Programmable Parts: A Demonstration of the Grammatical Approach to Self-Organization

J. Bishop, S. Burden, E. Klavins*, R. Kreisberg, W. Malone, N. Napp and T. Nguyen

Department of Electrical Engineering
University of Washington
Seattle, WA 98195
klavins@ee.washington.edu

***Abstract*— In this paper, we introduce a robotic implementation of the theory of *graph grammars*[12], which we use to model and direct self-organization in a formal, predictable and provably-correct fashion. The robots, which we call *programmable parts*, float passively on an air table and bind to each other upon random collisions. Once attached, they execute *local rules* that determine how their internal states change and whether they should remain bound. We demonstrate through experiments how they can self-organize into a global structure by executing a common graph grammar in a completely distributed fashion. The system also presents a challenge to the grammatical method (and to distributed systems approaches in general) due to the stochastic nature of its dynamics. We conclude by discussing these challenges and our initial approach to addressing them.**

I. Introduction

Engineering self-organizing processes presents us with the daunting problem of manipulating and coordinating vast numbers of objects so that they perform global tasks. Because of the potentially enormous quantities of objects involved, uniquely addressing and manipulating each one is impossible. Nevertheless, there are examples of complex machines, such as the ribosome or the motor in a bacterial flagellum, that seem to be built *in bulk* spontaneously out of large numbers of simple components. This seems to occur when simple components *self-organize* via local interactions into more complex aggregates which, in turn, self-organize into larger aggregates and processes. Our goal is to begin to understand the relationship between local interaction rules and the resulting global processes, and to use this understanding to engineer self-organizing systems.

To this end we have focused on two things: The mathematics of self-organization and the construction of systems that simultaneously validate and challenge the mathematics. These systems are interesting in their own right as idealized models of self-organization that may shed light on, for example, molecular self-organization. They may have practical import as well. For example, a group of robots that self-assembles into a solar array and maintains itself may be useful for planetary exploration. We are also investigating the possibility of implementing these ideas with MEMs in order to mass-produce 3D objects – currently difficult with standard fabrication techniques.

*This work is partially supported by NSF Grant #0347955.

Fig. 1. Four programmable parts partially assembled into a triangle. The parts bind upon random collisions and communicate via IR, deciding whether to remain bound or to detach. A graph grammar stored on the microcontroller of each part determines the ultimate global structure that will emerge. The parts are not self-motive but instead are "mixed" on an air table by overhead oscillating fans.

In this paper, we introduce an experimental self-organizing robotic system and show how we can use *graph grammars* [12] to direct its self-organization. Specifically, the system consists of a number of simple robotic *programmable parts* (Figures 1 and 2). As described in Section III, each part is capable of binding to other parts, communicating with the parts to which it is bound, and detaching from other parts. The parts rely on the environment for mobility: They float on an air-table and are mixed by fans. Thus, all interactions between parts begin with chance collisions.

The resulting dynamics are nondeterministic and concurrent: Many interactions may occur simultaneously and the order in which interactions occur is not determined. We, therefore, provide each part with a rule book (i.e. a *graph grammar*) that prescribes the outcome of each possible interaction. In our system, each part has a microcontroller that can store both a rule book and an internal state value. Each rule in the book is of the form $L \Rightarrow R$. If the internal states and local topology of a pair of interacting parts matches L, then the parts rewrite their states and local topology with R. These ideas are made

formal in Section IV.

It can be shown that a rule book can be designed so that the parts assemble into *any* desired structure[12]. Rules that produce limit cycles (i.e. processes such as sequences of global shape changes) can be generated as well. In this paper, we simply demonstrate the grammatical approach with the programmable parts by considering, in Section V, the problem of directing them to assemble into a hexagon (one of many possible global structures).

The reader may notice the resemblance of graph grammar rules to chemical reactions. In fact, there is a satisfying relationship between the two ideas. What is required to leverage the similarity is a model of the "reaction rate" of each rule based on geometry and the "concentration" of reactants (parts matching the left hand side of a rule). We discuss this relationship in Section VI. We also show how graph grammars that produce the same final assembly can differ in how long they take to form the assembly. Based on our observations, we argue that one may augment graph grammars with *kinetics* and determine the yields of various assemblies (we will report on this in more detail in a forthcoming paper).

The paper concludes with a discussion of the main questions raised so far by our initial experience with the testbed. The number and depth of the questions is a sign that the testbed is serving its purpose.

II. Previous and Related Work

The programmable parts described here are similar to *self-reconfiguring modular robots* built by many groups [20], [15], [17], [14], the main difference being that the present system consists of initially disconnected modules and the modules themselves may not change form. There is other work [19], [18] on building self-assembling robots similar to those described here. Our work may be distinguished from other work by the facts that our robots (1) have their own on-board power supplies and (2) are programmed with graph grammars.

The idea of the programmable part was inspired by surface-tension driven assembly of *passive* tiles [8], [2]. We were intrigued by the potential of having the parts in these systems actively "decide" whether to bind with others. An example of a system not quite passive and yet not fully actuated is Hosokawa, Shimoyama and Miura's "kinetics" experiment [7] consisting of triangular parts in a shaker. By careful placement of magnets in the parts, they were able to achieve interesting assemblies.

Similar to the graph grammar approach is idea of *conformational switching*, wherein particles undergo changes in shape upon assembly events. Saitou described this process symbolically with string assemblies [16]. Graph grammars were introduced [5], [3] more than two decades ago and have been used to describe a broad array of systems, from data structure maintenance to mechanical system synthesis. Graph grammars are, of course, a generalization of the standard "linear" grammars used in automata theory and linguistics and thus (incidentally), can perform arbitrary computation. The use of graph grammars to model distributed assembly, to the best of our knowledge, is new. We have used them to model assembly, ratcheting, self-replication and other processes [10], [11], [12].

Graph grammars as a means of programming modular robots is also new to this paper. Other software environments [21] deal with a lower level of abstraction (e.g. require more details about communication). The benefit of the grammatical approach is that the mathematics used to model self-organization is *equivalent* to the code used to program the parts.

III. The Testbed

A. The Programmable Part

A *programmable part* (Figures 1 and 2) consists of an equilateral triangular chassis that supports three controllable latching mechanisms, three IR transceivers, and a PIC18F242 based control circuit mounted on a custom made PC board. Each edge of the chassis is 12 cm long and the chassis is 1.2 cm high (although the motors add another 3 cm to the height of the part). The polyurethane chassis is cast in a Si-rubber mold, made from an original chassis printed using a 3D rapid prototyping tool. The motor mounts are constructed similarly.

Each latch consists of three NeFeB permanent magnets: one fixed and the other two mounted on the end of a small geared DC motor. The position of the movable magnet is determined using Hall Effect sensors and mechanical switches. The default position of the magnets is such that the north pole of the fixed magnet and the south pole of the movable magnet are pointing out (see Figure 2). When two latches from different parts come into contact, they temporarily bind – the fixed magnet of one attaching to the movable magnet of the other. At that point, a contact switch on each part is pressed and the parts may communicate. If they mutually decide to remain bound to each other, they do nothing. If at any point they mutually decide to detach from each other, each temporarily rotates its movable magnet 180°, forcing the parts apart. The movable magnets then return to their default positions. Because they are aggressively mixed, it is unusual for parts to immediately re-bind. The latch only requires power when switching, and does not require power to stay closed. Each part is powered by a 200 mAh Li polymer battery, which remains charged for approximately 2 hours in situations where motors are not rotated more than once every 10 seconds.

The 3.6 MHz PIC microcontroller on each part coordinates IR communications between neighboring parts and controls the motors during detach events. The decisions to detach or not are based on a *graph grammar* (Section IV) stored in memory. The requirements of a graph grammar are that an internal state or *label* is associated with each latch, that labels are updated according to rules when the combined labels of two neighboring parts match a rule, and that the parts are able to bind and detach according to the grammar rules. Thus, the program on the PICs is quite simple: It is essentially a "graph grammar interpreter".

B. Infrastructure

The parts float on a custom-made 2 m^2 air table. To maximize useful collisions, we use various methods for stirring

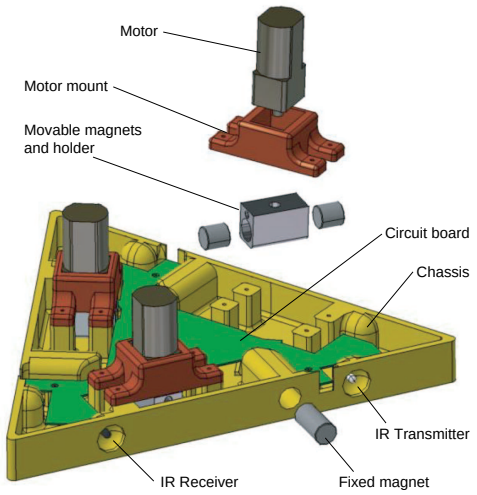


Fig. 2. The components of the programmable part include low power magnetic latches, infrared communications, and an on-board microcontroller.

the parts on the air table. For the experiments in this paper, oscillating fans were placed at the corners of the table.

A camera is mounted approximately 3 m above the surface of the table and is used to record the motion of the parts. We adorn the parts with triangular "hats" that render them easy to parse by the vision system (e.g. Figure 4). The positions and orientations of the parts are processed by a workstation at 30 Hz. Because each part looks identical to the vision system, we assign to them arbitrary IDs in software and track the parts to associate trajectories with the IDs. The trajectories are then exported to MATLAB for further analysis (e.g. Figure 5). In the future we expect to use the output of the vision system in real time to adjust the mixing strategy, thereby speeding up the dynamics.

C. Simulation Environment

At the time this paper was written, we had built six programmable parts. Because of the simplicity in their design, the materials for each cost less than $100 total. We plan to eventually build approximately 100 parts, which will allow us to do more elaborate experiments than the one reported here. In the meantime, we have built a realistic simulation environment that allows us to "experiment" with a greater number of parts. The simulation uses the *Open Dynamics Engine* [1] library, which can routinely compute trajectories of hundreds of parts and determine the result of collisions and contact situations quickly. The main simplifications in the simulation are that the magnets are modeled as pairs of point attractors to avoid explicitly modeling their magnetic fields; the effect of the fans is modeled as a time varying force field acting on only the centers of mass of the parts; and communication is not explicitly modeled.

IV. Graph Grammars for Self-Organization

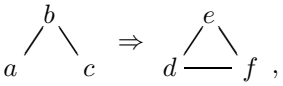
We give a brief overview of the graph grammar approach to directing the topological component of self-organization, and focus on how it applies to programmable parts. An in-depth treatment of our approach to graph grammars, a broad array of examples, and the fundamental properties can be found in our other papers on the subject [10], [11], [12].

A. Definitions

A *simple labeled graph* over an alphabet $\Sigma = \{a, b, c, ...\}$ is a triple $G = (V, E, l)$ where V is a set of *vertices*, E is a set of unordered pairs or *edges* from V, and $l : V \to \Sigma$ is a labeling function. We restrict our discussion to simple labeled graphs and thus simply use the term *graph*. We denote an edge $\{x, y\} \in E$ by xy. We denote by V_G, E_G and l_G the vertex set, edge set and labeling function of the graph G.

Definition 4.1: A *rule* is a pair of graphs $r = (L, R)$ where $V_L = V_R$. The graphs L and R are called the *left hand side* and *right hand side* of r respectively.

If (L, R) is a rule, we usually denote it by $L \Rightarrow R$. We also represent rules graphically as in



where the relative locations of the vertices represent their identities. In the above rule, for example, $V = \{1, 2, 3\}$ and vertex 1 is labeled by $l_L(1) = a$ in the left hand size and by $l_R(1) = d$ in the right hand side.

A *monomorphism* between graphs G_1 and G_2 is an injective function $h : V_{G_1} \to V_{G_2}$ that preserves edges: $xy \in E_{G_1} \Leftrightarrow h(x)h(y) \in E_{G_2}$. It is *label-preserving* if $l_{G_1} = l_{G_2} \circ h$.

Definition 4.2: A rule $r = (L, R)$ is *applicable* to a graph G if there exists a label-preserving monomorphism $h : V_L \to V_G$. An *action* on a graph G is a pair (r, h) such that r is applicable to G with witness h.

Definition 4.3: Given a graph $G = (V, E, l)$ and an action (r, h) on G with $r = (L, R)$, the *application* of (r, h) to G yields a new graph $G' = (V, E', l')$ defined by

$$E' = (E - \{h(x)h(y) | xy \in L\}) \cup \{h(x)h(y) | xy \in R\}$$

$$l'(x) = \begin{cases} l(x) \text{ if } x \notin h(V_L) \\ l_R \circ h^{-1}(x) \text{ otherwise.} \end{cases}$$

We write $G \xrightarrow{r,h} G'$ to denote that G' was obtained from G by the application of (r, h).

Definition 4.4: A *system* is a pair (G_0, Φ) where G_0 is the *initial graph* of the system and Φ is a set of rules (called the *rule set* or *grammar*).

Definition 4.5: A *trajectory* of a system (G_0, Φ) is a (finite or infinite) sequence

$$G_0 \xrightarrow{r_1,h_1} G_1 \xrightarrow{r_2,h_2} G_2 \xrightarrow{r_3,h_3} ...$$

If the sequence is finite, then we require that there is no rule in Φ applicable to the terminal graph. The set of all graphs reachable from G_0 via some finite trajectory is called the *reachable set* and is denoted $\mathcal{R}(G_0, \Phi)$. If C is a *component* of a reachable graph, it is called a *reachable component*. If no rules in Φ can alter a reachable component, the component is said to be *stable*. The set of reachable components is denoted $\mathcal{C}(G_0, \Phi)$ and the set of stable components is denoted $\mathcal{S}(G_0, \Phi)$.

B. Properties of Graph Grammars

There are many properties and algorithms associated with graph grammars. It is beyond the scope of this paper to describe them formally here. Therefore, we describe informally the highlights and limitations of the theory.

Examples: As a simple example, one can create a graph grammar that, starting with a disconnected soup of parts each initially labeled a, forms unstable chains of arbitrary length and stable cycles of length three or more:

$$\Phi_1 = \begin{cases} a & a & \Rightarrow & b-b \\ a & b & \Rightarrow & b-c \\ b & b & \Rightarrow & c-c. \end{cases}$$

One can also define rules sets that: make any given graph uniquely stable; make certain families of graphs stable (such as all k-connected graphs); make processes such as ratchets and walkers; cause a seed graph to self-replicate in a soup of "raw material" parts [11]; sort themselves according to given ordering function; etc.

Concurrency: Two actions (r_1, h_1) and (r_2, h_2) with $r_1 = (L_1, R_1)$ and $r_2 = (L_2, R_2)$ are said to *commute* if $h_1(L_1)$ and $h_2(L_2)$ are disjoint. A trajectory in which a set A of pairwise disjoint actions are executed sequentially can be rewritten as a *concurrent action* $G_i \xrightarrow{A} G_j$, since the order in which the actions in A are applied does not matter. Those rule sets that result in many rules being applied concurrently will result in faster self-assembly.

Topology and Communication: Small rules (involving few vertices) model local communication. We are usually concerned with those that involve only two parts. Distributed algorithms involving multiple hops, such as group communication and simple consensus protocols, can in fact be written as grammars whose rules involve only two parts. Nevertheless, there are limitations to local rules [13]. For example, due to fundamental topological properties, it is impossible to find a grammar with size-two rules that uniquely stabilizes a given graph. It is possible with size-three rules, however [12].

Synthesis Algorithms: The *synthesis problem* for graph grammars is: Given a specification of a desired global behavior and an initial graph G_0, generate a grammar Φ so that all trajectories of (G_0, Φ) meet the specification. We have devised algorithms that automatically generate grammars for the problem of making a desired graph stable (the *self assembly synthesis problem*) [12] and also algorithms that generate rules for processes (such as ratcheting).

Deadlock Avoidance: Ninety parts trying to form into copies of an assembly with twenty parts could get stuck in six non-final sub-assemblies of fifteen parts each, at which point no rule in Φ would apply. One way to avoid such situations is for the parts to randomly disassociate from sub-assemblies to which they have been bound for "too long". Under certain assumptions, it can be shown that this approach avoids deadlock [9].

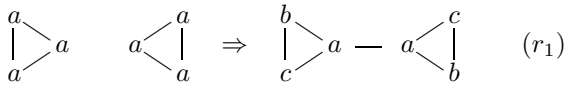
Limitations of Graph Grammars: There are at least two limitations to the approach. First, it models only changes in *topology*[1] – which parts are bound to which other parts. Clearly, the geometry of triangular parts prevents certain topologies from being formed. Presently, we have found that, in practice, designing simple grammars that respect geometrical constraints is not difficult. However, the formal relationship between the graph grammars and their possible embeddings into various geometrical settings is not presently understood; many open mathematical questions in this area remain to be explored.

Second, rules are applied non-deterministically (as opposed to probabilistically, for example) and without an explicit notion of time. Thus, for example, a graph may be reachable in the abstract sense, but in a realistic setting its appearance in a trajectory may be extremely unlikely. One may augment the notion of state with a continuous component describing, for example, the positions and velocities of all the robots. This leads to a *hybrid system* and the various accompanying difficulties. Alternatively, one may make various assumptions about the probability that a given action will apply and how long it is likely to take. This leads to a *kinetics* based interpretation of graph grammars, which we describe in slightly more detail in Section VI.

C. Rules for Triangular Programmable Parts

We associate a vertex with each of the latches on a given programmable part and connect them with edges (permanently) to indicate that latches are physically attached to each other. Thus, we have embedded the purely topological object C_3 (a cycle of three vertices) into $\mathbb{R}^2$ as indicated in Figure 3. The *state* stored by the microcontroller is a triple of labels (a, b, c), one for each latch. We do not distinguish between latches, so that two states are equivalent if they vary only by rotation. For example, $(a, b, c) = (b, c, a)$. In practice, always require that the embedding of C_3 into $\mathbb{R}^2$ is *counterclockwise*. Since the parts are confined to motion in the plane, this orientation will not change as the parts assemble.

When two parts interact, they compare their triples against the stored grammar to determine the result of the interaction. As an example, consider the system defined by the single rule:



(r_1)

Suppose that a set of parts is initialized so that each has state (a, a, a) and that their interactions are governed by the above rule only. When two parts labeled (a, a, a) collide, their states match the left hand side of the above rule, so they remain attached and change their states to (a, b, c), each associating the latch involved in the connection with the label a and the other latches with the labels b and c, going counterclockwise from the active latch. The result is a "dimer" assembly consisting of two triangles as in Figure 3(a). One can easily show that dimer assemblies are stable components and that "monomer" assemblies are not.

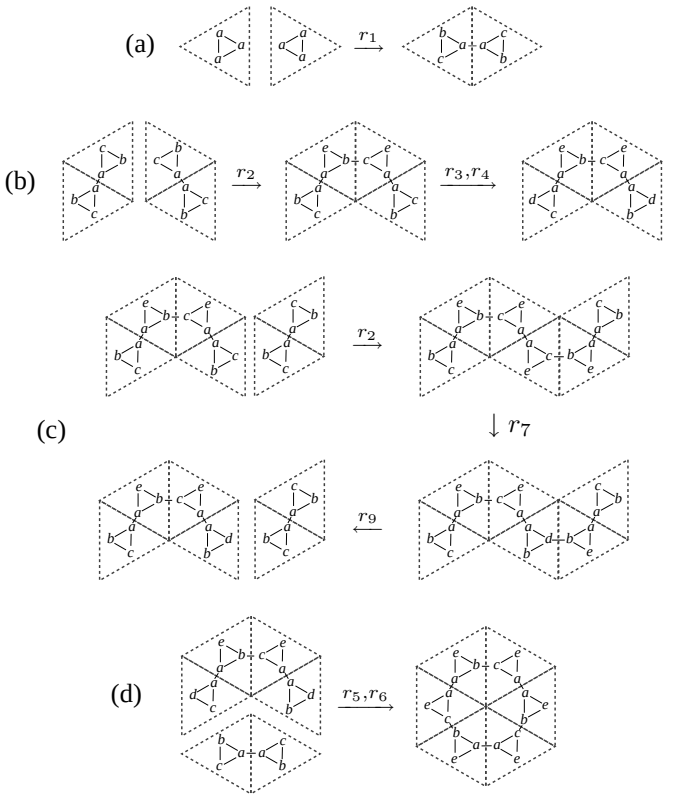


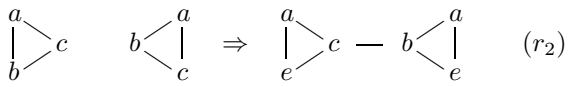
Fig. 3. The steps in the self-assembly of a hexagon using the rules described in Section V. Note that the geometry of the embedding represented here is for convenience. Graph grammars are purely topological, describing only the way the network topology of the system changes.

V. Hexagon Formation

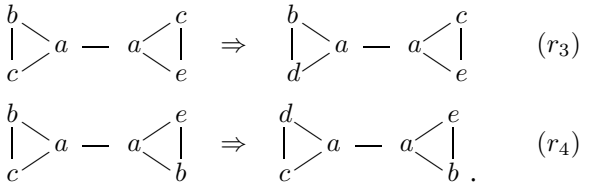
In this section, we describe in detail a grammar for self-assembling hexagons and show the results of using the grammar with the programmable parts. We also demonstrate that the choice of grammar is crucial by comparing the average assembly time of three different grammars whose stable components are hexagons.

A. A Graph Grammar for Hexagons

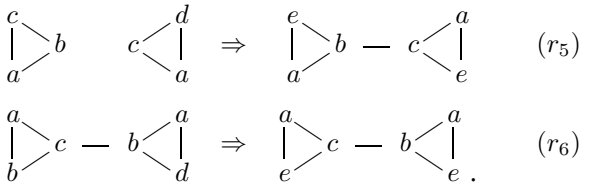
Suppose the programmable parts start in the state (a, a, a). The grammar consists of two subsets: $\Phi = \Phi_c \cup \Phi_u$. The first set Φ_c corresponds to *controllable rules*, that is, rules initiated by the parts. The second set, Φ_u corresponds to rules that are initiated by the environment due to, for example, a high velocity collision breaking apart two bound parts. We first describe Φ_c. To assemble a hexagon, we first form dimers by including the rule (r_1) from the previous section. Next, we include three rules for joining dimers:



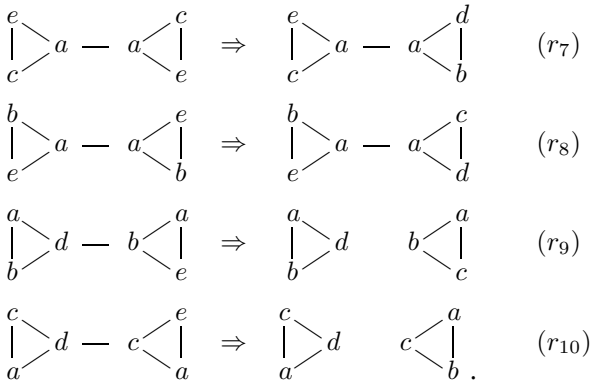
[1]Specifically, a graph is considered to be a 1D simplicial complex.



Rule (r_2) is used when parts from two different dimers collide as in Figure 3(b). It results in a 4-mer. However, since only two parts are involved in the binding event, the fact that the rule was applied needs to be propagated to the ends of the 4-mer. This is accomplished by rules (r_3) and (r_4). A dimer attaching to a 4-mer completes completes the hexagon. This is done with two more rules, as shown in Figure 3(d):



Another possibility must accounted for, namely, that a dimer attaches to a 4-mer using rule (r_2) before rules (r_3) and (r_4) propagate through the 4-mer. We have the following rules:



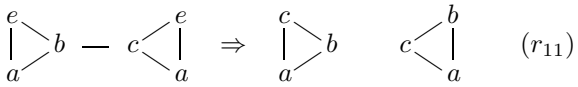
Either (r_7) and (r_9) are used to detach the errant dimer or (r_8) and (r_{10}) are used, depending on where the dimer attached to the 4-mer. See Figure 3(c).

The rules in Φ_c are sufficient to build hexagons in most situations. In particular, if G_0 consists of copies of C_3 all labeled by a, then $\mathcal{S}(G_0, \Phi_c)$ contains exactly the hexagon shown in Figure 3(d)(right).

Several problems arise however. First, it is possible that a 4-mer and a dimer continually bind and detach, preventing other actions from occurring (i.e. *there is a potential for livelock*). However, because the mixing is essentially random (see Section VI), this is unlikely. Second, it is possible, with 12 parts for example, for three 4-mers to form at which point no rules apply (i.e. there is a potential for deadlock). We do not address this problem here, although it can be handled in a straightforward manner [9]. Finally, it is possible for a subassembly to break apart[2] due to a collision of sufficiently high energy.

[2]In fact, it is a statistical certainty that assemblies will eventually break apart. This inadvertently solves the deadlock problem by breaking apart "stale" subassemblies so that their parts can be incorporated into others.

To address this last case, we define the set Φ_u to deal with uncontrollable events such as these. The set we use in our experiments contains seven break rules and four propagation rules, which (for lack of space) we do not list here. For example, the following rule corresponds to a 4-mer breaking in two:



(r_{11})

The propagation rules (not shown) are used to relay the fact that the subassembly has broken to the other parts in it. They relabel the parts and may even cause further breaks as when, for example, a single part breaks off of a 4-mer resulting in a 3-mer that forces itself to break again into a singleton and a dimer so that assembly the rules in Φ_c apply.

Note that combining the rules in Φ_u to Φ_c *destabilizes* the hexagon. However, the rules in Φ_u are less likely to occur in our setting, and thus we expect the hexagon to be more stable than other assemblies. We hope to address this with a rigorous probabilistic argument in a future paper.

B. Experiments

We programmed six programmable parts with the grammar $\Phi = \Phi_c \cup \Phi_u$ and initialized them each to the state (a, a, a). Then we placed the parts on the air table and mixed them with four oscillating fans, as described in Section III-B. Figure 4 shows several frames taken from video of an example run. After approximately one minute (which was typical), the hexagon was complete.

Another way to visualize progress of the system toward the complete assembly of a hexagon is by plotting the distance of each part to the center of mass of the parts. Since, in this experiment, there are only six parts, this distance will reach a minimum when all the parts are assembled. We show this data, recorded using the overhead camera, in Figure 5 for the same experiment shown in Figure 4.

C. Simulation

In simulation we can experiment with more parts. We are particularly interested in evaluating different grammars with the same stable components. For example, the grammar we describe above is only one of many grammars that form hexagons. Another way to form hexagons is to add parts one by one and grow the hexagon linearly. Still another way is to first form 3-mers and then hexagons from them.

We tried each of these grammars in simulations using 50 parts. For each grammar, we simulated the system 10 times from random initial conditions for 20 minutes and recorded the *yield*, which is defined to be the number of hexagons formed divided by the total number of hexagons possible. In Figure 6 we show the average yield for ten runs of each grammar starting from random initial conditions. The dimer grammar (Figure 6(c)) described above reached 70% yield on average. The 3-mer grammar (Figure 6(b)) reached less than 30% yield on average. The one-by-one grammar (Figure 6(a)) often formed no hexagons after 20 minutes. In the next section, we discuss why these grammars might perform so differently.

VI. Macroscale Kinetics

Following in the footsteps of Hosokawa et. al [7], we discuss briefly the applicability of the standard chemical kinetics model to our system by showing that the parts essentially undergo diffusive motion and that one can in principle make reasonable estimates about the *reaction rates* between subassemblies (that is, the rate at which rules in the grammar apply). We plan to report on the formal aspects of combining graph grammars with reaction-diffusion dynamics in a future paper. In particular, we intend only to motivate the potential usefulness of the kinetics model here.

A. Evidence of "Molecular" Motion

A graph grammar is a *non-deterministic* model, meaning that it describes all of the trajectories that can occur. However, the system we describe in this paper has considerably more structure: Some trajectories are much more likely to occur than others. In principle, it may be possible to determine the exact motions of the parts and predict the resulting trajectories, but this seems impractical. Instead, we would like to assume that the system is undergoing *reaction-diffusion* dynamics. Our data suggest that this is a good assumption.

Particles that diffuse make random walks with Maxwellian velocity distributions. Figure 7(a) shows the distribution of the magnitude of the translational velocities of 50 parts in a 20 minute simulation. In the simulation, the motions of the mixing fans were simple oscillations. The distribution is approximately Maxwellian.

Particles undergoing Brownian motion (essential to diffusion) "forget" their velocities quickly [4]. In Figure 7(b) we show the velocity autocorrelation for a particular programmable part among six parts in a hardware experiment using $\Phi = \varnothing$. The velocity of the part is essentially uncorrelated with its initial velocity after approximately 2.5s. The data strongly support the assumption that the parts diffuse across the air table.

B. The Binding Mechanism and Reactivity

One can determine the "reaction probability" of binding events with a simple experiment. We programmed six parts with $\Phi = \varnothing$ (so that they immediately detach upon binding) and allowed the system to run. Using the vision system, we recorded the part trajectories and recorded the number of collisions between any two parts and, for each collision, whether it resulted in a binding event. In this experiment, $154/211 = 73\%$ of the collisions resulted in binding events.

Figure 7(c) shows the data with circles for successful bindings and dots for unsuccessful bindings. The horizontal axis is the sum of the kinetic energies of the parts involved in the collision. The vertical axis is the distance between the centers of the parts involved in the collision, which is lowest when the parts collide with two faces parallel, and highest when the parts collide vertex on vertex. The data shows that most collisions occur somewhere in between the two extremes and that there is a slightly higher chance that low energy collisions will result in bindings. Combined with the diffusion rate, the probability of a binding given a collision gives us

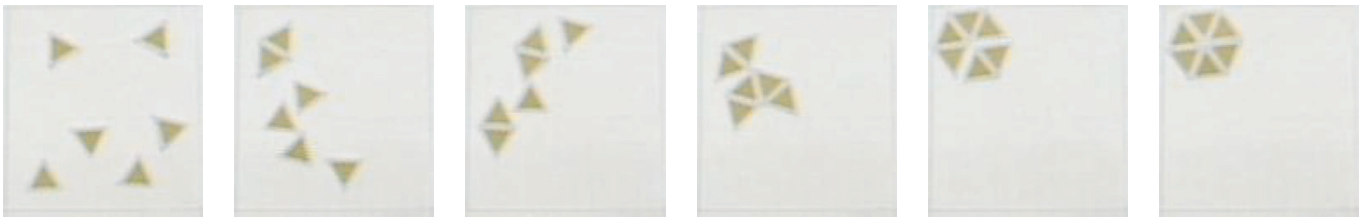


Fig. 4. A series of false color frames from video data showing a collection programmable parts forming into a hexagon. The data in Figure 5 is from the same run. Note: The airflow through the table is slightly uneven and parts tend to move toward the upper left corner.

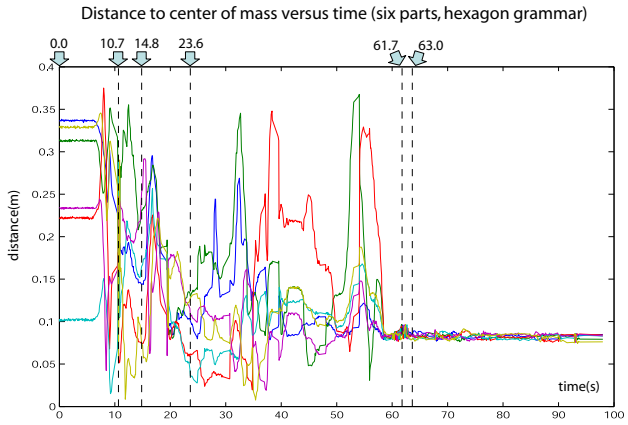


Fig. 5. The distance from each part to the center of mass of all parts plotted as a function of time during hexagon formation. When the distances are the same, the parts have formed a hexagon assembly.

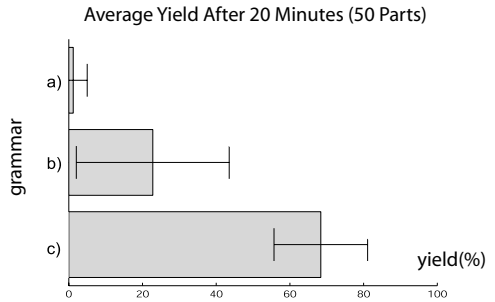


Fig. 6. Each of three grammars (a)-(c) that produce hexagons were used in ten simulations of 50 parts. The hexagon yield after 20 minutes is shown for each grammar.

the *stochastic reaction rate* [6] between two programmable parts, which is the probability that two given parts will react at any given time. Interestingly, one could obtain these values theoretically based on the geometry and dynamics of the parts. This would be similar to determining chemical reaction rates from Schrödinger's equation, albeit a significantly lower dimensional problem. We plan to report on our effort in this direction in a future paper.

C. Rule Rates

If we can determine the stochastic reaction rate between two parts, we can determine the rate at which two larger assemblies react via a given graph grammar rule by the same process as described above. A simpler approximation (that is independent of geometry and, therefore, quite approximate) is to define the *rule rate* for an given graph. Formally, suppose $G, H \in \mathcal{R}(G_0, \Phi)$ are reachable graphs. Define the rate $k_{G,H}(r)$ of a rule $r \in \Phi$ to be the number of distinct ways it can be applied to G to produce H:

$$k_{G,H}(r) \triangleq |\{(r,h) \mid G \xrightarrow{r,h} H\}|.$$

We may now build a Markov chain whose states are integer valued vectors v over the reachable components $\mathcal{C}(G_0, \Phi)$. In particular, v corresponds to the set of all reachable graphs G for which the number of occurrences (up to isomorphism) of component C in G is $v(C)$. We call this set $[\![v]\!]$. For two vectors u and v, we define the rate from u to v by

$$k_{u,v} \triangleq \sum_{G \in [\![u]\!]} \sum_{H \in [\![v]\!]} \sum_{r \in \Phi} k_{G,H}(r).$$

Note that it may also be possible to combine $k_{u,v}$ with the bindings-per-collision data to produce an adjusted rate that accounts for geometry.

As an example, consider the hexagon forming grammar described above. Suppose u_1 corresponds to a state with one 4-mer and one dimer. The dimer can react via (r_5) with a given 4-mer in two different ways and via (r_6) in two different ways. If v_1 corresponds to a state with one hexagon, then $k_{u_1,v_1} = 2 \cdot 2 = 4$.

In contrast, consider the grammar (which the reader can readily construct) that first forms 3-mers and then hexagons. Suppose u_2 is the state with two 3-mers and v_1 is the state with one hexagon. Then $k_{u_2,v_2} = 2$. This, together with the geometrical fact that two 3-mers have to be very well aligned to react, suggests that it may take quite some time for the system to escape the state u_2, whereas the state u_1 (with respect to the previous grammar), may be easier to escape.

VII. Discussion

We have built a testbed consisting of *programmable parts* that self-organize in a predictable fashion according to the mathematics of graph grammars. We demonstrated this by programming the parts with a graph grammar whose unique stable component is a hexagonal arrangement. Only six parts were used in the experiment, however. We intend to build approximately 100 parts so that we can investigate even more compelling behaviors. We believe that this will be possible based on the simplicity and low cost of the programmable part design.

We also showed, through experiment and simulation, that the parts diffuse and react in a way qualitatively similar to the behavior of chemical systems. Formally extending the theory

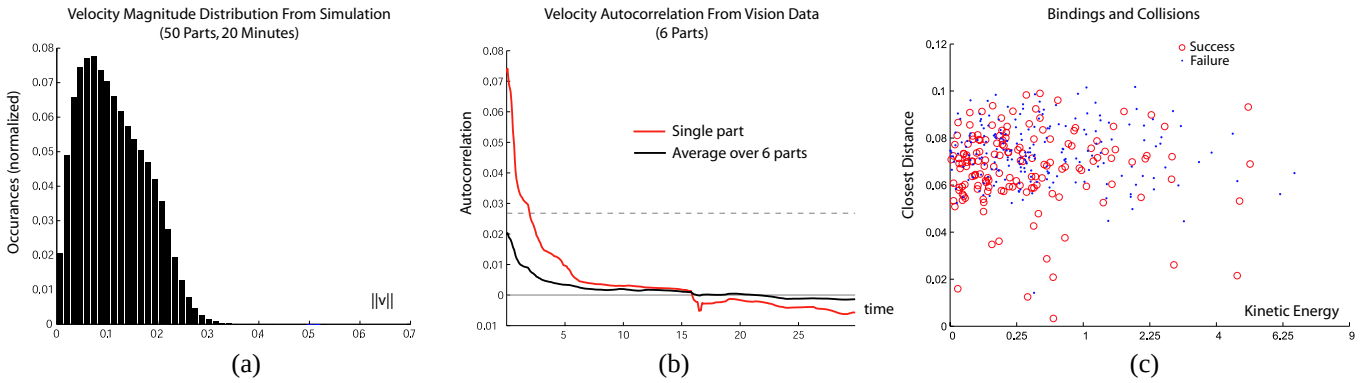


Fig. 7. (a) The distribution of the square of the part velocities from simulation. It is similar to a squared Gaussian distribution. (b) The velocity autocorrelation for one part in an experiment with 6 parts and $\Phi = \varnothing$ taken from vision data. Also shown is the average autocorrelation for all six parts. The horizontal line is at $R(0)e^{-1}$ and shows the *correlation time*. (c) Experimental data from the vision system with 6 programmable parts using $\Phi = \varnothing$ from Section IV-C. Each data point represents a collision between two tiles. If the collision resulted in a binding, a circle appears, otherwise a dot appears.

of graph grammars to accommodate these ideas is a main area of our current research.

There are many other ideas we plan to pursue that build on the results presented here. We list some of them here. We plan to investigate other grammars that assemble other shapes and that define *processes* such as locomotion, self-repair and transport. We hope to determine the binding probability for two parts and for two subassemblies from first principles and from that predict the rates at which grammar rules are applied in our system. We plan to develop a method for analytically evaluating an entire grammar using rule rates and geometry to predict its yield.

Unlike with chemistry, however, with programmable parts we will be able to engineer global behaviors quite exactly. Of course, our programmable parts are not nearly as useful as molecules. However, we believe that through building and studying self-organizing systems, we may eventually be able to apply the priciples we discover to engineering self-organization at all scales.

Acknowledgments: E. Klavins and N. Napp are partially supported by NSF Grant #0347955: “CAREER: Programmed Robotic Self Assembly”. We would like to thank R. Gordeon, A. Horike and C. Matlack for their contributions to the project and S. Johnston and M. Ganter for allowing us to use their 3D rapid prototyping facility. Many of the ideas in this paper were discussed at great length with R. Ghrist at UIUC.

References

[1] ODE: The open dynamics engine. http://www.ode.org/.

[2] N. Bowden, A. Terfort, J. Carbeck, and G. M. Whitesides. Self-assembly of mesoscale objects into ordered two-dimensional arrays. *Science*, 276(11):233–235, April 1997.

[3] B. Courcelle. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, chapter on Graph Rewriting: An Algebraic and Logic Approach, pages 193–242. MIT Press, 1990.

[4] K. Dill and S. Bromberg. *Molecular Driving Forces*. Garland Science, 2003.

[5] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69, 1979.

[6] D. T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *Journal of Physical Chemistry*, 81:2340–2361, 1977.

[7] K. Hosokawa, I. Shimoyama, and H. Miura. Dynamics of self-assembling systems: Analogy with chemical kinetics. *Artifi cal Life*, 1(4):413–427, 1994.

[8] K. Hosokawa, I. Shimoyama, and H. Miura. Two-dimensional micro-self-assembly using the surface tension of water. *Sensors and Actuators A*, 57(2):117–125, November 1996.

[9] E. Klavins. Automatic synthesis of controllers for distributed assembly and formation forming. In *Proceedings of the IEEE Conference on Robotics and Automation*, Washington DC, May 2002.

[10] Eric Klavins. Directed self-assembly using graph grammars. In *Foundations of Nanoscience: Self Assembled Architectures and Devices*, Snowbird, UT, 2004.

[11] Eric Klavins. Universal self-replication using graph grammars. In *The 2004 International Conference on MEMs, NANO and Smart Systems*, Banff, Canada, 2004.

[12] Eric Klavins, Robert Ghrist, and David Lipsky. A grammatical approach to self-organizing robotic systems. Sumbitted to the *IEEE Transactions on Automatic Control*, 2005.

[13] I. Litovsky, Y. Métevier, and W. Zielonka. The power and limitations of local computations on graphs and networks. In *Graph-theoretic Concepts in Computer Science*, volume 657 of *Lecture Notes in Computer Science*, pages 333–345. Springer-Verlag, 1992.

[14] S. Murata, H. Kurokawa, and S. Kokaji. Self-assembling machine. In *International Conference on Robotics and Automation*, pages 441–448, 1994.

[15] D. Rus and M. Vona. Crystalline robots: Self-reconfi guration with compressible unit module. *Autonomous Robots*, 10(1):107–124, January 2001.

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

[17] W.-M. Shen, P. Will, and B. Khoshnevis. Self-assembly in space via self-reconfi gurable robots. In *International Conference on Robotics and Automation*, Taiwan, 2003.

[18] P. White, V. Zykov, J. Bongard, and H. Lipson. Three dimensional stochastic reconfi guration of modular robots. In *Proceedings of Robotics Science and Systems*, Boston, MA, June 2005.

[19] P. J. White, K. Kopanski, and H. Lipson. Stochastic self-reconfi gurable cellular robotics. In *Proceedings of the International Conference on Robotics and Automation*, New Orleans, LA, 2004.

[20] M. Yim. A reconfi gurable modular robot with many modes of locomotion. In *JSME International Conference on Advanced Mechatronics*, Tokyo, Japan, 1993.

[21] Y. Zhang, K. Roufas, and M. Yim. Software architecture for modular self-reconfi guable robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Hawaii, October 2001.