we propose naturally supports additional constraints pertaining to which intermediate graphs are allowed during the growth of the graph.

The solution and the problem itself are laid down in the language of concurrency theory, and the algorithm is written in a rule-based process algebra that one could view as a simplified version of Milner's π-calculus [5]. Although the self-assembly algorithm we present is independent of this particular choice, there is a good reason for such a formal approach. More often than not, one can go wrong in the description of such synchronisation procedures, and the use of formal methods seems legitimate in this context, since they allow for a clear statement of correctness, and a correctness proof based on a well-established notion of equivalence known as *barbed bisimulation* [6].

Our formal treatment is made relative to abstract or logical space. Including true space and explicit motorization in the agents supposes a significant extension of the usual concurrency models and as such represents an interesting challenge to formal methods. Such an extension would in particular allow a refined description of the agents behaviour in the case of a group being dislocated. This is a matter to which we plan to return in a further work. For now, we provide a crude treatment of such "crashes" by introducing non deterministic alarms. The correctness of the algorithm enriched with alarms is also proved. A demo illustrating the algorithm is available on line.[1]

The self-assembly question we address here was inspired by similar questions raised in the context of formal molecular biology [7, **?**]. Indeed, a strong structural property that one might look for when defining a formal language for protein-protein interaction is precisely whether the formation of complexes (assemblies of proteins) can be explained in terms of only local interactions. In the context of biology, there is an additional constraint, namely that the self-assembly algorithm doesn't build in the agents unrealistic computational prowess. With robots however, agents can be taken to be computationally strong and no such objection stays on the way of a completely satisfying result.

2 Graph rewriting

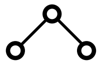
2.1 Agents and Networks

In order to handle graphs and the kind of local graph rewriting our agents will perform, we introduce first a notation for graphs inspired by π-calculus, where nodes are agents, and edges are represented by name-sharing. Let $\mathcal{C}$ be a countable set of *names* ranged over by x, y, z, ..., one defines an *agent* as a finite set $C \subset \mathcal{C}$, written $\langle C \rangle$, where the set C itself is refered to as the agent *interface*. Agents can be arranged in *networks* according to the following grammar:

$$G := \varnothing \mid \langle C \rangle \mid G, G \mid (\nu x)G$$

where $\varnothing$ is the empty network, G_1, G_2 stands for the juxtaposition of G_1 and G_2, and $(\nu x)G$ stands for G where the name x has been made private to G.

Here is an example:

 becomes $(\nu\ x, y)\ (\langle x \rangle\ ,\ \langle x, y \rangle\ ,\ \langle y \rangle)$

Our algebraic notation is redundant in that there are many distinct ways to represent the same graph. The notion of *structural congruence* below will take care of this redundancy.

[1] `http://www.pps.jussieu.fr/~tarissan/self`

The "new" operator, written in symbols ν, is a binder for names in $\mathcal{C}$ and allows for a smooth treatment of name creation. It comes along with the usual inductive definition of *free names*:

$$\begin{aligned}
\mathsf{fn}(\varnothing) &= \varnothing \\
\mathsf{fn}(\langle C\rangle) &= C \\
\mathsf{fn}(G, G') &= \mathsf{fn}(G) \cup \mathsf{fn}(G') \\
\mathsf{fn}((\nu x)G) &= \mathsf{fn}(G) \smallsetminus \{x\}
\end{aligned}$$

An occurrence of name is said to be bound if not free. The operation of renaming bound variables is often called *α-conversion*.

Definition 1 *Structural congruence, written $\equiv$, is the smallest congruence relation closed under α-conversion and such that:*

1. *$(\mathcal{N}/{\equiv}$, ',' , $\varnothing)$ is a symmetric monoid*
2. *$(\nu x)(\nu y)G \equiv (\nu y)(\nu x)\ G$*
3. *$(\nu x)G \equiv G$ if $x \notin \mathsf{fn}(G)$*
4. *$(\nu x)G, G' \equiv (\nu x)(G, G')$ if $x \notin \mathsf{fn}(G')$*

There is a unique network, up to structural congruence, representing a given isomorphism class of graphs, which is what we wanted. Based on this, we will now consider graphs as networks and don't distinguish them notationally. We also observe in passing that our notation also accomodates the description of hypergraphs.

Working up to structural congruence, and using the last three clauses, one can use any of the equivalent notations $(\nu\{x, y\})$, (νxy) or $(\nu x)(\nu y)$. Sets of names will be sometimes denoted succinctly by $\tilde{x}$.

2.2 Reactions and Transition systems

Now that we have our notation for graphs in place, we turn to the definition of a notion of graph rewriting which will be expressive enough for our needs.

Definition 2 *A reaction is a pair L, $(\nu\tilde{x})R$, also written $L \to (\nu\tilde{x})R$, where $L = \langle L_1\rangle, \ldots, \langle L_n\rangle$, $R = \langle R_1\rangle, \ldots, \langle R_n\rangle$, and $\mathsf{fn}((\nu\tilde{x})R) \subseteq \mathsf{fn}(L)$.*

When in addition, $n \leq 2$, one will say the reaction is *local*. Local reactions express point-to-point communications and will be used to state the self-assembly problem.

Names occurring in a reaction fall naturally in three classes: the names *created* by the reaction $\tilde{x}$, the names *erased* by the reaction $\mathsf{fn}(L) \smallsetminus \mathsf{fn}((\nu\tilde{x})R)$, and the rest $\mathsf{fn}(L) \cap \mathsf{fn}((\nu\tilde{x})R)$. The condition in the definition above makes sure that any name occurring in R is either created, or already occurs in L.

To fire a reaction in a network G, one looks for an instance of L in G and then replaces it with the right hand side $(\nu\tilde{x})R$. More precisely, given a set of reactions $\mathfrak{R}$, one defines inductively a binary relation $\rightarrow_{\mathfrak{R}}$ as follows:

$$\text{(DIR)} \quad \frac{G \rightarrow_{\mathfrak{r}} G' \quad \mathfrak{r} \in \mathfrak{R}}{G \rightarrow_{\mathfrak{R}} G'} \qquad \text{(GROUP)} \quad \frac{G_1 \rightarrow_{\mathfrak{R}} G_2}{G_1, G \rightarrow_{\mathfrak{R}} G_2, G}$$

$$\text{(NEW)} \quad \frac{G_1 \rightarrow_{\mathfrak{R}} G_2}{(\nu x)\ G_1 \rightarrow_{\mathfrak{R}} (\nu x)\ G_2} \qquad \text{(STRUCT)} \quad \frac{G_1 \equiv G_1' \quad G_1' \rightarrow_{\mathfrak{R}} G_2' \quad G_2' \equiv G_2}{G_1 \rightarrow_{\mathfrak{R}} G_2}$$

with $G \rightarrow_{\mathfrak{r}} G'$, if $\mathfrak{r} = \langle L_1 \rangle, \ldots, \langle L_n \rangle \rightarrow (\nu \tilde{x})(\langle R_1 \rangle, \ldots, \langle R_n \rangle)$, and there exists an injection $\mathfrak{i}$ from $\mathsf{fn}(L) \cup \tilde{x}$ to $\mathcal{C}$ such that:

$$\begin{aligned} G\ &= \langle \mathfrak{i}(L_1) \rangle, \ldots, \langle \mathfrak{i}(L_n) \rangle \\ G' &= (\nu \mathfrak{i}(\tilde{x}))(\langle \mathfrak{i}(R_1) \rangle, \ldots, \langle \mathfrak{i}(R_n) \rangle) \end{aligned}$$

One also writes $G \rightarrow^*_{\mathfrak{R}} G'$, whenever G' can be obtained from G by repeatedly firing reactions in $\mathfrak{R}$. This includes the case when no reaction is used and $G' = G$. A *transition system* is a pair G, $\mathfrak{R}$, where G is a graph, called the *initial state*, and $\mathfrak{R}$ is a set of reactions. A transition system G, $\mathfrak{R}$ is said to be *local* when each reaction in $\mathfrak{R}$ is local.

2.3 Self-assembly

Suppose now we want our agents to self-assemble according to a given connected graph $G = (V, E)$. Write $|V|$ for the number of nodes in G, $\langle\rangle_n$ for the network $\langle\rangle, \ldots, \langle\rangle$ consisting of n empty agents, and $(\langle\rangle_n, \{r_G\})$ for the transition systems associated to G, with initial states $\langle\rangle_n$ and only reaction $r_G := \langle\rangle_{|V|} \rightarrow G$.

The *self-assembly* problem for G is to find a set of *local* reactions $\mathfrak{R}_G$ and a map θ from agents to some suitable notion of enriched agents, such that for all n, the transition system $(\theta(\langle\rangle_n), \mathfrak{R}_G)$ simulates —in a sense yet to be defined— the original system $(\langle\rangle_n, \{r_G\})$.

Two points need to be clarified here. First we need to explain how to enrich agents. An enriched agent will no longer be a mere set of names but a list of sets of names or integers. The θ function above will take care of structuring the initially empty agents according to the enriched agent format. Second, we need a definite statement about what we mean when we say that $(\theta(\langle\rangle_n), \mathfrak{R}_G)$ simulates $(\langle\rangle_n, \{r\})$, and this is where bisimulation comes in the picture.

The first point will be addressed in the course of the construction of the local transition system $(\theta(\langle\rangle_n), \mathfrak{R}_G)$ to which we turn now, while the second will be the object of the next section devoted to correctness.

3 The construction

Transition systems over enriched agents are similar to the ones defined before and we don't go over all the definitions of the preceding section.

We proceed to our construction in two steps. First we define the notion of a *growth scenario*, in essence a transition system describing which intermediate graphs one should seek for. Then we obtain the corresponding local transition system.

3.1 Growth scenarios

Agents manipulate concrete representations of graphs and we have to be careful to distinguish these from abstract graphs. Specifically, a *concrete graph* will be taken to be a graph of the form $(\{1, \ldots, n\}, E)$ with $n > 0$. The *trivial* graph $(\{1\}, \varnothing)$ will be denoted by $\mathbf{1}$. Given G a concrete graph, we write $[G]$ for its isomorphism class, that is to say the corresponding abstract graph. Next, we define two operations on concrete graphs:

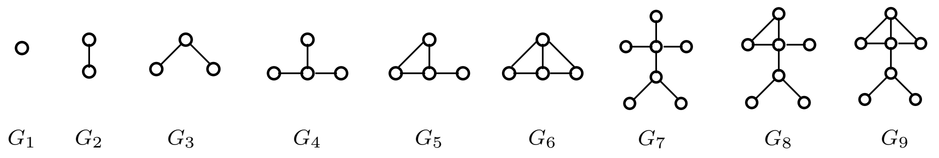
Definition 3 *Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be concrete graphs, the* join *of G_1 and G_2 via $u \in V_1$, $v \in V_2$, written $G_1.u \oplus G_2.v$, is defined as:*
— $V = \{1, \ldots, |V_1| + |V_2|\}$, and
— $E = E_1 \cup \{\{u, v + |V_1|\}\} \cup \{\{a + |V_1|, b + |V_1|\} \mid \{a, b\} \in E_2\}$
and the self-join *of G_1 via u, $v \in V_1$, written $G_1.(u, v)$, is defined as $V = V_1$, and $E = E_1 \cup \{\{u, v\}\}$.*

Note that in the binary join operation, the nodes of G_2 are shifted by $|V_1|$, and as a consequence the result is again a concrete graph. These operations naturally extend to abstract graphs, and they define together a partial order, written $<$, on concrete as well as on abstract graphs.

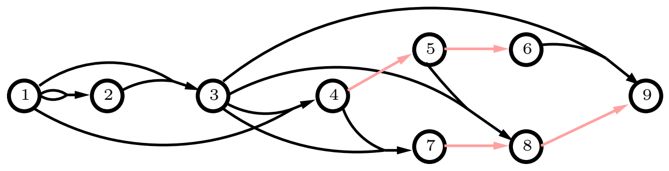
Definition 4 *A* growth scenario *$\mathcal{G}$ is a set of abstract graphs such that for all non trivial $G \in \mathcal{G}$, either $G = G_1.u \oplus G_2.v$, for some G_1, $G_2 \in \mathcal{G}$, or $G = G_1.(u, v)$ for some $G_1 \in \mathcal{G}$.*

All graphs within a scenario are connected, and conversely any connected graph is obviously contained in some scenario.

Here is an example of a growth scenario $\mathcal{G} = \{G_1, \ldots, G_9\}$:



The idea is that agents wants to self-assemble to reach the target graph G_9, and $\mathcal{G}$ specifies all intermediate graphs they are allowed to construct in so doing. The figure below, where nodes stand for graphs in $\mathcal{G}$, bi-edges correspond to joins, and mono-edges to self-joins gives a visual proof that $\mathcal{G}$ is indeed a growth scenario. Note that we do note require scenarios to be downward closed with respect to $<$, and indeed $\mathcal{G}$ is not.



To each scenario $\mathcal{G}$ corresponds naturally a set of reactions, denoted by $\mathfrak{R}(\mathcal{G})$, obtained by translating as reactions the joins and self-joins under which $\mathcal{G}$ is closed. These reactions are quite specific, since they create only one name, and delete none. If we return to the example, the bi-edge linking G_1 and G_2 to G_3 corresponds to the following join reaction in $\mathfrak{R}(\mathcal{G})$:

$$\langle x\rangle, \langle x\rangle, \langle\rangle \rightarrow (\nu y)(\langle x\rangle, \langle x, y\rangle, \langle y\rangle)$$

3.2 The local transition system

With these definitions behind us, and supposing given a target graph G, and a scenario $\mathcal{G}$ with G as only maximal graph, the self-assembly problem can now be rephrased as the problem of finding a set of local reactions that will simulate $\mathfrak{R}(\mathcal{G})$.

Let us begin with a still informal description of our algorithm and its local reactions. Each agent has in its internal state an integer called the *role* meant to describe its coordinate on the map being circulated between the agents in a same component. By allowing at any time only one agent to modify a component, we prevent concurrent modifications of the structure. This agent is chosen to be the only one holding a map of the current component, since only him needs it. So to speak, the map is itself the activity token. This makes both the internal state of an agent and the update phase simpler. Update is then reduced to the propagation of the new role played by the agents in the new enlarged component.

Going back to definition 3, we see that joins affect only the roles in one of the two connected components. Therefore, role updates are only required in one component which we take in the implementation to be the smaller one. Second, we see also that the new role is easily determined from the old one, since it is just a shift of the old one by the number of nodes, say N of the other component. Agents involved in an update phase will then simply transmit this integer N.

The case of a self-join is easier, since roles are all unchanged. Role updates are not needed, it is just the map of the active agent which is modified.

To avoid conflicts arising during update in case the graph is cyclic, we use a tree spanning the component to transmit the shift. This spanning tree is itself a dynamic structure that grows along with the component. A new edge is added to it at each join. To reflect this structure in their states, enriched agents partition their name set in two classes, written respectively S (for span) and C (for cycles), depending on whether the corresponding edge belongs to the spanning tree or not. Take note that this tree is undirected. The direction of transmission is also dynamically determined and varies over time.

Following these informal explanations, we define the interface of an *enriched agent* as a tuple $[S, C, g, r, m]$ where:
— S is a set of names which represents neighbours in the spanning tree
— C is a set of names which represents the set of the remaining neighbours
— g is a name used as a group identifier for agents in a same component
— r is an integer referring to the role played by the agent in the component
— m is the agent *running mode*:

- P when the agent is in *passive* mode
- $Act(G)$ when the agent is in *active* mode, with G a concrete graph
- $Up(L, N)$ when the agent is in *update* mode, with L a set of names referring to the neighbours still to be updated, and N the shift

3.3 Local Reactions

Given $\mathcal{G}$ a growth scenario, we may now define our family of local reactions $\mathfrak{R}^l(\mathcal{G})$ simulating $\mathfrak{R}(\mathcal{G})$. First come the *join reactions*, with $[G_1.r_1 \oplus G_2.r_2] \in \mathcal{G}$ and $g_1 \neq g_2$:

$$\begin{matrix}[S_1, C_1, g_1, r_1, Act(G_1)], \\ [S_2, C_2, g_2, r_2, Act(G_2)]\end{matrix} \longrightarrow (\nu x) \begin{matrix}[S_1 + \{x\}, C_1, g_1, r_1, Act(G_1.r_1 \oplus G_2.r_2)], \\ [S_2 + \{x\}, C_2, g_1, r_2 + |G_1|, Up(S_2, |G_1|)]\end{matrix}$$

Next come the *self-join reactions*, with $[G.(r_1, r_2)] \in \mathcal{G}$:

$$\begin{matrix}[S_1, C_1, g, r_1, Act(G)], \\ [S_2, C_2, g, r_2, P]\end{matrix} \longrightarrow (\nu x) \begin{matrix}[S_1, C_1 + \{x\}, g, r_1, Act(G.(r_1, r_2))], \\ [S_2, C_2 + \{x\}, g, r_2, P]\end{matrix}$$

Then the *update reactions*:

$$\begin{matrix}[S_1, C_1, g_1, r_1, Up(L + \{x\}, N)], \\ [S_2 + \{x\}, C_2, g_2, r_2, P]\end{matrix} \longrightarrow \begin{matrix}[S_1, C_1, g_1, r_1, Up(L, N)], \\ [S_2 + \{x\}, C_2, g_1, r_2 + N, Up(S_2, N)]\end{matrix}$$

and the *end-of-update reactions*:

$$[S, C, g, r, Up(\varnothing, N)] \longrightarrow [S, C, g, r, P]$$

Finally we need the *switch reactions*:

$$\begin{matrix}[S_1 + \{x\}, C_1, g, r_1, Act(G)], \\ [S_2 + \{x\}, C_2, g, r_2, P]\end{matrix} \longrightarrow \begin{matrix}[S_1 + \{x\}, C_1, g, r_1, P], \\ [S_2 + \{x\}, C_2, g, r_2, Act(G)]\end{matrix}$$

This last switch reaction, which allows the activity to circulate around in a component, can only be fired if the agents share the same group identifier. We may also note that in the update reactions, the updated agent simultaneously changes its role, and group, while entering himself in update mode. In the end-of-update reaction, the agent goes passive because his contact list is empty.

We may now complete our definition:

Definition 5 *Given a growth scenario $\mathcal{G}$, one defines the local transition systems associated to $\mathcal{G}$ as $(\theta(\langle\rangle_n), \mathfrak{R}^l(\mathcal{G}))$, where $\mathfrak{R}^l(\mathcal{G})$ is defined above, and the initial state $\theta(\langle\rangle_n)$ is given by $(g_1)\langle\varnothing, \varnothing, g_1, 1, Act(\mathbf{1})\rangle, \ \ldots, (g_n)\langle\varnothing, \varnothing, g_n, 1, Act(\mathbf{1})\rangle$.*

4 Correctness

We have completed the formal definition of our algorithm, and it remains to explain in which sense our local transition systems, $\mathfrak{R}^l(\mathcal{G})$, behave as the global ones, $\mathfrak{R}(\mathcal{G})$. We use the notion of barbed bisimulation to do this.

Given a network G, we will write $G \downarrow_n C$, if the connected component C occurs n times in the network G.

Definition 6 *Two transition systems* $(G_0, \mathfrak{R})$ *and* $(G'_0, \mathfrak{R}')$ *are bisimilar if there exists a binary relation* $\sim$ *over* $\mathcal{N}$ *such that:*
— $G_0 \sim G'_0$,
— *if* $G \sim G'$*, then* $G \downarrow_n C$ *if and only if* $G' \downarrow_n C$*;*
— *if* $G \sim G'$ *and* $G \rightarrow_{\mathfrak{R}} H$*, there exists* H' *such that* $G' \rightarrow^*_{\mathfrak{R}'} H'$ *and* $H \sim H'$*;*
— *if* $G \sim G'$ *and* $G' \rightarrow_{\mathfrak{R}'} H'$*, there exists* H *such that* $G \rightarrow^*_{\mathfrak{R}} H$ *and* $H \sim H'$*.*

We can now state the correctness of our distributed algorithm. Note that the obtained property is slightly more general than the one we wanted, since it does not assume that there is a unique maximal graph in the scenario $\mathcal{G}$.

Proposition 1 *For all* n *and all growth scenarios* $\mathcal{G}$*, the local and global transition systems,* $(\langle\rangle_n, \mathfrak{R}(\mathcal{G}))$ *and* $(\theta(\langle\rangle_n), \mathfrak{R}^l(\mathcal{G}))$*, are bisimilar.*

The key to proving this proposition is to prove that active agents always have a *consistent* view of their components. That is to say, for any G which is reachable from the initial state $\theta(\langle\rangle_n)$, and any active agent in G, both the map and the role occurring in the interface of this agent are the actual ones in G. Once this is done, it is relatively easy to construct a bisimulation between the global and local systems.

5 Escaping deadlocks

The local transition systems defined above are monotonic in the sense that edges can only be added. Different components representing partially grown target graphs could compete and be deprived of resources.

Klavins suggests a simple timeout method to grow, starting with a given population of agents, the maximum possible number of copies of the target graph. We can easily incorporate an abstract version of this deadlock escape mechanism in our algorithm.

First we extend our notion of growth scenario by allowing each connected component to dislocate, with the constraint that as soon as it has begun to do so, it has no choice but keeping on breaking down to unconnected agents. To enforce this, we allow only active agents to fire an alarm. Second, we introduce a new *alarm mode*, written Al. And third, we add the accompanying reactions, starting with the *breaking-loose reactions*:

$$[S, C, g, r, Act(G)] \longrightarrow [S, C, g, r, Al]$$

and the *alarm propagation reactions*:

$$\begin{array}{lcl} [S_1 + \{x\}, C_1, g_1, r_1, Al], & \longrightarrow & [S_1, C_1, g_1, r_1, Al], \\ [S_2 + \{x\}, C_2, g_2, r_2, \_] & & [S_2, C_1, g_2, r_2, Al] \end{array}$$

$$\begin{array}{lcl} [S_1, C_1 + \{x\}, g_1, r_1, Al], & \longrightarrow & [S_1, C_1, g_1, r_1, Al], \\ [S_2, C_2 + \{x\}, g_2, r_2, \_] & & [S_2, C_1, g_2, r_2, Al] \end{array}$$

and finally the *alarm-end reactions*:

$$[\varnothing, \varnothing, g, c, Al] \longrightarrow (\nu g)[\varnothing, \varnothing, g, 1, Act(\mathbf{1})]$$

Note that one has two reactions to propagate the alarm depending on whether the alarm is shipped along the spanning tree or not. There are no longer any conditions on g_2 and r_2, since consistency is lost during dislocation. An agent goes active again only when it has broken all connections and spread the alarm to all its neighbours. Thus, active agents still view correctly their components, and we can easily extend proposition 1 to include the case of dislocation.

6 Conclusion

We have presented an abstract self assembly protocol by which a population of autonomous agents can grow an arbitrary network of connections in a distributed way. The target network is built incrementally. In any given connected component, a map of the current component circulates between the agents allowing them to make decisions concerning whether an edge should be added and where. One specificity of our approach is to cast the problem in the language of concurrent processes, actually in a simplified version of π-calculus, and be able to subsequently give a precise statement of the correctness of the proposed protocol. An implementation of the protocol extended with an alarm propagation mechanism avoiding deadlocks is available online.[2] A decidedly interesting extension would be to incorporate true space, if only because no ordinary process algebra models true space.

This protocol was inspired by self assembly questions in biological systems. Although we assumed computationally strong agents, we took care to keep their internals and interaction capacities, as embodied by the local reactions, to a minimum. It is agreed that they are still way stronger than anything that could be implemented today in the combinatorics of molecular biology. Nevertheless exploring such self-assembly procedures could help in building a working engineering intuition of biological self assembly.

References

1. Eric Klavins. Automatic synthesis of controllers for assembly and formation forming. In *Proceedings of the International Conference on Robotics and Automation*, 2002.
2. Eric Drexler and Richard Smalley. Controversy about molecular assemblers. Available at `www.foresight.org/NanoRev/Letter.html`, 2003.
3. Radhika Nagpal. Programmable self-assembly using biologically-inspired multiagent control. In *Autonomous Agents and Multiagent Systems Conference (AAMAS)*, July 2002.
4. Jeff Hasty, David McMillen, and James J. Collins. Engineered gene circuits. *Nature*, 420:224–230, November 2002.

[2] `http://www.pps.jussieu.fr/~tarissan/self`

5. Robin Milner. *Communicating and mobile systems: the π-calculus.* Cambridge University Press, Cambridge, 1999.
6. Robin Milner and Davide Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *Nineteenth Colloquium on Automata, Languages and Programming (ICALP) (Wien, Austria)*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.
7. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *Proceedings of the 12th European Symposium on Programming (ESOP'03, Warsaw, Poland)*, volume 2618 of *LNCS*, pages 302–318. Springer, April 2003.