Submitted, 2002 IEEE Int'l Conference on Robotics and Automation
http://www.cs.caltech.edu/~klavins/papers/klavins-icra2002.pdf

Automatic Synthesis of Controllers for Distributed Assembly and Formation Forming

Eric Klavins

Computer Science Department
California Institute of Technology
Pasadena, CA 91125
klavins@cs.caltech.edu

Abstract

We consider the task of assembling a large number of self controlled parts (or robots) into copies of a prescribed assembly (or formation). In particular, we introduce a way to synthesize, from a specification of the desired assembly, local controllers to be used by each part which, when taken together, have the global effect of assembling the parts. We pay careful attention to the time and space complexity of the synthesis procedure, showing that the size of the representation of the synthesized controller is polynomial in the size of the specification and that the computational power needed by the controller is low.

1 Introduction

We consider the problem of controlling hundreds of robots to perform a task in concert. This problem presents many fundamental issues to robotics, control theory and computer science. With a great number of robots, decentralization is critical due to the cost of communication and the need for fault tolerance. In decentralized control, each robot should act based only on information local to it. It then becomes difficult, however, to guarantee or even derive the behavior of the entire system given the behaviors of the individual components. In this paper we address this difficulty in a novel way: We begin with a specification of an assembly and develop methods that allow us to automatically *synthesize* individual behaviors so that they are guaranteed to produce the desired global behavior.

Specifically, we consider the task of assembling many disk-shaped parts in the plane into copies of a prescribed assembly (formation), which is specified by a graph. As shown in Figure 1 we suppose that each part can move itself and can play any role in an assembly, which makes the task particularly rich. The contribution of the paper is a means of synthesizing from the specified assembly, a set of identical controllers for the parts to run which have the net effect of moving the parts to form copies of the specified assembly without colliding. The idea is that parts should join together to into subassemblies which should in turn join together to make larger assemblies and so on. In Section 3, a theory is developed along with algorithms which compile a specified assembly into a list of allowable subassemblies. In Section 3.3, we show how to produce a lookup table from the list which can be used as a discrete event controller (Figure 2) that guides parts through a "soup" of other parts and subassemblies. In Section 4, we add a continuous motion controller based on the assembly rules represented by the lookup table from Section 3.3. Various deadlock situations occur with the initial class of controllers we synthesize. In Section 4.2 we describe a means of avoiding this situation. Finally, we present our initial investigation into the effects of various compile-time choices on the performance of the resulting controllers. Throughout we pay careful attention to the time and space complexities of our algorithms — showing that they are polynomial in the size of the specified assembly.

1.1 Related Research

We are most strongly inspired by the work of Whitesides and his group [2, 3] in meso-scale self-assembly. In this work, small, regular plastic tiles with hydrophobic or hydrophylic edges are placed on the surface of some liquid and gently shaken. Tiles with hydrophobic edges are attracted along those edges while hydrophylic edges repel. Striking "crystals" emerge as larger and larger structures self assemble.

By using different shapes and edge types, different gross structures can be created. A similar idea is used on a much smaller scale in [13] where strands of DNA are attached to tiny gold balls in solution. Complementary strands attract and a gross structure is revealed. By choosing which strands go where, the "programmer" has some control over the resulting emergent structure. At least two next steps are apparent. First, these and similar [1] methods generally produce arrays or lattices of parts, meaning that there is no way to *terminate* a regular pattern at, say, a 5×5 array of parts (There has been work on changing the function of parts as they combine [16]). Second, there is no know formal method of starting with a *specification* of the desired emergent structure and devising the structure of the individual parts. In this paper we address both of these issues by supposing that each part can run a *program* that tells it when to join with another part, and when to repel it, based on some state information. Of course, this is a far way away from the reality of small plastic parts or gold balls, but our ideas could easily be implemented with teams of robots and may even, when developed further, present the chemist with new tools.

The motivation for considering disk shaped parts in the plane and for the potential field construction in Section 4 comes from the work of Koditschek and others [11, 7] in assembly. There, a global artificial potential function over the configuration space of n disk shaped parts is used to guide the parts to their assembled state, corresponding to the unique minimum of the potential function. The approach is not distributed, however, because it requires that each part know the full state of the system to act. Other work has applied similar ideas, in a distributed fashion [14], although without a means of assuring or even defining the resulting behavior. Still other approaches to the control of a group of robots [4] assume a leader. In contrast, the present paper commits to a strong degree of decentralization, using decentralized potential fields merely as a *primitive* in a more sophisticated hybrid control scheme.

The ideas in this paper also grow from our own work in controller synthesis in manufacturing systems [9, 8]. Our approach to manufacturing has been to synthesize a decentralized automated factory description from a description of a product. The description includes the layout of the factory and the control programs the robots should run to produce the product. In that sense, the present work is an extension of the idea, although it assumes fewer constraints on the topology of the workspace.

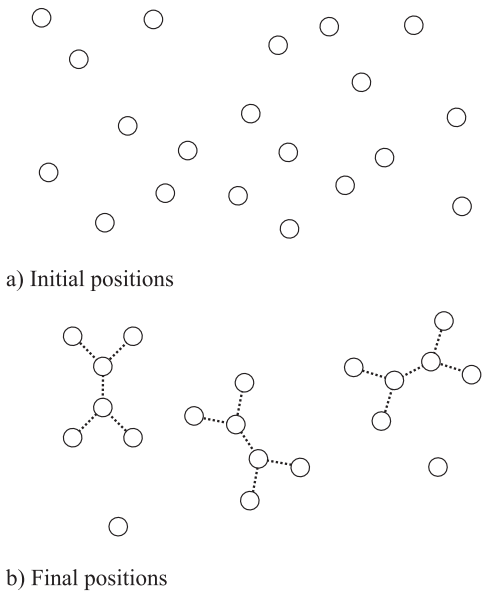

Figure 1: The goal of the assembly problem. Each disk shaped part must move from its initial position (a) to a position in an assembly (b). Dashed lines show the resulting adjacency relationship E. There may be leftover parts.

2 The Problem

We consider a simple form of assembly process by assuming that parts are programmable and able to sense the position and state of other nearby parts. We start with m disk-shaped parts (of radius r) confined to move in $\mathbb{R}^2$. Denote the position of part i by the vector x_i. We desire that each part move smoothly, without colliding with other parts, so that all n parts eventually take some role in an *assembly* or *formation*. This is shown graphically in Figure 1. For simplicity, we assume that the dynamics of each disk are given by $\ddot{x} = u_i$. We believe that control of parts with more complicated dynamics can be based on the control algorithms we develop for this simple situation. In this section we describe the goal of assuming a role in a formation formally.

Let $G = (V, E)$ be a finite undirected, acyclic graph. Thus, V is a finite set of nodes (in this paper, $V = \{1, ..., n\}$) and E is a collection of edges of the form $\{a, b\}$ with $a, b \in V$ and $a \neq b$. In this paper, we will call such a graph an *assembly* and only consider the case where G is a tree (i.e., contains no cycles). There are technical details, which are solvable but not addressed in this paper except briefly, that prevent

the direct application of the methods in this paper to general graphs.

Given an assembly $G = (V, E)$ with $|V| = n$, consider the case where $m = n$. The problem is to produce a control algorithm to be used by each part that will control the m parts to move, without colliding, from arbitrary initial conditions to positions such that there exists a permutation h of $\{1, ..., m\}$ such that

1. If $\{h(i), h(j)\} \in E$ then $k_{nbr} - \epsilon < ||x_i - x_j|| < k_{nbr} + \epsilon$;
2. If $\{h(i), h(j)\} \notin E$ then $||x_i - x_j|| > k_{nbr}$.

Here $k_{nbr} > 0$ and $\epsilon > 0$ are parameters. The image $h(i)$ of i is called the *role* of i in the assembly. We furthermore require that these assemblies be stable to disturbances in the sense that the set of points $x_1, ..., x_m$ satisfying the above conditions is an attractor of the closed loop dynamics we will construct. If $m = kn$ for some $k \in \mathbb{Z}$ then we still require the above except now with respect to a disjoint union of k copies of G. And of course, if m is not a multiple of n, then we require that as many parts as possible form assemblies in the obvious way.

A part j such that $\{h(i), h(j)\} \in E$ is called a neighbor of i. Suppose that i has n_i neighbors each distance k_{nbr} from it and equally spaced around it. Then it is simple to show that the distance between the neighbors is $2k_{nbr} \cos(\frac{(n_i-2)\pi}{2n_i})$. Thus, for the above condition (2) to hold, $n_i \leq 5$. Thus, we assume that all nodes in the assemblies we specify have degree less than 6.

2.1 Controller Structure

In general we will assume that parts have limited sensing and communication capabilities and we allow them to store a discrete state, s_i, along with their control programs. In particular, we assume that part i can sense its own position and the positions and discrete states of other parts within some range $d_{max} > 0$ of x_i.

The methods we develop below will, given a description of the desired assembly structure, *synthesize* a hybrid controller H_i of the form shown in Figure 2. The goal is that when each part runs a copy of H_i (from different initial conditions), the parts will self assemble.

The controller H_i is described by a continuous control law F_i, a predicate $\mathcal{A}$ called the *attraction predicate* and a discrete update rule g. F_i describes the force that the part should apply to itself. $\mathcal{A}(s_i, s_j) \in \{\text{true}, \text{false}\}$ determines whether parts i and j with

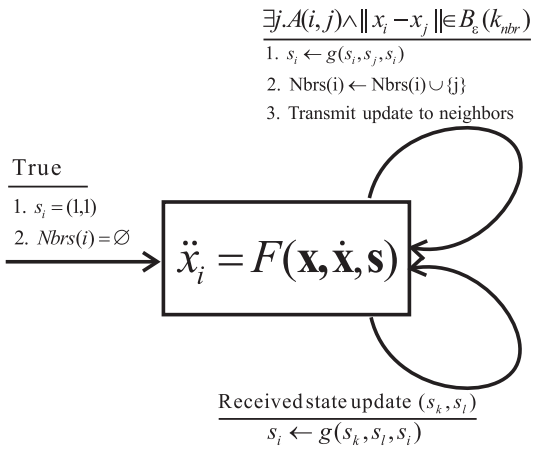

Figure 2: The structure of the hybrid controller that is constructed by the compilation scheme in this paper. Arcs denote transitions and are labeled by a predicate/action pair. When an arc's predicate becomes true, the action is taken and control transfers from the source of the arc to the target of the arc.

states s_i and s_j should try to become neighbors, thereby forming a larger assembly. The update rule $g(s_i, s_j, s_k)$ determines the new discrete state of part k based on the joining of parts i and j. Loosely, the operation of H_i is as follows. Part i starts with some initial position $x_i(0)$, the initial state $s_i(0) = (1, 1)$ and no neighbors. It then applies the control force $F_i(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{s})$ to itself until either a new neighbor is detected or it receives a state update from a neighbor. Here $\mathbf{x}$, $\dot{\mathbf{x}}$ and $\mathbf{s}$ are m dimensional vectors describing the complete state of the system. However, F_i may only use the states of the parts within distance d_{max} of part i. The force F_i is computed based on the position, velocity and discrete state of part i and on the discrete states of the sensed parts.

The task of an automatic synthesis procedure, performed by what we are calling a *compiler*, is to take a description of a desired assembly and produce H_i — in this case, F_i, $\mathcal{A}$ and g. The construction of $\mathcal{A}$ and g are described in Section 3 and the construction of F_i, which requires $\mathcal{A}$, is discussed in Sections 4 and 4.1.

3 Compilation of Assembly Rules from Specifications

In this paper, an assembly can be specified simply by listing which roles in the assembly are adjacent — that is, by a graph. As mentioned above, we restrict

ourselves to the situation where the adjacency graphs are trees, leaving the detail of arbitrary graphs to future work (see Section 5). In any case, we believe that assembling an arbitrary graph will start with the assembly of a spanning tree of that graph.

The goal of this section is to produce the attraction predicate $\mathcal{A}$ and the update rule g from a specified assembly $G_{spec} = (V_{spec}, E_{spec})$, which we assume is a tree. This requires first generating a set of subassemblies of G_{spec} (Section 3.2) and then compiling $\mathcal{A}$ and g from the set (Section 3.3).

3.1 Discrete State of a Part

We intend that the parts control themselves to first form subassemblies of G_{spec}, and from those subassemblies form larger subassemblies and so on until G_{spec} is finally formed. The discrete state of a part must, therefore, include a reference to the subassembly in which it currently plays a role. To this end, we build a list (in Section 3.2) of the particular (connected) subassemblies we will allow: $\mathcal{G} = \{G_1, ..., G_p\}$. We require that each $G_i \in \mathcal{G}$ is of the form (V_i, E_i) where $V_i = \{1, ..., |V_i|\}$ and $E_i \subseteq V_i \times V_i$. Although this representation of subgraphs in $\mathcal{G}$ is arbitrary, because the vertices in V_i could have been named in other ways, some common scheme is required for a graceful definition of the states of parts.

Now, the discrete state of a part consists of a pair $s_i = (j, k) \in \mathbb{Z}^2$ where j is the index of a subassembly in $\mathcal{G}$ and $k \in V_i$ is a *role* in that subassembly.

3.2 Generating Assembly Sequences

Define an operation on assemblies G_1 and G_2 as follows

Definition 3.1 *The* **join** *of G_1 and G_2 via vertices $u \in V_1$ and $v \in V_2$, denoted $G_1.u \oplus G_2.v$, is defined as $G_1.u \oplus G_2.v = (V, E)$ where*

$$V = \{1, ..., |V_1| + |V_2|\} \text{ and}$$

$$E = E_1 \cup \{\{a+|V_1|, b+|V_1|\} \mid \{a, b\} \in E_2\} \cup \{u, v+|V_1|\}.$$

For example

$$(\{1,2\}, \{\{1,2\}\}).2 \oplus (\{1,2\}, \{\{1,2\}\}).1$$

$$= (\{1,2,3,4\}, \{\{1,2\}, \{2,3\}, \{3,4\}\}).$$

We will also use the notations $i.j \oplus k.l$ and $(i,j) \oplus (k,l)$ to mean the join of the assemblies with indices i and k in a given $\mathcal{G}$ via the vertices with indices j and l.

The set of subassemblies $\mathcal{G}$ must have the following property:

Algorithm A_1:

Input: $G = (V, E)$, a tree
$u \in V$, a node
Output: $\mathcal{G}$, a list of subassemblies

$S \leftarrow$ Neighbors(v)
$G_{new} = (V_{new}, E_{new}) \leftarrow (\{1\}, \emptyset)$
$\mathcal{G} \leftarrow \{G_{new}\}$
While $S \neq \emptyset$
  $v \leftarrow$ ElementOf(S)
  $V_{new} \leftarrow V_{new} \cup \{v\}$
  $E_{new} \leftarrow E \mid_{V_{new}}$
  $\mathcal{G} \leftarrow \mathcal{G} \cup \{$MakeStandard($G_{new}$)$\}$
  $S \leftarrow (S - \{v\}) \cup ($Neighbors(v) $-$ $V_{new})$

Figure 3: Psuedocode to generate a size-m set of subassemblies of a given assembly of size m. In the code, Neighbors(v) is the set of neighbors v with respect to G and ElementOf(S) is a randomly chosen element of the set S. The function *MakeStandard* takes an assembly and renames the nodes so that they form the set $\{1, ..., |V_{new}|\}$.

Property 3.1 *For all $G \in \mathcal{G}$ there exist $G_1, G_2 \in \mathcal{G}$, $u \in V_1$ and $v \in V_2$ such that $G_1.u \oplus G_2.v \simeq G$ unless $G = \{\{1\}, \emptyset\}$ and there does not exist a $G' \in \mathcal{G} - \{G\}$ with $G \simeq G'$.*

Here "$\simeq$" means isomorphic in the usual sense: $(V_1, E_1) \simeq (V_2, E_2)$ if there exists a function $h : V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(h(u), h(v)) \in E_2$. Such an h is called a *witness* of the isomorphism. Witnesses are used in this paper to "translate" the representation of the join of two graphs to the representation of that graph in $\mathcal{G}$. Property 3.1 assures that any assembly can be constructed from exactly two other assemblies, so that only pairwise interactions between parts need be considered by the ultimate controller, and that there is only one representation of each subassembly in the list.

The simplest means of automatically constructing $\mathcal{G}$ from G_{spec} is to simply set $\mathcal{G}$ to be all possible connected subgraphs of G up to isomorphism, producing a set of size $O(2^n)$. This set can be computed using a simple exhaustive search. Since $\mathcal{A}$ and g will be obtained from a table constructed from $\mathcal{G}$ (see Section 3.3), this may be an impracticably large set for large G_{spec}, although for small assemblies the set of all subassemblies is quite reasonable and produces good controllers. A $\mathcal{G}$ thus constructed trivially satisfies Property 3.1.

Another means of constructing $\mathcal{G}$ is to build subtrees of G_{spec} one node at a time, starting at some base node. A simple algorithm, A_1, for doing this is shown in Figure 3. It requires an assembly G_{spec} and a base node i. It produces a set $\mathcal{G}_{A_1,i}$ of size exactly n, there being one subassembly for each size 1 to n. The set $\mathcal{G}_{A_1,i}$ constructed using A_1 satisfies Property 3.1 easily since each subassembly (except the singleton assembly) can be obtained by joining the next smallest subassembly with $\{\{1\}, \emptyset\}$.1. Richer subassembly sets can be made by calling A_1 again, starting with a different base node, and combining it with the first set. In this manner a set of size $O(cn)$ can be constructed from a set of c nodes $U \subseteq V_{spec}$. Call this set $\mathcal{G}_{A_1,U}$. It satisfies Property 3.1 because each of the sets $\mathcal{G}_{A_1,i}$ for $i \in U$ do. The process of combining the sets requires some computation, however, because we must maintain the second part of Property 3.1. To combine the list $\mathcal{G}_{A_1,i}$ with list $\mathcal{G}_{A_1,j}$ we must compare each element of the first list with each element of the second list to make sure they are not isomorphic. If they are, we keep only one of them for the combined list. Although there is no known polynomial time algorithm for checking the isomorphism of two graphs, checking the isomorphism of two trees of size n takes $O(n^{3.5})$ steps [15]. Thus, combining two size n lists takes time $O(n^{5.5})$. The reader can check that the combination of sets satisfying Property 3.1 also satisfies Property 3.1.

3.3 Generating Update Rules

From an assembly set $\mathcal{G}$ satisfying Property 3.1, we can state the definition of $\mathcal{A}$ simply:

Definition 3.2 *Given $\mathcal{G}$ satisfying Property 3.1, the* **attraction predicate** *$\mathcal{A}$ is defined as*

$$\mathcal{A}(s_i, s_j) = \text{true} \Leftrightarrow \exists\, G \in \mathcal{G} \text{ such that } s_i \oplus s_j \simeq G.$$

We can also define the update rule g.

Definition 3.3 *Given $\mathcal{G}$ satisfying Property 3.1 and states s_i and s_j with $\mathcal{A}(s_i, s_j) =$ true, the* **update rule** *g is defined as follows. Suppose $G \simeq s_i \oplus s_j$ has index k in $\mathcal{G}$, suppose $h : s_i \oplus s_j \to G$ witnesses this isomorphism and suppose $s_l = (a, b)$. Then*

$$g(s_i, s_j, s_l) \doteq (k, h(b'))$$

where $b' \in V(s_i \oplus s_j)$ is the name of vertex b after taking disjoint unions in Definition 3.1 of the join operation. If $\mathcal{A}(s_i, s_j) =$ false then the update rule is not defined: $g(s_i, s_j, s_l) \doteq \perp$.

The procedure for determining the values of $\mathcal{A}$ and g require determining tree isomorphisms — which is likely too time consuming to be done online. We can, however, perform all the necessary computations offline by compiling $\mathcal{G}$ into a table. The result is that H_i can make all discrete transitions essentially instantaneously because all that is required is a table lookup. Furthermore, the size of the table is $O(|\mathcal{G}|^2 n^3)$. As was shown, $|\mathcal{G}|$ can taken to be cn, so that even complicated assemblies require only $O(n^5)$ storage.

The construction proceeds in two steps. First, we determine a representation of the update function g resulting from a join of $G_i.j$ with $G_k.l$. Second we build a table of all possible joins between all possible pairs of distinct graphs taken from $\mathcal{G} - G_{spec}$. The result is a four dimensional table T where each entry $T_{i,j,k,l}$ is the representation of $G_i.j \oplus G_k.l$.

Given $G_i.j$ and $G_k.l$, let $G = (V, E) = G_i.j \oplus G_k.l$. We must first determine whether there exists a $G' \in \mathcal{G}$ such that $G \simeq G'$ then, we require a witness h of this isomorphism because we must have a means of translating the new roles of each part in the new assembly into their representations in $\mathcal{G}$. Suppose such an h exists. Then we represent the table entry $T_{i,j,k,l}$ as a pair

$$(\ index(G'),\ \langle h(1), ..., h(|V_i| + |V_j|)\rangle\ ).$$

Otherwise, set $T_{i,j,k,l} = \perp$. The procedure for constructing T is shown in Figure 4, it takes time $O(|\mathcal{G}|^3 n^{6.5})$ because of the added complexity of finding a witness for each join.

To summarize, given G_{spec}, constructing $\mathcal{A}$ and g, the discrete part of the controller H_i, proceeds in two steps. First, a list of subassemblies $\mathcal{G}$ is build from G_{spec} using one of the methods discussed in Section 3.2. Second, using algorithm A_2, a table T is built from the $\mathcal{G}$. $\mathcal{A}(s_i, s_j)$ can be computed simply by checking whether $T_{s_i,s_j} \neq \perp$ and $g(s_i, s_j, (a, b))$ can be determined by looking up T_{s_i,s_j} and reading off $h(b)$.

4 Implementation of Assembly Rules

Completing the controller H_i shown in Figure 2 requires a definition of F_i as well as some method by which parts can communicate. In the example in Section 4.1, we define an F_i and assume a simple communications scheme that works in simulation and about which we have a preliminary analytical understanding [10].

We suppose that parts can only communicate with their neighbors. The difficulty is then that two parts

Algorithm A_2:

Input: $\mathcal{G}$, a list of subgraphs with Property 3.1
Output: T, a tabular representation of $\mathcal{A}$ and g

For $i = 1$ to $|\mathcal{G}| - 1$
  For $k = i$ to $|\mathcal{G}| - 1$
    For $j = 1$ to $|V_i|$
      For $l = 1$ to $|V_k|$
        If $\exists\, G \in \mathcal{G}$ with $i.j \oplus k.l \simeq G$
          Let h be the witness
          $T_{i,j,k,l} = (\ index(G'),\ \langle h \rangle)$
        Else $T_{i,j,k,l} = \perp$

Figure 4: The procedure for constructing a table of size $O(|\mathcal{G}|^2 n^3)$ from a list of subassemblies $\mathcal{G}$ of a specified tree G_{spec}. The predicate $\mathcal{A}$ and the update rule g can be read off the resulting table in constant time.

playing roles in the same subassembly might try to update the state of that subassembly simultaneously. Thus, such an update requires a means of obtaining consensus among all parts in the subassembly. Consensus can be difficult or even impossible if the processing is asynchronous and there are process or link failures [12], although approximate algorithms exist for these situations [6]. In what follows, we assume a good consensus algorithm no process of communication failures. Consideration of the many complications we may add, although important, would take us too far afield of the present topic and are somewhat independent of methods we have so far described.

4.1 An Example Implementation

For each part i, we can decide, using $\mathcal{A}$, whether part i should move toward j or not. To this end define

$$\begin{aligned} S(i) &= \{j \mid ||x_i - x_j|| < d_{max}\} \\ Attract(i) &= (\{j \mid \mathcal{A}(s_i, s_j)\} \cup Nbrs(i)) \cap S(i) \\ Repel(i) &= (\{j \mid \neg\mathcal{A}(s_i, s_j)\} - Nbrs(i)) \cap S(i). \end{aligned}$$

$S(i)$ is the set of parts that i can sense. Note that these sets are easily computed from a table compiled from a given G_{spec}. One way of forming the control law F_i is to sum, for each $j \in Attract(i)$ a vector field F_{att} which has an equilibrium set at distance k_{nbr} from x_j and for each $j \in Repel(i)$ a vector field F_{rep} which has x_j as a repellor. We can construct these fields from the potential functions defined by

$$\begin{aligned} V_{att}(x_i, x_j) &= \left(\frac{||x_i - x_j|| - k_{nbr}}{||x_i - x_j|| - r}\right)^2 \\ V_{rep}(x_i, x_j) &= \left(\frac{1}{||x_i - x_j|| - r}\right)^2. \end{aligned}$$

Recall that r is the radius of the (disk shaped) parts. Then we set

$$\begin{aligned} F_{att}(x_i, x_j) &= -\frac{1}{||x_i - x_j||}\frac{\partial V_{att}}{\partial x_i}(x_i, x_j) \\ F_{rep_1}(x_i, x_j) &= -\frac{1}{||x_i - x_j||}\frac{\partial V_{rep}}{\partial x_i}(x_i, x_j) \\ F_{rep_2}(x_i, x_j) &= \begin{cases} 0 \text{ if } ||x_i - x_j|| > k_{nbr} + \delta \\ F_{rep_1}(x_i, x_j) \text{ otherwise} \end{cases} \end{aligned}$$

where $\delta > 0$ is some small constant. We have scaled the gradients of the potential functions by $||x_i - x_j||^{-1}$ so that the "influences" of parts nearest i are felt most strongly. We have also defined two versions of the repelling field. We use F_{rep_2} because it is only active when parts violate condition (2) from Section 2. We will see the reason for this shortly.

For the complete control law we use

$$\begin{aligned} F_i(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{s}) &= \sum_{j \in Attract(i)} F_{att}(x_i, x_j) \\ &\quad + \sum_{j \in Repel(i)} F_{rep_2}(x_i, x_j) - b\dot{x}_i \end{aligned}$$

where $b > 0$ is a damping parameter. In practice we assume a maximum actuator force, setting $u_i = \max\{u_{max}, F_i(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{s})\}$. It can be shown that the following holds:

Proposition 4.1 *Suppose G_{spec} has maximum degree 5, $n = |V_{spec}|$, and the neighbors relation $Nbrs$ induces a graph isomorphic to G_{spec}. Then all points in the set*

$$\begin{aligned} B = \{(\mathbf{x}, \dot{\mathbf{x}}) \mid (j \in Nbrs(i) \rightarrow ||x_i - x_j|| = k_{nbr}) \\ \wedge (j \notin Nbrs(i) \rightarrow ||x_i - x_j|| > k_{nbr} + \delta)\} \end{aligned}$$

are equilibrium points of $F = (F_1, ..., F_n)$.

In [10] we present an initial investigation of the local stability of B with respect to $F = (F_1, ..., F_n)$. The reason for using F_{rep_2} instead of F_{rep_1} is now evident: The repelling component of F_i is zero in B allowing us to ignore the repelling part of F_i when determining the equilibriums of F.

Simulations of the above system, from a variety of initial conditions, with varying numbers of agents (from tens to hundreds), and various specifications of the desired assembly G_{spec} can be viewed at http://www.cs.caltech.edu/~klavins/rda/.

4.2 Deadlock Avoidance

Even if B (in Proposition 4.1) is locally stable, it is certainly not globally stable with respect to the hybrid controller H_i. Two deadlock situations arose in our initial simulations. First, F may have spurious stable equilibriums which prevent attracting pairs from moving toward each other. Second, it is possible that the set of currently formed subassemblies admit no joins in $\mathcal{G}$. That is, it may be that at some time there do not exist parts i and j such that $\mathcal{A}(s_i, s_j)$ is true.

To avoid these situations, we employ a simple deadlock avoidance method. For each subassembly $G_k \in \mathcal{G}$ we define a *stale time* $stale(k) \in \mathbb{R}$. Any subassembly that has not changed state within $stale(i)$ seconds of its formation time should (1) break apart, setting the state of each part in it to $(1, 1)$ and (2) have each part "ignore" other parts from that same assembly for $stale(k)$ seconds. If k_{spec} is the index of G_{spec} in $\mathcal{G}$, we set $stale(k_{spec}) = \infty$. The result is a new controller $H_{d,i}$ that checks for staleness and implements (1) and (2) above, but is otherwise similar to H_i in Figure 2. We also change the definitions of $Attract(i)$ and $Repel(i)$. Suppose that $Ignore(i)$ is the set of all part indices that part i is presently ignoring due to a staleness break-up. Then

$$\begin{aligned} Attract_d(i) &= Attract(i) - Ignore(i) \\ Repel_d(i) &= Repel(i) - Ignore(i). \end{aligned}$$

F_i is then changed accordingly. Using this deadlock avoidance measure, we have not yet seen a set of initial conditions for any G_{spec} we tried for which our simulation did not converge upon a maximum number of parts playing roles in a final assembly.

4.3 Comparing the Performance of Various Compilation Options

The rate of assembly of a specified tree G_{spec} depends on the number of join interactions possible at any given time. By increasing the size of $\mathcal{G}$ — that is, the number of subassemblies possible — we can increase the number of opportunities that a given part has to join with other parts. Figure 5 shows the affect of the size of $\mathcal{G}$ on the rate of assembly. Starting with a specification G_{spec} of a ten part assembly, we generated two subassembly lists: $\mathcal{G}_{all}$ and $G_{A_1,1}$. The first contained all possible subassemblies (17 of them) while the second, obtained using algorithm A_1 contained 10 subassemblies. We compiled each list into a controller with deadlock avoidance then ran 50 simulations of each controller from randomly chosen initial

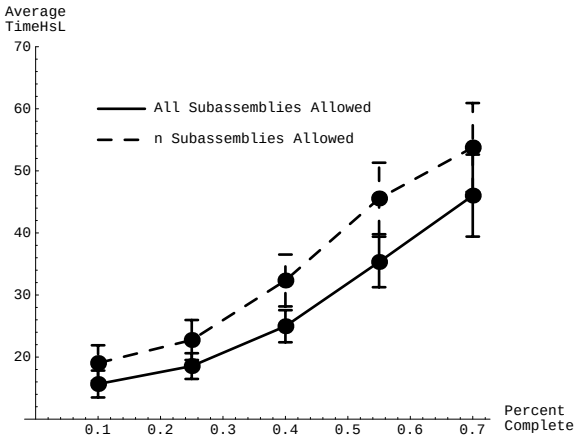

Figure 5: Comparison of assembly times for $\mathcal{G}_{all}$ and $G_{A_1,1}$. See Section 4.3.

conditions using 110 parts. The time in seconds until $110p$ parts found roles in a copy of the final assembly was recorded for $p \in \{0.1, 0.25, 0.4, 0.55, 0.7\}$. The figure shows the time until $p\%$ of the parts were assembled versus p averaged over the 50 runs. Error bars denote standard deviation. The solid line shows the controller with $\mathcal{G}_{all}$ and the dashed line shows the controller with $G_{A_1,1}$. Apparently, the increased number of interactions possible afforded by just seven extra subassemblies results in an increase in the "reaction rate" of the system. Thus, the price of using algorithm A_1 is apparent. We expect that the cost of using a linear or quadratic size list of subassemblies instead of the full exponential sized list will increase dramatically as the size of G_{spec} increases, although we do not yet have an analytical means of proving so.

5 Conclusion

The ideas in this paper represent only the first steps toward understanding and realizing specifiable, programmable self assembly. Many relatively unexplored and apparently fruitful issues remain. First, although simulations suggest that the implementation (particular choice of F_i) given in Section 4.1 combined with the deadlock avoidance procedure in Section 4.2 produces controllers that assemble a maximum number of parts safely (without collisions), this must be verified analytically: A substantial problem requiring possibly new tools in hybrid systems analysis. We also hope to develop a formal understanding of the dependence of the rate at which parts assemble on the various com-

pile time options presented in Section 3. Many variations on the theme presented here should be explored: hierarchical assembly with intermediate goal assemblies, three dimensional assembly, assembly of non-homogeneous parts, assembly of parts with complex dynamics (e.g. nonholonomic), and so on. Finally, we are exploring hardware implementations of these algorithms so that the issues of asynchronous processing, inaccurate sensors and faulty communications may be addressed.

Acknowledgements

Thank you to Dan Koditschek with whom I have discussed many of the ideas. The research is supported in part by DARPA grant numbers JCD.61404-1-AFOSR.614040 and RMM.COOP-1-UCLA.AFOSRMURI.

References

[1] E. Bonabeau, S. Guerin, D. Snyers, P. Kuntz, and G. Theraulaz. Three-dimensional architectures grown by simple 'stigmergic' agents. *BioSystems*, 56:13–32, 2000.

[2] N. Bowden, L. S. Choi, B. A. Grzybowski, and G. M. Whitesides. Mesoscale self-assembly of hexagonal pates using lateral capillary forces: Synthesis using the "capilary" bond. *Journal of the American Chemical Society*, 121:5373–5391, 1999.

[3] T. L. Breen, J. Tien, S. R. J. Oliver, T. Hadzic, and G. M. Whitesides. Design and self-assembly of open, regular, 3D mesostructures. *Science*, 284:948–951, 1999.

[4] J. P. Desai, V. Kumar, and J. P. Ostrowski. Control of changes in formation for a team of mobile robots. In *IEEE International Conference on Robotics and Automation*, Detroit, May 1999.

[5] J. A. Fax and R. M. Murray. Graph laplacians and vehicle formation stabilization. Technical Report CDS 01-007, California Institute of Technology, 2001. Submitted to IFAC 2002.

[6] M. Franceschetti and J. Bruck. A group membership algorithm with a practical specification. To appear in *IEEE Transactions on Parallel and Distributed Systems*.

[7] S. Karagoz, H. I. Bozma, and D. E. Koditschek. Event driven parts moving in 2d endogenous environments. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1076–1081, San Francisco, CA, 2000.

[8] E. Klavins. Automatic compilation of concurrent hybrid factories from product assembly specifications. In *Hybrid Systems: Computation and Control Workshop, Third International Workshop*, Pittsburgh, PA, 2000.

[9] E. Klavins and D.E. Koditschek. A formalism for the composition of concurrent robot behaviors. In *Proceedings of the IEEE Conference on Robotics and Automation*, 2000.

[10] Eric Klavins. Toward a proof of stability for a distributed assembly system. http://www.cs.caltech.edu/∼klavins/papers/tr-rda.pdf, unpublished.

[11] D. E. Koditschek, , and H. I. Bozma. Robot assembly as a noncooperative game of its pieces. *Robotica*, 2000. to appear.

[12] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[13] C. A. Mirkin. Programming the assembly of two- and three-dimensional architectures with dna and nanoscale inorganic building blocks". *Inorganic Chemistry*, 39(11):2258–2272, 2000.

[14] H. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. In *Proceedings of the 1994 Workshop on the Algorithmic Foundations of Robotics*. A.K.Peters, Boston, MA, 1995.

[15] S. W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM Journal on Computing*, 6:730–732, Dec 1977.

[16] K. Saitou. Conformational switching in self-assembling mechanical systems. *IEEE Transactions on Robotics and Automation*, 15(3):510–520, 1999.