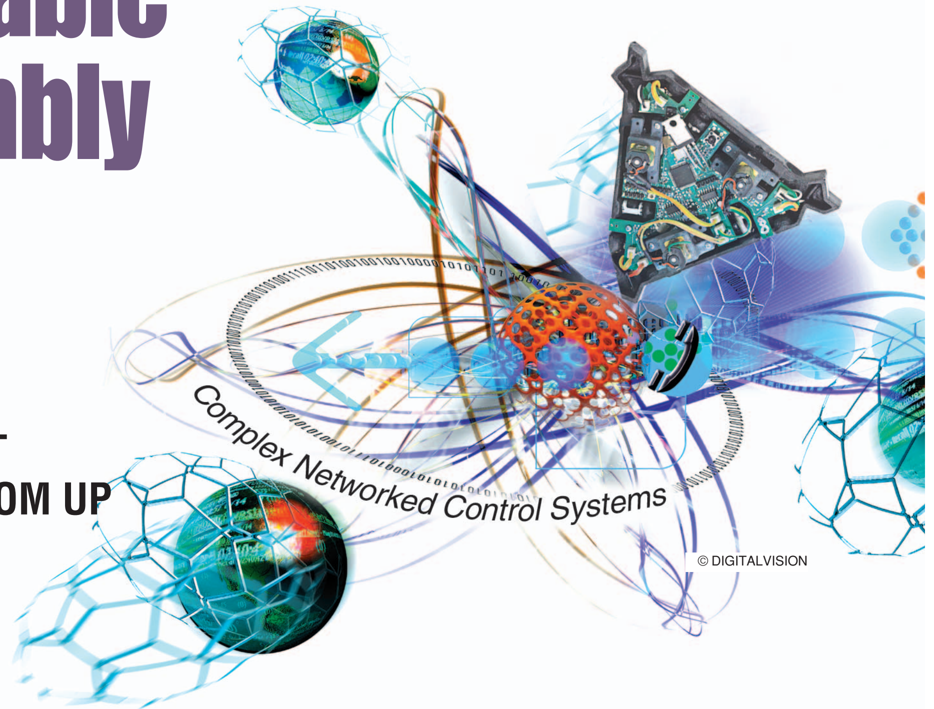
Programmable Self-Assembly

ERIC KLAVINS

CONTROL OF CONCURRENT SYSTEMS FROM THE BOTTOM UP



© DIGITALVISION

Self-assembly is the phenomenon in which a collection of particles spontaneously arrange themselves into a coherent structure. Self-assembly is ubiquitous in nature. For example, virus capsids, cell membranes, and tissues are self-assembled from smaller components in a decentralized fashion. Self-assembly is beginning to find its way into engineering, through various technologies ranging from molecular [1], [2] to robotic [3], [4].

Self-assembly comes in two modes, *passive* and *active*. In passive self-assembly, particles interact according to their geometry or surface chemistry and tend toward a thermodynamic equilibrium in which the particles are assembled. For example, phospholipids stick to each other along hydrophobic regions to form membranes. In active self-assembly, each particle may expend energy to accept some interactions with other particles while rejecting others, according to a program. Examples range from proteins in cells, whose conformational switching patterns may determine the order in which they interact with other molecules, to multirobot systems, where small groups of robots determine the outcome of encounters according to their internal programming.

In this article we consider the task of programming active self-assembling and self-organizing systems at the level of interactions among particles in the system. To demonstrate the approach, we use it to control an experimental system called the *programmable parts testbed* (PPT), shown in figures 1, 2, and 3. We also consider several illustrative examples, including polymerization, a model of a molecular ratchet, and a cooperative control scenario. In all of these systems, we provide each particle or robot with a local interaction rule book called a *graph grammar*.

A graph grammar can be used to model the physics of the particles by describing the outcomes of interactions among them, and it can be used to program the desirable outcomes of interactions among particles. In the latter setting, a grammar is a description of a communication protocol and is thus intended to be coupled with a physical model of the environment that mediates the interactions. In particular, a suitably designed grammar can precisely describe and direct the changing network topology of a self-organizing system.

The main questions in the area of programmed self-organization concern the ability to engineer the global behavior of a system by means of local rules. In a wide variety of settings, we can design local rules that yield a specified behavior, and we can reason about the correctness of the result. In some circumstances, we can provide algorithms that automatically generate such a set of rules.

Recent results in diverse areas [1], [5], [6] indicate that the emergent behavior of a self-organizing system can be precisely predicted and controlled, although there is much work to be done to understand the physics, dynamics, and implementation of self-organization. Progress in this area promises to usher in a new era of bottom-up engineering of systems ranging from programmable nanoscale molecular machines to controlled swarms of interacting autonomous robots.

THE PROGRAMMABLE PARTS

In robotics, self-organization is explored with increasing success using modular robots [7], which are small

1066-033X/07/$25.00©2007IEEE

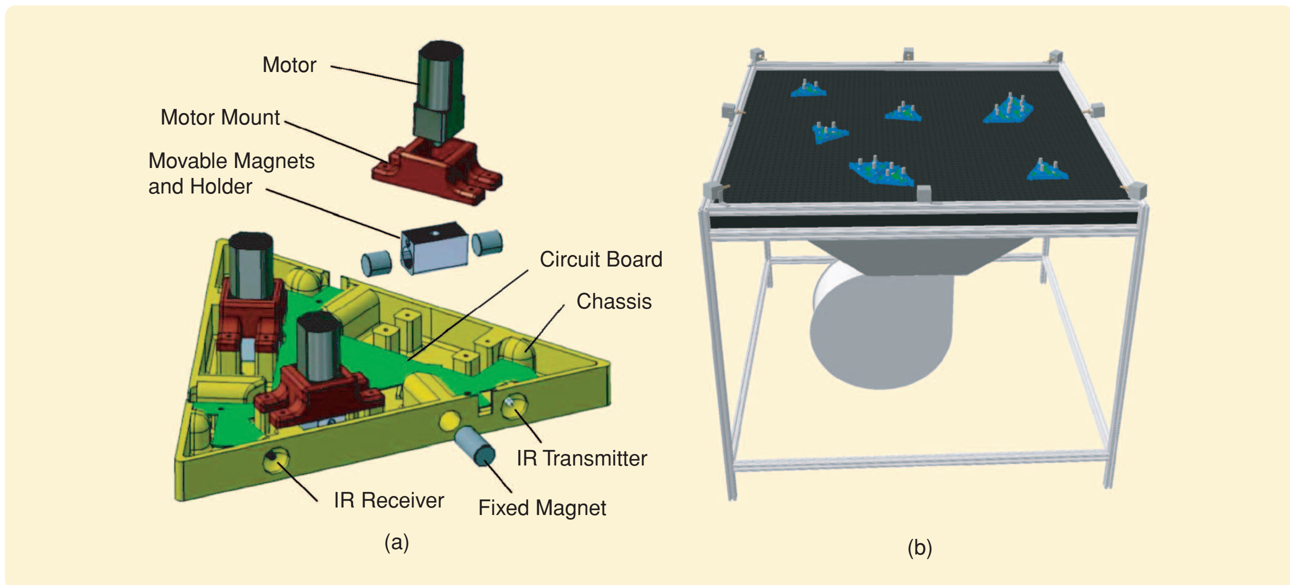


FIGURE 1 The programmable parts testbed (PPT). (a) A diagram of a programmable part showing the latching mechanism. Parts occasionally bind upon chance collisions. If the parts later jointly decide to detach, motors rotate permanent magnets to repel the parts. The latch requires power only when switching. (b) A schematic of the laboratory setup. The parts float on an air table and are randomly stirred by air jets to induce collisions. An overhead video camera (not shown) is used to collect data.

electromechanical devices that can latch on to each other, rotate or translate with respect to each other, and communicate with each other by means of peer-to-peer communication devices. Through local interactions, a group of modular robots can reconfigure into a variety of shapes, repair itself [8], and even self-replicate [9], [10].

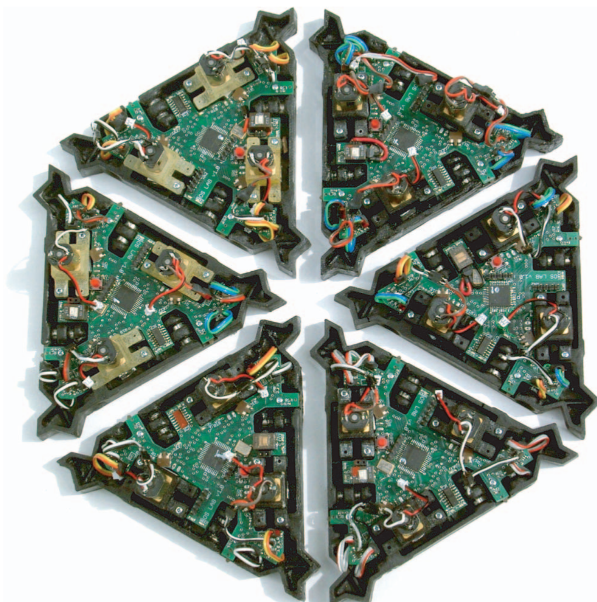


FIGURE 2 A photo of six programmable parts. Each part is 12.5 cm on a side. The robots are programmed with a set of local interaction rules called a graph grammar. Unlike nanoscale systems, the behavior of the programmable parts can be observed directly. PPT provides a rich, reconfigurable, and complex system for investigating and demonstrating ideas in engineered self-assembly.

PPT [4] is well suited to implementing the ideas described in this article. PPT consists of about 20 small robots, called *programmable parts*. The parts float passively on an air table and are randomly stirred by air jets. The idea is to emulate a well-mixed chemical reaction wherein each programmable part is programmed to behave like a specific kind of molecule.

Each programmable part consists of an equilateral-triangle-shaped chassis that supports three controllable latching mechanisms, three infrared transceivers, a microcontroller, and control circuitry. Each edge of the chassis supports a latching mechanism and a transceiver. Each latching mechanism consists of two permanent magnets, one fixed and the other mounted on the end of a small geared dc motor. The north pole of the fixed magnet and the south pole of the movable magnet point outward in the default configuration. Thus, when two programmable parts collide edge to edge, the latching mechanisms bind them together. At that point, the two programmable parts exchange information with each other by means of the infrared transceivers on their bound edges. If the two programmable parts mutually decide to detach from each other, each programmable part temporarily rotates its movable magnet 180°, forcing the programmable parts apart. The movable magnets then return to their default positions. Both programmable parts must coordinate to break the bond. Figure 4 summarizes the manner in which programmable parts interact.

When two programmable parts bind and communicate their internal states as described above, they may update the internal states stored in the memories of their microcontrollers, and they may choose to stay bound or detach. The resulting global behavior depends on their programming. As shown in Figure 3, one program might result in assemblies of a certain shape, while another program might result in assemblies of a different shape. Furthermore, high-energy collisions between robots may cause an assembly to break apart. The engineering goal is to program the particles so that a desired structure emerges with high probability, despite the stochastic nature of the system. To address this goal, we use *graph grammars* to both model and program the ways in which programmable parts interact.

GRAMMARS MODEL DYNAMICAL SYSTEMS

We explain graph grammars [11] informally by showing how to build a graph grammar that models polymerization, the process by which particles form into chains. In particular, suppose we wish to model a collection of interacting particles, wherein each particle can be bound to zero, one, or two other particles. To keep track of the configuration of a particle, let a denote the state of a particle having no bonds, let b denote the state of a particle having one bond, and let c denote the state of a particle having two bonds. The dynamics of the system are as follows. If two particles in state a interact, they bind to each other and both of their states change to b. If one particle in state a and another in state b interact, they bind to each other and their states change to b and c, respectively. Finally, if two particles in state b interact, they bind to each other and both of their states change to c. No other interactions lead to a new bond. As in [12], we use the term *label* for the symbols a, b, and c, saying, for example, that particle i *is labeled* by a.

We represent the interaction rules describing the polymerization example with the graph grammar

$$\Phi = \begin{cases} a \quad a \rightharpoonup b - b & (r_1), \\ a \quad b \rightharpoonup b - c & (r_2), \\ b \quad b \rightharpoonup c - c & (r_3). \end{cases} \qquad (1)$$

Rule r_1, for example, states that two particles labeled a and that are not attached to each other can attach to each other, in which case they change their labels to b.

For now, we suppose that each particle is initially labeled a, that time proceeds in discrete steps, and that bonds form instantaneously. Finally, the choice of which two particles interact at any step is nondeterministic, resulting in a set of allowable trajectories, as opposed to a single trajectory or distribution over trajectories. Later we describe a stochastic interpretation of local rules that refines the notion of a trajectory. The first several steps of one possible trajectory in the polymerization example, starting with a collection of nine particles, is shown in Figure 5.

The graph grammar formalism is treated formally in [5] and [13]. However, with the above example in mind, we describe the main definitions. The state of the system is described by an assignment of labels to particles and by a set of connections between particles. This description corresponds to a mathematical object called a *labeled graph* [12], [14]. Specifically, a labeled graph has the form

$$G = (V, E, l),$$

where the *vertex set* V is a finite or countably infinite set of positive integers used to index the nodes (representing robots or particles, for example) of the system, the *edge set* E is a set of unordered pairs from V, and the *labeling function* $l : V \to \Sigma$ is a function associating with each particle $i \in V$ a label $l(i) \in \Sigma$, where $\Sigma = \{a, b, c, \ldots\}$ is the set of

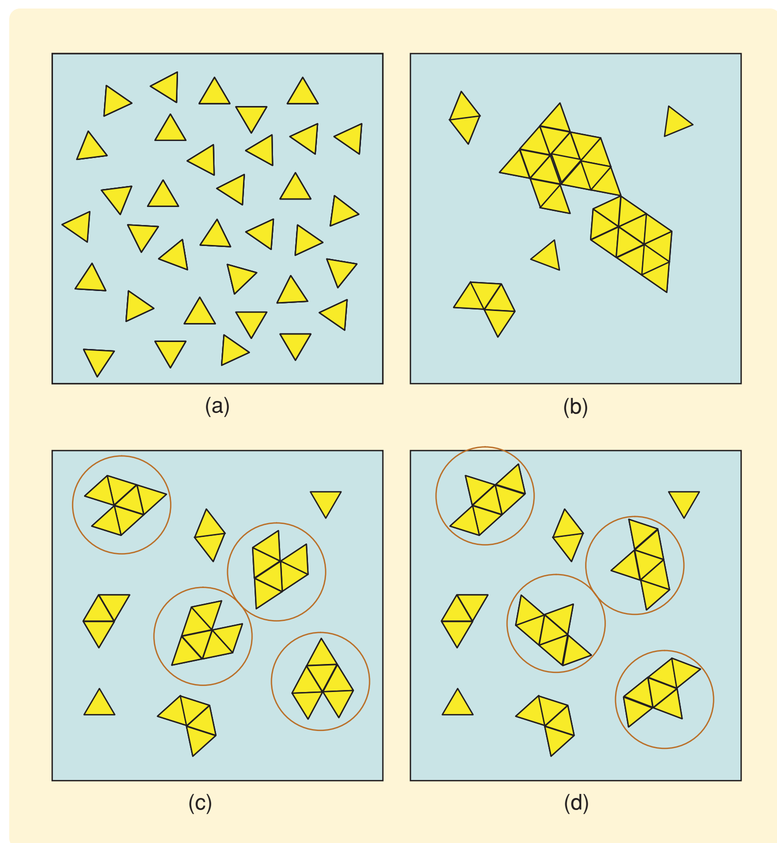


FIGURE 3 The reconfigurability of the programmable parts testbed. The programmable parts can be programmed to self-assemble into a specified structure by means of a graph grammar. (a) Initially, the programmable parts are not assembled and have identical internal states. (b) With no programming, the parts aggregate into a crystalline structure whose overall shape cannot be precisely predicted. (c), (d) Different graph grammars result in the predictable emergence of structures.

labels. The vertices, edges, and labels of a labeled graph G are sometimes denoted V_G, V_E, and V_l when more than one graph is under consideration. We draw labeled graphs using only labels and edges, as in the boxes in Figure 5. Since rules do not identify the indices on which they operate, we consider labeled graphs with the same label and edge structure as equivalent. In graph theory, such graphs are said to be *isomorphic*.

Formally, a rule $L \rightharpoonup R$ such as r_1 in (1) is a pair L and R of labeled graphs having the same vertex set. For example, the rules r_1, r_2, and r_3 in the polymerization example (1) are described using graphs with two vertices. Rules need not involve exactly two vertices, however, and the examples in the next section use rules involving several vertices. However, rules involving a small number of particles provide better models of local interactions, since, in many of the systems we consider (such as PPT), it is unlikely that a large number of particles could be considered to be local to each other.

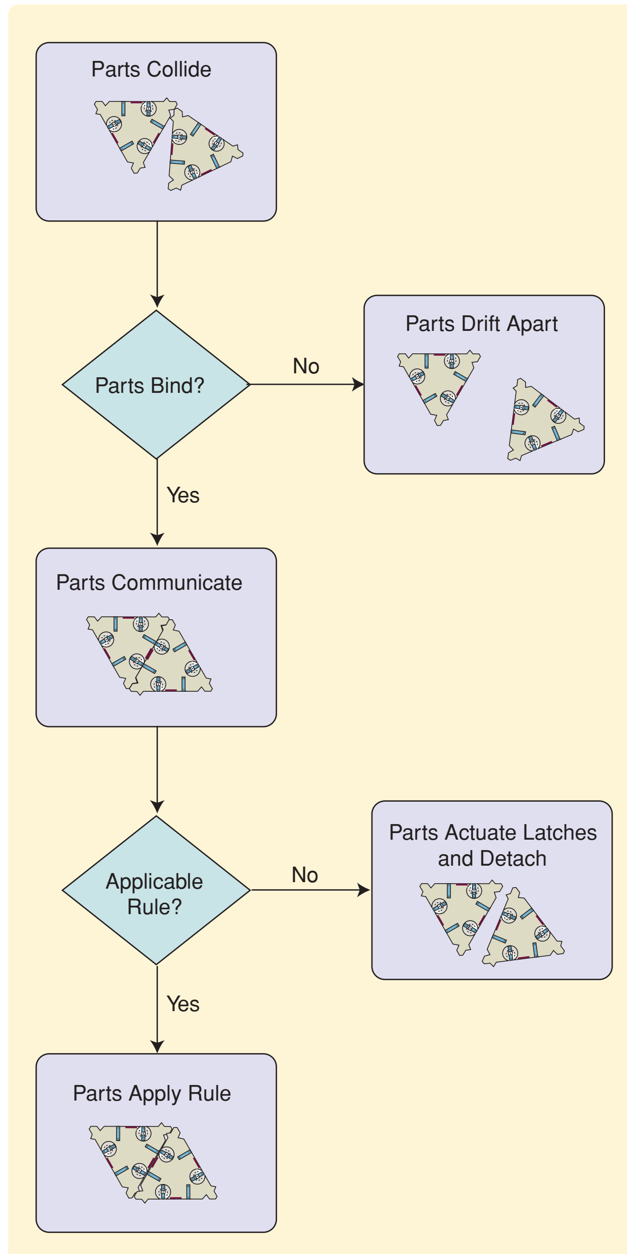


FIGURE 4 How local rules are interpreted by the programmable parts. When two randomly stirred programmable parts collide, they may bind to each other. Once bound, the parts communicate their current label, stored internally in the memory of their microcontrollers. For each rule in the graph grammar, each part then checks whether the rule applies to the shared labels. If a rule does apply, the programmable parts change their labels according to the rule. Otherwise, the programmable parts detach from each other by temporarily rotating their permanent magnets.

Rule application leads to dynamic behavior. A rule r of the form $L \rightharpoonup R$ is applicable to a graph G_k representing the state of the system at step k if there is a *copy* of L present somewhere in G_k. The injective edge- and label-preserving map $h: V_L \to V_{G_k}$ taking L to a copy of L in G_k is used to define where in G_k the rule r is applied. The pair (r, h) is a *rule application*. We write

$$G_k \xrightarrow{r,h} G_{k+1}$$

to indicate that G_{k+1} is obtained from G_k by replacing the copy of L in G_k identified by h with a copy of R.

A *completely connected subgraph* of a graph representing the state of the system describes an *assembly* of particles. In the polymerization example (1), assemblies of various types are self-assembled by the rules in Φ. For example, in the last box of Figure 5, a cycle of six particles and a chain of three particles are formed. An assembly is *reachable* if there is a trajectory

$$G_0 \xrightarrow{r_{i_1},h_1} G_1 \xrightarrow{r_{i_2},h_2} \cdots \xrightarrow{r_{i_k},h_k} G_k \tag{2}$$

such that the assembly occurs as a completely connected subgraph of G_k. Furthermore, an assembly is *stable* if no rules in Φ can alter it. An inductive argument shows that, starting with an infinite number of particles labeled a in the polymerization example (1), every chain is a reachable assembly and every cycle of three or more particles is a stable assembly.

SELF-ASSEMBLED GRAPHS

A key question is whether or not it is possible to define a grammar such that the only stable assemblies produced are isomorphic to a prespecified graph, supposing that the initial state of the system is an infinite graph with no edges and all of whose particles are labeled a. This problem is the *self-assembly problem*. It is shown in [5] that for all graphs there exists a grammar such that the only stable, reachable assembly of the grammar is the given graph. Specifically, for an acyclic graph T, there exists a grammar Φ_T each of

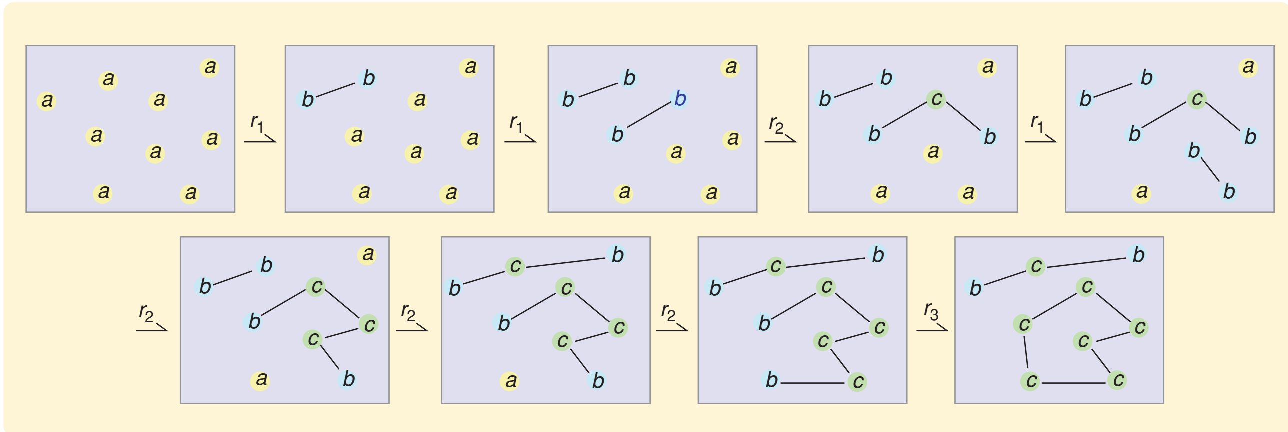


FIGURE 5 A trajectory of the polymerization system. The states of the system are labeled, undirected graphs. A transition can occur if the left-hand side of a rule matches a subgraph, in which case the subgraph is replaced by the right-hand side of the rule.

whose rules involves at most two vertices (*binary rules*), such that T is the only stable, reachable assembly. Furthermore, for an arbitrary graph C, possibly having cycles, there exists a grammar Φ_C each of whose rules involves at most three vertices (binary or *ternary* rules), such that C is the only stable assembly arising in trajectories of Φ_C.

An algorithm for generating these grammars is described in [5]. Unfortunately, rules involving three particles cannot be implemented in PPT, since three-way communication is not possible. However, the need for ternary rules is a fundamental limitation of local-rule-based systems, as shown next.

LIMITATIONS OF LOCAL RULES

A binary rule, which involves two particles, is a basic building block in a system described by local rules. In a peer-to-peer communication protocol, rules involving more than two particles must be implemented by means of a sequence of binary communications.

In fact, there exist assembly tasks that are impossible to perform using binary rules. In [15] it is shown that there exist classes of graphs, such as the *planar* graphs, that cannot be recognized by the nodes in the graph no matter what set of local rules is used. That is, if a set of particles, all with the same initial label, are connected together into a particular graph, there does not exist a set of local rules that the particles can use to determine whether nor not the graph they inhabit is, for example, planar.

To get a flavor for why determining global topology is difficult, consider the problem of making a length-three cycle the only stable assembly of a grammar. Suppose that the binary rule

$$w \quad x \rightharpoonup y - z \tag{3}$$

is used by a grammar to close a reachable length-three path into a length-three cycle. Suppose that particles i and j are labeled w and x, respectively, before the rule is used. These particles need not be in the same assembly but may each be a part of a different copy of the length-three path. Thus, rule (3) could be used twice to combine two length-three paths into a length-six cycle, which is not the desired assembly; see Figure 6. This phenomenon is related to impossibility results in distributed systems [16, ch. 3].

As shown in [5] this phenomenon is general and applies essentially to any rule set whose rules do not contain cycles in their right-hand sides. This fact might seem a bit strange. For example, servers on the Internet can determine their interconnection topology, However, servers have unique identifiers given by IP addresses, which they use to distinguish each other. In our examples we assume that the particles involved should be numerous and simple, so the luxury of IP addresses is not available. Often this problem can be resolved with the particle geometry, which is not

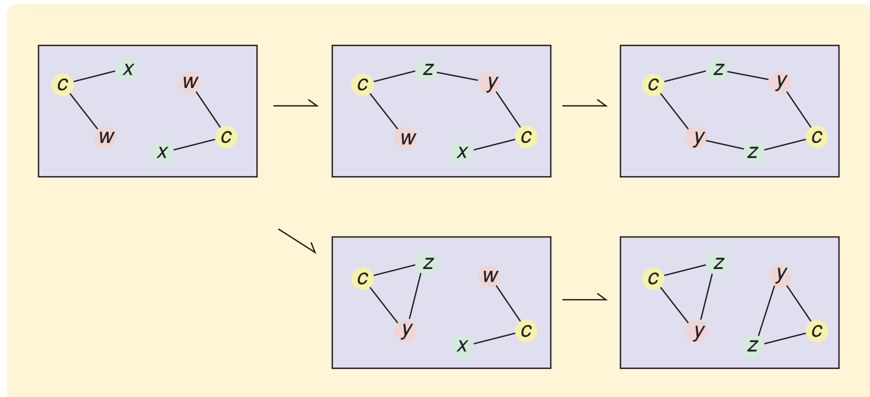


FIGURE 6 A limitation of local rules. The rule $w \; x \rightharpoonup y - z$ can be used to close a path into a cycle, or to attach two paths into a longer cycle. No binary rule set can distinguish between these situations unless the parts have unique identifiers.

modeled by a graph grammar, to resolve ambiguities. For example, in PPT, the programmable parts cannot form cycles of any length because the parts are triangular.

MODELING CONCURRENCY

A *concurrent system* is one in which, at any given time, there are multiple subsystems simultaneously operating independently of each other. As time progresses, subsystems might join or split, changing the coupling relationship. Concurrency is a crucial part of modeling large systems.

In graph grammars, concurrency is modeled by the *commutativity* of rule applications. For example, consider the first two steps in the trajectory shown in Figure 5. The particles involved in the first rule application are distinct from the particles involved in the second rule application. Thus, it does not matter which rule application occurs first, since all rules effectively occur concurrently.

Concurrency leads to a notion of time different than that encountered in standard dynamical systems. When a set of rule applications is pairwise commutative, the rule applications in the set might occur concurrently. When a set of rule applications is not commutative, however, the rule applications must occur serially. As an example, consider the number of concurrent steps required to construct a chain of length n using binary rules. In the first step, we could concurrently add edges between particles $2i+1$ and $2i+2$, for each i from 1 to $n/2$. In the second step, we could concurrently add the remaining edges. The process takes two steps; see Figure 7(a). On the other hand, consider the number of concurrent steps, using binary rules, required to construct the *star graph* with n edges shown in Figure 7(b). Any binary-rule grammar that accomplishes this task must take at least n steps, since the center particle must be involved in each rule; see Figure 7(b). Thus, while the notion of trajectory shown in (2) describes the assembly process, it does not capture the degree to which the assembly process can be parallelized. In [5] we describe a formulation of concurrency that distinguishes between concurrent and serial executions.

The minimum number of concurrent steps required to assemble a graph, which is a property inherent to the graph, is the *worst case assembly complexity*. Related complexity measures can be defined, such as the minimum number of rules needed, or the minimum number of labels.

USING GRAMMARS TO REPRESENT DISTRIBUTED ALGORITHMS

A grammar can also be thought of as a distributed algorithm. In this interpretation, we suppose that a physical process initiates the interaction between particles and that the local rules used by the particles determine what the outcome of each interaction should be. In this section, we describe several such examples.

Programming the Programmable Parts

We associate a vertex with each latch on each programmable part and permanently connect vertices with edges to indicate that latches are physically attached to each other by the chassis of the robot. The state stored by the microcontroller on a programmable part 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 by rotation only. For example, $(a, b, c) = (b, c, a)$. By convention, we require that the embedding of the cycle representing a programmable part into $\mathbb{R}^2$ be counter-

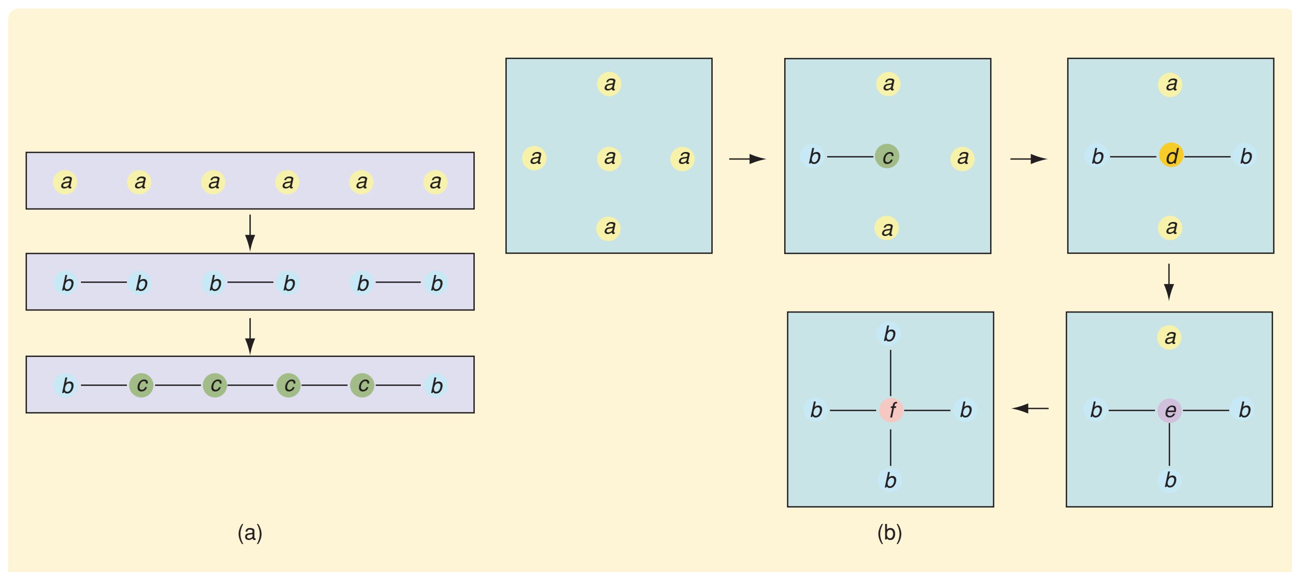
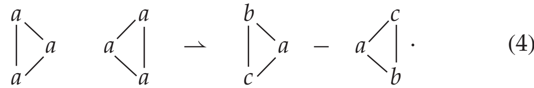


FIGURE 7 Concurrent rule application. (a) Every chain can be self-assembled with binary rules in two concurrent steps, as illustrated with a six-particle chain in this example. (b) Assembling a star graph with binary rules is necessarily a serial process. In this example, a star graph with five particles requires four steps.

clockwise. Since each programmable part is confined to motion in the plane, this orientation does not change as the parts assemble and disassemble.

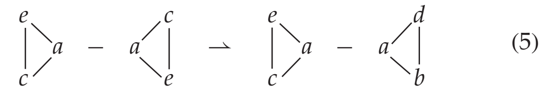
When two programmable parts initially bond, they compare their triples against the stored grammar to determine the result of the interaction; see Figure 4. As an example, consider the system defined by the rule



Suppose that a set of programmable parts is initialized so that each part has state (a, a, a) and that their interactions are governed by rule (4) only. When two parts labeled (a, a, a) collide, their states match the left-hand side of the rule, so they remain attached, changing their states to (a, b, c) to associate the latch involved in the connection with the label a and the remaining latches with the labels b and c, located counterclockwise from the active latch. The result is a *dimer* assembly consisting of two programmable parts as in Figure 8(a). When this rule is the only programming used, dimers are the only stable, reachable assembly type. The notion of a stable, reachable assembly refers only to assemblies reachable by means of programmed rules. In the discussion below, we include rules that model decay events as well, but these rules are not considered to be part of the program.

As a more substantial example, consider the problem of building hexagons by programming rules that form dimers, rules that direct dimers to form a *tetramer*, and rules that direct a tetramer and a dimer to form a hexagon. Figure 8 summarizes the main events in the assembly of a hexagon in this manner. Note that, at each step, only two programmable parts need to communicate.

Subtleties in designing rule sets for the programmable parts arise. First, we use rules of the form



to direct a programmable part to send information to its neighbors when it attaches to a new assembly. These rules do not add new edges to the underlying graph but rather change labels. For example, in the construction of a hexagon, there are three possible types of tetramers that can form; see Figure 9. One tetramer is useful, while the other two are not. By checking the embedding of the labels

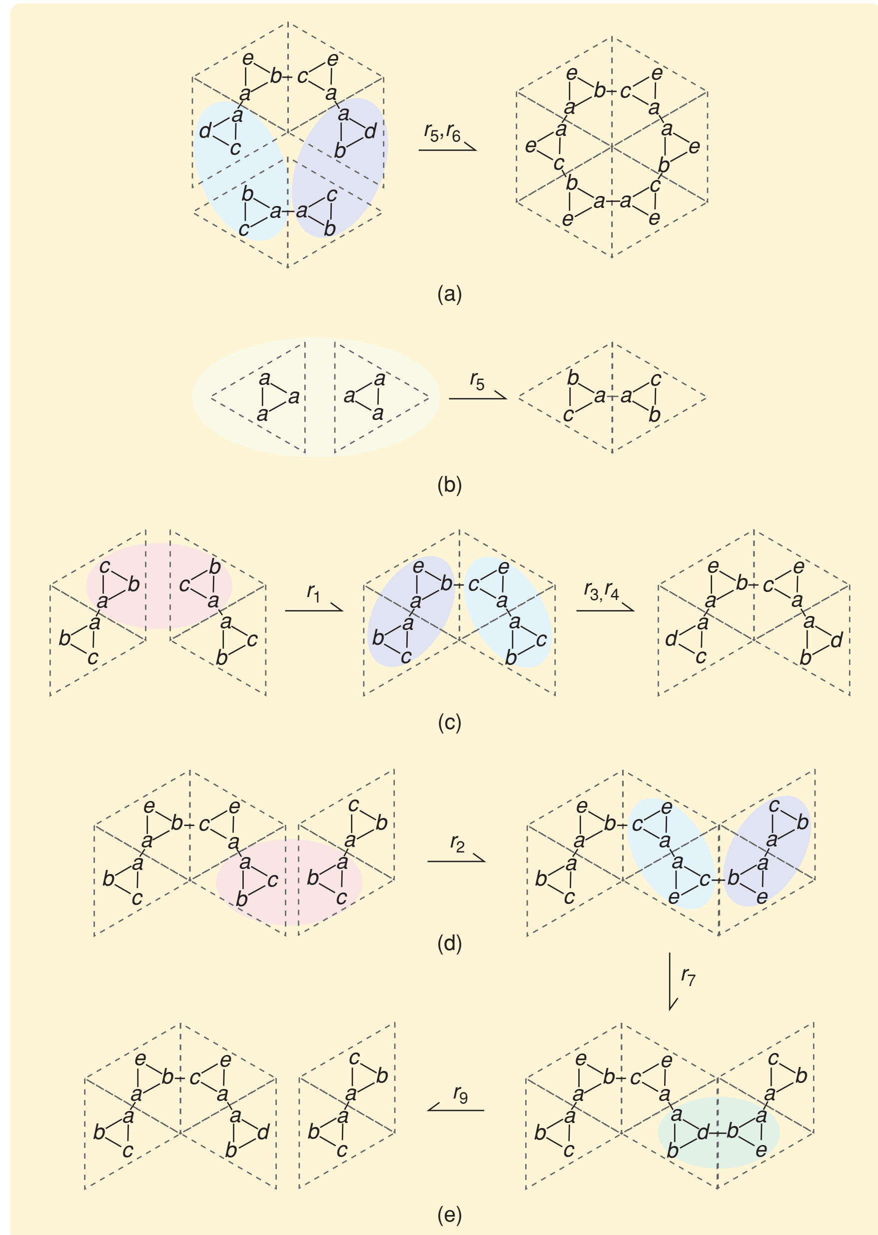


FIGURE 8 The steps in the self-assembly of a hexagon using the dimers-first rules. (a) One rule is used to update the labels of two parts when they form a dimer. (b) Three rules are used to update the labels when two dimers form a tetramer. (c) When a dimer and a tetramer combine, several rules are used to check whether or not a hexagon has been formed. If a hexagon has not formed, the new bond is rejected. (d) Two rules are used to update labels when a dimer and a tetramer combine to form a hexagon.

locally using a rule such as (5), the programmable parts can detect this situation. Second, in our system, it is possible that an assembly breaks apart due to, for example, a high-energy collision with another assembly. We can model this aspect of the environment with a rule that deletes an edge. When an edge is spontaneously deleted in this fashion, however, the labels on the programmable parts may reflect an incorrect assembly state. Therefore, we include rules that direct the robots to recover from breaks by changing their labels to reflect their new situation. For example, if a dimer breaks, the programmable parts involved reset their labels to (a, a, a). The details of the hexagon grammar are described in [17].

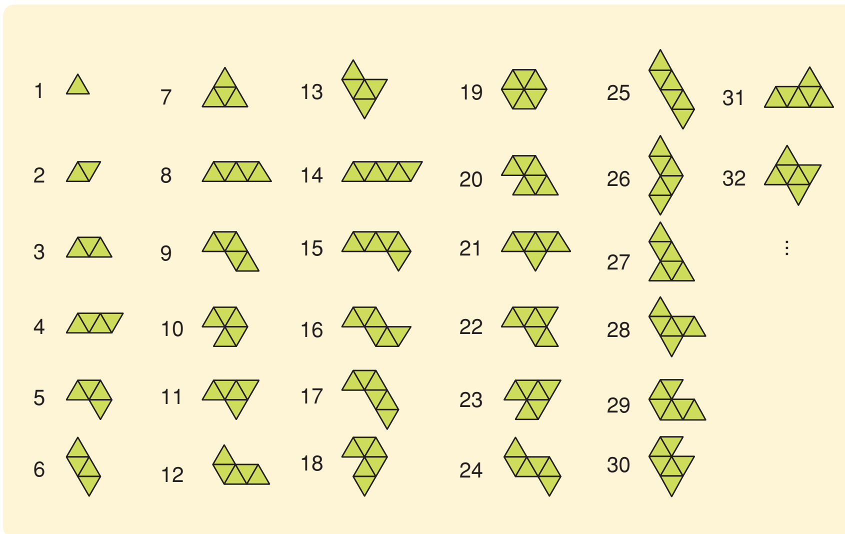


FIGURE 9 The indexing scheme used to identify assemblies of programmable parts. Assemblies are listed by size, then shape. Assembly 19, for example, is the hexagon C_{19}, the construction of which is addressed in this article.

Cooperative Control

In [18] and [19] the notion of a graph grammar is extended to include a time-varying embedding of the graph into a real-valued state space wherein the individual particles can control their own motion, instead of floating randomly as in PPT. The result is a hybrid model of distributed cooperative control, called an *embedded graph grammar*. We describe this approach by means of an example.

Consider a system consisting of three types of autonomous agents, namely, scouts, intruders, and bases. The setting is a large playing field. Each scout explores the field in a distributed fashion, looking for intruders using a motion controller to direct its motion, instead of floating randomly as with PPT. When a scout finds an intruder, it chases the intruder and attempts to recruit additional scouts to help. When three scouts are chasing an intruder, it is considered captured. The scouts then transport the intruder to a base and resume their exploration. Figure 10 shows several snapshots of how the system might evolve.

The graph grammar rules shown in Figure 11(a) describe how the scouts explore, chase, capture, and deliver an intruder. The left-hand side of each rule specifies what structure the labels and interconnections of a group of agents must have for the rule to be applied. The right-hand side specifies the result of the rule in terms of new labels and interconnections. In this example, an edge means

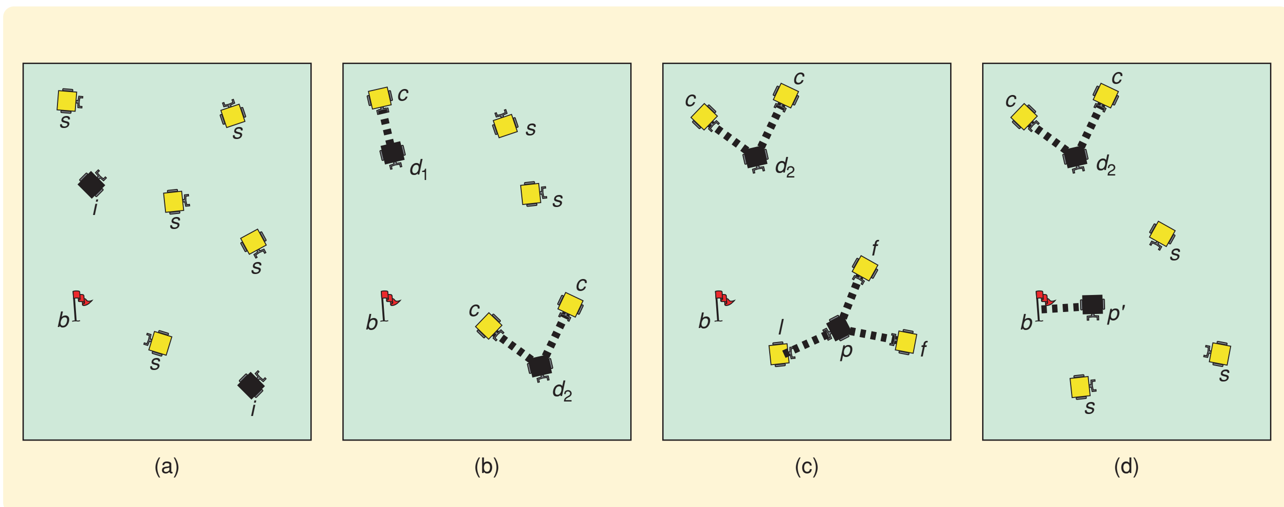


FIGURE 10 Snapshots of the scouts-intruders scenario. (a) Initially, all scouts are searching (label s) for intruders (label i). (b) Scouts begin chasing (label c) intruders, labeling them as partially detained (label d_k). (c) When an intruder is surrounded by three scouts, the intruder becomes a prisoner (label p), the scouts elect a leader (label l), and the remaining robots become followers (label f). (d) The leader escorts the group to a base (label b) to deposit the prisoner and allow the scouts to continue searching.

The main questions in the area of programmed self-organization concern the ability to engineer the global behavior of a system by means of local rules.

that the motion of the connected robots is coupled. See [18] for a description of the motion controllers used by the agents.

The rules in Figure 10(a) are sufficient to control the system in some, but not all, situations. Deadlock can occur when there are enough intruders to attract all of the scouts without having more than two scouts per intruder. Inclusion of the rules listed in Figure 10(b) results in a system wherein all intruders are eventually captured. This result is proved using an integer-valued Lyapunov function whose value is the number of intruders. Under the assumption that the search algorithm used by the scouts results in their eventually finding an intruder if one exists, it can be shown that the value of the Lyapunov function is decreasing or zero.

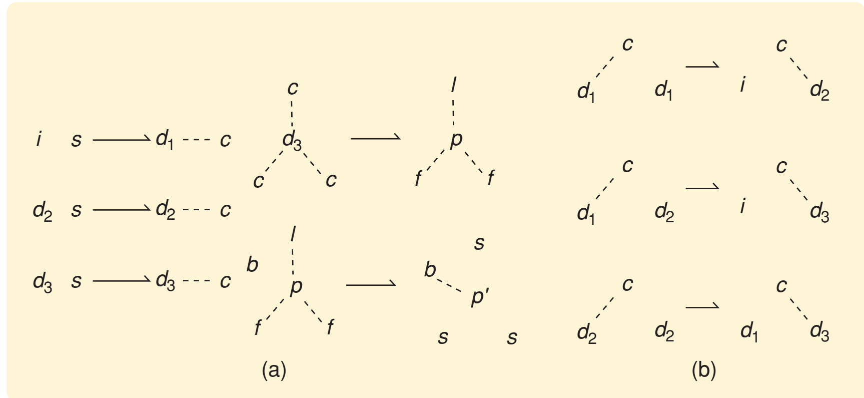


FIGURE 11 Rules for the scouts-intruders scenario. (a) The first set of rules defines the local interactions that specify how scouts chase and capture intruders. (b) The addition of the second set of rules allows scouts to switch from chasing one intruder to chasing another, which prevents deadlock.

Associated with each agent is a mapping from its label and local topology to a continuous motion controller. These controllers specify, for example, a leader-follower law for the situation in which an agent is chasing an intruder. To integrate the graph grammar formalism with the motion controllers, we augment [18] the notion of a rule with *guards*, which are local, geometric conditions that specify when the rule can be applied. Reasoning about such systems uses control-theoretic tools based on continuous Lyapunov functions as well as progress and safety arguments, such as found in the concurrency literature [20].

A Model of a Molecular Motor

A molecular motor, such as *myosin* [21], moves in a ratchet-like fashion along a molecular substrate by means of chemical bonds that connect and disconnect from the substrate according to the motor's changing conformation. We model such a system using a graph grammar. The main point of this example is to show that a graph grammar can describe cyclic behaviors. A simplified model of a molecular motor is given by the rules

$$\Phi_3 = \begin{cases} a - c \rightharpoonup d \quad e, & (r_4) \\ e \overset{b}{\diagup\diagdown} g \rightharpoonup c \overset{b}{\diagup\!\!\!-\!\!\!\diagdown} h, & (r_5) \\ b - h \rightharpoonup f - b, & (r_6) \\ d - f \rightharpoonup g - a, & (r_7) \end{cases}$$

A trajectory of Φ_3 is shown in Figure 12. The sequence starts with a cycle of particles labeled a, b, and c attached to a chain of particles labeled g, which form the substrate. The rule set directs the particle to move along the substrate, without allowing it to move backwards. 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. In particular, this rule forces the vertex labeled e to attach to the next g in the sequence. When the substrate of vertices labeled g is infinite or circular, there are no stable assemblies. All of the reachable assemblies are isomorphic to those shown in Figure 12.

THE STOCHASTIC INTERPRETATION OF GRAPH GRAMMARS

When reasoning about the behavior of a graph grammar, we consider all possible trajectories. For example, by showing that, along every trajectory, no star with six particles can form, we obtain a result about every possible behavior, called a *safety assertion*. Safety assertions can be proved by induction on the number of rule applications.

In contrast, a *progress assertion*, which states that a given assembly forms in every trajectory, is more difficult. Often it is the case that desired progress assertions are not true. For example, although we may want a cycle of exactly length five to eventually form in every trajectory of the polymerization example (1), length-five cycles form in some trajectories but not in others. Furthermore, in many of the applications we consider, rules are applied with probabilities that arise from the physical properties of the system. In

this context, we want to determine the expected number of a desired assembly present when the system is equilibrated.

PPT can be modeled as a reaction-diffusion system [4]. We use a simulator, called the *microstate simulator*, which simulates PPT using mass, velocity, friction, collisions, mixing forces, contact forces between programmable parts, and binding forces. Using the microstate simulator, the rates at which various assemblies form and break naturally due to the "thermal" agitation of the air table as well as random collisions can be estimated. Figure 13 lists several reaction rates.

The reaction rates capture the essence of the dynamics. Consider the task of building a chain of four programmable parts, specifically, assembly C_4 or C_6 in the indexing scheme of Figure 9. In particular, consider the following grammars for this task:

1) Φ_1: build chains one by one
2) Φ_2: build dimers and then chains of length four from dimers
3) Φ_3: allow all interactions, even those that build assemblies containing a hexagon. When a superstructure containing C_4 or C_6 occurs, detach superfluous subassemblies.

Figure 14(a) shows the results of using the microstate simulator with each grammar. We use ten programmable parts, an average kinetic energy of 5×10^{-4} J, and a density of 5 parts/m^2. Each curve represents the number of assemblies of type C_6 averaged over 15 trajectories. Figure 14 shows that Φ_3 substantially outperforms the remaining grammars in terms of the average number of product assemblies at the end of the simulation and in terms of the initial rate at which product assemblies are formed.

The data indicate that the grammar used to construct a given assembly has dramatic effects on how quickly the assembly forms. We next describe an approach to improving the performance of a reaction-diffusion system governed by graph grammar.

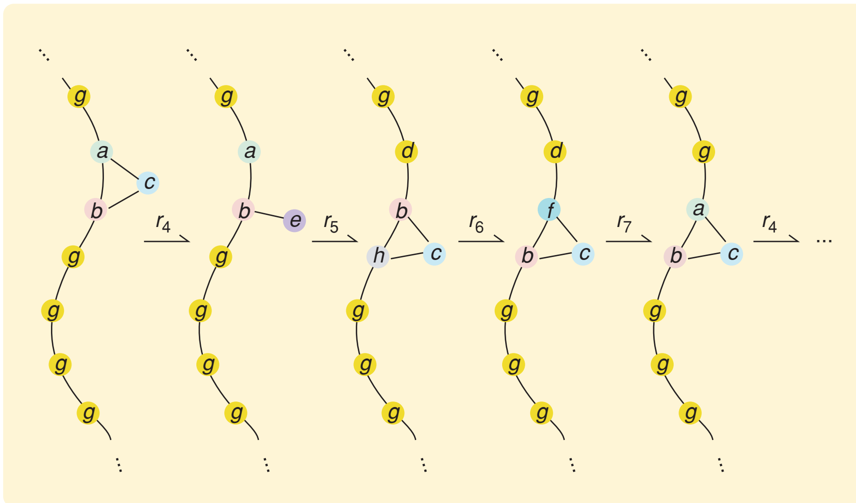


FIGURE 12 Illustrative trajectory of the ratchet system. A grammar need not describe a self-assembling system. In this example, the grammar instead determines how a particle moves along a substrate in a repeating sequence of four basic steps.

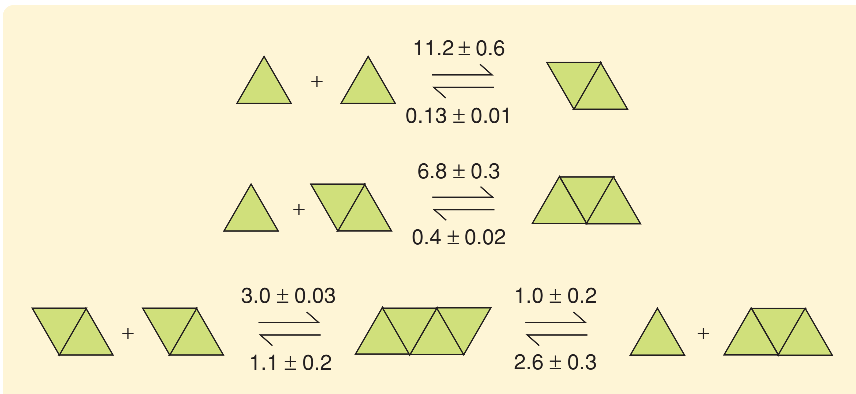


FIGURE 13 Kinetic rate constants. These constants are measured in reactions per 100 s measured from simulations of the programmable parts system with 12 parts at a density of 5 parts/m^2 and average kinetic energy 5×10^{-4} J per programmable part. These rates along with approximately 200 more are used to find an optimal scheme for assembling a hexagon.

OPTIMIZATION OF LOCAL RULES

The relationship between a graph grammar and the measured reaction rates of PPT is that new assemblies form and decay at rates determined by the physics of the system, where the interaction rules allow the robots to keep, reject, or recover from reactions. If two assemblies react to form a larger assembly, the programmable parts within the new assembly use a graph grammar to determine what the new assembly is and whether to keep the new bond or dissolve it. If the robots choose to dissolve the bond, they might have several choices of how to do so. If an assembly breaks due to a high-energy collision, the parts can use a graph grammar to determine the new situation and change their labels accordingly.

The main question, then, is what to do with a new assembly when it forms. When the goal is to build assembly type C_5, a few choices are obvious. For example, when a C_2 and C_3 react to form an assembly C_{10}, the programmable parts comprising the C_5 assembly must discard a single part in order to leave C_5; see Figure 15. Assuming that the reaction $C_2\ C_3 \rightharpoonup C_{10}$ occurs at the rate k and that the communication rate k_{comm} between the parts is much greater than k, the programmed reaction can be modeled by

$$C_2\ \ C_3 \rightharpoonup C_5\ \ C_1 \tag{6}$$

with the same rate k. Note that reactions such as (6) are interpreted as graph grammar rules. For consistency with graph grammar notation, we omit the "+" in (6), which appears in chemical reactions.

It might be useful to disassemble the result of a reaction whether it contains the desired assembly or not. Figure 15 shows all of the choices for disassembling C_{10}. The parts involved in a reaction of this type could choose to disassemble the resulting assembly according to these choices, flipping a many-sided coin to decide. The probabilities u_j weight each option, where $0 \le \sum_{j=1}^{k} u_j \le 1$.

Thus, with each reaction i, we associate a set of probabilities $u_{i,j}$. By tuning the probabilities, we can construct a system whose global performance can be tuned or optimized for a specific task.

It is straightforward to frame this problem in the language of chemical kinetics. First, index the set of all assembly types C_i by $\mathcal{C} = \{1, \ldots, M\}$. Next define a *macrostate* to be an assignment $v : \mathcal{C} \to \mathbb{N}$ describing how many of each assembly type are present. A forward reaction is a vector that describes an assembly event as in, for example,

$$\mathbf{a} = (-1\ \ -1\ \ 1\ \ 0\ \ \cdots)^T,$$

which describes the reaction $C_1\ \ C_2 \rightharpoonup C_3$. The result of a reaction $\mathbf{a}$ in the macrostate $\mathbf{v}$ is

$$\mathbf{v}' = \mathbf{v} + \mathbf{a}$$

as long as all of the components of the vector $\mathbf{v}'$ are nonnegative. In this case, $\mathbf{a}$ is said to be *applicable* to $\mathbf{v}$. Reverse reactions are given by $-\mathbf{a}$ for each forward reaction $\mathbf{a}$.

A programmed forward reaction has the form

$$\widetilde{\mathbf{a}}_i = \mathbf{a}_{i,0} + \sum_j u_{i,j}\mathbf{a}_{i,j},$$

where j ranges over the partitions of the original reaction product C_i. Here $\mathbf{a}_{i,0}$ has one or two negative entries, corresponding to the reactants consumed, and each $\mathbf{a}_{i,j}$ has positive entries corresponding to the indices of possible outcome assemblies.

We interpret the system of reactions as a Markov process. The states of the process are the N possible macrostates, which we enumerate by recursively applying all possible actions to the initial state. Define $\mathbf{S} = (\mathbf{v}_0\ \ldots\ \mathbf{v}_N)$ to be the $\mathcal{C} \times N$ matrix of reachable macrostates, and let $x_i(t)$ be the probability that the system is in macrostate $\mathbf{v}_i$ at time t. The ensemble behavior of the system is given by

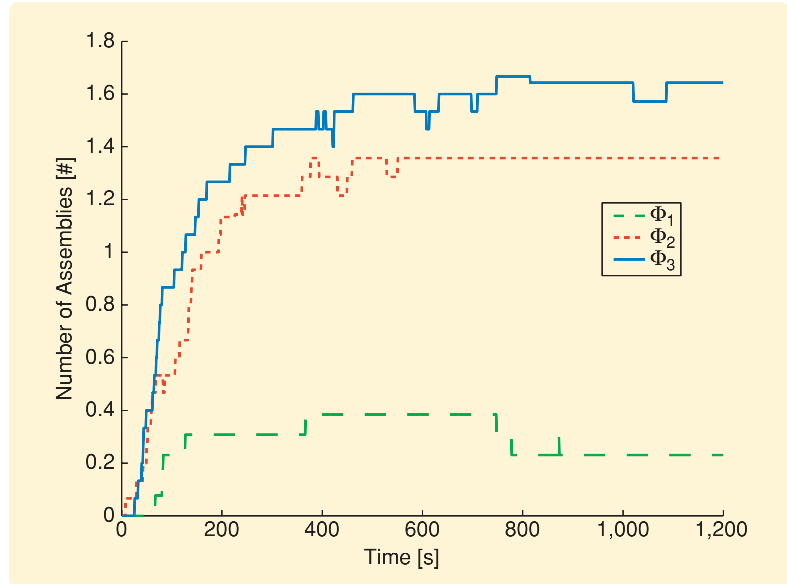


FIGURE 14 A comparison of the systems induced by Φ_1, Φ_2, and Φ_3. These plots are based on the average of 15 simulations. The first grammar directs the assembly in a part-by-part fashion. The second grammar directs the assembly by first allowing dimers to form, and then chains. The last grammar first accepts all possible reactions if they contain a chain of length four and then breaks off nonessential subassemblies.

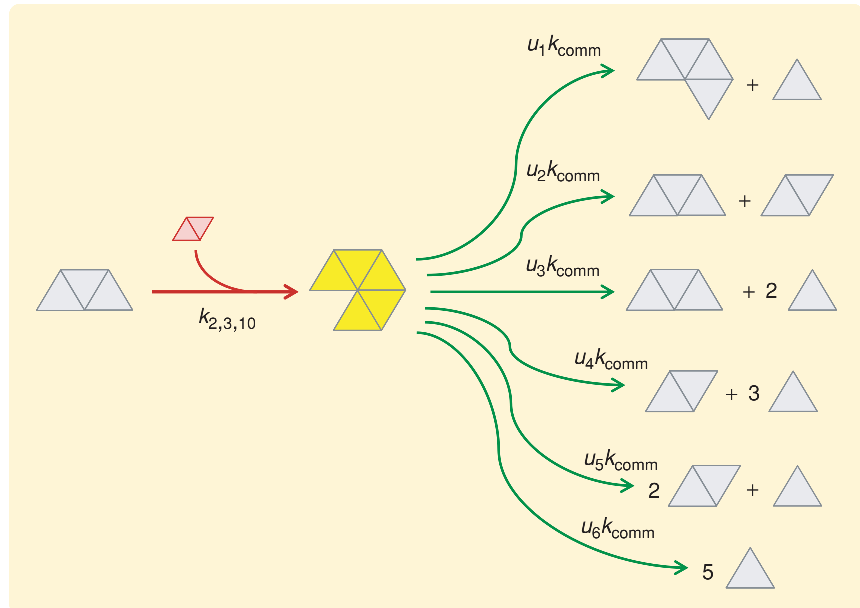


FIGURE 15 A programmed reaction. The parts flip a many-sided coin to determine which option to use when the environment introduces a reaction. The result is a new reaction network, programmed by the graph grammar rules that implement the choice and update rules.

$$\dot{\mathbf{x}} = \mathbf{Q}(\mathbf{u})^T\mathbf{x}, \quad (7)$$

where $\mathbf{x} = (x_0, \ldots, x_N)^T$. Equation (7) is Kolmogorov's forward equation [22, pp. 85–86]. The steady-state distribution $\mathbf{x}_\infty$ of the system is obtained by solving $\mathbf{Q}^T(\mathbf{u})\mathbf{x}_\infty = \mathbf{0}$. Because each reaction is reversible, $\mathbf{x}_\infty$ is unique in these problems [22, pp. 46–50].

To determine the probabilities $\mathbf{u}$ in (7) that maximize the number of the desired assembly at equilibrium, we solve the following optimization problem: Maximize

$$J_{\text{assem}}(\mathbf{x}, \mathbf{u}) = \mathbf{c}^T\mathbf{S}\mathbf{x}$$

subject to

$$\mathbf{Q}^T\mathbf{x} = \mathbf{0}$$

and the constraints

$$\sum_j u_{i,j} = 1 \quad \text{for all } \ i,$$

where $\mathbf{x}$ is a probability distribution. This problem is bilinear in $\mathbf{x}$ and $\mathbf{u}$. The software PENPOT [23] can be used to find a local maximizer of $J_{\text{assem}}(\mathbf{x}, \mathbf{u})$ in polynomial time depending on the number of probabilities in $\mathbf{u}$ and the dimension $N+1$ of $\mathbf{x}$. However, N depends exponentially on the number M of assembly types. Nevertheless, small examples or fragments of larger ones can be addressed. Furthermore, approximate algorithms for the optimization problem can be used to tackle larger problems.

To illustrate the optimization method with the programmable parts, we consider the problem of assembling the hexagon C_{19} in Figure 9. The goal is to use the measured rate data to determine the best hexagon-forming program possible.

A hexagon is grown from subassemblies C_1, C_2, C_3, C_5, and C_{10}. Scaffold assemblies such as C_8 can also be allowed, keeping in mind that they may later react with, for example, a C_3 to form an assembly that contains a hexagon. Hundreds of assembly types can result from interactions among this small set, and the set of rates for these interactions result in a large matrix $\mathbf{Q}$. Furthermore, for each assembly type that does not contain a hexagon as a subassembly there is a choice of how to break it down according to the vector $\mathbf{u}$.

The assembly C_{10} rarely reacts with C_1 to form a hexagon. For example, note the relatively long period of time between the fourth and sixth frame in Figure 16. Thus, we consider only how to break C_{10} according to the options illustrated in Figure 15, to which we associate the probabilities $u_{10,i}$, $i \in \{1, \ldots, 6\}$. Starting with nine parts, there turn out to be 34 states, resulting in a 34×34 matrix $\mathbf{Q}(\mathbf{u})$.

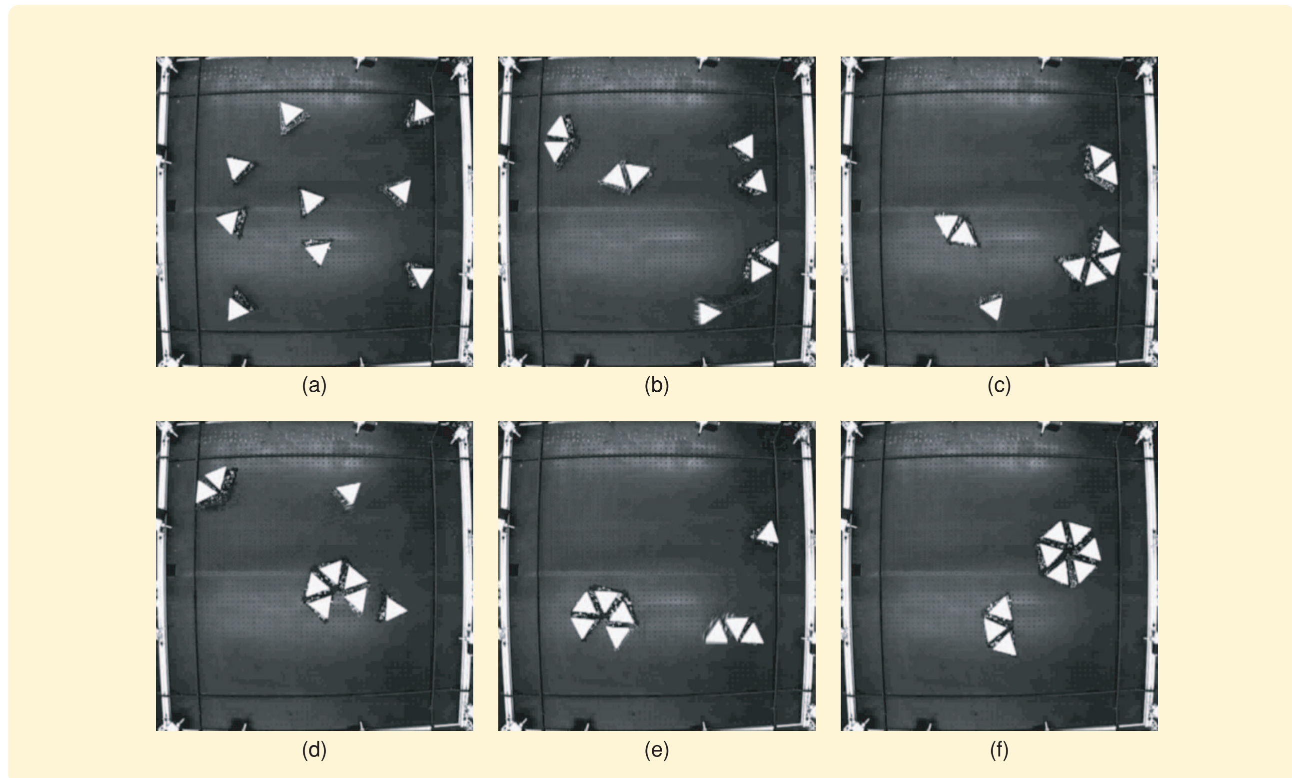


FIGURE 16 A series of frames from video data showing a collection of programmable parts forming into a hexagon using a grammar that accepts all reactions that form a subassembly of a hexagon. At (a) 0 s, all parts are isolated. By (b) 10 s into the experiment, several dimers have formed. By (c) 23 s, a tetramer has formed. By (d) 43 s, a single part has combined with the tetramer to form a pentamer. After (e) 230 s, the pentamer is still present. Finally, after (f) 382 s, the hexagon is formed.

Running PENOPT on this example, the optimization problem corresponding to breaking C_{10} results in

$$u_{10,2} = 0.5, \quad u_{10,3} = 0.5\,,$$

while the remaining probabilities are equal to zero. This choice is not obvious, although in hindsight, using option 3 in Figure 15 half of the time increases the population of C_2 assemblies, which can then react with C_5 assemblies. The optimal choice depends on the initial number of programmable parts, although empirically there seems to be no substantial difference between the resulting performance.

The assumption underlying this example is that disassembling C_{10} correctly in this restricted setting is also the right thing to do using a more complete set of rules. To test this assumption and the validity of the approach, we encode a complete set of hexagon rules for the programmable parts and average the results of several simulations for two different choices for $\mathbf{u}$. The first choice is the "greedy" choice for handling C_{10}, in which assemblies of type C_{10} are not disassembled. The second is the optimal choice computed above. The results of these simulations are shown in Figure 17, where the difference in performance is evident.

To implement programmed assembly actions with programmable parts, a graph grammar updates each robot about the assembly type it is a part of and the role it takes in that assembly. When a new topology is determined, one of the programmable parts flips a coin according to the probabilities obtained in the offline optimization step and then starts a cascade of the appropriate rule applications to signal to superfluous robots in the assembly that they must disconnect from the assembly.

This example shows that a graph grammar can be used to define the underlying procedure by which an object is self-assembled. Consequently, the implementation of the graph grammar can be tuned to optimize the performance of the assembly process. This distinction between structure and tuning is analogous to the distinction between a control algorithm, such as full state feedback, and the choice of the gain matrix used in the algorithm.

CONCLUSIONS

In this article we describe several problems in self-organization. We show that aspects of these problems are captured by the formalism of graph grammars, while others, such as the association of a position or

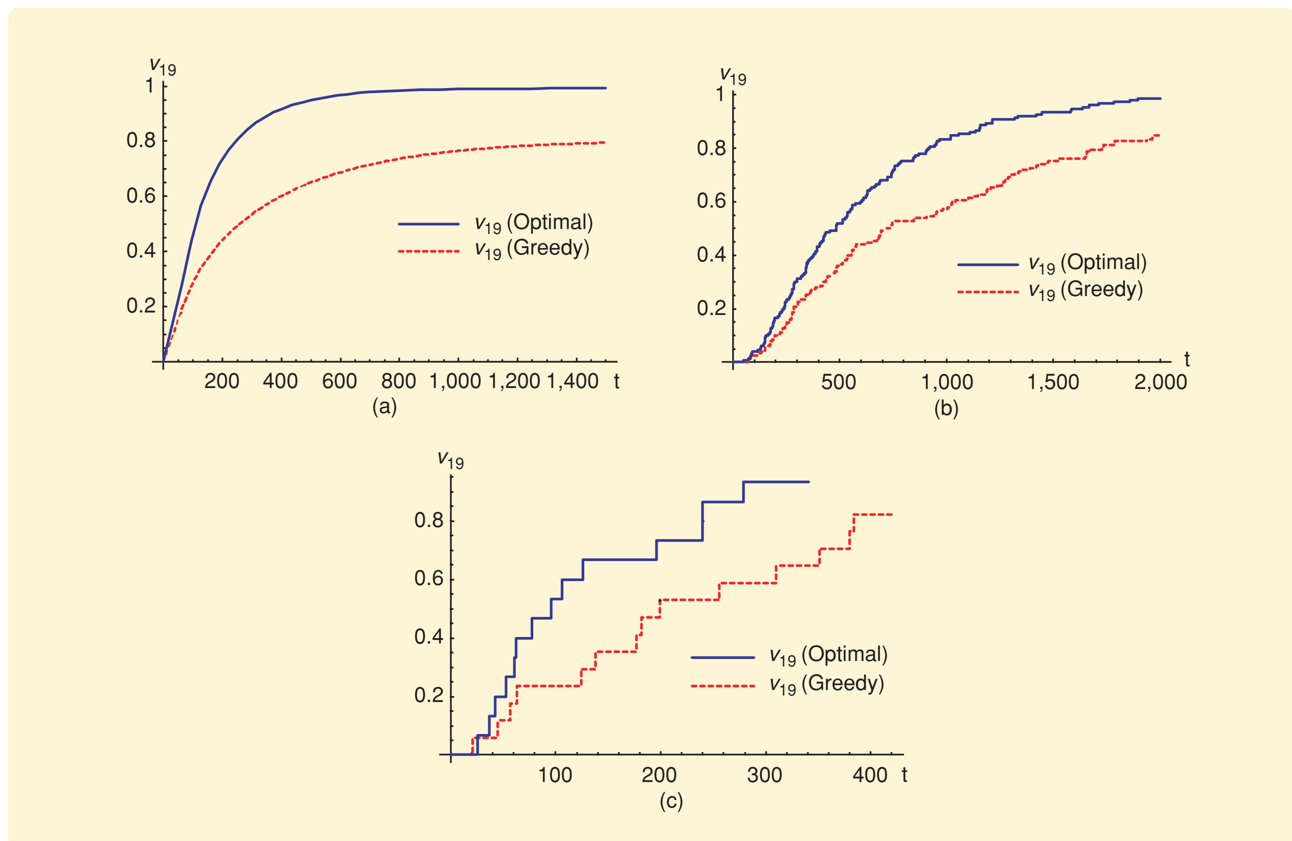


FIGURE 17 Comparison of the optimal choice and greedy choice for how to dismantle C_{10} when assembling C_{19}. (a) Mass-action kinetics with an initial concentration of 9 parts/m^2. (b) The average of 256 trajectories from a high-fidelity mechanics-based simulation of the robots starting with nine parts. (c) Data averaged over 15 runs with nine programmable parts and the universal grammar. The data are collected using an overhead camera and subsequent software analysis of the video data. The various time scales are likely due to different communication rates, specifically, immediate in (a), fast in (b), and somewhat slow in (c), the actual experiment.

velocity with the particles in the system or the association of reaction rates to rule applications, require extensions to the theory.

The formulation of the problem of controlling a system to self-assemble has two parts. First, we define a graph grammar that determines the outcomes of interactions between particles or robots. The grammar can also model aspects of the environment. The result is a transition system on graphs. Second, we define a stochastic process over this transition system and pose an optimization problem that maximizes the yield of a desired assembly type. This distinction between the system defined by a program and the tuning of parameters within a system appears to be a fundamental design principle, which in current work we continue to refine and expand.

The examples we describe in this article are small in scale, especially compared to the vastly more complex self-organizing systems found in nature, from ecologies to cells. Although these examples serve as tantalizing evidence that the phenomenon of self-organization can be harnessed and controlled, engineering large self-organizing systems remains enormously challenging. Addressing these challenges promises new directions and paradigms for control systems research.

ACKNOWLEDGMENTS

The results reported in this article are due in large part to the work of my students Samuel S. Burden, Nils Napp, John-Michael McNew, Joshua Bishop, and Fayette Shaw. In particular, Nils Napp led the effort to design, build, and program the programmable parts, and Samuel S. Burden created the PPT simulator. The work described here is partially supported by NSF Grant 0347955: CAREER: Programmed Robotic Self-Assembly. Robert Ghrist of the University of Illinois in Urbana-Champaign, Mehran Mesbahi of the University of Washington, and Erik Winfree and Niles Pierce of the California Institute of Technology contributed many ideas and helpful criticisms.

REFERENCES

[1] E. Winfree, "Algorithmic self-assembly of DNA: Theoretical motivations and 2-D assembly experiments," *J. Biomolecular Structure Dynamics*, vol. 11, no. 2, pp. 263–270, May 2000.

[2] R.C. Mucic, J.J. Storhoff, C.A. Mirkin, and R.L. Letsinger, "DNA-directed synthesis of binary nanoparticle network materials," *J. Amer. Chem. Soc.*, vol. 120, no. 148, pp. 12674–12675, 1998.

[3] P. White, V. Zykov, J. Bongard, and H. Lipson, "Three dimensional stochastic reconfiguration of modular robots," in *Robotics: Science and Systems I,* S. Thrun, G. Sukhatme, S. Schaal, and O. Brock, Eds. Cambridge, MA: MIT Press, pp. 161–168, June 2005.

[4] E. Klavins, S. Burden, and N. Napp, "Optimal rules for programmed stochastic self-assembly," in *Robotics: Science and Systems II,* G.S. Sukhatme, S. Schaal, W. Burgard, and D. Fox, Eds. Cambridge, MA: MIT Press, 2007, pp. 9–16.

[5] E. Klavins, R. Ghrist, and D. Lipsky, "A grammatical approach to self-organizing robotic systems," *IEEE Trans. Automat. Contr.*, vol. 51, no. 6, pp. 949–962, June 2006.

[6] H.G. Tanner and A. Kumar, "Formation stabilization of multiple agents using decentralized navigation functions," in *Robotics: Science and Systems I*, S. Thrun, G. Sukhatme, S. Schaal, and O. Brock, Eds. Cambridge, MA: MIT Press, pp. 49–56, 2005.

[7] M. Yim, Y. Zhang, and D. Duff, "Modular robots," *IEEE Spectr.*, vol. 39, no. 2, pp. 30–34, Feb. 2002.

[8] S. Murata, E. Yoshida, H. Kurokawa, K. Tomita, and S. Kokaji, "Self-repairing mechanical system," *Autonomous Robots*, vol. 10, no. 1, pp. 7–21, 2001.

[9] J. Suthakorn, A.B. Cushing, and G.S. Chirikjian, "An autonomous self-replicating robotic system," in *Proc. 2003 IEEE/ASME Int. Conf. Advanced Intelligent Mechatronics*, 2003, pp. 137–142.

[10] S. Griffith, D. Goldwater, and J.M. Jacobson, "Self-replication from random parts," *Nature*, vol. 437, no. 9, pp. 636–636, Sept. 2006.

[11] B. Courcelle, "On graph rewriting: An algebraic and logic approach," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, J. van Leeuwen, Ed. Cambridge, MA: MIT Press, 1990, pp. 193–242.

[12] B. Courcelle and Y. Métivier, "Coverings and minors: Application to local computations in graphs," *Eur. J. Combinatorics*, vol. 15, no. 1, pp. 127–138, 1994.

[13] H. Ehrig, "Introduction to the algebraic theory of graph grammars," in *Graph-Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig, and G. Rozenberg, Eds. London: Springer-Verlag, 1979, pp. 1–69.

[14] B. Bollobás, *Modern Graph Theory*. New York: Springer-Verlag, 1991.

[15] I. Litovsky, Y. Métivier, and W. Zielonka, "The power and limitations of local computations on graphs and networks," in *Graph Theoretic Concepts in Computer Science (WG'92)*. New York: Springer-Verlag, 1993, pp. 333–345.

[16] N. Lynch, *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann, 1996.

[17] J. Bishop, S. Burden, E. Klavins, R. Kreisberg, W. Malone, N. Napp, and T. Nguyen, "Self-organizing programmable parts," in *Proc. Int. Conf. Intelligent Robots Systems*, 2005, pp. 3684–3691.

[18] J.M. McNew, E. Klavins, and M. Egerstedt, "Solving coverage problems with embedded graph grammars," in *Hybrid Systems: Computation and Control,* A. Bemporad, A. Bicchi, and G. Buttazzo, Eds. London: Springer-Verlag, 2007, pp. 413–427.

[19] J.M. McNew and E. Klavins, "A grammatical approach to cooperative control: The wanderers and scouts example," in *Cooperative Control,* D. Grundel, R. Murphey, P. Pardalos, and P. Prokopyev, Eds. New York: Springer-Verlag, 2005, pp. 117–139.

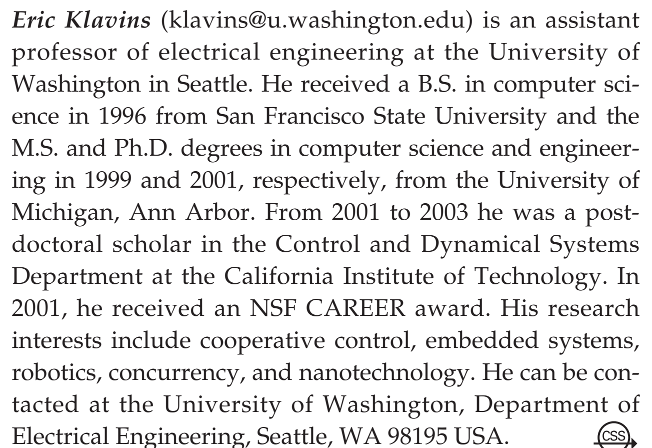
[20] L. Lamport, "The temporal logic of actions," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 872–923, May 1994.

[21] T.C. Daniel, A.C. Trimble, and P.B. Chase, "Compliant realignment of binding sites in muscle: Transient behavior and mechanical tuning," *Biophys. J.*, vol. 74, pp. 1611–1621, Apr.1998.

[22] D.W. Strook, *An Introduction to Markov Processes*. New York: Springer-Verlag, 2005.

[23] PENOPT home page [Online]. Available: http://www.penopt.com/

AUTHOR INFORMATION



Eric Klavins (klavins@u.washington.edu) is an assistant professor of electrical engineering at the University of Washington in Seattle. He received a B.S. in computer science in 1996 from San Francisco State University and the M.S. and Ph.D. degrees in computer science and engineering in 1999 and 2001, respectively, from the University of Michigan, Ann Arbor. From 2001 to 2003 he was a postdoctoral scholar in the Control and Dynamical Systems Department at the California Institute of Technology. In 2001, he received an NSF CAREER award. His research interests include cooperative control, embedded systems, robotics, concurrency, and nanotechnology. He can be contacted at the University of Washington, Department of Electrical Engineering, Seattle, WA 98195 USA.