A Grammatical Approach to Self-Organizing Robotic Systems

Eric Klavins* Robert Ghrist† David Lipsky†

*Electrical Engineering
University of Washington
Seattle, WA 98195

†Mathematics
University of Illinois
Urbana, IL 61801

Abstract

In this paper we define a class of graph grammars that can be used to model and direct parallel robotic self-assembly and similar self-organizing processes. We give several detailed examples of the formalism and then focus on the problem of synthesizing a grammar so that it generates a given, prespecified assembly. In particular, to generate an acyclic graph we synthesize a binary grammar (rules involve at most two parts), and for a general graph we synthesize a ternary grammar (rules involve at most three parts). In both cases, we characterize the number of concurrent steps required to achieve the assembly. We also show a general result that implies that no binary grammar can generate a unique stable assembly. We conclude the paper with a discussion of how graph grammars can be used to direct the synthesis of robotic parts floating in a fluid or for self-motive robotic parts.

1 Introduction

Engineering processes in the realm of the very small presents us with the daunting problem of manipulating and coordinating vast numbers of objects so that they perform some global task. Because of the potentially enormous quantities of objects involved, uniquely addressing and manipulating each one is impossible. There are of course examples of sophisticated machines, such as the ribosome or the mechanical motor in the bacterial flagellum, that seem to be built *in bulk* spontaneously our of large numbers of simple interacting components. One hypothesis for how this occurs is that the simple small components *self-assemble* into more complex aggregates which, in turn, self assemble into larger aggregates or function as simple machines. The goal of the present work is to understand how the interactions between the components of a system can give rise to coordinated behavior in a self-organizing (i.e. decentralized and unsupervised) way. In particular, we suppose that the components themselves are (very) simple robots, whose general capabilities we will specify in the body of this work.

Our starting point is the idea of *conformational switching* [29]: Each component (robot, molecule, particle, etc.) exists in one of several *conformations* or shapes. When two components come into close proximity, they attach or not based on whether their conformations are complimentary. If they do attach, their conformations change (mechanically for example), thereby determining in what future interactions the components may partake.

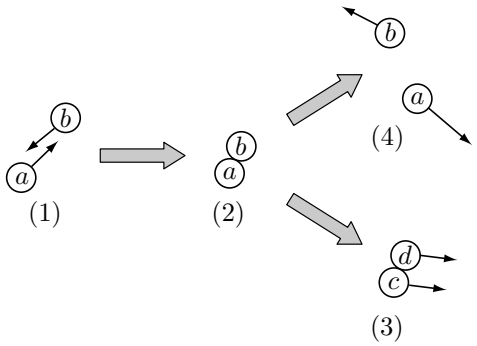


Figure 1: The particle interaction model considered in this paper. (1) Two part with labels a and b (2) occasionally collide. If their labels correspond to a rule of the system (Definition 3.4), then (3) the parts stick (Definitions 3.2 and 3.3) and their labels change (to c and d in this example) according to the rules. Otherwise, (4) the particles simply bounce off of each other. In their new states, the parts may stick to yet other parts (or release other parts to which they were previous attached) so that aggregates form.

As in other work [28, 15], we consider the conformation of a part or *robot* as corresponding to a discrete symbol, and we model an assembly as a simple graph labeled by such symbols. Vertices in these graphs represent robots, and the presence of an edge between two robots represents the fact they are attached. A *rule* is a pair of labeled graphs and a grammar is a set of such rules interpreted as follows. If a subset of robots together with their labels and edges matches the first part some rule, then the subset may be replaced by the second part of the rule to achieve a new state of the system. That is, we use the rule to *rewrite* a part of the current graph. Continuing to apply rules one may produce a class of trajectories whose properties one might want to guarantee.

We are mainly in interested in the situation where the parts decide in a distributed fashion whether and how to execute an assembly rule. In this sense, a graph grammar defines a *distributed algorithm*, but not one that works on a fixed-topology network. Instead, the fact that the processors are located on robotic parts means that the network topology changes as the robots move through their environment.

A main motivation for the theory presented in this paper is a robotic testbed under construction at the University of Washington wherein programmable parts execute local rules to form global assemblies. Figure 2 is a photograph of a prototype trangular "programmable part." On each edge of the part is a magnetic latching mechanism and an infrared (IR) transceiver. The battery-powered part is controlled by an onboard micro-controller programmed according to rules of the sort presented in this paper. Many such parts are floated on an air table as in Figure 3(a) and experience essentially random motions. When two parts collide, their default behavior is to attach to each other. Only then may they communicate over their IR link. By sharing their internal states, the parts may decide whether to remain attached (when an assembly rule matches their states) or to dis-attach (when no assembly rule matches their states). Using the theory presented in this paper, this basic interaction protocol can be used to form complex assemblies of parts, such as those shown in Figure 3(b) (see also Example 4.3). The point of the testbed is to demonstrate the feasibility of the approach described in this paper. With relatively simple devices and a formal algorithmic and synthesis approach, we are able to

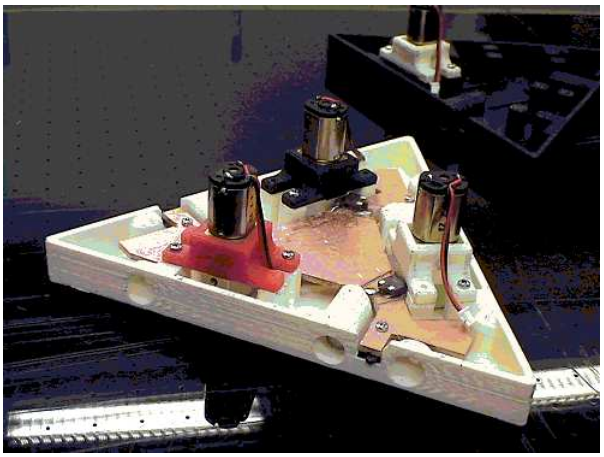
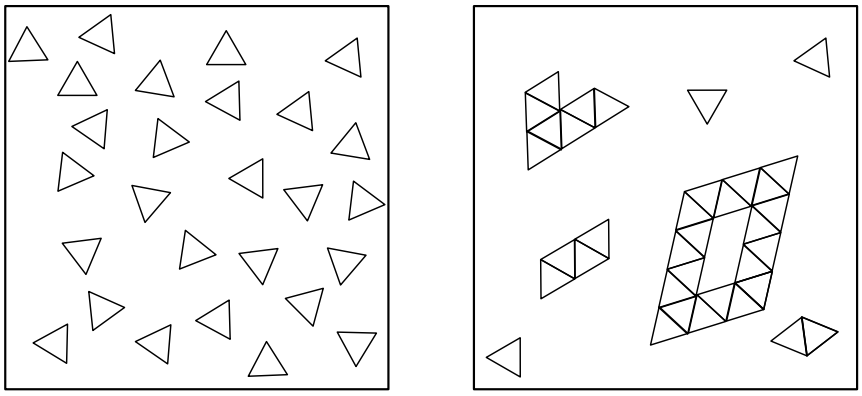


Figure 2: A robotic "programmable part". Each edge of the trangle may attach via a magnetic latch to another part. Once attached, the parts may decide to remain attached or may dis-attach by rotating a permanent magnet, which disengages the latch.

control this system to self-assemble into a variety of forms. We discuss breifly how this is done with graph grammars in Example 4.3. We plan to report on the details of the latching meachnism, stirring algorithms, rule sets specific to the programmable parts and other experimental results in a separate paper.

The main problem we consider in this paper is the *synthesis* problem, defined roughly as follows. Given a specification S (i.e. a logical statement about trajectories) and an initial configuration of robots G_0, define a grammar Φ so that all trajectories arising from G_0 via the application of rules in Φ meet the specification S. The specification may require that some cyclic process occur among the robots (as for example in the locomotion of *Metamorphic Robots* [6, 20, 23]) or that some undesirable set of configurations be avoided (for example, certain network configurations may lead to unstable dynamics in a control system [11]). In particular, we address the following synthesis problem: Given an arbitrary graph H define Φ so that only copies of H emerge as stable components of the system. For example, one might desire that the robots all form rings of 10 robots, or arrange themselves into a long line. This problem is stated more concisely at the end of Section 3.

Specifically, the contributions of this paper are as follows. In Section 3 we introduce a class of graph grammars suitable for describing distributed assembly. We give a set of detailed examples that elucidate the definitions and demonstrate the capabilities of the formalism in Section 4. We next examine some basic properties and limitations of graph grammars. In particular, in Section 5 we prove a theorem about the impossibility of assembling a unique stable graph using rules consisting of acyclic graphs (to be defined). We also explain how to view graph grammars as describing a concurrent process in Section 6 and associate to each trajectory of a system a canonical trajectory that captures the number of *concurrent* steps required to assemble a given component. Then in Section 7 we describe two algorithms that take a graph G and produce a grammar Φ such that the only *stable* assemblies of Φ are isomorphic to G. In first algorithm requires that G is acyclic, and the second algorithm (which uses the first) works for any given graph. We show the algorithms are correct and also explain how quickly the desired graphs can be assembled. In Section 8 we discuss how assembly rules can be implemented in a distributed fashion by simple communication protocols and finally, in Section 9, we introduce



(b) (c)

Figure 3: (a) An initial configuration of programmable parts on an air table. The parts are stirred by air jets (not shown) so as to produce collisions. (b) An assembly of programmable parts after the assembly process has completed. A complete final assembly and several partial assemblis are shown.

a physical model — based on programmable parts floating passively in fluids — that can be used to implement a graph grammar.

2 Previous and Related Work

Conformational switching was first described as a symbolic process for self assembly by Saitou [28], who considered the assembly of strings in one dimension. Berger et al. showed that a conformational switching model could mimic the assembly of virus capsids in simulation [2]. Self assembly as a graph process has been described by Klavins [15], although the graph grammar formalism is new to this paper, and a rule synthesis procedure for trees was given that is somewhat more complex than the one described in Section 7. A method for using potential fields and a certain deadlock avoidance scheme to implement local rules with a group of mobile robots was also described [15]. The proof of Theorem 5.1 is topological, utilizing tools from covering space theory (see e.g., [13]). The notion of concurrency we use in this paper is similar to that of processes as described by Reisig [26] and our notion of concurrent equivalence is directly related to Abrams' and Ghrist's work on State Complexes [1].

Graph grammars were introduced [10, 7] 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 come in many flavors. The variety we use in this paper is called *Graph Rewriting Systems on Induced Subgraphs* (IGRS) by Litovsky et al. [21] (The requirement that rules are applied on induced subgraphs is equivalent to our requirement in Definition 3.2 that witnesses be monomorphisms). Our results about the reachable and stable sets being closed under covers are related in spirit to the results of Courcelle that the classes of graphs that can be *recognized* by local computations (i.e. by an appropriate graph grammar) must be closed under covers [8].

The study of graph grammars usually takes place in a much different context than in the present paper, as can be seen by examining the somewhat different names usually given to the objects we define in this paper: The

initial graph is often called the *axiom*, trajectories are called *derivations*, re-write rules are called *productions* and so on. The terminology in the present work is meant to correspond more closely to the dynamical systems literature. 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.

There are many other models of self assembly besides graph grammars, a complete list of which is beyond the scope of this paper. But, for example, several groups [35, 3] have explored self assembly using passive *tiles* floating in liquid. The tiles attach along complimentary edges (due, for example, to capillary forces or the hybridization of complimentary strands of DNA) upon random collisions. A simple dynamical model of the *physics* of tile assembly has been described [16]. Somewhat similar to the *stable set* in this paper (Definition 3.8), the identification of "unique" assemblies has been explored [31]. There is also other work on supplying assembling robots with state information [5, 14, 30], although not in a graph-theoretic context, and also initial work by other groups [34] on building a self-assembling robot similar to that described in the introduction to this paper. Tile systems can in fact be used to perform arbitrary computations [33] and are best understood as two or three dimensional symbolic processes. Another approach uses geometrical constraints on part-part interactions to model, for example, the assembly of proteins into spherical shells called *capsids* [2]. The addition of simple processing to each part, similar in capability to that assumed in the present paper, is considered in models of the assembly of the T4 bacteriophage [32].

3 Definitions

3.1 Rules and Systems

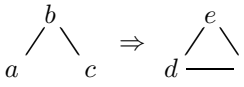
In this section we provide the basic definitions related to our notion of a graph grammar. We save examples for Section 4. The reader may wish to read the examples in Section 4.1 concurrently with this section to help ground the definitions.

A *simple labeled graph* over an alphabet Σ is a triple $G = (V, E, l)$ where V is a set of *vertices*, E is a set of 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 or by V, E and l when there is no danger of confusion. We usually use the alphabet $\Sigma = \{a, b, c, ...\}$.

In this paper, a graph is a model of the *network topology* of an interconnected collection of robots, vehicles or particles. In most examples, a vertex x corresponds to a robot, or, more exactly, the *index* of a robot in an indexed set. The presence of an edge xy corresponds to an attachment (via a physical and/or communication link) between robots x and y. The label $l(x)$ of robot x corresponds the *state* of the robot and will be used to keep track of local information in a self-organizing algorithm.

Definition 3.1 *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. The* size *of r is $|V_L| = |V_R|$. Rules whose vertex sets have one, two and three vertices are called* unary*,* binary *and* ternary*, respectively.*

If (L, R) is a rule, we sometimes denote it by $L \Rightarrow R$ to emphasize its use as a *rewrite rule* or *grammatical production*. 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.

We may refer to rules as being *constructive* ($E_L \subset E_R$), *destructive* ($E_L \supset E_R$) or *mixed* (neither constructive or destructive). A rule is called *acyclic* if its *right hand side* contain no cycles (n.b., the left hand side of an acyclic rule may contain cycles).

In this paper, a rule describes how a (small) sub-collection of an aggregate of robots can change its local network topology and states. In particular, if an induced subgraph of a graph G matches the left hand side L of a rule (L, R), that subgraph may be replaced by the right hand side R. The size (number of vertices) of a rule is a measure of "how local" it is: A unary rule can be executed by a robot without any communication. A binary rule requires the cooperation of two robots via one communication event. Larger rules require even more communication, since a larger group of robots must somehow realize that they collectively match the left hand side of a rule. Typically, one prefers to use small rules. Also notice that the rules depend *only* on the labels (or states) of the robots and not on their underlying indices in the graph. That is, the indices we use for underlying vertex set V do not give particular robots any special properties. The following definitions make formal the notion of rule application.

A *homomorphism* between graphs G_1 and G_2 is a function $h : V_{G_1} \to V_{G_2}$ which preserves edges: $xy \in E_{G_1} \Leftrightarrow h(x)h(y) \in E_{G_2}$. A *monomorphism* is an injective homomorphism. An *isomorphism* is a surjective monomorphism: in this case, G_1 and G_2 are said to be *isomorphic*. A homomorphism h is said to be *label preserving* if $l_{G_1} = l_{G_2} \circ h$.

Definition 3.2 *A rule $r = (L, R)$ is* applicable *to a graph G if there exists a label-preserving monomorphism $h : V_L \to V_G$. In this case, the function h is called a* witness. *An* action *on a graph G is a pair (r, h) such that r is applicable to G with witness h.*

Definition 3.3 *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*

$$\begin{aligned} E' &= (E - \{h(x)h(y) \mid xy \in L\}) \cup \{h(x)h(y) \mid xy \in R\} \\ l'(x) &= \begin{cases} l(x) \text{ if } x \notin h(V_L) \\ l_R \circ h^{-1}(x) \text{ otherwise.} \end{cases} \end{aligned}$$

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

Remark: The vertex set V of the the graph G in the above definition remains the *same* upon the application of a rule. Only the presence or absence of edges and the labels on the vertices change.

A set of rules (i.e. a *grammar*) defines an algorithm for a group of robots to follow. We suppose that each robot has a copy of the rule set initially and that by communicating with other nearby robots they can decide

in a distributed fashion if there is some applicable rule and, if so, apply it. Continuing this process changes the states of the robots and their connections to other robots. The result is a system with a well defined set of behaviors, or trajectories. These are the objects we examine in this paper.

Definition 3.4 *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*).*

We sometimes refer to a system simply by its rule set Φ and frequently suppose that the initial graph is the infinite graph defined by

$$G_0 \triangleq (\mathbb{N}, \varnothing, \lambda x.a) \tag{1}$$

where $a \in \Sigma$ is the *initial symbol* (here $\lambda x.a$ is the function assigning the label a to all vertices). The idea is to model systems with vast numbers of robots or parts all of which have the same initial internal state.

Definition 3.5 *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. We denote the set of trajectories of a system by* $\mathcal{T}(G_0, \Phi)$.

We denote a trajectory by, for example, $\sigma \in \mathcal{T}(G_,\Phi)$ and use the notation σ_j to mean the j^{th} graph in the trajectory.

Remark: Many authors call G_0 the *axiom* and $\sigma \in \mathcal{T}(G_0, \Phi)$ a *derivation* when the goal is to study the language generated by the grammar. By our choice of terminology in the present paper we intend to emphasize the dynamical properties of grammars. For example, the grammars we define can exhibit non-trivial limit-cycles (as in Example 4.5).

Definition 3.5 defines what many authors call the *interleaving semantics* of a concurrent system [4]. In this model, actions cannot occur simultaneously. However, a set of successive actions that operate on disjoint parts of a graph may be interleaved in any order and still produce essentially the same behavior. In Section 6 we provide an alternate characterization of trajectories that better accounts for concurrency. In either case, we see that the pair (G_0, Φ) defines a non-deterministic transition system whose states are the labeled graphs over V_{G_0}. Non-determinism arises due to the fact that, at any given step, several rules in Φ may be simultaneously applicable, each possibly via several different witnesses. Our goal is to reason about all such behaviors of a given system.

3.2 Reachable Graphs

Given a system (G_0, Φ) of robots and rules, one naturally wishes to know (1) what types of aggregates can be built and (2) which of these are "finished products" with regards to the rule set.

Definition 3.6 *A graph* G *is* reachable *by the system* (G_0, Φ) *if there exists a trajectory* $\sigma \in \mathcal{T}(G_0, \Phi)$ *with* $G = \sigma_k$ *for some* k. *The set of all such reachable graphs is denoted* $\mathcal{R}(G_0, \Phi)$.

For assembly problems, we are particularly interested in the connected components of reachable graphs, as these correspond to aggregates of robots that are connected by physical or communication links. Recall that a graph G is *connected* if any pair of vertices can be connected by a sequence of edges in G. The connectivity relation partitions any graph into connected *components*.

Definition 3.7 *A connected graph H is a* reachable component *of a system (G_0, Φ) if there exists a graph $G \in \mathcal{R}(G_0, \Phi)$ such that H is a component of G. The set of all such reachable components is denoted $\mathcal{C}(G_0, \Phi)$.*

A reachable component may be temporary. That is, there may be some rule in Φ that operates on part of it. Other components are permanent: once they show up in a trajectory, they remain forever.

Definition 3.8 *A component $H \in \mathcal{C}(G_0, \Phi)$ is* stable *if, whenever H is a component of $G_k \in \mathcal{R}(G_0, \Phi)$ via a witness f, then H is also a component via f of every graph in $\mathcal{R}(G_k, \Phi)$. The stable components are denoted $\mathcal{S}(G_0, \Phi) \subseteq \mathcal{C}(G_0, \Phi)$. Reachable components that are not stable are called* transient.

In other words, the stable components are those that no applicable rule can change. Note, however, this does not mean that a stable component may not take part in an action, merely that it is left unchanged by it.

We illustrate these definitions with several examples in Section 4. In Section 7 we will describe rule-synthesis algorithms that solve the problem of how to assemble a unique stable component:

Problem 3.1 *(Rule Synthesis for Assembly) Given any graph H and an initial graph G_0, find a set of (preferably small) rules Φ such that $\mathcal{S}(G_0, \Phi) = \{H\}$.*

4 Examples

4.1 Paths and Cycles

In this section we illustrate all the definitions in Section 3 by examining a simple system that assembles paths and cycles from individual parts. Define a constructive rule set by

$$\Phi_1 = \begin{cases} a \quad a \;\Rightarrow\; b - b, & (r_1) \\ a \quad b \;\Rightarrow\; b - c, & (r_2) \\ b \quad b \;\Rightarrow\; c - c. & (r_3) \end{cases}$$

We have named the rules r_1, r_2 and r_3. Recall that we use the position of the vertices in the presentation of the rules to denote the re-labeling. For example, the first rule in Φ corresponds to

$$\begin{aligned} L &= (\{1,2\}, \varnothing, \lambda x.a) \text{ and} \\ R &= (\{1,2\}, \{12\}, \lambda x.b) \end{aligned}$$

We suppose that the rule set is used by a very large set of robots all initially labeled by the symbol a. Thus, the initial graph is defined by Equation (1). At first, the only applicable rule is r_1 since initially no vertices are labeled by either b or c. After r_1 is applied, both r_1 and r_2 are applicable. The rule r_3 eventually becomes applicable as soon as two unconnected vertices labeled b arise.

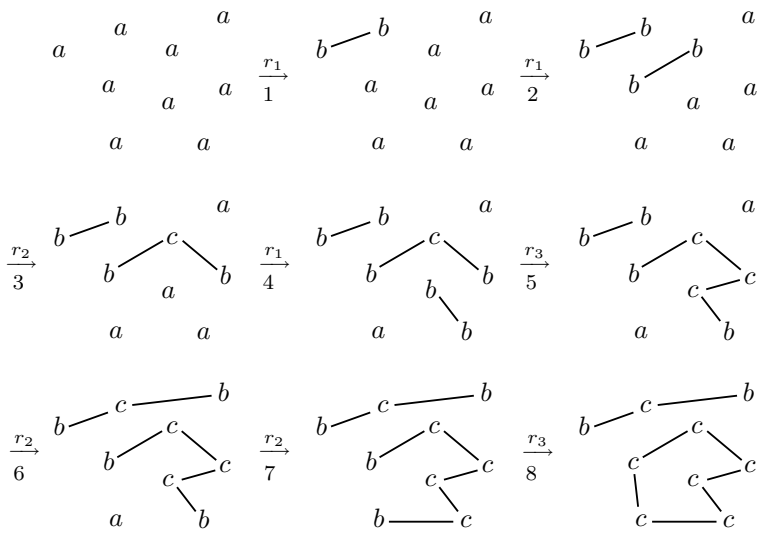


Figure 4: A trajectory of the system defined in Example 4.1. The relative positions of the nodes denote the identities of the nodes and do not change after each rule application. The rules used are shown above the arrows. For convenience (in the discussion in Section 6.3) the actions are numbered sequentially by the integers 1, ..., 8 appearing below the arrows.

An example trajectory of (G_0, Φ_1) is shown in Figure 4. In the figure, the relative positions of the nodes denote the identities of the nodes and do not change after each rule application. The names of the rules used in each action are shown above the arrows. Witnesses are not shown because they can be inferred from the diagram.

From the figure, it is apparent that the grammar produces paths P_k starting and ending with vertices labeled b and with internal vertices labeled c, except for the length-one path, which is labeled a. The grammar also produces cycles C_k with all vertices labeled by c. This can be proved by explicitly constructing a sequence of actions that realizes any given component.

No other components are reachable, which can be shown by induction: The initial graph contains only P_1. If at any point in a trajectory only paths and cycles have been produced then the application of any rule in Φ can only: make P_2 from two copies of P_1 (using r_1); make a path of length P_{j+1} from copies of P_j and P_1 (using r_2); make a path of length P_{j+k} from copies of P_j and P_k (using r_3); make a cycle C_k from a copy of P_k (using r_3). Thus we have:

Proposition 4.1 $\mathcal{C}(G_0, \Phi_1) = \{P_1, P_2, P_3, ...\} \cup \{C_3, C_4, C_5, ...\}$.

The stable components of (G_0, Φ) are the cycles, since no rule in Φ has a left hand side with a vertex labeled c. The paths are all transient.

Proposition 4.2 $\mathcal{S}(G_0, \Phi_1) = \{C_3, C_4, C_5, ...\}$.

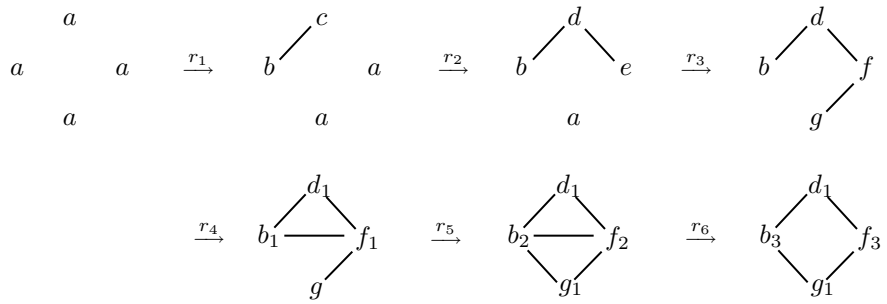


Figure 5: An example trajectory of the system defined in Section 4.2.

4.2 A Uniquely Stable Cycle

In the previous example, all cycles were stable components. There is, in fact, a fundamental limitation, as Theorem 5.1 implies (Section 5): a rule set containing only binary rules cannot have a stable set consisting of exactly one cycle. However, using larger rules we can "stabilize" a cycle of a particular size. For example, consider the following rule set.

$$\Phi_2 = \begin{cases} a \quad a \;\Rightarrow\; b - c, & (r_1) \\ a \quad c \;\Rightarrow\; e - d, & (r_2) \\ a \quad e \;\Rightarrow\; g - f, & (r_3) \\ \begin{array}{c} d \\ b \quad\; f \end{array} \;\Rightarrow\; \begin{array}{c} d_1 \\ b_1 \!-\!\!- f_1 \end{array}, & (r_4) \\ \begin{array}{c} f_1 \\ b_1 \quad\; g \end{array} \;\Rightarrow\; \begin{array}{c} f_2 \\ b_2 \!-\!\!- g_1 \end{array}, & (r_5) \\ b_2 - f_2 \;\Rightarrow\; b_3 \quad f_3. & (r_6) \end{cases}$$

Once again, we take the initial graph to be G_0, defined in Equation (1).

An example trajectory for Φ_2 is shown in Figure 5. The three constructive binary rules yield chains of length 4. The two ternary rules "triangulate" the cycle. The last rule removes the first triangulating edge to yield a copy of the cycle C_4, which is the unique stable graph of the system.

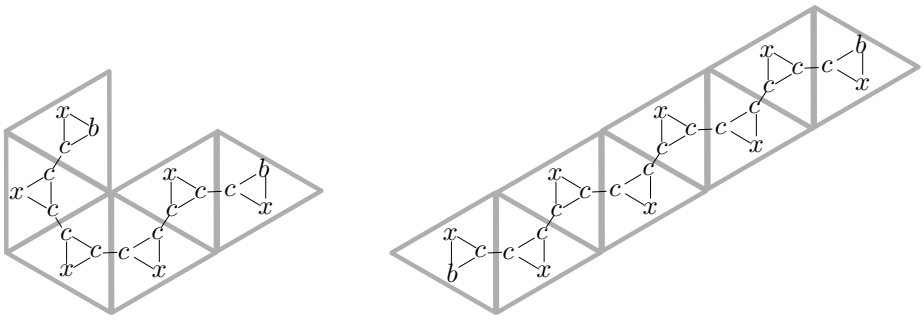
Proposition 4.3 $\mathcal{S}(G_0, \Phi_2) = \{C_4\}$.

If instead of the "scaffolding" rules above, we simply used the rule

$$b \quad g \;\Rightarrow\; b_1 - g_1,$$

to close P_4, then C_4 would indeed be stable, but so would C_8, C_{12} and so on. This is because, for example, two copies of P_4 could combine to form C_8 via two applications of the above rule.

In Section 7, we describe a general procedure for building cyclic graphs via triangulation.

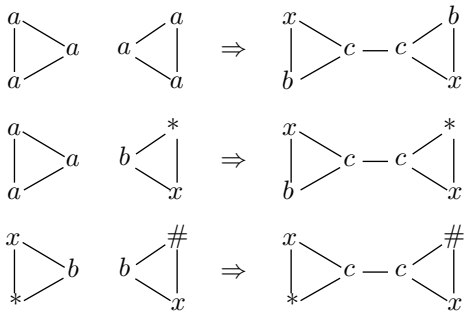


(a) (b)

Figure 6: Two reachable components of the grammar described in Section 4.3.

4.3 Rules for Programmable Parts

The triangular programmable parts shown in Figures 2 and 3 each have three latches with which they can connect to other parts. We associate a label with each of these latches. The *state* stored by the micro-controller is then a triple of labels, and communication between parts must compare a pair of triples against a rule set. As an example, define a simple rule set with the following rules:



This presentation uses the "wildcard" symbols * and # to stand for either b or c. Thus, the second line above is a schema representing two rules and the third line is a schema representing four rules. The symbols a, b and c are used in a way similar to that in Example 4.1. The symbol x is used to effectively "turn off" an edge.

In Figure 6 we shown two reachable components of this grammar, with the actual geometry of the parts shown as well. Note that the orientation of the graphs in the above rules is not specified (i.e. the x can be either immediately clockwise of the binding edge or counter-clockwise (part (a) of the figure) because graphs define only topology. In an actual implementation, however, one can easily guarantee that the x is always clockwise of the binding edge (part (b) of the figure). In this paper we do not address this problem or the (difficult) problem of geometry in general.

In general, any physical implementation of a graph grammar will require careful reasoning about the resulting geometric embedding of the rules, a subject we hope to report on in a future paper.

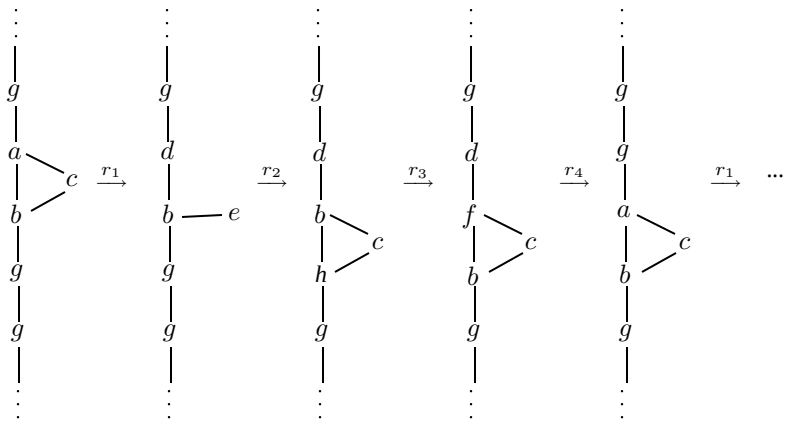


Figure 7: An example trajectory of the ratchet defined in Section 4.4.

4.4 A Ratchet

A rule set need not define a simple assembly process. For example, consider the following rule set.

$$\Phi_3 = \begin{cases} a - c \;\Rightarrow\; d \quad e, & (r_1) \\ \begin{array}{ccc} & b & \\ e & & g \end{array} \;\Rightarrow\; \begin{array}{ccc} & b & \\ c & \text{—} & h \end{array}\ , & (r_2) \\ b - h \;\Rightarrow\; f - b, & (r_3) \\ d - f \;\Rightarrow\; g - a. & (r_4) \end{cases}$$

A trajectory of Φ_3 is shown in Figure 5. The sequence starts with a cycle of robots labeled by a, b and c attached to substrate of robots labeled g (i.e. this structure defines G_0 for this example). As illustrated in the figure, the rule set "ratchets" the cycle along the substrate. The ternary rule is used to prevent the "loose" stage of the ratchet (in the second graph in the trajectory) from attaching to the wrong substrate vertex (i.e. the rule forces the vertex labeled e to attach to the *next* g in the sequence).

If the substrate of vertices labeled g is infinite (or circular), then $\mathcal{S}(G_0, \Phi) = \varnothing$. The reachable components are all isomorphic to those shown in Figure 7. We plan to focus on cyclic and other more general processes defined by graph grammars in a future paper.

4.5 Bubble Sort

Graph grammars are commonly used to describe distributed algorithms in a fixed network topology. In fact, it is rare to find a rule with a disconnected left hand side in the graph grammar literature [27]. In this section, we illustrate a very simple and traditional distributed algorithm for sorting an array. Although perhaps not particularly useful in robotics, the example hopefully gives the reader an idea of the possibilities of our approach.

We suppose that each robot in a chain has a unique id and an integer in an array to be sorted. The robots are initially arranged in order of increasing id, but their array values are not in order. Robots may trade their values with neighboring robots when doing so improves the order of the array.

To describe this algorithm as a graph grammar, we use the alphabet $\Sigma = \mathbb{N} \times \mathbb{N}$ where $(id, val) \in \Sigma$ denotes the process identifier and current value of a process. The initial graph G_0 is any graph of the form

$$(1, v_1) - (2, v_2) - ... - (n, v_n)$$

where $v_k \in \mathbb{N}$. Without loss of generality, we suppose the underlying vertex set is $\{1, ..., n\}$ arranged so that $l_{G_0}(k) = (k, v_k)$. The rule set is defined by

$$\Phi_{sort} = \{(i, x) - (i+1, y) \;\; \Rightarrow \;\; (i, y) - (i+1, x) \;\mid\; i \in \mathbb{N} \text{ and } x > y\}.$$

Each step in a trajectory of (G_0, Φ_{sort}) increases the order of the array. This can be shown by considering the "Lyapunov function"

$$V(G) = \sum_{i,j} [val_G(i), val_G(j)].$$

In this expression, $val(i)$ is the value of vertex i (i.e. if $l_G(i) = (i, x)$ then $val_G(i) = x$) and $[x, y]$ is equal to 1 if $x < y$ and is equal to 0 otherwise. If $\sigma \in \mathcal{T}(G_0, \Phi_{sort})$ then the sequence

$$V(\sigma_1), V(\sigma_2), ..., V(\sigma_m)$$

is strictly decreasing to 0, where the array is sorted. Thus, we have the following.

Proposition 4.4 *The single stable component of* (G_0, Φ_{sort}) *is*

$$(1, x_1) - (2, x_2) - ... - (n, x_n)$$

where $\{x_1, ..., x_n\} = \{v_1, ..., v_n\}$ *and* $x_i \leq x_{i+1}$.

4.6 Other Examples Not Covered in This Paper

In this section we briefly mention interesting grammars and systems we or others have considered but whose full consideration is beyond the scope of this paper.

Graph Recognizers A graph recognizer [21] for a class of graphs $\mathcal{G}$ is a grammar Φ containing rules that do not add or delete edges and with the following property: If $G \in \mathcal{G}$ then all trajectories of (G, Φ) arrive at a copy of G with each vertex labeled by a and if $G \notin \mathcal{G}$ then all trajectories of (G, Φ) arrive at a copy of G with each vertex labeled by b. For example, the class of k-regular graphs, the class of 2-colorable graphs and the class of graphs with a given fixed diameter can be recognized this way. However, other classes, such as the class of planar graphs cannot be recognized [21]. This related to the fact that planar graphs are not closed under coverings (see our discussion of coverings in Section 5), a difficulty with the "local" nature of graph grammars discussed in more detail by Courcelle and Métevier [8].

Distributed Algorithms Distributed algorithms, such as consensus and leader election, that are often expressed in terms of somewhat sophisticated I/O automata [22] can be easily expressed in graph grammars as well [21]. This opens up the possibility of easily combining such algorithms with the self-organization algorithms we describe in this paper — that is, using the same formalism.

Planar Tilings Graph grammars as we have described them here are oblivious to the geometry of the robots or parts on which they operate. An important geometric question is: When are the reachable graphs of a grammar planar? Planarity of the reachable set is desirable when, for example, the grammar is to be implemented with flat robots (or tiles) that can connect only via their edges. We have begun to explore this problem and have proved several simple results [12]: (1) one can tile arbitrarily large portions of the plane with a finite grammar (finite number of rules); (2) there exist aggregates of tiles in the plane that cannot by uniquely stabilized by any finite grammar. However, the connection between graph grammars and their geometrical realizations is, for the most part, uncharted territory.

Self-Replication In other work, we focus on the design of a rule set that can replicate *any* suitable labeled "seed" graph [19]. The products of the replication, being identical to the seed, continue to replicate until all "raw materials" are used up. The main point is that the desired behavior is not encoded in the rules of the system, but is instead encoded by the seed assembly. In the grammar we designed for self replication [19], we assume that the initial seed graphs are strings of, for example, the form

$$s_1 - a_1 - b_1 - a_1 - a_1 - t_1,$$

consisting of an initial vertex labeled s_1, a terminal vertex labeled t_1 and an internal (and arbitrary) string of symbols labeled by a_1 or b_1. The rest of the initial graph consists of unconnected copies of the "raw materials": a_0, b_0, s_0 and t_0. During the course of the replication, each part type can be in one of several "conformations" (denoted by, for example, a_0, a_1, a_2, ...). In all, 21 symbols and 38 rules are used in the grammar. In the above cited paper, we prove the grammar is correct by explicitly constructing the reachable set and its dynamic structure.

5 Properties

It is intuitively clear that a grammar containing only binary rules cannot "distinguish" between a cycle of length $2N$, and two identical cycles of length N (cf. [21]). This has implications to the problem of constructing uniquely stable components. We extend this intuition in to a much larger class of rule sets. Given a system (G_0, Φ), we bound the size of the reachable and stable sets using a basic topological tool: covering space theory.[1] A clear and comprehensive introduction to these classical techniques can be found in the text by Hatcher [13, pp. 60-68].

Definition 5.1 *Given a graph G, an n-*fold cover *of G is a graph $\widetilde{G}$ such that, equivalently:*

1. *There exists a label-preserving n-to-1 homomorphism $p : V(\widetilde{G}) \to V(G)$ that preserves degree (i.e., the image of an degree k vertex is a degree k vertex).*
2. *There exists a label-preserving n-to-1 continuous map $p : \widetilde{G} \to G$ (where G and $\widetilde{G}$ are thought of as 1-d cell complexes) which is a local homeomorphism.*

[1] We have simplified the definitions and suppressed most of the explicit topological terminology for the sake of clarity.

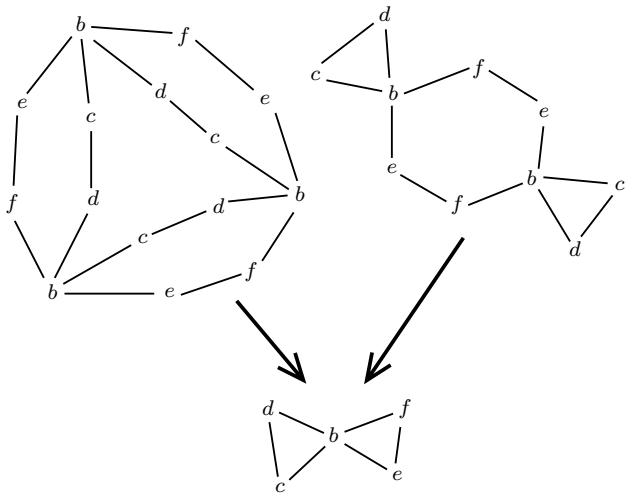


Figure 8: Two examples (above) of covers of a graph (below). The cover on the left is a 3-fold cover; that on the right is 2-fold.

The latter definition is the standard one in covering space theory; the former is particular to graphs and easier to verify. It is straightforward to demonstrate that these two definitions are equivalent. Examples appear in Figure 8. Note that in the case of a *trivial* 1-fold cover, p is an isomorphism.

Theorem 5.1 *For (G_0, Φ) an acyclic rule set, $\mathcal{C}(G_0, \Phi)$ is closed under covers. In particular, $\mathcal{C}(G_0, \Phi)$ contains infinitely many isomorphism types of graphs if it contains any graph with a cycle.*

Proof: Since we use tools from covering space theory in this proof, we will consider graphs to be topological objects: 1-d cell complexes in particular. All maps between graphs will be continuous maps, and covering projections will be local homeomorphisms as in the second half of Definition 5.1.

Assume that $H \in \mathcal{C}(G_0, \Phi)$ is a component of σ_k for some trajectory $\sigma = (\sigma_i, (r_i, h_i))$. Consider any n-fold covering projection $p : \widetilde{H} \to H$ of this component. We will reverse the trajectory σ and lift this disassembly procedure to build a trajectory $\widetilde{\sigma}$ with $\widetilde{H}$ a component of $\widetilde{\sigma}_{nk}$.

Denote by $\widetilde{\sigma}_{nk}$ the graph consisting of the disjoint union of $\widetilde{H}$ with n disjoint copies of the complement $\sigma_k - H$ (see Figure 9). Define the projection $p_k : \widetilde{\sigma}_{nk} \to \sigma_k$ via p on $\widetilde{H}$ and via the obvious projection of the complementary copies.

The image of the right hand side of the k^{th} rule $h_k(R_k)$ is a subtree of σ_k. Since the rule set is acyclic, the *lifting criterion* [13, pp. 61-62] is automatically satisfied, and it follows that the inverse image $\widetilde{R_k} := p_k^{-1}(R_k)$ is a disjoint union of isomorphic copies of R_k.

Replace each copy of R_k in $\widetilde{R_k}$ with its left hand side L_k, reversing the assembly step. There are n such replacements to be performed; any order is acceptable.[2] Denote by $\widetilde{\sigma}_{(k-1)n}$ the graph obtained after all n replacements, and denote by $\widetilde{L_k}$ the disjoint union of n copies of L_k within $\widetilde{\sigma}_{(k-1)n}$. Define a new projection map $p_{k-1} : \widetilde{\sigma}_{(k-1)n} \to \sigma_{k-1}$ to be (1) p_k on the complement of $\widetilde{L_k}$; and (2) the natural projection $\widetilde{L_k} \to L_k$

[2]These n rules are commutative, see Section 6.

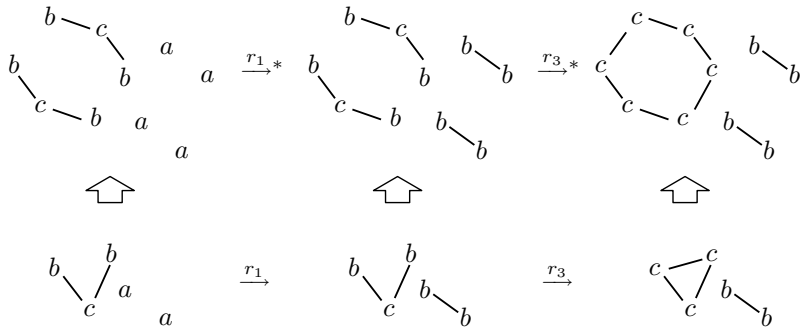


Figure 9: Part of a trajectory and its lifting via a 2-fold covering for the system defined in Example 4.1. In this figure, the symbol the $*$ after the yields relation denotes two applications of the rule r_i.

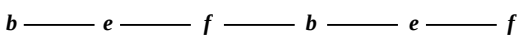


Figure 10: A rule with this left hand side can de-stabilize the covers in Figure 8 even though it does not apply to the (stable) graph being covered.

identifying the disjoint copies. This graph $\widetilde{\sigma}_{(k-1)n}$ is clearly an n-fold cover of σ_{k-1} via p_{k-1} since labels and indices are preserved.

Continue this procedure inductively, lifting R_i to $\widetilde{R_i}$, replacing it in parallel with copies of L_i, and then defining the projection map $p_{i-1} : \widetilde{\sigma}_{(i-1)n} \to \sigma_{i-1}$. This terminates in a covering projection $p_0 : \widetilde{\sigma}_0 \to \sigma_0$. Since $\sigma_0 = G_0$ and the lift of any discrete set is again a discrete set, we have that $\widetilde{\sigma}_0$ is isomorphic to G_0. Thus, $\widetilde{\sigma} \in \mathcal{T}(G_0, \Phi)$ and $H \in \mathcal{C}(G_0, \Phi)$.

In the case where H possesses a cycle, one has that there are infinitely many non-isomorphic covers (corresponding to subgroups of the fundamental group of H: see [13, Thm. 1.38]). □

The stable set, $\mathcal{S}(G_0, \Phi) \subset \mathcal{C}(G_0, \Phi)$, is, however, not necessarily closed under n-fold covers. Indeed, there may be additional rules in Φ which have "large" connected regions of $\widetilde{H}$ in their left hand sides. If the left hand sides are sufficiently large, then these rules can apply to covers even though H itself is inert. See Figure 10.

The following is a sample of the type of result that can be obtained using covering space theory.

Theorem 5.2 *Assume that Φ is an acyclic rule set, and that the stable set contains a component $H \in \mathcal{S}(G_0, \Phi)$ but not any of its covers. Then for each edge $e \in E(H)$, there exists a rule in Φ whose left hand side contains a copy of every edge of some cycle in H passing through e.*

Proof: Consider the 2-fold cover of H lifted along the edge e; that is, take two copies of H, snip each copy of e, and chain them end-to-end to form the connected graph $\widetilde{H}$, as in Figure 8[right] using the subgraph $e - f$. One checks that $p : \widetilde{H} \to H$ is indeed a covering projection.

Theorem 5.1 implies that $\widetilde{H}$ is a component of some $G_k \in \mathcal{R}(G_0, \Phi)$. By hypothesis, this component is not stable; hence there is some rule $(r, h) \in \Phi$ applicable to $\widetilde{H}$. Consider the image $p(h(L))$ of L in H. If this image were isomorphic to $h(L)$, then (r, h) would be applicable to H, contradicting the fact that H is stable. But it

follows from the lifting criterion that the only subgraphs of H that do not lift to isomorphic copies in $\widetilde{H}$ are those having a loop in H passing through the edge e. □

Corollary 5.1 *If Φ is an acyclic rule set all of whose left hand sides have no edges, then $\mathcal{S}(G_0, \Phi)$ is closed under covers.*

Proof: Since the left hand sides have no edges, there can be no long chains to wrap around loops in a cover. □

These results are best interpreted as topological bounds on the maximal amount of communication required to build a unique stable graph. Additional results and extensions to non-acyclic rules or to initial graphs different than G_0 are possible if one carefully tracks cycles.

6 Concurrency

6.1 Interleaved Versus Concurrent Trajectories

A graph grammar essentially describes a (non-deterministic) *parallel* algorithm or dynamical system. Two actions can be executed in parallel if they operate on disjoint parts of whatever is the current graph: that is, if they are physically independent of one another. In Definition 3.5, however, we give what is traditionally called an *interleaved semantics* for trajectories [4] where we suppose nothing truly parallel ever happens, but rather that any two actions that could occur in parallel are ordered in time somehow to create two different trajectories. From the point of view of concurrency, then, two trajectories as we have defined them could really be representatives of the *same* behavior.

In this section, we relate the interleaved semantics with a more faithful *concurrent* or *partial order* semantics [25]. Concurrency allows us, for example, to reason more naturally about the number of steps required to assemble a structure with a given graph grammar. The definition we give is similar to that found in the Petri Net literature [26], Higher Dimensional Automata, [24] and State Complexes [1].

Definition 6.1 *Let $A = \{a_1, ..., a_k\}$ with $a_i = ((L_i, R_i), h_i)$ be a set of actions each applicable to a graph G. The set A is called* commutative *if for all i and j*

$$h_i(L_i) \cap h_j(L_j) = \varnothing.$$

If A is commutative with respect to G_0 and $G_0 \xrightarrow{r_1,h_1} ... \xrightarrow{r_k,h_k} G_k$, then we write $G_0 \xrightarrow{A} G_k$.

When A is commutative, we are justified in writing $G \xrightarrow{A} G'$ since the graph G' is independent of the order in which the actions in A are applied. Using this notion, we form the set of *concurrent trajectories* of a system.

Definition 6.2 *A* concurrent trajectory *of a system (G_0, Φ) is a (finite or infinite) sequence*

$$G_0 \xrightarrow{A_1} G_1 \xrightarrow{A_2} G_2 \xrightarrow{A_3} ...$$

where A_{i+1} is a commutative set of actions applicable to G_i for each i. If the sequence is finite, then we require that there is no rule in Φ applicable to the terminal graph.

One can compress an interleaved trajectory by applying several consecutive but physically independent actions simultaneously (under the working assumption that all rules execute in unit time). Conversely, any concurrent trajectory can be transformed into an interleaved trajectory by simply choosing some ordering of the elements within each commutative step A_i.

We call two concurrent trajectories *similar* if they can be transformed into the same interleaved trajectory, and hence model essentially the same behavior. Clearly, similarity is an equivalence relation. We argue that within each equivalence class, there is a canonical representative which is optimal with respect to parallelizability.

6.2 Left-Greedy Concurrent Trajectories

Recall that a *partial order* $(P, \preceq)$ is a set P with an order relation $\preceq$ that is reflexive, antisymmetric and transitive [9, ch. 1]. An element $x \in P$ is *minimal* if there does not exist an element $y \in P$ with $y \preceq x$ and $y \neq x$. The set of all minimal elements of $(P, \preceq)$ is denoted $\min(P, \preceq)$.

There is a natural partial order associated with any trajectory $\sigma \in \mathcal{T}(G_0, \Phi)$. Suppose that

$$\sigma_0 \xrightarrow{a_1} \sigma_1 \xrightarrow{a_2} ... \xrightarrow{a_k} \sigma_k$$

is an interleaved trajectory where $a_i = ((L_i, R_i), h_i)$. Let $P = \{1, ..., k\}$ and define a relation $\sqsubset$ by

$$i \sqsubset j \Leftrightarrow i \leq j \text{ and } h_i(L_i) \cap h_j(L_j) \neq \varnothing.$$

The partial order relation $\preceq$ is then given by the reflexive and transitive closure of $\sqsubset$. This partial order gives rise to a canonical concurrent trajectory as follows.

Definition 6.3 *Let $\sigma \in \mathcal{T}(G_0, \Phi)$ denote a trajectory with partial order $(P, \preceq)$. Define*

$$\begin{aligned} A_1 &= \{a_i \mid i \in \min(P, \preceq)\} \\ A_2 &= \{a_i \mid i \in \min(P - A_1, \preceq)\} \\ &\vdots \\ A_{j+1} &= \{a_i \mid i \in \min(P - \bigcup_{i=1}^{j} A_i, \preceq)\} \\ &\vdots \end{aligned}$$

Then the left-greedy concurrent trajectory *arising from σ is defined to be $\overline{\sigma}$ where $\overline{\sigma}_0 = \sigma_0$ and*

$$\overline{\sigma}_0 \xrightarrow{A_1} \overline{\sigma}_1 \xrightarrow{A_2} ... \xrightarrow{A_j} \overline{\sigma}_j.$$

We denote the set of all left-greedy trajectories of a system by $\overline{\mathcal{T}}(G_0, \Phi)$.

Note that the left-greedy concurrent trajectory obtained from a given interleaved trajectory is certainly not the *only* concurrent trajectory that can be formed. However, these trajectories perform as many commutative steps as possible as early as possible, hence the name. They are both 'canonical' and 'optimal' as we demonstrate below.

Lemma 6.1 *An arbitrary concurrent trajectory*

$$\overline{\sigma}_0 \xrightarrow{A_1} \overline{\sigma}_1 \xrightarrow{A_2} ...$$

is left-greedy if and only if for every i and every $a \in A_{i+1}$, the collection $A_i \cup \{a\}$ is not commutative.

Proof: This follows directly from Definition 6.3. □

Theorem 6.1 *Any finite concurrent trajectory (or prefix of an infinite trajectory) is similar to a unique left-greedy concurrent trajectory.*

Proof: We begin with existence. Let

$$\overline{\sigma}_0 \xrightarrow{A_1} \overline{\sigma}_1 \xrightarrow{A_2} ... \xrightarrow{A_n} \overline{\sigma}_n$$

be a concurrent trajectory. If $\overline{\sigma}$ is not in left-greedy form, then there is some step i and some action $a \in A_{i+1}$ for which $A_i \cup \{a\}$ is commutative, by Lemma 6.1. Form a trajectory $\overline{\sigma}'$ by shifting the action a from A_{i+1} to A_i. (If A_{i+1} is now empty, then it is deleted, and the sequence is re-indexed.) Shifting an action clearly does not change the similarity class of the trajectory. This shifting strictly decreases the quantity $\sum_i i \cdot |A_i|$; thus, repeating the process terminates in a trajectory which, by Lemma 6.1 is left-greedy.

To prove uniqueness, suppose $\overline{\sigma}$ has two different left-greedy forms

$$\overline{\beta} = \overline{\beta}_0 \xrightarrow{A_1} \overline{\beta}_1 \xrightarrow{A_2} ... \quad \text{and} \quad \overline{\rho} = \overline{\rho}_0 \xrightarrow{B_1} \overline{\rho}_1 \xrightarrow{B_2} ...$$

and suppose that $A_i = B_i$ for $i = 1, ..., k$ but that $A_k \neq B_k$. Then, without loss of generality, there is some action $a \in A_k - B_k$. Because the two trajectories are similar and because $A_{k-1} = B_{k-1}$, it must be the case that $a \in B_{k+1}$. But then $B_k \cup \{a\}$ is commutative (since it is a subset of A_k), which is a contradiction. □

Corollary 6.1 *The left-greedy form of a trajectory has the minimal number of steps (commutative sets of actions) among all trajectories in its similarity class.*

Proof: The left-greedy path minimizes the quantity $\sum_i i \cdot |A_i|$ within its similarity class. □

Remark: These two results together are a restatement, in the language of partial orders, of geometric results about paths in cubical complexes [1]. The notion of similarity corresponds to homotopy of paths in a certain cubical complex, and the left-greedy trajectories correspond normal forms for geodesics on these spaces.

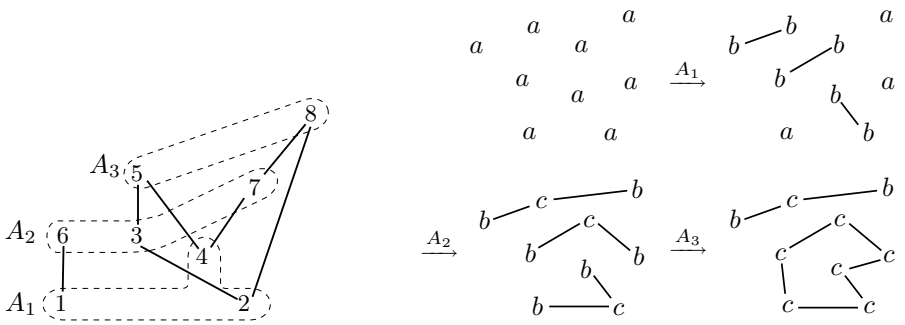
This motivates the following:

Definition 6.4 *For $H \in \mathcal{C}(G_0, \Phi)$ a reachable component, the* assembly time $\tau_H(\overline{\sigma})$ *of H in the trajectory $\overline{\sigma} \in \overline{\mathcal{T}}(G_0, \Phi)$ is the smallest integer i such that H is a component of $\overline{\sigma}_i$ (or ∞ if there is no such integer). The* best case assembly time *of H is the minimum of the set*

$$\{\tau_H(\overline{\sigma}) \mid \overline{\sigma} \in \overline{\mathcal{T}}(G_0, \Phi)\}.$$

The worst case assembly time *of H is the maximum of the set*

$$\{\tau_H(\overline{\sigma}) \mid \overline{\sigma} \in \overline{\mathcal{T}}(G_0, \Phi) \text{ and } H \text{ occurs in } \overline{\sigma}\}.$$



(a) (b)

Figure 11: (a) The partial order associated with the trajectory in Figure 4. The sets of commutative actions A_1 through A_3 as in Definition 6.2. (b) The left-greedy concurrent trajectory associated with the trajectory in Figure 4.

Note that a given reachable component H may not occur in every trajectory. If this is the case, we could have declared its worst case assembly time to be ∞. However, in the above definition, we only consider the worst case among those trajectories that assemble H, and so perhaps we should use the term "worst case realizable assembly time." We believe this is a more useful and informative number for characterizing the assembly time of a component with respect to a given grammar.

6.3 Chains and Cycles Revisited

To illustrate the notions of concurrency and assembly time, we revisit the example system that assembles paths and cycles, described in Section 4.1. In Figure 4, the actions in a trajectory of this system are numbered $1, 2, ..., 8$.

The partial order associated with the trajectory in Figure 4 is shown (as a *Hasse Diagram*) in Figure 11(a). The commutative sets A_1, A_2 and A_3 from Definition 6.2 are also shown. The associated concurrent trajectory is shown in Figure 11(b). If this trajectory is called $\overline{\sigma}$, then from the figure it can be seen that, for example, the assembly time $\tau_{P_3}(\overline{\sigma})$ of P_3 is 2. Also, the assembly time $\tau_{C_6}(\overline{\sigma})$ of C_6 is 3.

In general, the best case assembly time for P_k (with $k > 2$) is 2 steps since we could use rule r_1 in parallel to make $k/2$ copies of P_2 and then use rule r_3 in parallel to join these copies of P_2 into a copy of P_k. The best case assembly time for C_k is similarly 2 steps or 3 steps depending on whether k is even or odd, respectively.

If we consider only those trajectories in which P_k occurs, then we see that the worst case assembly time is k: Rule r_1 is used to make P_2 and then r_2 is used repeatedly. Similarly, the worst case assembly time for C_k, in those trajectories in which it occurs, is also k steps.

Proposition 6.1 *In the system (G_0, Φ_1), the best case assembly time for P_k is 2, the best case assembly time for C_k is 2 if k is even and 3 if k is odd. The worst case assembly time for both P_k and C_k is k.*

Algorithm 1 *MakeTree*(V, E)

Require: $T = (V, E)$ is an unlabeled tree

1: **if** $V = \{x\}$ **then**
2: **return** $(\varnothing, \{(x, a)\})$
3: **else**
4: **choose** $xy \in E$
5: **let** (V_1, E_1) be the component of $(V, E - xy)$ containing x
6: **let** (V_2, E_2) be the component of $(V, E - xy)$ containing y
7: **let** $(\Phi_i, l_i) = \mathit{MakeTree}(V_i, E_i)$ for $i = 1, 2$
8: **let** u, v be new labels
9: **let** $\Phi = \Phi_1 \cup \Phi_2 \cup \{\ l_1(x) \quad l_2(y) \quad \Rightarrow \quad u - v\ \}$
10: **let** $l = (\ l_1 - \{(x, l_1(x))\}\ ) \ \cup\ (\ l_2 - \{(y, l_2(y))\}\ ) \ \cup\ \{(x, u), (y, v)\}$
11: **return** (Φ, l)
12: **end if**

7 Synthesis Algorithms

In this section we consider the *self assembly problem*: Given a graph G, find a rule set Φ such that $\mathcal{S}(G_0, \Phi) = \{G'\}$ and $G' \simeq G$., where G_0 is defined by equation (1). We first describe an algorithm that constructs a rule set to assemble a given tree (i.e. an acyclic graph) and then use it to build an algorithm that constructs a rule set to assemble an arbitrary graph.

7.1 Trees

We define in Algorithm 1 a recursive procedure *MakeTree* that, given any tree T, produces a set of binary rules Φ_T so that $\mathcal{S}(G_0, \Phi_T) = \{T\}$ (up to isomorphism and not including labels). The procedure takes as an argument an unlabeled tree (V, E) and returns a pair (Φ, l) where Φ is a rule set and l is a labeling function on V. For convenience, the function l is here denoted by subset of $V \times \Sigma$.

The base case of the procedure (lines 1-2) labels the singleton graph with the label a. For the recursive step, an edge (x, y) is chosen and *MakeTree* is called recursively on the two components that result from removing (x, y) from E. The recursive calls return rule sets Φ_1 and Φ_2 and labeling functions l_1 and l_2. The rule set Φ for (V, E) is constructed from these in line 9 and the labeling function l is constructed in line 10. The construction uses two new labels u and v that we suppose have not been used before by any recursive call to *MakeTree*.

Theorem 7.1 *Let* $(\Phi, l) = \mathit{MakeTree}(V, E)$ *where* (V, E) *is an unlabeled tree. Then* $\mathcal{S}(G_0, \Phi) = \{T\}$.

Proof: By general induction on the size of T. □

Notice that the *MakeTree* procedure defines an acyclic assembly graph as shown, for example, in Figure 12. The structure of this graph essentially defines the partial order associated with any trajectory of the resulting system (G_0, Φ). In some cases, the assembly graph is balanced and the concurrent assembly time of (V, E), using Φ, is $O(\log |V|)$. However, in other cases the assembly graph cannot be balanced (as with, for example, a

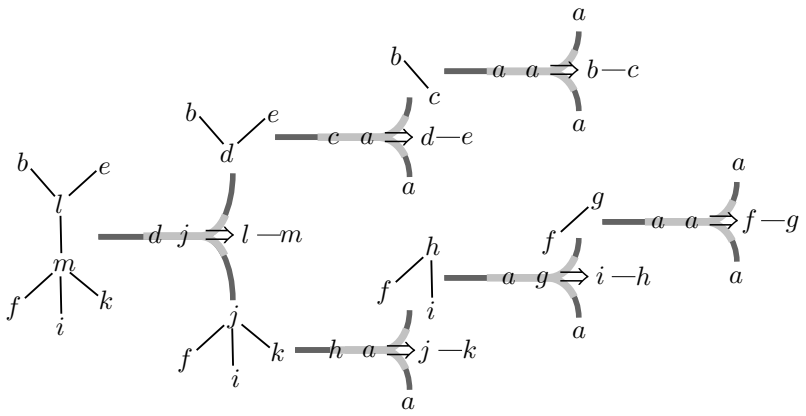


Figure 12: An example assembly graph produced by the *MakeTree* procedure.

star graph consisting of a root vertex connected directly to all other vertices). In this case, the assembly time of (V, E) is $O(n)$ since none of the rules produced by *MakeTree* commute. In either case, we may state:

Theorem 7.2 *Let $(\Phi, l) = MakeTree(V, E)$ where (V, E) is an unlabeled tree. The best and worst case assembly times of T in the system $\mathcal{S}(G_0, \Phi)$ are both $O(|V|)$.*

Of course, the best and worst case assembly times may be considerably better, depending on the structure of the input graph. For example, *MakeTree* will produce a rule set that assembles the chain C_n in $O(\log n)$ steps as long as the edge chosen in step 4 partitions the tree into approximately equal parts.

Note that the procedure $MakeTree(V, E)$ produces exactly $|V| - 1$ rules and uses $2|V| - 2$ labels.

7.2 Arbitrary Graphs

Given a graph G, Algorithm 2 defines a function *MakeGraph* that produces a rule set Φ_G such that $\mathcal{S}(\Phi_G) = \{G\}$ using rules that are *at most* ternary in size. A priori, this would appear to be difficult, given the constraints of Theorem 5.2. The ingredient that makes this low-communication synthesis possible involves building a temporary "scaffold" to close each cycle.

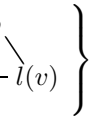
The procedure begins in lines 1-2 by finding a maximal spanning tree $T = (V, E_T)$ for (V, E) and constructing a rule set Φ for this tree using *MakeTree*. We choose r (for root) to be a vertex on the first edge chosen in the call to MakeTree. In a trajectory, when r is eventually labeled by $l(r)$, we can be certain that the tree is assembled and that we may begin to close the cycles.

The loop in lines 6-13 adds rules to Φ that, in effect, triangulate the spanning tree until the resulting graph contains a subgraph isomorphic to G. At each step, E_S denotes the edges that have been used in this triangulation.

The triangulation procedure, defined by *Triangulate* and called in lines 7-8 of *MakeGraph*, is as follows. For each *chord* $uv \in E - E_T$ we find a shortest path in (V, E_S) from r to u, and form ternary rules that add edges from r to each successive vertex along this path. We then update E_S to include these edges, and then repeat the procedure for a minimal path from r to v. Then in lines 9-10 of *MakeGraph* we add an edge from u to v,

Algorithm 2 *MakeGraph*(V, E)

Require: $G = (V, E)$ is an unlabeled connected graph

1: **let** $T = (V, E_T)$ be a maximal spanning tree of (V, E)
2: **let** $(\Phi, l) = \mathit{MakeTree}(V, E_T)$
3: **let** r be a vertex on the first edge chose by *MakeTree*
4: **let** $x = l(r)$
5: **let** $E_S = E_T$
6: **for all** $uv \in E - E_T$ **do**
7: **let** $(\Psi_i, E_S, x) = \mathit{Triangulate}((V, E_S, l), r, u, x)$
8: **let** $(\Psi_i, E_S, x) = \mathit{Triangulate}((V, E_S, l), r, v, x)$
9: **let** y be a new label
10: **let** $\Phi = \Phi \cup \Psi_1 \cup \Psi_2 \cup$ 
11: **let** $E_S = E_S \cup \{uv\}$
12: **let** $x = y$
13: **end for**
14: **for all** $v \in V$ **do**
15: **if** $rv \in E_S - E$ **then**
16: **let** $\Phi = \Phi \cup \{\ x - l(v) \ \Rightarrow \ x \ \ l(v)\ \}$
17: **end if**
18: **end for**
19: **return** Φ

which can be done with a ternary rule since r will be connected via the scaffolding to both u and v. Note that if the distance from r to v in *Triangulate* less than 3, the loop on line 5 will not execute so that no rules are added.

After each new rule is added, we change the label on the root node, ensuring that the rules can only be applied in the order given. The triangulation for one chord cannot begin until the triangulation for the previous one is complete (i.e. none of the triangulation rules commute). After the loop on lines 6-14 has finished, the resulting rule set will have a single graph in its stable set, and this graph will contain a subgraph isomorphic G. The rules added by the loop on lines 14-18 serve to remove the excess edges between the root node r and the other vertices. The left side of each of these rules contains a label that only appears on the root node after the entire triangulation process is complete. In effect, the label on the root node tracks the progress towards completion of the triangulation. After all the rules given in lines 14-18 have been applied, the resulting graph will be isomorphic to G. As shown above, this graph will be the only element of the stable set for Φ. An example trajectory illustrating this procedure is shown in Figure 13.

This discussion constitutes the following:

Theorem 7.3 *Let* $(\Phi, l) = \mathit{MakeGraph}(V, E)$ *where* $G = (V, E)$ *is an unlabeled graph. Then* $\mathcal{S}(\Phi) = \{G\}$.

Algorithm 3 *Triangulate*(G, r, v, x)

Require: G is a labeled graph, $r, v \in V_G$ and x is a label

1: **let** $\Psi = \varnothing$
2: **let** $w = x$
3: **let** $E = E_G$
4: **let** $(v_1, \ldots, v_n)$ be a shortest path in G from r to v
5: **for** $i = 2$ to $n - 1$ **do**
6: **let** z be a new label
7: **let** $\Psi = \Psi \cup \left\{ \begin{array}{c} l(v_i) \\ a \quad l(v_{i+1}) \end{array} \Rightarrow \begin{array}{c} l(v_i) \\ b \text{ — } l(v_{i+1}) \end{array} \right\}$
8: **let** $E = E \cup \{rv_{i+1}\}$
9: **let** $w = z$
10: **end for**
11: **return** $(\Psi,\ E',\ w)$

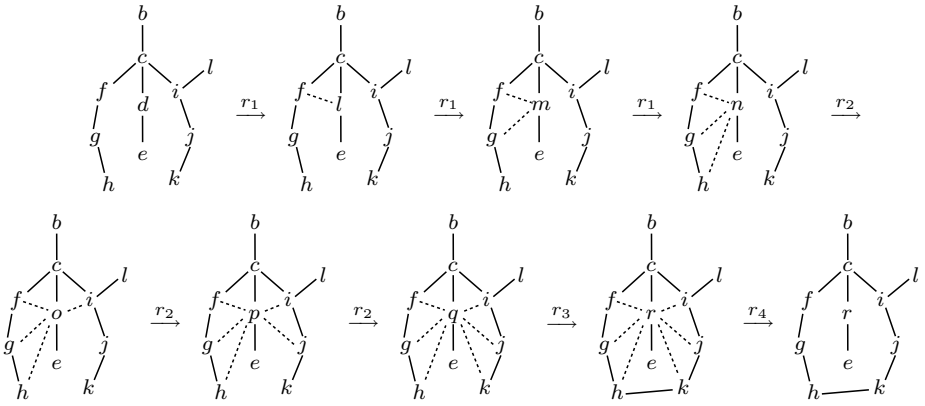


Figure 13: A partial trajectory arising from the *MakeGraph* rules for a particular graph. Once the spanning tree is formed (first graph), the root vertex (labeled d initially) is used to form a scaffolding. First the path from the root to the vertex v (labeled by h in this example) is triangulated (productions labeled r_1). Then the path from the root to the vertex u (labeled by k in this example) is triangulated (productions labeled r_2). The loop is then closed (in the production labeled r_3) and finally the scaffolding edges are removed (in the production labeled r_4, which consists of six actions).

The *MakeGraph* rules produces an essentially a serial process, once the spanning tree has been assembled. If there are $|E|$ edges in the input graph, then rules for adding $c \triangleq |E| - n + 1$ of them will created by *MakeGraph* (the rest are created by the *MakeTree* rules). The end-vertices of each chord are, in the worst case, a distance $O(|V|)$ apart in the graph (V, E_S) in lines 7-8 of *MakeGraph*. Thus, $O(c|V|)$ rules will be added for each chord. Since these rules do not commute, the assembly times of the rules Φ produced by *MakeGraph* are easily characterized:

Theorem 7.4 *Let $\Phi = MakeGraph(V, E)$ where (V, E) is an unlabeled connected graph with $|E| = c + |V| - 1$. The best and worst case assembly times of (V, E) in the system $\mathcal{S}(G_0, \Phi)$ are both $O(c|V|)$.*

The algorithm of course does not give an optimal rule set for any given graph. However, it does demonstrate the triangulation approach to closing cycles, some variation of which will be needed by any rule set that assembles a single stable cyclic graph.

8 Communication

In the above sections, we assumed that the robots (or parts) do not have unique identifiers or any other qualities that distinguish them from other robots or parts with the same labels and local network topologies. This assumption along with the fact that we usually suppose that all vertices in our initial graphs have the same label is the main reason that binary rules do not suffice for many problems. However, if we suppose that each robot has a unique identifier (e.g. an IP address), then we can get away with smaller rules. In this section we explore first how a simple communication protocol between robots can implement binary rules and then how adding unique identifiers allows us to simulate larger rules (in our example ternary rules) with a set of binary rules. The result is a *higher level* communication protocol for implementing ternary rules.

8.1 Communication Protocols for Binary Rules

It is a simple matter to implement a binary rule set with a communications protocol that does not use unique identifiers. To see this, suppose that each robot can (1) store its label; (2) attach to or detach from other robots; (3) communicate with nearby robots. The main assumption we make is that robots can distinguish which other robots they are talking to using some scheme. like: "send a message to the robot on my right" or "receive a message from the robot connected to port 7".

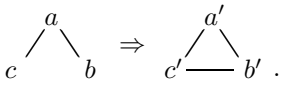
To implement a binary grammar Φ, a robot i first chooses a nearby robot j with whom to communicate and sends it a message containing $l(i)$. If i and j are attached, then robot j checks to see if the left hand side of some rule in Φ matches $l(i) - l(j)$, otherwise it checks to see if the left hand side of some rule in Φ matches $l(i)\ \ l(j)$. If robot j finds a match, it updates its label according to the rule and sends a message to i to the effect that it should change its label according to the rule. Furthermore, if the right hand side of the rule has an edge, robot j also communicates to i that they should attach to each other (or stay attached), otherwise they should detach (or not attach in the first place). Finally, if robot j does not find a rule that matches, it does not change its label and sends i a message that it should not do anything. Notice that at no time do the robots communicate the indices i and j, which are simply used above to explain the protocol.

We have implemented this communication protocol using the *Computation and Control Language* (CCL) [17] in a large scale simulation of the system described in Section 9.2. It is easily adapted to the situation where messages may be lost, as may be the case with the hardware platform we are building. We plan to report on this effort in more detail in a future paper.

8.2 Binary Rules for Ternary Rules

Using an infinite set of symbols and an infinite set of rules, we can, for example, implement ternary rules with a set of binary rules. We do this by constructing binary rules that essentially implement a multi-hop version of the original ternary rule. Note that Corollary 5.1 tells use that we cannot do this with a simple graph grammar containing a finite set of rules.

We, therefore, from the original alphabet Σ, we introduce a new, infinite set of labels of the form $x : i : S$ where $x \in \Sigma$, $i \in \mathbb{N}$ and $S \subset \mathbb{N}$. The symbol x denotes the original label from Σ and the number i denotes the unique identifier of the part. The set S consists of a set of other identifiers that have committed to executing a given ternary rule[3]. For example, suppose we wish to implement, with binary (and unary) rules, the ternary rule r defined by



(2)

We can do so with the following rules:

$$\Phi_r = \begin{cases} b{:}k{:}\varnothing - a{:}j{:}\varnothing & \Rightarrow & b{:}k{:}j - a{:}j{:}k & (S1) \\ c{:}i{:}\varnothing - a{:}j{:}\varnothing & \Rightarrow & c{:}i{:}j - a{:}j{:}i & (S2) \\ b{:}k{:}\varnothing - a{:}j{:}i & \Rightarrow & b{:}k{:}j - a{:}j{:}ik & (S3) \\ c{:}i{:}\varnothing - a{:}j{:}k & \Rightarrow & c{:}i{:}j - a{:}j{:}ik & (S4) \\ c{:}i{:}j \quad b{:}k{:}j & \Rightarrow & c{:}i{:}jk - b{:}k{:}ij & (S5) \\ a{:}j{:}ik \Rightarrow a'{:}j{:}\varnothing & & & (S6) \\ b{:}k{:}ij \Rightarrow b'{:}k{:}\varnothing & & & (S7) \\ c{:}i{:}jk \Rightarrow c'{:}i{:}\varnothing & & & (S8) \end{cases}$$

We have, in fact, specified an infinite set of rules: Eight for each triple $(i, j, k) \in \mathbb{N}^3$, with i, j, k distinct.

The set S in the new labels denotes the state of a protocol for executing r. For example, the symbol $b{:}k{:}\varnothing$ is the state of a robot k labeled by b that has not started to execute r. The symbol $b : k : j$ is the state of a robot k that has initiated execution of r by communicating with a robot j labeled by a. The symbol $b : k : ij$ denotes a robot that has proceeded through the execution of r by communicating with robots i and j labeled by a and c respectively. Notice that in the rule $(S5)$, where the edge between c and b is actually added, we require that c and b are connected to the same robot j labeled with a. This restriction is possible since the parts have unique identifiers. Without these identifiers, it could be that the robots labeled b and c are connected to different robots labeled a. It follows that for any G

$$\mathcal{R}(G, \Phi_r) = \mathcal{R}(G, \{r\}) \quad \text{and} \quad \mathcal{S}(G, \Phi_r) = \mathcal{S}(G, \{r\}).$$

This does not contradict Theorem 5.1 since, by assigning unique identifiers to robots, we violate the hypotheses of having an infinite supply of identically labeled vertices as the start graph G_0. In fact, the in the initial graph for the new system, we would suppose that each robot i is initially labeled by $a{:}i{:}\varnothing$.

There are two possible concurrent trajectories that can arise from the rules in Φ applied to the appropriate initial graph (one matching the left hand side of the original ternary rule). Either rule $(S1)$ can be applied first

[3]We use the notation ijk (or jki, etc) to represent the set $\{i, j, k\}$.

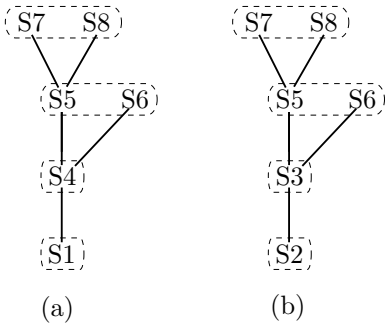


Figure 14: The partial orders associated with the two possible trajectories from Φ_r.

or rule $(S2)$ can be applied first. The resulting concurrent trajectories, shown in Figure 14 both have length four. Thus, we can conclude that *it is possible to implement a ternary rule in four concurrent binary steps*. This (the number four) is in a sense the *cost* of a ternary rule in terms of the basic currency of binary rules.

9 Physical Models of Assembly

We describe briefly several physical robotic systems that can implement graph grammars to self-organize. Each of the following sub-sections is only an overview meant to convince the reader that graph grammars can be easily employed in a variety of robotic systems. A complete description of the systems is beyond the scope of this paper, however. In the future, we plan to report on these efforts in great detail.

9.1 Self-Motive Robots

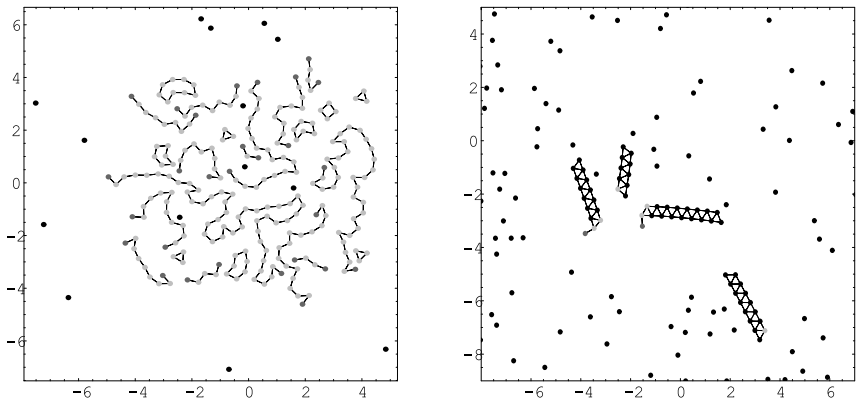
In previous work [15] we showed how to use graph grammars as a basis for multi-robot formation forming (without explicit grammatical structures). To each robot i we associate two sets: The set $\mathcal{A}_i$ of robots j such that either i is "attached" to j or the graph $l(i)\ \ l(j)$ matches a rule in Φ, and the set $\mathcal{R}_i$ of the robots where no rule matches. We then define an *artificial potential function* U_i of the form

$$U_i = \sum_{j \in \mathcal{A}_i} U_{attract}(x_i, x_j) + \sum_{j \in \mathcal{R}_i} U_{repel}(x_i, x_j)$$

where x_k is the position of robot k. The function $U_{attract}$ is defined so that $-\nabla_{x_i} U(x_i, x_j)$ has the set $||x_i - x_j|| = R$ as an attractor. The function U_{repel} is defined so that $-\nabla_{x_i} U(x_i, x_j)$ has the set $||x_i - x_j|| = 0$ as a repellor. Here R is the desired distance between "attached" robots. Each robot i simply follows the negative gradient of U_i to move toward robots in $\mathcal{A}_i$ and away from robots in $\mathcal{R}_i$. When two attracting robots come in close proximity, they execute the appropriate rule and change their states. This has the effect of changing $\mathcal{A}_k$ and $\mathcal{R}_k$, and therefore U_k, for each k.

If the number of robots is finite, then deadlock can occur in the above system due to groups forming incompatible subassemblies. We thus add the rule

$$(V, E, l) \Rightarrow (V, \varnothing, \lambda x.a)$$



(a) (b)

Figure 15: (a) Snapshots of a simulation of Equation 3 with disk-shaped parts and a complete communications protocol implementing the paths and cycles rule set from Example 4.1. The location of the part and its label (a, b or c) is suggested by the gray level used. The force due to stirring is modeled as the sum of five divergent force fields focused in different locations on the plane (which may result from jets perpendicular to the plane of the figures). The magnitudes of the components of the field oscillate out of phase. The parts start out evenly distributed and eventually form chains and cycles. (b) A snapshot of a system that builds *trusses*. The rule set that produces these structures and several other examples are described elsewhere [18].

for each $(V, E, l) \in \mathcal{C}(G_0, \Phi) - \mathcal{S}(G_0, \Phi)$. With low probability, robots randomly choose to disassociate using this rule. Note that this rule may be large. A simple consensus algorithm can be implemented by the robots in each component to decide whether to execute the rule.

9.2 Stirred Robotic Parts

We now introduce a model of "robotic" assembly that requires less capable robots, in that they do not need to be able to move themselves. Instead we suppose that large number of robotic parts *float* in a stirred fluid. Upon colliding (by chance), two parts will latch onto each other or not based on whether their current labels match the left hand side of a rule in Φ (See Figure 1). If they do latch together, then they change their labels according to the rule. A similar scheme works for mixed or destructive rules (see Section 8).

To model this system, we suppose that each part i with position x_i has a latch variable $L_{i,j} \in \{0, 1\}$ associated with every other robot j. When the parts come in close proximity, they communicate their current labels to each other in an attempt to find an applicable rule. If one is found, they both set their latch variable for the other to 1 and change their labels. Otherwise the latch variable is set to 0 and the parts bounce off each other. The dynamics of robot i are

$$m\ddot{x} = F_1(x_i, t) - c\dot{x}_i + \sum_{j \neq i} L_{i,j} L_{j,i} F_2(x_i, x_j) \tag{3}$$

where F_1 is a time varying force field that models the effect of the fluid on the part and

$$F_2(x_i, x_j) \triangleq -\nabla_{x_i} U(x_i, x_j) - b \frac{\dot{x}_i (x_j - x_i)}{||x_j - x_i||} (x_j - x_i)$$

is a damped nonlinear spring, with spring potential U, modeling the latching mechanism. We suppose that U has a minimum at $||x_i - x_j|| = R$ as before. In other work [16] we describe, for example, how such a mechanism would work using capillary forces.

We have explored the behavior of the above model in simulation using the rule systems we explored in this paper. Figure 15(a) shows snapshots of a system of parts using the rules in Example 4.1 and Figure 15(b) shows a snapshot of a system using a more complex rule set.

We are currently constructing a physical instantiation of this model with triangular "programmable parts", as described in the introduction to this paper (see Figures 2 and 3 and also so Example 4.3). Each part employs a graph grammar rule set in essentially the above fashion using a simple microcontroller mounted on its back and simple magnetic latches for attaching and detaching. The parts float on an air-table and are stirred by overhead controllable blowers. The main differences between the platform we are building and the above model are (1) that *when two parts collide, they stick to each other by default* and (2) that the parts a polygonal instead of being point-masses. Once parts are attached, they can easily communicate (via IR transceivers in our case) and implement any applicable rules, or detach if there are not any applicable rules. This obviates the need for the robots to sense their local surroundings. We will report on this effort in an upcoming paper.

10 Discussion

We have defined a class of graph grammars that describe self organizing robotic systems. We focused on the properties of the reachable and stable graphs that rule sets produce. We noted in Section 5 that acyclic rule sets (those without cycles in their right hand sides) cannot produce a unique stable cyclic graph and are thus less powerful than more general, cyclic rule sets. We then presented two algorithms to synthesize rule sets for arbitrary trees and graphs as unique stable outputs. Topological results inform these algorithms: cyclic rules are used for the general case. Using the notions of concurrency described in Section 6, we determined the number of time-steps required to assemble the desired graph using these synthesized rule sets. In Section 8 we introduced infinite rule sets that can, with binary rules, implement ternary cyclic rules. To work around Corollary 5.1, we added a unique identifier to each node, in a similar fashion to symmetry-breaking methods found in distributed algorithms [22]. Finally, we discussed how graph grammar rules can be used as a basis for robot assembly by describing two physical models appropriate to the task.

The methods we propose show that algorithms for collective tasks, here distributed self-assembly of a pre-specified structure, can be *engineered*. The model is one of controlling the local interactions of a system so that global properties result. We believe that similar methods will be applicable to other systems and may potentially provide a predictive model for distributed tasks observed in nature.

There are many natural problems in our model to be explored more completely in the future:

Assembly Complexity For any given graph G, which grammar $\Phi(G)$ assembles G as a unique stable component in the fewest number of concurrent steps in the worst case? A constructive description of Φ would be vastly preferable to the algorithm of Section 7.

Rule Set Optimization Given a collection of graphs S, find the 'smallest' rule set Φ such that $\mathcal{S}(G_0, \Phi) = S$. Smallness can be measured in terms of number of rules, sizes of rules, and/or numbers of labels required.

Other Synthesis Problems The specification in the synthesis problem is probably better described using a *temporal logic* formula F on trajectories of graphs. The formula F could describe the assembly problem we have posed here, or it could describe, for example, the ratchet in Example 4.4. The synthesis procedure, given F and an initial graph G_0 is then to find Φ so that $(G_0, \Phi) \models F$ (i.e. so that every trajectory $\sigma \in \mathcal{T}(G_0, \Phi)$ has the property F). A related problem is the *control problem*: Given F and (G_0, Φ), find additional rules Υ so that $(G_0, \Phi \cup \Upsilon) \models F$. That is, determine rules Υ that regulate the behavior of the original rules Φ so that they meet some desirable performance criterion F.

Stochastic Models The model we have described here is suitable for situations wherein interactions between parts have deterministic outcomes. This assumption is not valid in many situations of interest, for example, where the "robotic parts" are in fact single molecules. Thus, the rules in Φ need to be annotated with rates or probabilities. The result is the study of stochastic processes arising from a given grammar instead of trajectories.

Geometry Although it is easy in some cases to embed a grammar onto a physical, geometrical robot, our model is not geometrical. However, geometry is crucial in most self-assembly processes from meso-scale tile assembly to the formation of supra-molecular aggregates. Also, even if it may somehow be possible to encode a finite set of labels in the *conformation* of a part, the infinite set of identification symbols we propose to recover binary rules is untenable. However, a set of appropriately shaped parts (using a graph grammar or not) can easily form a uniquely stable, cyclic graph. We believe the interplay between symbolic descriptions of self-assembly and geometric descriptions is worth considerable study.

Physical Instantiations Finally, a main focus of our future research will be to design devices capable of executing the graph grammars, although there are at present many existing platforms (multi-robot swarms and multi-vehicle systems for example) that could easily use graph grammars for formation forming and other tasks. The use of a graph grammar in these situations would be coupled with control of motion, perhaps as suggested in Section 9. In our own labs, we have already begun to design approximately 10cm robot "programmable parts" with magnetic latches, MEMs scale parts that bind using capillary forces, and we are also exploring ways to implement grammatical systems on the molecular scale.

Acknowledgments

The first author would like to thank his students Nils Napp, Tho Nguyen, Josh Bishop, William Malone and Richard Kreisberg who helped build the prototype programmable part described in the introduction. Klavins is supported in part by NSF CAREER grant number #0347955. Ghrist is supported in part by NSF CAREER grant number DMS-0337713. Lipsky is supported in part by NSF VIGRE grant number DMS-9983160.

References

[1] A. Abrams and R. Ghrist. State complexes for metamorphic robot systems. To appear, *Intl. J. Robotics Research*.

[2] B. Berger, P.W. Shor, L. Tucker-Kellogg, and J. King. Local rule-based theory of virus shell assembly. *Proceedings of the National Academy of Science, USA*, 91(6):7732–7736, August 1994.

[3] 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.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addision-Wesley, 1988.

[5] I. Chen and J. W. Burdick. Determining task optimal modular robot assembly configurations. In *International Conference on Robotics and Automation*, 1995.

[6] G. Chirikjian. Kinematics of a metamorphic robotic system. In *International Conference on Robotics and Automation*, 1994.

[7] 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.

[8] B. Courcelle and Y. Métivier. Coverings and minors: Application to local computations in graphs. *European Journal of Combinatorics*, 15:127–138, 1994.

[9] B. A. Davey and H. A. Preistley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[10] 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.

[11] J. A. Fax and R. M. Murray. Information flow and cooperative control of vehicle formations. *IEEE Transactions on Automatic Control*, 2004. submitted.

[12] R. Ghrist and D. Lipsky. Grammatical self assembly for planar tiles. In *The 2004 International Conference on MEMs, NANO and Smart Systems*, Banff, Canada, 2004.

[13] A. Hatcher. *Algebraic Topology*. Cambridge University Press, 2001.

[14] C. Jones and M. J. Matarić. From local to global behavior in intelligent self-assembly. In *International Conference on Robotics and Automation*, Taipei, Taiwan, 2003.

[15] 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.

[16] E. Klavins. Toward the control of self-assembling systems. In A. Bicchi, H. Christensen, and D. Prattichizzo, editors, *Control Problems in Robotics*, pages 153–168. Springer Verlag, 2002.

[17] E. Klavins. A language for modeling and programming cooperative control systems. In *Proceedings of the International Conference on Robotics and Automation*, New Orleans, LA, 2004.

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

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

[20] K. Kotay, D. Rus, M. Vona, and C. McGray. The self-reconfiguring robotic molecule: Design and control algorithms. In P. Agrawal, L. Kavraki, and M. Mason, editors, *Algorithmic Foundations of Robotics*. A. K. Peters, 1998.

[21] 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.

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

[23] A. Nguyen, L. Guibas, and M. Yim. Controlled module density helps reconfiguration planning. In *Workshop on the Algorithmic Foundations of Robotics*, Dartmouth, NH, March 2000.

[24] V. Pratt. Modeling concurrency with geometry. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, 1991.

[25] W. Reisig. Partial order semantics versus interleaving semantics for csp-like languages and its impact on fairness. In J. Paredaens, editor, *Automata, Languages and Programming*, volume 172 of *Lecture Notes in Computer Science*, pages 403–413. Springer, 1984.

[26] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, 1985.

[27] G. Rozenberg. Personal Communication.

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

[29] K. Saitou and M. Jakiela. Automated optimal design of mechanical conformational switches. *Artificial Life*, 2(2):129–156, 1995.

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

[31] Y. S. Smentanich, Y. B. Kazanovich, and V. V. Kornilov. A combinitorial approach to the problem of self assembly. *Discrete Applied Mathematics*, 57:45–65, 1995.

[32] R. L. Thompson and N. S. Goel. Movable finite automata (MFA) models for biological systems I: Bacteriophage assembly and operation. *Journal of Theoretical Biology*, 131:152–385, 1988.

[33] H. Wang. Notes on a class of tiling problems. *Fundamenta Mathematicae*, pages 295–305, 1975.

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

[35] E. Winfree. Algorithmic self-assembly of DNA: Theoretical motivations and 2D assembly experiments. *Journal of Biomolecular Structure and Dynamics*, 11(2):263–270, May 2000.