In *Proceedings of the 2002 International Conference on Robotics and Automation*

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[1]

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

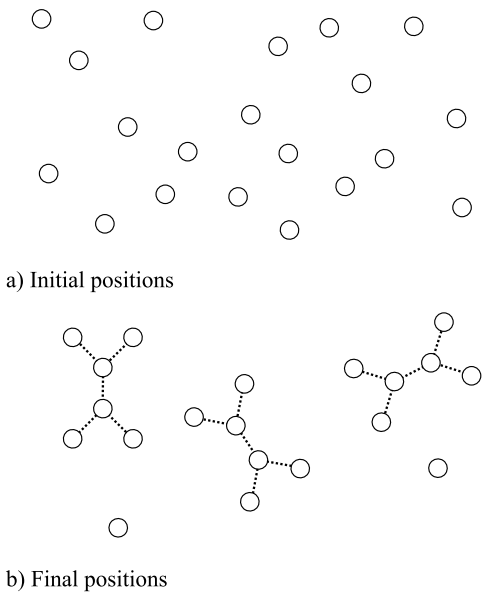

Figure 1: The goal of the assembly problem. Each disk shaped part must move from its initial position (a) to some position in a copy of an assembly (b). Dashed lines show the resulting adjacency relationship E. There may be leftover parts.

[1]This research was supported in part by Darpa SEC grant number F33615-98-C-3613 and AFOSR grant number F49620-01-1-0361.

join together to into subassemblies which should in turn join together to make larger assemblies and so on. In §3, a theory is developed along with algorithms which compile a specified assembly into a list of allowable subassemblies. In §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 §4, we add a continuous motion controller based on the assembly rules represented by the lookup table from §3.3. Various deadlock situations occur with the initial class of controllers we synthesize. In §4.2 we describe a means of avoiding this situation. Finally, we summarize a proof (given in [9]) that the discrete dynamics given by the lookup table and the deadlock avoidance mechanism are correct. The proof assumes a certain logical model of the dynamics which accounts for the discrete interactions between parts (forming neighbor relationships) but neglects the continuous dynamics.

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.

The motivation for considering disk shaped parts in the plane and for the potential field construction in §4 comes from the work of Koditschek and others [11, 6] 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 and uses 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 [10, 7]. 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.

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 $\ddot{x} = u_i$.

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.

We note that not all trees can be embedded in the plane in such a way that neighbors are distance d_{nbr} apart and non-neighbors are distance greater than d_{nbr} apart. For simplicity in what follows, we restrict the assemblies we specify to those that can be so embedded.

2.1 Controller Structure

In general we 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 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

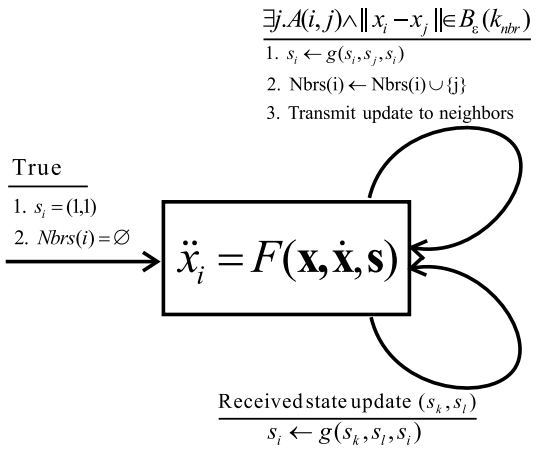

Figure 2: The structure of the hybrid controller that is constructed by the compilation scheme in this paper. Arcs denote transitions and are labeled by a predicate/action pair. When an arc's predicate becomes true, the action is taken and control transfers from the source of the arc to the target of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

$F_i(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{s})$ 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 §3 and the construction of F_i, which requires $\mathcal{A}$, is discussed in §4 and §4.1.

3 Compilation of Assembly Rules from Specifications

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} (§3.2) and then compiling $\mathcal{A}$ and g from the set (§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 §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 $E_i \subseteq V_i \times V_i$.

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*

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

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

We 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:

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 §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 and adding nodes to the leaves of subtrees. This algorithm, which we call A_1, 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 look-up. Furthermore, the size of the table is $\mathrm{O}(|\mathcal{G}|^2 n^3)$. As was shown, $|\mathcal{G}|$ can taken to be cn, so that even complicated assemblies require only $\mathrm{O}(n^5)$ 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 takes time $\mathrm{O}(|\mathcal{G}|^3 n^{6.5})$ 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 §3.2. Second 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 §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.

We suppose that parts can only communicate with their neighbors. The difficulty is then that two parts 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 [5]. In what follows, we assume a good consensus algorithm no process of communication failures.

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 §2.

For the complete control law we use

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

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 easily shown that the configurations representing assembled products are stable equilibria, using the obvious composition of V_{att} as a Lyapunov function.

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

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. In the next section, we suggest more formally why this is so.

5 Correctness

In this section, we summarize a the proof given in [8] that the assembly rules and the deadlock avoidance mechanism we defined are correct, with respect to the following simplified discrete model of the dynamics. We ignore the continuous state completely and concentrate on the discrete state. Initially, all parts have state $(1, 1)$. At each step, two parts i and j for which $\mathcal{A}(s_i, s_j)$ is true are picked, their states and the states of their current assemblies are updated according to g and their neighbor relations are updated as well. If no such pair exists and there are at least two non-final subassemblies, then the smallest subassembly is broken up according to the deadlock avoidance mechanism. At every step k, we define $\Gamma(k)$ to be the graph induced by the neighbor information kept by each part. In general $\Gamma(k)$ will be a *forest* whose components correspond to subassemblies. This model is defined formally as an automaton in [8].

Two properties hold about the system defined above. The first is a *safety* property, asserting that only subassemblies in $\mathcal{G}$ form during executions of the system. The second is a *progress* property, asserting essentially that the number of components of $\Gamma(k)$ decreases as k increases. From this property we can conclude that every run of the system ends with a maximum number of final subassemblies being formed.

Theorem 5.1 *For all $k \in \mathbb{N}$, every component of $\Gamma(k)$ is isomorphic to some graph $G' \in \mathcal{G}$.*

We define a new property on $\mathcal{G}$ that is that we require of assembly sequences in addition to Property 3.1:

Property 5.1 *$\{\{1\}, \emptyset\} \in \mathcal{G}$ and for all $G \in \mathcal{G}$ there is a $u \in V(G)$ such that $G.u \oplus \{\{1\}, \emptyset\}.1$ is isomorphic to some graph in $\mathcal{G}$, unless G is the final assembly.*

Theorem 5.2 *Suppose Property 5.1 holds for $\mathcal{G}$. Then every sequence of states in the discrete model of the assembly dynamics ends with a maximum number of final assembles being formed.*

6 Conclusion

The ideas in this paper represent only the first steps toward understanding and realizing specifiable, programmable self assembly. Many relatively unexplored

and fruitful issues remain. First, although simulations and the results in §5 suggest that the implementation (particular choice of F_i) combined with the deadlock avoidance procedure produces controllers that assemble a maximum number of parts safely (without collisions), this must be verified analytically using the tools in §5 and tools from non-linear dynamical systems.

Arbitrary graphs (as opposed to trees) require certain embeddings of their subassemblies in order to assemble themselves. For example, suppose we assemble a graph by first assembling a spanning tree of the graph and then "closing" it. If we require the closing procedure to respect the d_{nbr} distance requirements we have used, then the tree can not *cross over* itself while closing. This means the tree must assemble to an appropriate embedding class — a constraint we do not yet deal with, but plan to address soon.

Many variations on the theme presented here should also be explored: hierarchical assembly with intermediate goal assemblies, three dimensional assembly (which has fewer "closing" problems than in two dimensions), 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 realistically addressed.

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 plates 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] M. Franceschetti and J. Bruck. A group membership algorithm with a practical specification. To appear in *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[6] 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.

[7] 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.

[8] E. Klavins. Automatically synthesized controllers for distributed assembly. In *Proceedings of the Conference on Cooperative Control and Optimization*. Gainsville, FL, Nov 2001.

[9] E. Klavins. Automatically synthesized controllers for distributed assembly: Partial correctness. In *Cooperative Control and Optimization*, 2001. Presented at Conference. Submitted to Edited Volume.

[10] 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.

[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.