Towards a common comparison framework for global-to-local programming of self-assembling robotic systems

Justin Werfel and Radhika Nagpal

TR-14-07

Computer Science Group
Harvard University
Cambridge, Massachusetts

Towards a common comparison framework for global-to-local programming of self-assembling robotic systems

Justin Werfel and Radhika Nagpal
Harvard University CS Technical Report TR-14-07

October 30, 2007

Abstract

Self-assembling robotic systems are a class of modular robots where many identically-programmed modules mix randomly and attach together to assemble into complex shapes. Recently several groups have proposed global-to-local compilers for such systems and for related robotic systems; the compilers take a global shape description and automatically generate appropriate local module attachment rules. A current challenge is understanding how these approaches compare in terms of complexity, parallelism, and expressivity.

In this paper we present some initial steps towards a common framework for comparing algorithmic approaches to self-assembly. We specify a generic 2D self-assembly framework and then use this framework to compare self-assembly algorithms based on two distinct global-to-local compilers: the MakeGraphGrammar approach for generating topological graphs and the CollectiveConstruction approach for mobile robots that build structures using communicating blocks. We show that: (1) Both approaches generate module programs with the same high complexity, O(total modules); (2) Both approaches have dramatically different best-case parallelism: while the number of steps required to build a shape with CollectiveAssembly is bounded by the diameter of that shape, MakeGridGrammar has a best case of no more than four parallel steps to complete any shape it can handle; (3) Both approaches can provably generate 2D shapes without defects, but the class of shapes is much larger for CollectiveConstruction and poses less stringent module requirements.

1 Introduction

Self-assembling robotic systems are a class of modular robotic systems composed of many identical modules that, when mixed randomly, can bind together to assemble into complex shapes [1, 2, 5, 6, 14]. Modules in these systems have dynamic state and local binding rules that drive the overall self-assembly process. A key question is how to generate module rules such that the system assembles to form a particular prespecified global shape.

Recently, several groups have demonstrated decentralized, local agent rules for shape formation, not only in the context of self-assembly but also for self-reconfigurable robots and collective construction by mobile robots [3, 4, 7–10, 12]. An important feature of these particular examples is that they achieve *global-to-local programming:* there is a principled way to derive local rules to achieve a prespecified global shape from a given class. In some cases, it has also been shown that the derived local rules are provably correct regardless of variations in the order and timing of agent movements and actions. It may be possible to apply many of these algorithmic approaches in the context of self-assembling modular robots. However, it is often difficult to understand and compare the theoretical properties of these global-to-local compilers because of the different settings—both the assumptions about agents (shape, computation and communication capability, self-propelling or not, homogeneous or bipartite, physical movement constraints, etc.) and the global shape descriptions can differ significantly between frameworks. For example, the MakeGraphGrammar algorithm by Klavins *et al.* [7] operates on homogeneous point agents that are randomly mixed, and generates agent rules that provably will terminate only at the completion of an arbitrary desired topological graph. By contrast, the family of CollectiveConstruction algorithms by Werfel *et al.* [12, 13] operates on bipartite agents (self-propelled robots and passive square blocks) with potentially restrictive movement assumptions, and provably produces arbitrary desired geometric structures in 2D without holes. Given such different settings, it is often difficult to see whether

these algorithms can solve the same problems, and if so how they would compare in measures such as complexity, correctness, parallelism and expressivity.

In this technical report, we present some initial steps toward comparing the theoretical properties of different global-to-local programming approaches in the context of self-assembling robotic systems. We do so by adapting the approaches to work in a common and generic framework. We present three aspects of this problem:

1. A generic framework for 2D self-assembly (§2): We specify a simple 2D self-assembly framework that accommodates many different systems currently under investigation. We also describe a specific method, geometric grammar rules, for specifying module programs. Together these elements provide a concrete formal model in which one can describe and reason about the complexity and performance of different approaches.

2. We recast two existing global-to-local compilers to work in this common setting (§3, §4): the MakeGraphGrammar approach from [7] and the "communicating blocks" variant of the CollectiveConstruction approach from [12]. The former requires modifying the MakeGraphGrammar approach to respect geometry; we present a modified algorithm, MakeGridGrammar, and show that this algorithm can provably create a restricted class of 2D solid geometric shapes while guaranteeing no defects or deadlock. We also modify the CollectiveConstruction approach to work with homogeneous agents that are not self-propelling. Doing so requires expressing the shape formation process using explicit grammar rules. We show that the mapping in this case is quite natural and so the CollectiveAssembly algorithm preserves all the algorithmic guarantees of the original setting, such as provable construction of 2D solid shapes with no defects or deadlock, and without requiring modules to travel down narrow tunnels.

3. We compare the two global-to-local compilers based on four main criteria (§5): (1) complexity of the module program and scalability with structure size; (2) best-case parallelism; (3) expressiveness of the provable class of shapes; (4) module motion requirements. We show that both approaches generate module programs with the same complexity, $O(n)$, where n is the number of modules in the desired shape, regardless of the details of the actual shape. Although the complexity is the same for both approaches, the rules themselves are quite different, resulting in dramatically different best-case parallelism: the number of assembly steps for CollectiveAssembly is bounded below by the diameter of the shape, whereas MakeGridGrammar in the best case can produce a complex shape in O(1) steps. Finally, both approaches can provably generate a large class of 2D shapes without defects, although CollectiveAssembly can generate a much larger class of shapes than can MakeGridGrammer and with less stringent requirements on module motion.

A key goal of this work is to demonstrate that one can reason theoretically about important properties of the module programs that these global-to-local compilers generate, and that it is possible to give precise answers over large classes of shapes. While the properties above do not cover all the important properties of compiled module programs, they do provide a basis for evaluating the tradeoffs in the design of global-to-local compilers. We conclude with discussion of how one can modify the compilers to take advantage of different tradeoffs.

2 Self-Assembly Framework

In this section we describe a simple, generic model for self-assembling robotic systems that can be used as a framework for comparing different algorithms. The framework has the following four parts, also illustrated in Figure 1.

1. **Module Shape and Actions:** Modules are homogeneous square tiles that move in a 2D space. They can decide to attach to, or detach from, other modules along any of their four faces depending on their interactions. Modules may be self-propelled [1] or driven (possibly randomly) by external forces [6, 14].

2. **Global Shape Definition:** A desired global shape for the assembly is described as a contiguous 2D grid region.

3. **Module State:** Modules have a state associated with each of their four faces ("face state") and a state associated with the module as a whole ("body state"). Both can change as a result of interactions with other modules.

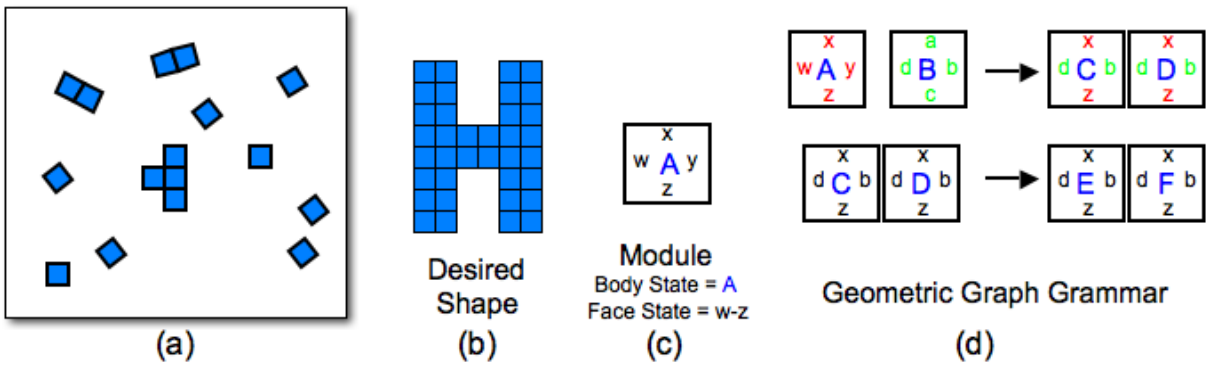

Figure 1: Abstract Self-Assembly Framework: (a) We assume square modules moving in a 2D space (b) The desired goal is specified as a 2D grid shape (c) Each module has body and face states (d) Modules interact according to geometric grammar rules. The top rule shows that if two modules with body states A and B meet along the y and d faces respectively (LHS), they can attach together and as a result of their attachment both their body and face states change in an orientation specific manner (RHS). It is also possible to have rules that operate on attached modules and only change the body/face states.

4. **Module Ruleset:** The module's local behavior is described by a set of interaction rules, expressed as a *geometric graph grammar*. All modules are identically programmed with the same ruleset. Rules in this grammar specify the initial state of the modules on the left hand side, and the final state of the modules on the right hand side. If modules are in the correct initial state, then the rule applies and the modules transition to the final states on the right. Figure 1 shows some examples of grammar rules. Rules may be involve two modules (binary) or three modules (tertiary). A rule most often involves two modules that may become attached as a result of the interaction; however, rules may also apply to modules that are already attached. A key feature is that rules involve both body and face states; the use of face states constrains the relative geometric direction (orientation) of interactions between modules.

This framework is inspired by similar work on a generic model for self-reconfigurable modular robots [8], which was created to facilitate the design of decentralized algorithms using simple abstract modules that can be mapped to many different styles of physical modules. Here we focus on self-assembling systems where, unlike the self-reconfigurable case, the modules need not remain connected but also may not be self-propelled (capable of individual movement control). Expressing the module program as a geometric graph grammar is inspired by the graph grammars for topological self-assembly by Klavins et al. [7]. The main difference in our framework is that a module has both body and face states (while in the Klavins et al. framework, modules have only the equivalent of body state). The face states can be used to constrain the relative orientation in which rules may apply. This allows rulesets to express geometric rules as well as topological ones.

Using this framework we are interested in comparing the module programs generated by different global-to-local approaches. The criteria for comparison are:

1. **Module Program Complexity**: The complexity of a module program can be measured as (a) the number of unique body/face labels plus (b) the number of distinct rules in the module ruleset. We are especially interested in how the complexity of the automatically generated grammar scales with the dimensions and complexity of the desired global shape.[1]

[1]In this framework, we assume that the ruleset is stored as a lookup table and thus each additional rule increases the complexity. However, in many systems, the agent may have the ability to compute more complex functions than just accessing a lookup table, in which case the rules may be expressed much more succinctly.

2. **Best-Case Parallelism**: Given a grammar for forming a particular shape, the best-case parallelism is measured by the length of the shortest sequence of parallel module attachments that results in the desired shape. We assume that a module can bind/evaluate a rule along only one face at a time. This metric is a lower bound on the running time of an algorithm and reflects the theoretical ability of the grammar to leverage the available parallelism of assembly.

3. **Provably-Achievable Shape Class**: The expressiveness of the global-to-local compiler is determined by the class of shapes for which the compiler can generate module rulesets that are provable. Provable implies that for the given shape class, it can be shown that the desired shape will be formed correctly regardless of the order in which modules act.

4. **Module Restrictions**: Different approaches make different assumptions about module movement abilities, for example, whether modules are able to slide through narrow (module-width) tunnels. This consideration reflects how easy it would be to physically realize a system that takes a given approach—relying on module movement through more confined spaces would make a system more difficult to construct.

The above criteria help us understand how different self-assembly approaches might create the same shape while making different tradeoffs in module complexity and parallelism. The criteria depend mainly on algorithmic properties and thus, as we will show, many of these can be proven theoretically over entire classes of shapes. However there are also other important criteria requiring more concrete models of module movement, e.g.:

1. *Average expected running time, or likelihood of best-case parallelism:* Depending on how modules move, it may be very unlikely that the best-case sequence will occur, and one grammar may be more efficient than another in practice. This may be due to the fact that the partly assembled pieces are more likely to align [**?**], or because correct assembly requires modules to move through highly constrained spaces that in practice are unlikely to happen. Calculating these probabilities requires a more realistic model of module movement and alignment, whether through physical models or experimentally calculated probabilities. In this technical report we do not address average running time directly, however we do discuss qualitatively how module movement restrictions (metric 4) impacts the practicality of assembly. We hope to address this metric more quantitatively in the future.

In the next two sections, we use this common framework to recast two recent examples of global-to-local compilers for shape formation: the MakeGraphGrammar approach from [7], and the CollectiveConstruction approach using communicating blocks from [12]. Both examples require some modifications to work in this setting; however, we try to remain as close to the original approaches as possible.

3 MakeGridGrammar

In [7], Klavins et al. describe an approach to self-assembly that uses graphs to describe a desired goal, and graph grammars to express how nodes should interact in order to generate the desired graph topology. We focus on the algorithm MakeTreeGrammar, which can automatically generate a graph grammar given a description of the tree. It has been shown that the generated grammars are provably correct—they will only create the desired tree and not create intermediate stable structures (e.g., cycles or other trees) that can not progress towards a correct tree through subsequent attachment rules. In this section, we recast this algorithm into our 2D generic model and modify the basic idea to respect both topology and geometry.

One basic approach to generate 2D shapes is to embed a topological tree in the desired 2D shape, and then use MakeTreeGrammar to generate a grammar for that tree. However, a limitation of MakeTreeGrammar is that the resulting grammar only respects topology. This results in two different kinds of geometry problems (Figure 2): (1) the generated shape may end up being any topological equivalent to the requested one; (2) the intermediate trees may create physically enclosed gaps that block modules from completing the assembly. While the same group has demonstrated how one can hand-design graph grammars for physical modules with geometry constraints, to our knowledge the global-to-local compiler has not been extended to the that space.

Here we present an algorithm MakeGridGrammar, and show that it can provably create a restricted class of 2D solid geometric shapes while guaranteeing no defects or deadlock. MakeGridGrammar works by embedding a tree into

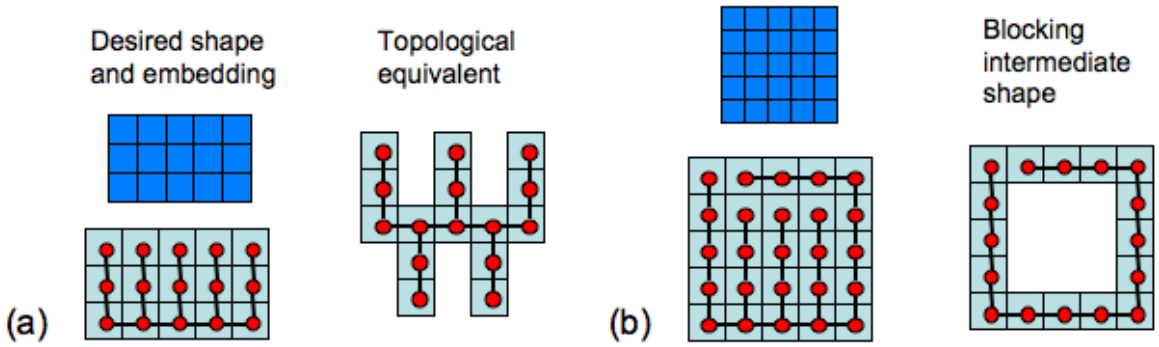

Figure 2: MakeTreeGrammar provides grammars that preserve topology but not geometry. Simply embedding a tree in a 2D shape can result in two different kinds of problems (a) the generated shape is a topological equivalent of the desired shape (b) the topologically-valid, partially-generated, shape blocks physical access to modules.

the desired 2D shape in a principled way, and modifying MakeTreeGrammar to generate grammar rules that preserve orientation through the use of face states.

1. **Embedding a Tree in a 2D Shape**: We express a shape as a west-east backbone, with north-south chains of different lengths attached to the backbone. Thus the tree consists of attached linear chains.[2] This expression allows for a restricted class of 2D shapes, where for each node that is not in the backbone, there must be a linear fully connected path to the backbone. Figure 3 shows some examples of shapes that can and can not be expressed in this form. We will call this class of shapes *2D-backbone shapes*.

2. **Generating the Grammar**: To generate the grammar, we modify the basic MakeTreeGrammar algorithm. The basic algorithm removes an edge from the desired tree, and then runs MakeTreeGrammar on the two subtrees. To connect the two subtree grammars, it adds a new rule that connects the labels of the roots of the two subtrees and gives them new unique labels. In the base case, a subtree is just a node which receives the label a. In order to make the grammar respect geometry, we make two changes. First we assume that all base case nodes are labeled with a body state a and face states *N, S, E, W* to indicate a handedness for all initial modules. Second, when two subtree grammars are merged, the new rule that merges them uses both body and face states to specify the orientation of the attachment, enforcing the correct geometry. Thus, at each step of the tree self-assembly, the handedness and orientation of attachments is respected.

Note that we purposefully retain as much of the structure of MakeTreeGrammar as possible, so that the new algorithm inherits most of the properties and behavior of the original algorithm.

3.1 Algorithm Pseudocode

A shape is represented as a tree (V, E) where each edge in E also has an orientation labeling. Thus an edge is represented by $((x, F_x), (F_y, y))$ where x and y are the names of the nodes in V and F_x, F_y are the faces of the nodes that meet in an edge. Note that F_x, F_y are from the set N, S, E, W

In the algorithm below, we will represent a geometric grammar rule by

$a, F_x + F_y, b => u, F_x - F_y, v$

Note that this is simply a textual representation of the type of rules described before. If modules with body labels a and b meet along the faces F_x, F_y, then they should attach and their body states should change to u and v respectively. Note that MakeGridGrammar generates rules that use face states, but do not change them.

[2]Note that the north-south-east-west labelling is only to preserve relative connections and does not imply that the shape is created in a orientation relative to some external compass.

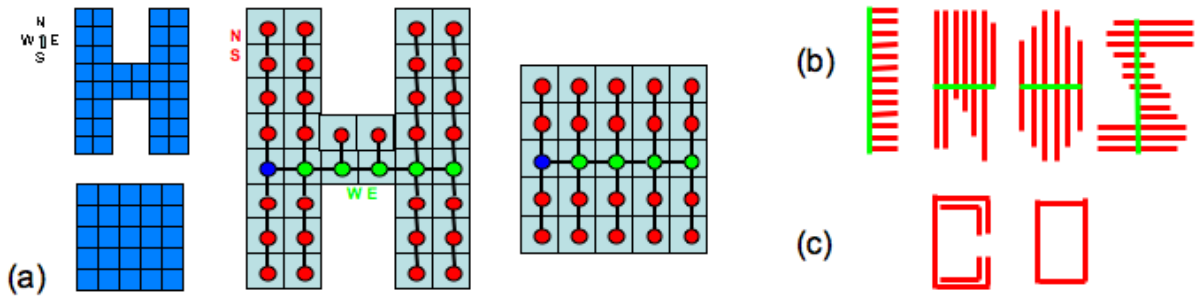

Figure 3: MakeGridGrammar: (a) Given a desired shape, the embedding creates a tree with a linear chain backbone and sidechains (b) Example of shapes in the 2D-backbone shape class (c) Examples of shapes that cannot be formed.

```
Let (V,E) represent a tree with a set of nodes V and edges E
Let G represent a set of grammar rules
Let L represent a mapping from nodes in V, to body state labels.

ALGORITHM: MakeGridGrammar(V,E)

 If E is not empty, then

   // Divide the tree into two subtrees and run MakeGridGrammar on both
   let (x,y) be an edge of (V,E)
   let (V1,E1) be the subtree of (V,E - {x,y}) containing x
   let (V2,E2) be the subtree of (V,E - {x,y}) containing y
   let (G1, L1) be the result of MakeGridGrammar(V1,E1)
   let (G2, L2) be the result of MakeGridGrammar(V2,E2)

   // Merge the two grammars, and add a new rule connecting the two
   // subtree roots. Also update the node to body state mapping
   let u,v be new unique labels
   let G = G1 U G2
           U {L1(x),Fx  + Fy,L2(y) => u,Fx -- Fy,v}
   let L = L1 U L2
           - (x,L1(x)) - (y,L2(y)) // remove the old labels for x, y
           + (x,u) + (y,v)         // and add the new ones
   return (G,L)

 Else if E is empty
   // Base case, give the solitary node the label
   // a with N-S-E-W face orientations
   return(emptyset, a(N,S,E,W))
```

Note that above algorithm produces a grammar with binary rules, each of which is an attachment rule (the left hand side modules are not attached, the right hand side is). Also the rules require that correct face states be presented when the modules meet, but face states are not changed after attachment. The single nodes are labeled as "a" with the north-south-east-west face states, which is the initial module type that the process starts out with. Note that in this algorithm multiple rules will in general be produced for the situation a–a, which requires that the initial nodes pick different outcomes with some probability (we assume equal likelihood).[3]

[3]An alternative is to have the algorithm's base case give a unique label to every solitary node corresponding to position in the final tree, and

3.2 Proof of Correctness

To prove that the algorithm generates the correct geometric structure, without deadlocks or blocking, we can leverage some of the results from the original MakeTreeGrammar algorithm [7]. The proof of MakeTreeGrammar guarantees that (1) the only reachable state is a topological equivalent of the desired tree (2) any substructures formed in the process are always subtrees of the original graph.

The MakeGridGrammar algorithm is also guaranteed to produce only the desired topological tree because the grammar, without the face states, is exactly the same as one generated by MakeTreeGrammar. To show that the process generates the correct geometric tree, we need to show that at each step the handedness and orientation of attachments is preserved. This can easily seen for the base case, which assumes all starting elements have the same handedness. For the inductive step, the merge operation adds an edge between the subtree roots that preserves the orientation of the labels across the edge. Thus at each step the handedness and orientation are preserved.

Finally we need to show that the system will not produce defects by creating enclosed gaps. This property is ensured because any connected subtree of the shape never includes an enclosed gap. This point is the motivation for the specific method of embedding the tree with a backbone and sidechains. MakeTreeGrammar ensures that only subtrees of the desired structure are created during the self-assembly process; thus we avoid constructing any shapes with enclosed gaps.

3.3 Criterion 1: Complexity

The complexity of the module program generated by MakeGridGrammar is determined by the number of edges in the final tree. In particular, for each edge we generate a new rule and two new labels for the body states. In the case of a tree the number of edges is $n - 1$, where n is the number of nodes. Face states take on only a constant number of labels, therefore they do not fundamentally affect the complexity.

Thus the module program complexity is O(n) state and O(n) rules, where n is the number of modules. This was also shown for MakeTreeGrammar [7]. Note that this result implies that the amount of memory required by individual modules increases as the overall number of modules used increases, which may be undesirable. Furthermore the complexity of the program is independent of the complexity of the desired shape—forming a large square is just as complex as forming the letter "R", if both contain the same number of modules. This suggests that the global-to-local compiler is unable to exploit symmetries and repeated structures in the desired shape.

3.4 Criterion 2: Best-Case Parallelism

The geometric graph grammar dictates a self-assembly process—in this particular case, many parts of the structure are allowed to assemble in parallel, providing the opportunity for considerable parallelism. An illustrative example is that of a creating a chain of length n. Depending on the order in which edges are removed, the MakeGridGrammar algorithm can generate two very different types of chain formation processes, which are illustrated in Figure 4.

- Method 1: Remove edges in a left-to-right order. This results in a grammar where the assembly of the chain occurs from right to left. Thus the minimum number of steps to assemble is n.
- Method 2: Remove even edges first and then odd edges. In this case alternate links can form in parallel: the nodes can form pairs, and then the pairs can link together. Thus the entire chain can be formed in just 2 steps, regardless of its length. Note that 2 steps is a tight lower bound since we allow a module to only bind along one face at a time.

By exploiting the edge removal order in method 2, we can also produce best-case parallelism for all members of the backbone-shapes class. Note that a strict lower bound on the time taken is the number of neighbors of any module—thus if a module in the backbone has four neighbors, then the assembly process can not take less than 4

initializing modules heterogeneously so that all such labels are represented, rather than initializing all in the same state. So doing would make all rules deterministic; the cost, aside from the heterogeneity of initialization, is that assembly from solitary modules will rely on the right specific labels encountering each other (rather than any two solitary modules), which will in general slow the assembly process. We assume that all solitary nodes get the label a which is consistent with the original MakeTreeGrammar algorithm.

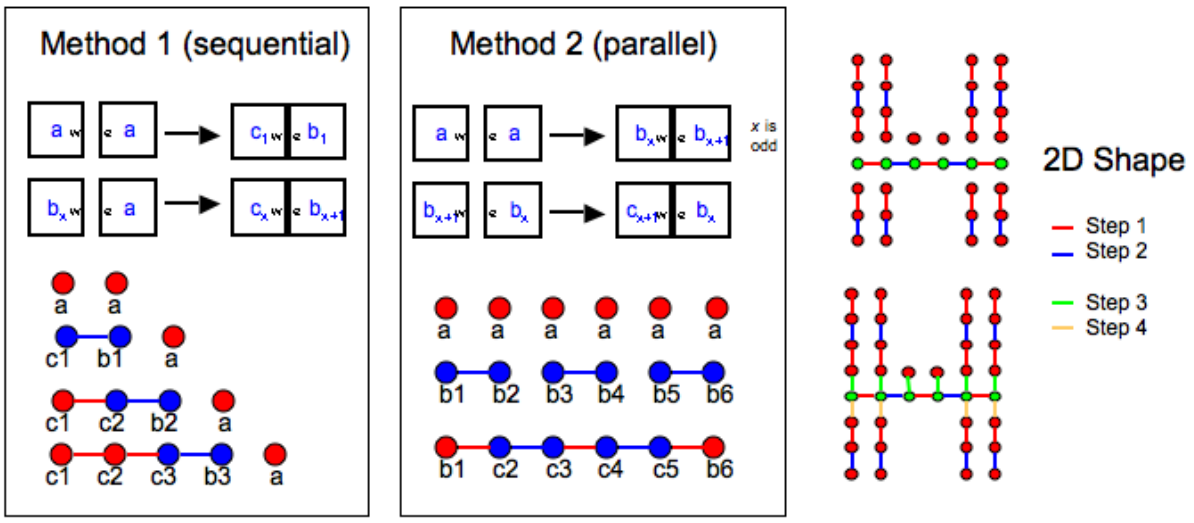

Figure 4: Best-case parallelism: Two different grammars for a chain that create two different self-assembly processes. Method 1 forces the chain to form sequentially, while method 2 allows the chain to assemble its part in parallel. By exploiting parallel chain formation, it is possible to generate any 2D-backbone shape in just 4 steps.

steps. Here we show that four steps is in fact an achievable parallelism. The basic self-assembly strategy is as follows: allow each of the chains (backbone and the side chains) to form in parallel using method 2. Then allow the backbone to attach in parallel to all north chains, then in parallel to all south chains. The ordering can be enforced by removing the edges connecting the backbone to the side chains first in MakeGridGrammar, similar to how method 1 imposes an order on the order of chain formation. The complete process takes a total 4 steps.

Thus the **best-case parallelism for MakeGridGrammar to construct any member of the backbone-shape class is 4 steps**, regardless of the size or detailed geometry of the shape.

3.5 Criterion 3 and 4: Provably-Achievable Shape Class and Module Restrictions

The 2D-backbone class is a restricted class of 2D shapes; it may be possible to create more expressive methods to extend MakeTreeGrammar to other shape classes. Figure 3 shows some examples of shapes that can, and can not, be created by specifying a backbone.

The method described relies on certain key assumptions about module movement. The modules and sub-assemblies of modules must be able to move freely and even move within restricted spaces in order to assemble into the final structure. In particular, fully or partially formed chains must be able to travel down tunnels of width equal to that of the chain. While assuming this capability allows for better theoretical parallelism, in practical settings the requirement may be restrictive. It is possible to make a tradeoff between parallelism and module movement—for example, one can create a backbone chain grammar that forces the side chains to attach in a left-to-right order. In that case the best-case parallelism is proportional to the length of the backbone, but module chains will not need to travel down tunnels to form the final structure. These alternative self-assembly orders can be achieved by systematically forcing an order on edge removal in the MakeGridGrammar process.

4 CollectiveAssembly Grammar

In [12], we describe a family of algorithms by which a swarm of independent robots can assemble arbitrary solid 2D structures using square building blocks. One of these algorithms uses "communicating blocks", which can store state and communicate with physically attached neighbors. That situation corresponds directly to programmed self-assembly in the framework we consider here, where blocks correspond to modules, and robots are responsible for module movement. In this section, we adapt the CollectiveConstruction approach presented in that paper to the self-assembly framework explicitly.

In this approach, a desired shape is described as a 2D grid with an internal coordinate system ("shape map"). The basic idea is that modules explicitly store the shape map and their own coordinates (initially modules have unassigned coordinates). Assembly begins starting from a single "seed" module which knows its own coordinate within the desired shape. The structure growing outward from that seed, as unassigned modules attach to the growing structure and acquire their coordinate from their neighbors.

In order to avoid defects, it is necessary that the structure grow in a conservative manner. Furthermore, we assume that modules can not enter spaces one module width wide or travel down tunnels one module width wide. As shown previously, this translates into a simple constraint that must be maintained throughout the shape formation process: two non-adjacent modules must never be attached in the same row or column within the desired shape, if all sites between them are meant to be occupied. Therefore the rule is to attach modules in desired sites, subject to the above constraint. It has been shown that the rule will provably generate any desired 2D shape that (1) encloses no internal empty gaps, and (2) has no alleys narrower than two module-lengths wide [11]. To carry out this rule, the CollectiveAssembly grammar specifies that modules share a common coordinate system once bound, and that each of the four faces of each module maintains a state corresponding to whether modules have yet been attached in the row[4] the face borders.

Because the CollectiveAssembly algorithm is a direct translation of the CollectiveConstruction algorithm previously characterized, it preserves all the algorithmic guarantees of the original setting, such as provable construction of 2D solid shapes with no defects or deadlock and without requiring modules to travel down narrow tunnels.

4.1 Module grammar

In this approach, the module ruleset can be easily inferred from the desired shape. Therefore rather than give pseudocode for an algorithm to generate module grammars, we will give the module grammar more directly. The grammar relies on the explicit representation of the desired shape with an internal coordinate system (the "shape map"). We describe module states and rulesets in two separate parts: those having to do with the coordinate system and shape map (§4.1.1), and those having to do with face states to ensure that separated blocks are not inappropriately attached in the same row (§4.1.2, §4.1.3).

4.1.1 Coordinates and shape

All modules in the assembly need to share a common coordinate system and orientation. The assembly begins with a single module that has a body state equal to a coordinate, say $(2, 3)$, and four face states that have a (N, S, E, W) orientation. Note that the assembly does not need to begin from the origin. All other modules have a body state $(?, ?)$ and unassigned face state. Figure 5 gives the rules by which a new module joining the assembly acquires its coordinates and orientation.

Note that this process is very local; each attachment rule is a binary rule. Furthermore, an attachment rule only exists if that coordinate needs to be filled in the desired shape. For instance, if there should be no module at (3,3), the ruleset will not contain the first rule in the figure for an existing module at (3,2). Thus the ruleset describes the desired shape to be formed.

[4] Hereafter we use "row" to mean "row or column".

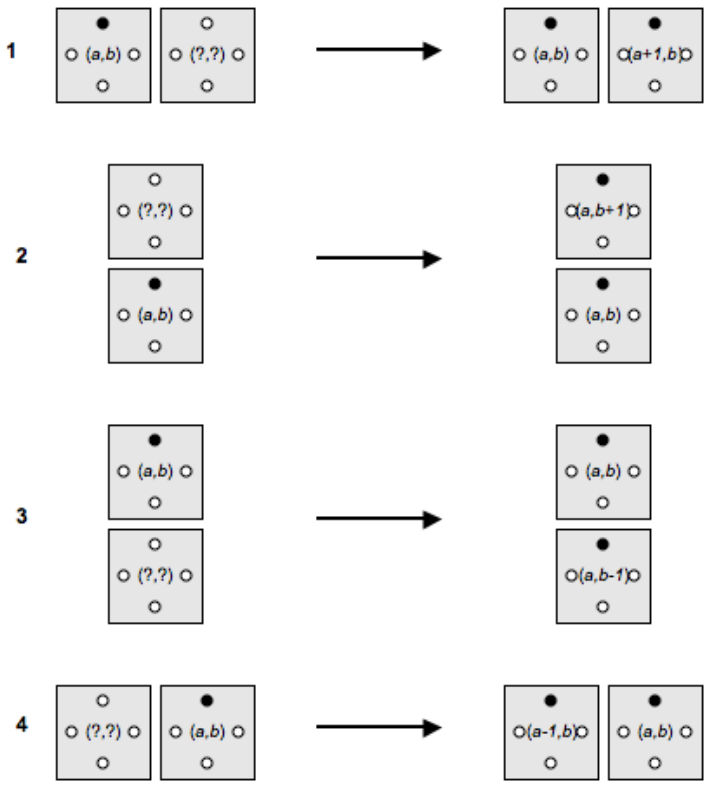

Figure 5: Four rules by which a new module becomes attached to an existing one at coordinates *(a,b)*, and acquires coordinates and orientation to match the rest of the assembly. Coordinates are given in the center of the module; orientation is shown by having the north face marked with a black dot.

4.1.2 Grammar for adding single modules

In addition to the coordinate rules described above, a second set of rules ensures that separated blocks are not inappropriately attached in the same row. We begin by discussing in this section the simpler case where only one module at a time ever binds to the structure, and all module state changes specified by the grammar run to completion before the next module arrives. In the next section we will discuss the more complicated case where many modules may be present at one time.

The basic intuition is that initially a row is open to attachment at any site, however as soon as the first module gets attached, the remaining row is closed to attachment. The only valid sites are adjacent to the current row modules. In order to achieve this process, we must propagate information through the existing, partially-assembled, structure. To achieve that, we use grammar rules that operate on attached modules and change their face states.

Figures 6–8 show the details of the grammar for the single-module case. Faces here have states from the set {*open, closed, corner, done, start, waiting, edge*}. A face in the state *start* has two further bits associated with it, shown as circles in Figure 6 as with Figure 5. The intuition is that a face may be *open*, indicating that nothing has yet been attached in the row the face borders; *closed*, indicating that a module has been attached in that row some distance away; *corner*, indicating that a module is attached at the next site over (so that attachment here is permissible without causing separation); *done*, when a module has been permanently attached to a face; or *edge*, when the shape map specifies that this face lies along the perimeter of the desired final shape and no module should ever be attached there. The *start* and *waiting* states are intermediate states used when information needs to be propagated further than one

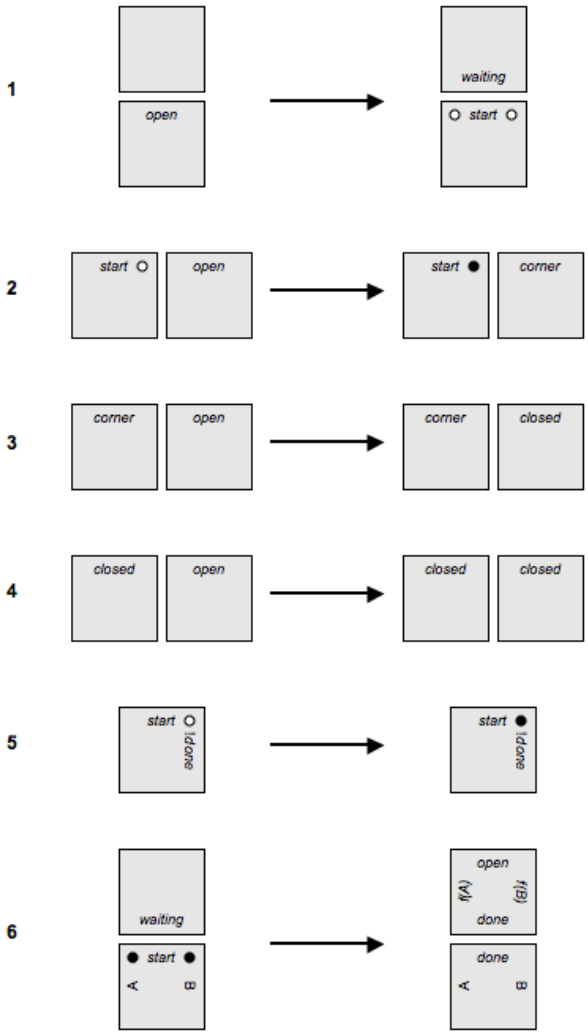

Figure 6: Rules for attachment in rows where no modules have yet been attached. In these pictures, any states not explicitly specified are to be treated as wildcards and may be any state. (The exception is the completely unmarked module, which represents the new one potentially becoming attached.) All rules shown as non-symmetric should be understood to apply in reflection as well.

1. When a new module approaches an *open* face, the former takes on a *waiting* state and the latter a *start* state. The two auxiliary bits will be used to make sure that *open* modules in both directions will become *closed*.
2. An *open* module in either direction is set to *corner*, and the corresponding start bit set to 1 to indicate that information about the attachment will be sent down the row in that direction.
3. The *corner* module will set an *open* module to *closed*...
4. ...and the *closed* one will set a further *open* to *closed*. This rule will continue to apply until the end of the row is reached.
5. If there is no module adjacent to the one where attachment is being considered, the face on the side will be in some state other than *done*. In this case, the corresponding start bit can be set to 1, since there are no modules in that direction to pass a message to.
6. Once both start bits are set, confirming that the attachment information is being propagated in both directions as necessary, the attachment is finalized. The states of the other faces of the newly added module are set based on the analogous faces of the previously existing module (A and B), with the function f defined in the text. (However, any faces bordering sites meant to be left empty according to the shape map are instead set to *edge*.)

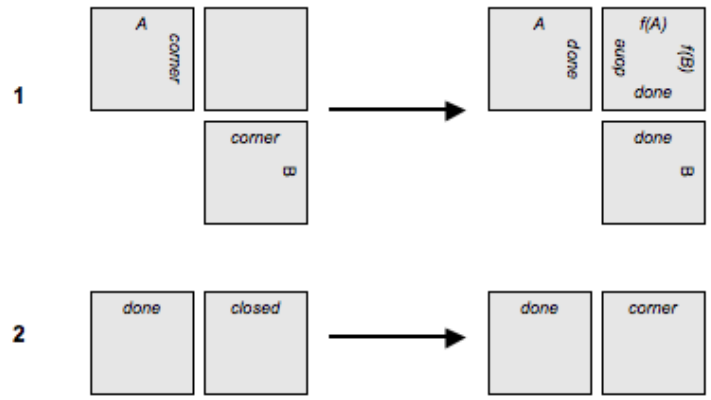

Figure 7: Rules for attachment in rows where modules have already been attached. In such rows, attachment is permitted only at *corner* sites.(1) The attachment can be completed immediately, without the need to propagate information down the row. The states of the other faces of the newly added module are set based on the analogous faces of the previously existing modules A and B, with the function f defined in the text. (2) Once attachment is completed, the next adjacent site (if any) is unlocked by setting it to *corner*.

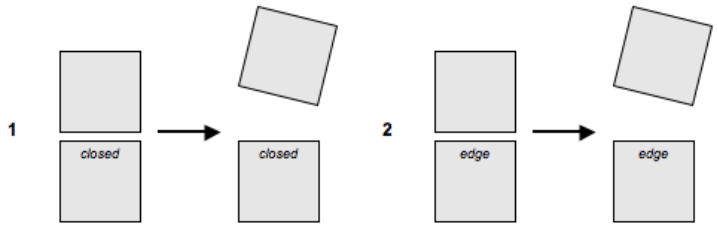

Figure 8: Rules for rejecting potential attachments. If a new module (blank) comes to a *closed* or *edge* face, it is rejected and the connection is broken.

module. Rows start out *open*; when the first module gets attached, a message is passed down the row of modules to put the corresponding faces into non-*open* states; thereafter, attaching additional modules in the same row affects the states of modules no more than one site away, and longer-range coordination is not needed.

Face states for a newly added module are based on face states of the module(s) the new one is attached to, as shown in the figures. The function f is defined as *f(done)=corner; f({open, edge})=open; f({corner, closed})=closed*.[5] The exception is if a face borders a site that should be left empty according to the shape map, in which case the face takes on state *edge*.

In some ways these rules look as though they could be condensed; nor do we claim they are minimal. However, some apparent redundancy is present to ensure that all necessary transitions occur: because the geometric graph grammars framework enforces no ordering on rule evaluation in situations where more than one rule applies, additional rules and states need to be added to prevent cases where a problem occurs if the wrong rule is evaluated first. For instance, the *start* state and its two associated bits are used to ensure that *open* faces in the rest of the row are closed off; without those bits, rule 6 in Figure 6 might be applied before rule 2, and the rest of the row would inappropriately remain *open*.

[5]In this section we assume that only one module is attached at a time, so no other faces will have states *start* or *waiting*, and f need not be defined on those states.

4.1.3 Grammar for adding multiple modules

In real self-assembly scenarios, many modules will be present on all sides of the assembly at once, and conflicts are possible if two modules approach the same *open* row simultaneously. More face states and rules are necessary to prevent the possibility of the assembly getting stuck in an incomplete configuration. A sufficient approach is based on Algorithm 5 in [11].

The idea is that before a new module is given permission to attach in an *open* row, all associated *open* faces must enter a state to lock out attachment of other modules in the same row; only after the entire row has been locked can the first module be given permission to permanently attach. To that end, a new module will trigger a wave of state changes in both directions along the *open* row; that wave must travel down to both ends and return to the potential attachment site to convey the information that the row has been successfully locked. If two modules try to attach to the same *open* row at once, and two such locking messages meet each other midway as a result, they trigger a different set of state changes returning to both potential attachment sites, directing both to reject attachment.[6] Finally, depending on whether attachment was allowed or forbidden, the appropriate module faces are set to *closed* as in the single-module case, or reset to *open*.

We implement this approach with several additional face states: {*request, ok, no, accept, reject, cancel*}. Like *start*, *accept* and *reject* states have two further associated bits; *request*, *ok*, *no*, and *cancel* states are associated with a direction along the row *f(request→)=←ok; f(ok→)=ok→; f(no→)=no→; f(accept)=corner; f(reject)=open; f(cancel→)=cancel→*. Figures 10–12 in the Appendix depict the rules for this case..

4.2 Proof of Correctness

The CollectiveAssembly algorithm is a direct translation of the CollectiveConstruction algorithm previously characterized in [11], therefore it preserves all the algorithmic guarantees of the original setting i.e. provable construction of 2D solid shapes with no defects or deadlock. The original proof uses induction to show that at each step, if the shape is not complete, then there must exist a viable attachment site that does not violate either the desired shape or the gap-avoidance requirements.

4.3 Criterion 1: Complexity

The complexity of the module program under CollectiveAssembly is based on the number of distinct locations there are within the desired shape—that is, the number of modules. Each unique coordinate corresponds to a unique body label, resulting in n state labels. Each face state can take on a constant number of labels (< 20), from orientation to open/closed/etc, however this does not depend on the shape size. Thus the complexity of the module state is $O(n)$. Similarly, the number of rules corresponds to the number of edges in the 2D shapes, which is at most $4n$. The gap avoiding rules contribute a constant number of additional rules, that again do not depend on the size of the desired shape.

Thus the module program complexity is $O(n)$ states and $O(n)$ rules, where n is the number of modules in the desired final shape. Again, this complexity is independent of the symmetry or asymmetry of the desired shape and depends only on the size of the desired shape.

4.4 Criterion 2: Best-Case Parallelism

The assembly process starts from a single initiation point and proceeds outward in all directions in a connected form. This observation suggest that a lower bound on the assembly time, for an arbitrary desired shape, is the longest shortest path from the seed to any point in the desired shape (call this the diameter of the shape). As we have previously shown for the CollectiveConstruction algorithm [13], this is a tight lower bound. One can construct a best-case assembly process that proceeds without creating any gaps in the structure and achieves the shape in exactly diameter steps: at each time step, if modules are added everywhere they may permissibly be added, then any site whose shortest path from the seed is m modules will become occupied on the mth time step.

[6] Another approach would be to give one module precedence over the other, so that one is allowed to bind and the other rejected. For simplicity, we present only the simpler grammar which rejects both.

Thus the best-case parallelism is $O(diameter)$ of the desired shape.

4.5 Criterion 3 and 4: Provably-Achievable Shape Class and Module Restrictions

As discussed above, the class of shapes this algorithm provably works for are any 2D shape which contains no enclosed spaces nor alleys as narrow as one site wide.

Furthermore, any shape from that class will be assembled subject to conservative assumptions about module movement. Note that the rules simplify considerably if modules can freely move in single-width tunnels. However this does not fundamentally change the complexity, since that is linked to the size of the desired shape rather than the module movement.

5 Comparison Summary

In this section we summarize and discuss the comparison of the two algorithms, MakeGridGrammar and CollectiveAssembly, according to the criteria described: the complexity and scalability of the generated agent program, the best-case parallelism possible, the class of shapes for which correctness is provable, and the assumptions about limitations on possible module movement that the algorithm is able to accommodate. We also qualitatively discuss the likelihood of each algorithm approaching best-case performance in practice.

- The MakeGridGrammar approach generates agent programs whose complexity scales with structure size in the same way as CollectiveAssembly ($O(n)$, where n is the number of modules in the desired assembly). This result suggests that approaches using explicit coordinates (e.g., [3,8,9,12]) are not inherently worse than automatically generated graph grammars [7].

  Note that neither approach takes advantage of symmetry, asymmetry or repetitive structures in the shape. The complexity of forming a n by n square is the same as a jagged complex shape with n^2 modules. However it may be possible to adapt both approaches to take more advantage of the shape. For example, by using a common grammar for all side-chains of similar length, the complexity of a grammar for a square can be transformed from n^2 to $2n$. In future work, we will show how this can be done systematically for any 2D-backbone shape.

- The MakeGridGrammar approach is able to exploit much greater best-case parallelism, with the potential for assembly of any shape in a constant $O(1)$ number of steps, compared to $O(diameter)$ steps for CollectiveAssembly. This result stems from the fact that in the former case, many subparts can assemble simultaneously and then merge, whereas the latter approach builds a connected assembly from a single initiation point.

  Note, however, that performance close to the best case is relatively unlikely for MakeGridGrammar and relatively more likely for CollectiveAssembly, given the same shape. The simultaneous assembly of subparts also requires that the appropriate subparts line up in the right relative locations and orientations to bind together. When there are many types of subparts, some of which may be large, that concordance may be slow to occur. Moreover, unbound modules may become scarce as they quickly bind into early-stage subparts, leaving them unavailable for later stages. For both these reasons, probabilistic unbinding rules become necessary for the approach to have a reasonable chance of producing a desired assembly in a reasonable time in practice [?]. By contrast, CollectiveAssembly has structures grow from a single point of origin; unbound modules remain free and "undifferentiated" until incorporated into their final location in the assembly, and if the density of free modules in the system is kept high, the waiting time for an available module to appear at a site where a module can be accomodated is likely to be small.

- MakeGridGrammar is provably correct for a smaller class of 2D structures than is CollectiveAssembly (Fig. 9A). The former generates rules that will terminate only at the desired assembly for assemblies that can be described by a straight backbone with straight side chains. This disallows simple shapes, such as the letter "C" where parts of the structure cannot be connected via a unbroken line. CollectiveAssembly can produce a much larger class of shapes, with many complex subparts, with the main restriction being the lack of enclosed gaps.

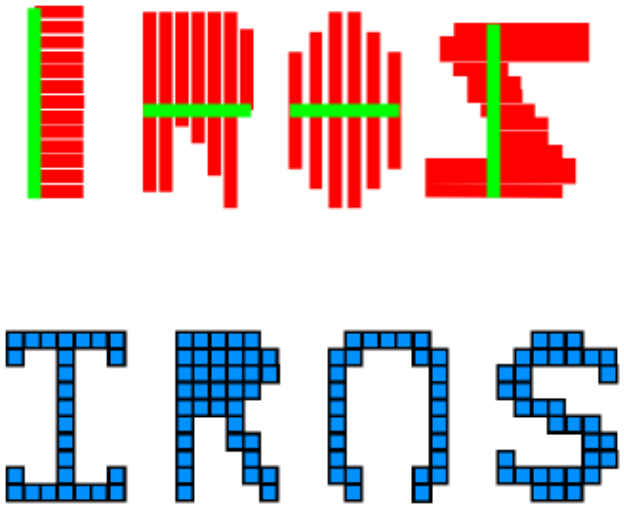

Figure 9: Examples of the structure classes for which MakeGridGrammar and CollectiveAssembly are provably correct. MakeGridGrammar (top): structures that can be drawn as straight backbones (green) plus orthogonal straight side chains (red). CollectiveAssembly (bottom): more general structures without internal holes.

- CollectiveAssembly accommodates modules with stricter motion constraints than does MakeGridGrammar. Many different possible constraints on physical movement of modules can be imagined: it may be that free modules are able to reach any space, even those entirely enclosed by other modules (e.g., if 2D modules can move in 3D); enclosed spaces may be unreachable, but modules may be able to move down narrow tunnels; or narrow tunnels may likewise be unnavigable.

  Currently, physically instantiated modules are likely to be closer to the more restrictive end of this spectrum of movement assumptions. CollectiveAssembly works with the most restrictive limitations on where modules can reach, while MakeGridGrammar assumes that modules and straight chains of modules are able to move down straight narrow tunnels. This difference may affect assembly speed and ease of physically realizing a system using each algorithm. For situations that do not match these assumptions, it is likely that CollectiveAssembly could be modified to be less restrictive and allow faster assembly, or that MakeGridGrammar could be modified to handle heavier restrictions on module movement at the cost of some parallelism; such modifications are left for future work.

6 Conclusion

In this paper, we have described some initial steps towards a framework for comparing algorithmic approaches to self-assembly. Our initial focus has been on 2D and two particular algorithms that describe dissimilar approaches to self-assembly. As we have shown, it is possible to use the framework to place both on an equal setting and provide theoretical statements about their behavior over large classes of shapes. Such comparisons are very important because they expose key elements of the approaches (why and when they work well) and expose new ways to improve their behavior.

We expect that in the future it will be possible to place other global-to-local shape algorithms in this common framework and compare them along the same axes. For example, many current approaches rely on explicit coordinate and connected formation which suggests that they will have similar complexity and best-case parallelism to CollectiveAssembly.

Finally, there are many types of shapes that have not been considered: for example, shapes with repeated/fractal structures, scale-invariant shapes and shapes that depend on the environment in which they are assembled. The two

algorithms presented assume that the desired shape is specified exactly. An interesting question is whether one can design global-to-local compilers that can self-assemble such shapes and whether such grammars will have inherently different complexity than the ones generated thus far.

References

[1] Daniel Arbuckle and Aristides Requicha. Active self-assembly. In *Proceedings of 2004 IEEE International Conference on Robotics and Automation*, pages 896–901, New Orleans, Louisiana, 2004.

[2] J. Bishop, S. Burden, Eric Klavins, R. Kreisberg, W. Malone, N. Napp, and T. Nguyen. Self-organizing programmable parts. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Edmonton, Canada, 2005.

[3] Michael De Rosa, Seth Goldstein, Peter Lee, Jason Campbell, and Padmanabhan Pillai. Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots. In *Proceedings of 2006 IEEE International Conference on Robotics and Automation*, Orlando, Florida, USA, May 2006.

[4] Robert Fitch, Zack Butler, and Daniela Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Las Vegas, USA, 2003.

[5] Ying Guo, Geoff Poulton, Phil Valencia, and Geoff James. Designing self-assembly for 2-dimensional building blocks. In *ESOA'03 Workshop*, Melbourne, Australia, July 2003.

[6] Chris Jones and Maja Matarić. From local to global behavior in intelligent self-assembly. In *Proceedings of 2003 IEEE International Conference on Robotics and Automation*, pages 721–726, Taipei, Taiwan, 2003.

[7] Eric Klavins, Robert Ghrist, and David Lipsky. A grammatical approach to self-organizing robotic systems. *IEEE Transactions on Automatic Control*, 51(6):949–962, June 2006.

[8] Keith Kotay and Daniela Rus. Generic distributed assembly and repair algorithms for self-reconfiguring robots. In *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

[9] Kasper Støy. How to construct dense objects with self-reconfigurable robots. In *Proceedings of European Robotics Symposium*, pages 27–37, Palermo, Italy, May 2006.

[10] Serguei Vassilvitskii, Mark Yim, and John Suh. A complete, local and parallel reconfiguration algorithm for cube style modular robots. In *Proceedings of 2002 IEEE International Conference on Robotics and Automation*, pages 117–122, Washington, DC, USA, 2002.

[11] Justin Werfel. *Anthills Built to Order: Automating Construction with Artificial Swarms*. PhD dissertation, Massachusetts Institute of Technology, MIT Computer Science and Artificial Intelligence Laboratory, May 2006.

[12] Justin Werfel, Yaneer Bar-Yam, Daniela Rus, and Radhika Nagpal. Distributed construction by mobile robots with enhanced building blocks. In *Proceedings of 2006 IEEE International Conference on Robotics and Automation*, Orlando, USA, 2006.

[13] Justin Werfel and Radhika Nagpal. Extended stigmergy in collective construction. *IEEE Intelligent Systems*, 21(2):20–28, March-April 2006.

[14] Paul White, K. Kopanski, and Hod Lipson. Stochastic self-reconfigurable cellular robotics. In *Proceedings of 2004 IEEE International Conference on Robotics and Automation*, pages 2888–2893, New Orleans, Louisiana, 2004.

7 Appendix

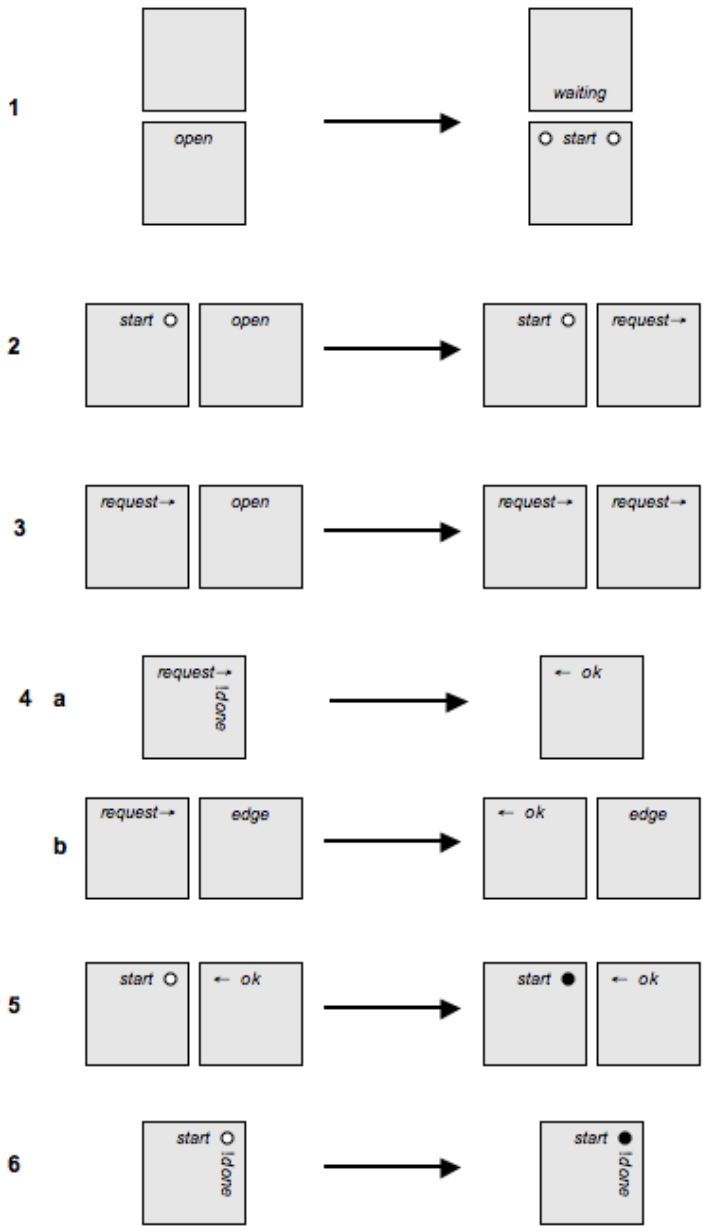

Figure 10: Modified rules for attachment when multiple modules may be present to try to bind at the same time to the same *open* row: part I.
1. Rule 1 of Fig. 6 continues to apply in this case, where a new module approaches an *open* face and sets off a sequence of state changes.
2. The *start* face sends *request*s in both directions.
3. A *request* continues to travel down an *open* row.
4. If a *request* reaches the end of a row (either (a) where there are no further modules yet, or (b) where the shape map specifies a site should be left empty), it sends an *ok* back in the other direction to indicate that all faces in this direction are now locked and attachment may proceed.
5. If an *ok* reaches the *start* face, the bit corresponding to that direction is set to 1.
6. As before, if there is no module adjacent to the *start* module in either direction, the corresponding *start* bit can be set to 1.

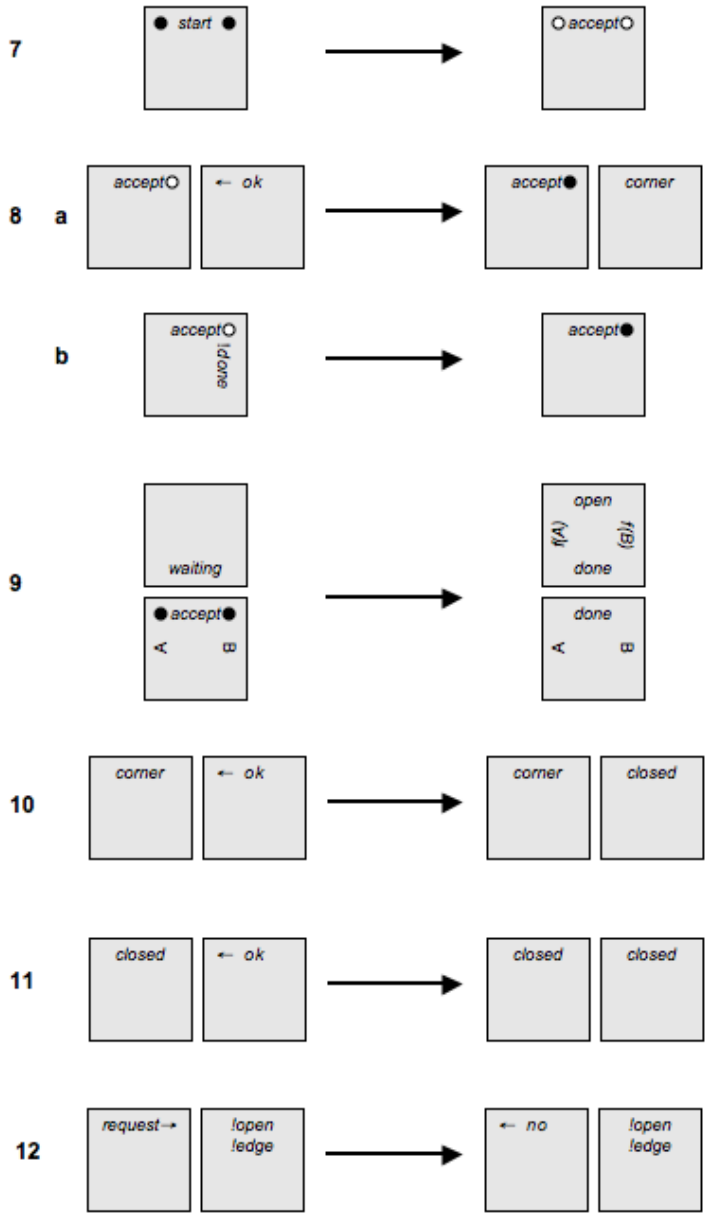

Figure 11: Modified rules for attachment when multiple modules may be present to try to bind at the same time to the same *open* row: part II.
7. If both directions have been confirmed as successfully locked, the *start* module can *accept* the attachment.
8. Doing so corresponds to granting attachment in the single-module case: first (a) the corresponding face of the adjacent module in each direction, if any, is set to *corner*, or (b) if there is no adjacent module, the bit in that direction is set accordingly ...
9. ... once both directions are accounted for, attachment is granted to the *waiting* module ...
10. ... the *corner* states turn their neighbors, if any, to *closed* ...
11. ... and the *closed* state propagates down the row.
12. If, instead, a *request* reaches a face which is neither *open* nor an *edge*, a *no* is sent back to indicate that attachment at the original site should be forbidden.

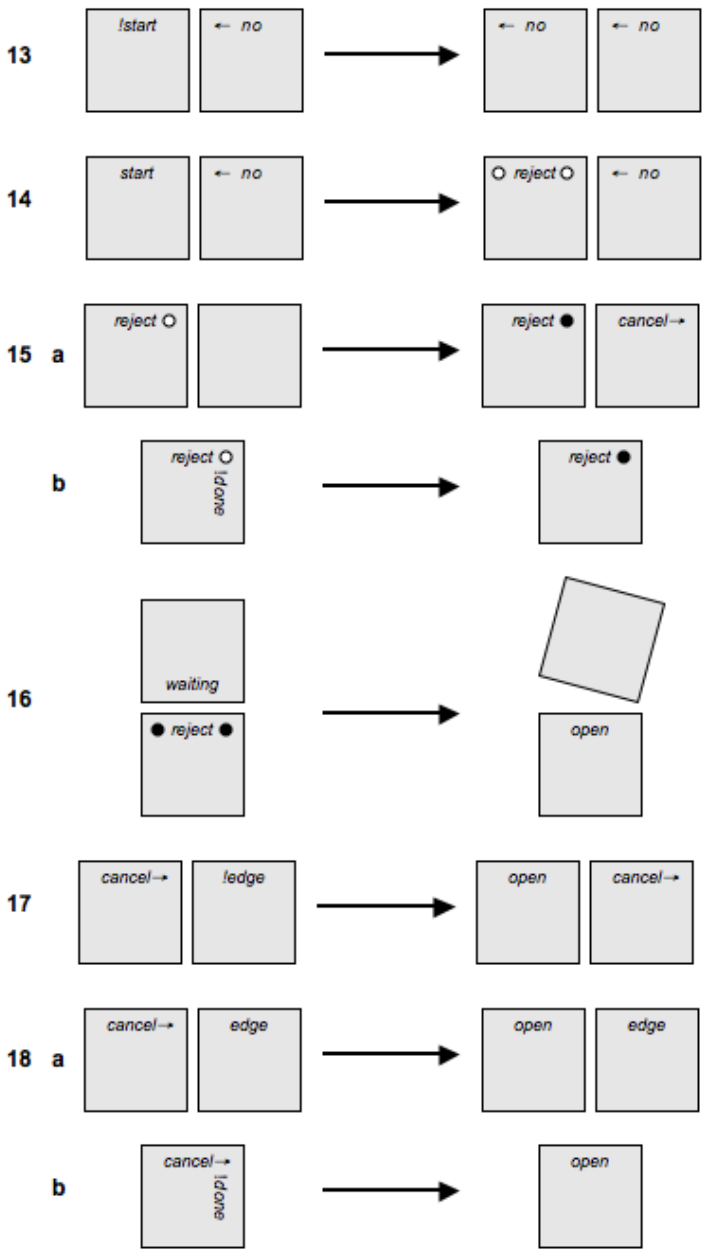

Figure 12: Modified rules for attachment when multiple modules may be present to try to bind at the same time to the same *open* row: part III.

13. A *no* propagates until it reaches the *start* face.
14. When it does, that face enters a *reject* state.
15. Now the process of rejection is analogous to that of granting permission: first (a) the corresponding face of the adjacent module in each direction, if any, is set to *cancel*, or (b) if there is no adjacent module, the bit in that direction is set accordingly ...
16. ... once cancellation in both directions is underway, the *waiting* module is rejected and returns to the unbound state, while the *reject*ing module becomes *open* again ...
17. ... the *cancel* message continues down the row, setting faces to *open* as it passes ...
18. ... until it reaches the end of the row, at which point it becomes *open*.

All the rules of Figs. 7 and 8 also apply in this multi-module case. New modules are rejected from *request*, *ok*, *no*, and *cancel* faces just as they are from *closed* and *edge* faces.