Self Assembling Graphs

Vincent Danos\textsuperscript{1} and Fabien Tarissan\textsuperscript{2}

\textsuperscript{1} Équipe PPS, CNRS & Université Paris VII
\textsuperscript{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 \cite{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 \cite{2}, amorphous computations \cite{3}, molecular biology \cite{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 \cite{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 $\pi$-calculus \cite{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, 8]. 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 \( x, y, z, \ldots \), one defines an agent as a finite set \( \langle C \rangle \subseteq \mathcal{C} \), written \( \langle C \rangle \), where the set \( C \) itself is referred to as the agent interface. Agents can be arranged in networks according to the following grammar:

\[
G := \emptyset \mid \langle C \rangle \mid G, G \mid (\nu x)G
\]

where \( \emptyset \) 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:

\[
\text{becomes } (\nu x)(\nu y)(\langle x \rangle, \langle x, y \rangle, \langle y \rangle)
\]

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.