Self-Reconfiguration Planning for a Class of Modular Robots

Arancha Casal [*a], Mark Yim[b]
[ab]Xerox Palo Alto Research Center
[a]Robotics Laboratory, Stanford University

ABSTRACT

Modular self-reconfigurable robots consist of large numbers (hundreds or thousands) of identical modules that possess the ability to reconfigure into different shapes as required by the task at hand. For example, such a robot could start out as a snake to traverse a narrow pipe, then re-assemble itself into a six-legged spider to move over uneven terrain, growing a pair of arms to pick up and manipulate an object at the same time.

This paper examines the self-reconfiguration problem and presents a divide-and-conquer strategy to solve reconfiguration for a class of problems referred to as *closed-chain reconfiguration*. This class includes reconfigurable robots whose topologies are described by one-dimensional combinatorial topology. A robot topology is first decomposed into a hierarchy of small "substructures" (subgraphs of modules) belonging to a finite set. Basic reconfiguration operations between the substructures in the set are precomputed, optimized and stored in a lookup table. The entire reconfiguration then consists of an ordered series of simple, precomputed sub-reconfigurations happening locally among the substructures.

Keywords: self-reconfiguration, modular robots, reconfiguration planning

1. INTRODUCTION

Modular self-reconfigurable robotics is a relatively new concept that is gaining in popularity as evidenced by the increasing number of research groups building and studying these systems [1-11]. The basic idea is that a complex robotic system can be constructed from a collection of many simple, self-contained elements or modules, whose functionality comes from the modules interacting together and being able to self-assemble in different ways.

The ability to change shape as needed translates into high task versatility. Potential applications fall within three broad categories: locomotion, manipulation, and static structures. The large range of possible shapes allow these systems to accomplish tasks in all three categories. A scenario that may require all three is during a search-and-rescue mission. A robot shape could be generated to traverse the narrow passages in a rubble pile and carry a camera to detect a potential victim. Another shape used to remove pieces of rubble, deliver communication and supplies. Finally a supporting structure can be created to protect the victim.

This paper uses the term modular self-reconfigurable robot to refer to a system with the following characteristics:

- It is comprised of basic units called modules, where the modules are all identical. This allows for homogenous treatment of the modules in the planning problem.
- A module can attach and detach from other modules and possibly have internal degrees of freedom.
- It can change its shape autonomously, by changing the connectivity among the modules.

A number of hardware designs exist which satisfy all of the above properties [1-10]. From the standpoint of reconfiguration, different implementations of the concept have resulted in three distinct "reconfiguration classes" differentiated by the module motion constraints and basic reconfiguration steps. These are presented in Section 2.

Despite the relative hardware simplicity of the modules (as compared to other traditional robots), there are challenging research issues associated with the cooperative, distributed nature inherent to these systems. Self-reconfiguration, the focus of this paper, is one such issue.

[*] Correspondence: Email: acasal@parc.xerox.com

The self-reconfiguration planning problem can be defined as the determination of a realizable sequence of module motions that changes an initial configuration into a desired goal configuration, without external intervention. When we consider systems with large numbers of modules, one of the main difficulties of self-reconfiguration resides in the fact that the attainable configurations can be extremely complex. A planner must compute a sequence of module motions to transform from an arbitrary initial configuration into the desired goal, both of which are possibly very intricate. A good planner must do so efficiently.

1.1 Optimal Reconfiguration

It would be desirable to design an optimal algorithm that minimizes the number of steps required to reach the final configuration, or some other measure of optimality (total actuator effort, time, etc).

However, there is no simple solution for computing the optimal sequence of moves required to reconfigure. The reason is that the search space (that is, the number of possible sequences of configurations) grows exponentially with the number of modules in the system. As a result, we must look for heuristic solutions with some guarantee of performance.

The optimal reconfiguration problem was first studied by Pamecha et al. [18, 21]. It is a combinatorial optimization problem that bears the hallmarks of a NP-complete problem, although no formal proof has been published yet. It is reminiscent of the classical Transformation problem in Operations Research, which is NP-complete. But in the latter problem an external operator is allowed to move the pieces to goal locations in the target shape without any motion constraints [19,20].

1.2 Related Research

To this date, few algorithms to solve reconfiguration planning have been published. Owing to the computationally complex nature of the problem, the proposed solutions are non-optimal and work for restricted cases. They are discussed below.

Pamecha et al. [21] present a near-optimal solution based on the minimization by simulated annealing of a formal "configuration distance" metric to measure the difference between an initial and a goal configuration. The method is only computationally feasible for one module moving at a time. For typical systems consisting of large numbers of modules, the restriction of single module motion may render this solution too slow.

Yim et al. [2] introduce a rule-based heuristic method for a three dimensional system that centers around the setting of ordering and priorities and relies on cooperation to achieve individual module goals. Results are shown for large systems involving multiple module motion, although pathological cases exist where complete reconfiguration is not achieved.

Murata et al. [4,5] present a "stochastic relaxation" method implemented on two and three dimensional systems. The method is based on potential fields and weighted probabilities to uniformly guide the motion of modules toward reachable positions in the target configuration. A way to describe target shapes is also introduced and used in the algorithm. However, results are not shown for large systems or arbitrary shapes.

All of the above algorithms were proposed for a class of reconfigurable systems in which modules nominally lie in discrete positions in a lattice. These kinds of systems have special characteristics, which determine which reconfiguration algorithms are suitable to consider and are referred to as the *substrate* reconfiguration class. To the authors' knowledge, this paper is the first time an approach has been published for the *closed-chain* reconfiguration class, and it can be subsequently used as a study of the characteristics inherent to this class of problems. These classes are more fully described in Section 2.

The ideas of modularity and reconfigurability in robotics have appeared in the literature for other related concepts. These include versatile manipulator design based on the assembly of different specialized modules which can be (manually) combined in different ways to suit various tasks [12,13,14,15]; and cooperative systems of heterogenous mobile robots which can attach together in varying degrees and separate to achieve a common goal [16].

2. RECONFIGURATION CLASSES

The term "configuration" is used in the paper to refer to a distinct topology of the system, that is, a particular connectivity arrangement of the connected modules[1].

We define the "reconfiguration space" to be the space of all configurations for a given set of labeled modules[28]. Two configurations are adjacent if one can be transformed into the other by one atomic set of actions which we call an *rmove*. The actions that define an *rmove* depend on the basic mobility of a module in a given set, typically consisting of one attach or detach and some motion of the modules' degrees of freedom. This mobility is used to differentiate modular self-reconfigurable systems into one of three "reconfiguration classes".

The three reconfiguration classes are: 1) *Mobile* Reconfiguration, 2) *Substrate* Reconfiguration and 3) *Closed-chain* Reconfiguration. The reconfiguration class impacts the kinds of algorithms that are suitable to consider.

2.1 Mobile Reconfiguration

Mobile reconfiguration consists of modules that can move in an environment without the assistance of other modules. These typically consist of a set of mobile bases that are individually maneuverable and that can also attach together. When modules disconnect and maneuver around the environment, the modules are no longer one single connected component. Since the reconfiguration space considers only the connected modules of the original set, we do not consider such instances to be configurations. We do not address the arrangements of multiple disconnected robots. The sequence of actions in an *rmove* is characterized by 1) a set of modules detach, 2) that set of modules maneuver in the environment, and 3) the set attaches to new location(s). Examples of existing hardware systems include [11,16].

2.2 Substrate Reconfiguration

The class we refer to as Substrate reconfiguration includes systems where modules can only stop and attach to other modules in discrete locations on a lattice. Motions between those locations occur only along prescribed paths. When a module is transitioning between two lattice positions, we do not consider such instances as configurations in the reconfiguration space. The sequence of actions in an *rmove* for *substrate* reconfigurations is typically 1) a set of modules detaches, 2) they move to new location in the lattice, and 3) they attach to these new locations. Figure 1 is an illustration of a reconfiguration sequence for a hardware system, Proteo, developed at Xerox PARC [2]. Several other hardware platforms in this class exist to date [4,5,6,7,10].

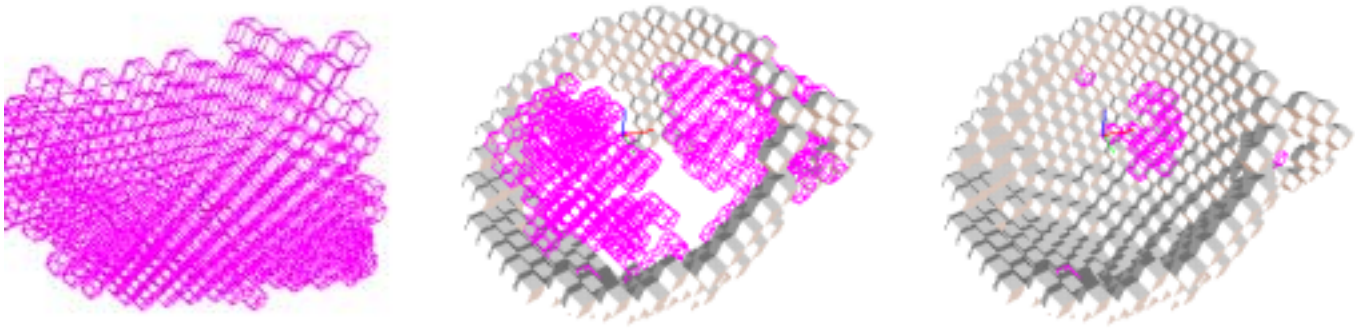

Figure 1: Proteo system reconfiguring from a square plate to a teacup shape.

2.3 Closed-chain Reconfiguration

By contrast, Closed-chain reconfiguration relates to robot systems made up of serial kinematic chains (open or closed) which can meet at common branch-out points to form complex configurations. In this class, the basic reconfiguration primitive involves a serial chain of modules reattaching its free end to a new point and possibly detaching from a different point. In the previous two classes, an *rmove* consisted of both attach and detach actions and intermediate steps were not considered to be a configuration in the reconfiguration space. In this class, however, intermediate steps between attach and detach actions are indeed considered to be configurations in reconfiguration space. Thus an *rmove* for this class is either a single attach or

[1] Note this is not the *configuration* of a robot in Configuration space, as used in motion planning [27].

detach together with possibly some motion of the modules' degrees of freedom. Figure 2 shows a reconfiguration sequence for this class. Hardware implementations of this concept include Polypod[1] and its successor PolyBot[8], and CONRO[9].

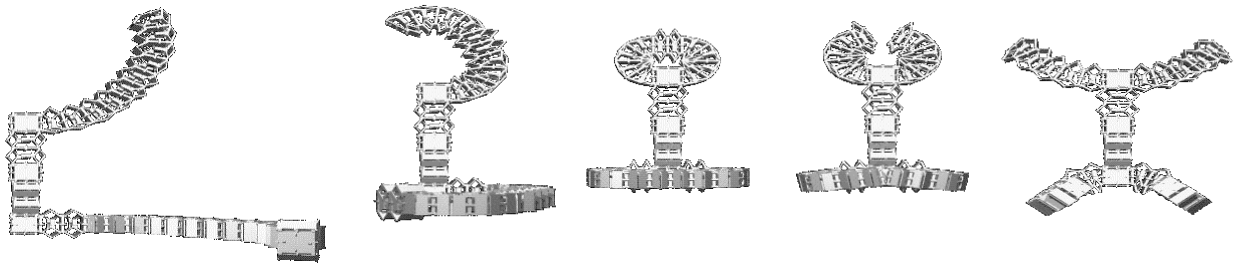

Figure 2: Reconfiguration sequence for Polypod belonging to the *closed-chain* class.

2.4 Characteristics

Mathematically, the space of possible configurations for the *closed-chain* class is formalized by a one-dimensional combinatorial topology [17], where topological complexes, graphs in this case, are constructed from the combination of two basic elements, or simplexes, namely edges and vertices. Surfaces, 3-D solids and polyhedra are excluded from this class, although skeletal framework approximations are possible.

Substrate systems have a more general topological space than the *closed-chain* type, and are particularly well suited to create surfaces. Furthermore, if the modules in a *substrate*-type system can deform, it could have the same basic reconfiguration mechanics as a *closed-chain* system, albeit perhaps not to an advantage. *Closed-chain* type systems, however, still offer important advantages. Their mechanical implementation is, in general, simpler than for *substrate*-type systems. For many practical locomotion and manipulation tasks, one-dimensional topological complexes suffice, and are in fact sometimes preferred. The algorithms presented in this paper apply to the *closed-chain* reconfiguration class. Nonetheless, the topological space of this class can be a subset of the *substrate* class, and so the method could be used selectively on the latter.

Since the motion trajectories involved in an *rmove* for the *substrate* class consist of prescribed discrete motions over a lattice the related inverse kinematics and motion planning problems are greatly simplified. On the other hand, *closed-chain* systems will, in general, require the generation of motion trajectories for chains made up of many modules. In these cases, the dimensionality of the related motion planning and kinematic considerations may be high. Optimal reconfiguration, as discussed in Section 1.1, is computationally intractable for both classes.

3. ALGORITHMS FOR CLOSED-CHAIN RECONFIGURATION

3.1 The Reconfiguration Planning Problem

A reconfiguration planner must produce a self-reconfiguration plan that is a sequence of adjacent configurations, or alternatively a sequence of *rmove*s, connecting an arbitrary configuration to a goal configurations in the reconfiguration space. For the *closed-chain* class, it must ensure in addition, that this sequence of connectivity changes be accompanied by physically realizable trajectories for the moving kinematic chains. To be feasible, these trajectories must 1) be reachable within the workspace of the chain, 2) not result in a loss of stability, under gravity, of the robot, and 3) be free of collisions with itself or otherwise.

The method presented in this paper produces a sequence of connectivity changes for reconfiguration disregarding any motion planning concerns. These considerations will be the subject of future work. The topological space covered by the method is that of planar graphs (a subset of 1-dimensional combinatorial topology, see Section 3.3).

3.2 Equivalence between Configurations

The underlying module-to-module connectivity in a given configuration can be represented as a "module graph", where vertices correspond to individual modules and an edge is drawn between two modules if they are connected neighbors. Figure 3 shows a robot configuration and its corresponding module graph. Note that from now on robot configurations will de depicted by their wireframe equivalent, as introduced in this figure.

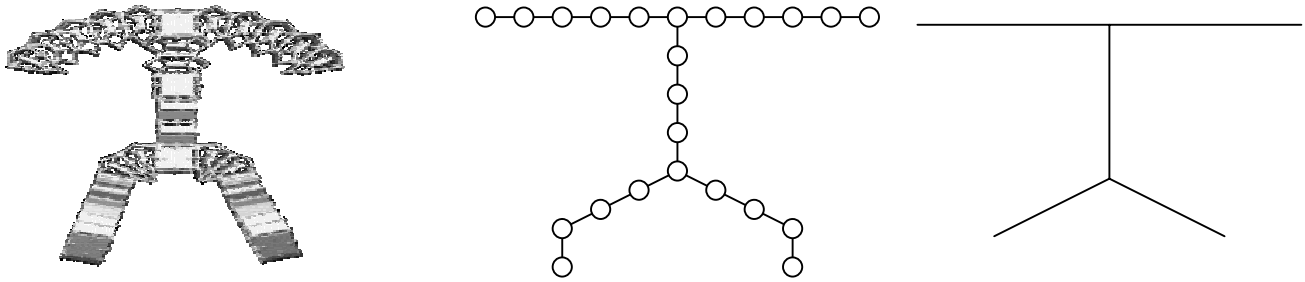

Figure 3: A Polypod configuration, its module graph representation and a wireframe depiction of the robot.

We define two configurations as *equivalent* iff their module graphs are the same. In the reconfiguration process, we can use this definition of equivalence to determine if the goal configuration has been reached by the sequence of *rmove*s output by an algorithm.

3.3 Approach

The method proposed in this paper is a reductionist approach whereby a configuration is decomposed into a hierarchy of small components, called *substructures*, which belong to a finite set. There are a number of operations defined on the set where, like an algebra, elements of the set can be combined to form another element in the set. Operations consist of simple reconfiguration sequences among substructures that are precomputed, optimized and stored in a lookup table for later use by the planner. Modular reconfigurable robots are inherently distributed systems. This divide-and-conquer approach easily lends itself to a distributed implementation.

The initial and goal configurations, **I** and **G** respectively, are decomposed as an ordered hierarchy of components (substructures). The reconfiguration process consists of performing a correct sequence of operations among the substructures of **I** to transform them into those of **G** and their corresponding hierarchical arrangement.

We refer to this approach as Hierarchical Substructure Decomposition (HSD). The proposed algorithms consist of two main stages:

1. decomposition of **I** and **G**
2. reconfiguration using basic substructure operations

The remainder of this section will introduce the set of substructures and operations, and the hierarchy construction. It will also discuss some of the characteristics inherent in the method, such as the non-uniqueness of decomposition, and their consequences. It will then present two similar HSD-based algorithms for reconfiguration.

3.3.1 Substructure Set

A set of substructures σ is selected as the basic elements with which to construct complexes (configurations). These substructures are topological groupings of several modules that satisfy the following characteristics:

1. They are topologically distinct or non-homeomorphic, i.e. one cannot change into the other without changing its connectivity.
2. They appear recurringly in a large number of configurations.
3. The reconfiguration operations between any two in the set are simple and well-known.

For instance, we can select σ to have two elements: *chain* and *loop*. These are defined as follows:

- A *chain* is a continuously connected series of modules.
- A *loop* is a *chain* whose two ends are connected or a cycle of *chains*.

It should be noted that this is an artificial decomposition, imposed to suit the purposes of this particular approach to reconfiguration. It is possible to choose a different set of substructures. For the remainder of this paper, we will assume that σ={*chain*, *loop*}. This choice satisfies the above properties, and as we shall see shortly it facilitates the hierarchical decomposition of a configuration.

A number of basic reconfiguration operations are defined for σ. These operations are:

- *transformation* into a different substructure in the set
- *merge* of two connected substructures
- *split* of a substructure in two
- *relocation* of a substructure with respect to its parent substructure

The basic operations are precomputed, optimized and indexed in a lookup table. The HSD-based algorithm achieves the global changes in connectivity required for reconfiguration by invoking an ordered sequence of these simple operations happening locally among substructures.

The first three operations all involve a small number of *rmove*s. The *relocation* operation also involves a small number of *rmove*s. Nevertheless, when we consider a *real* motion trajectory to realize the *relocation*, it may be the case that the number of *rmove*s needs to be large. For example, consider a substructure with short reachability that must nonetheless reach a target location a long distance away. In reality, several *rmove*s may be needed to complete the relocation operation, each of which move it increasingly closer to the target location. This paper, however, does not address the issue of reachability and motion planning for trajectories (see Section 2.4); thus *relocation* operations are indeed trivial.

3.3.2 Hierarchy Construction

The decomposition step first involves traversing the configuration module by module to identify substructures. Following the definition of the σ substructures above, there is no single unique way to group modules into substructures. This issue warrants further discussion and is presented in Section 3.5.

Once the substructures have been identified, we treat all the modules in a given substructure as a distinct, atomic unit. We no longer think of the configuration as a module graph, but having effectively partitioned the configuration into distinct groups of modules, we now view it as a connected *network of substructures* (*loop*s and *chains*), see Figure 4 below.

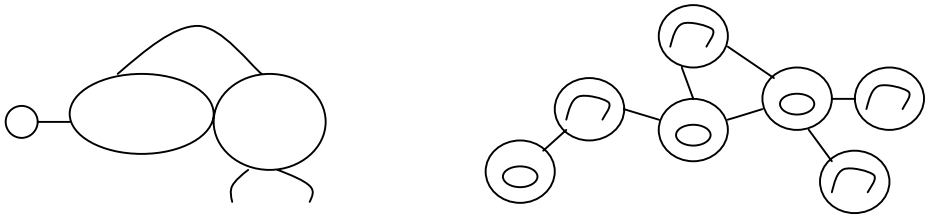

Figure 4: A wireframe depiction of a configuration and its corresponding partition into a network of substructures for σ={*chain*, *loop*}.

Once the configuration has been partitioned into substructures, we wish to describe how the substructures are connected relative to one another. In order to express a notion of order among the component substructures, we can describe their relative arrangement as a hierarchy, or tree. We can construct a "substructure tree" in a standard way [25] where each substructure becomes a node of the tree. The procedure the algorithms follow to construct the tree is detailed in Section 3.4.

In moving from a network of substructures to a hierarchy, we are effectively transforming a graph into a tree. A tree is an ordered, acyclic graph, meaning there must be no cycles between nodes (substructures). The choice of a *loop* as a basic element in σ effectively removes cycles. However, even if *loop*s are part of σ, there are still cases where cycles cannot be removed by the partitioning, as would be the case in Figure 4. This occurs when substructures form cycles among themselves. Such cases require special treatment and are the subject of Section 3.6.

A hierarchical decomposition has several advantages. It allows for a concise, ordered description of a configuration in terms of levels and substructures per level. This ordering can be used by an algorithm to guide the reconfiguration in a systematic way, and to quantify the difference between a current configuration and its desired goal at any point during the reconfiguration process.

Furthermore, a hierarchy seems like the preferred architecture of complex systems, natural and artificial. Complex systems might, in general, be expected to be constructed, or decomposable, as a hierarchy of levels where the components perform

particular sub-functions that contribute to the overall function. In nature, complex systems can evolve from simple systems more rapidly if there are stable intermediate configurations, which means the resulting complex form will be hierarchic. This argument has been put forward by several authors [22,23,24]. A complex modular robot configuration intended for manipulation or locomotion is also likely to be constructed hierarchically, with larger "macro" components performing coarse motions and mounted "mini" components performing shorter range, higher-precision (high mechanical bandwidth) operations.

3.4 HSD-based Algorithms

We now present two self-reconfiguration algorithms. As a first step, both perform a decomposition step where **I** and **G** are hierarchically organized as a tree of substructures, as discussed in Section 3.3.2. As an illustration, consider **I** and **G** in Figure 5a and 5b, and their corresponding decompositions. We will use these example configurations throughout the section to illustrate the algorithms.

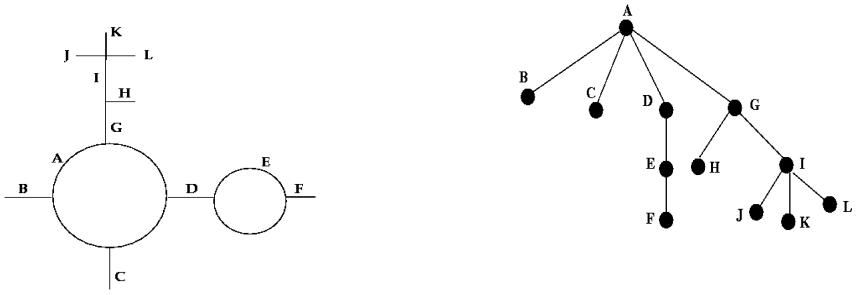

Figure 5a: Initial configuration, **I**, and resulting HSD. Nodes are labeled to indicate the associated substructure.

Figure 5b: Goal configuration, **G**, and resulting HSD. Nodes are labeled to indicate the associated substructure.

The procedure the algorithms follow to generate the substructure tree, for σ = { *chain*, *loop* }, is as follows:

1) Substructure selection.

The module-to-module connectivity of the configuration is traversed. A module is called a "vertex" module if it is connected to more than two neighbor modules (is at a branch-out point) or to only one neighbor (is at a free end). All other modules have two neighbors and lie between vertex modules.

A substructure is the shortest list of distinct modules lying between two vertex modules. If the two vertex modules are the same, the substructure is of type *loop*. Conversely, if the vertices are distinct the type is *chain*. The special case of a single cycle (with no vertex modules) is handled explicitly. A second step further creates *loop*s by searching for cycles among the existing *chain*s. If there are any, all the *chain*s involved in the cycle are now grouped together as a single *loop* substructure and no longer considered as a number individual *chain*s.

Each substructure has an associated data structure containing the following information: size (number of modules in the substructure), type of substructure (*chain* or *loop*), pointers to neighbor substructures (other substructures connected at its vertex modules).

2) Hierarchy creation

A substructure is selected as the root node (heuristics for this choice are discussed in Section 3.5). All neighbor substructures become a child node of the root, and the root their parent node. The children's children are recursively searched to create the corresponding parent/child relationships until childless nodes are reached.

Each node in the tree has an associated data structure that inherits the information of a substructure and the following: level in the tree, pointer to its parent, pointers to all its children, and list of "attachment points". We define *attachment points* as the relative locations (vertex modules) on a substructure where its children nodes are attached.

Once hierarchically decomposed, I and **G** can be compared for equivalence, as defined in Section 3.2. Specifically, **I** and **G** are equivalent if their trees are identical. That is, they have the same: 1) number of levels, 2) number of substructures per level, 3) type of substructures per level, 4) size of the substructures and 5) "attachment points" on each substructure.

Conveniently, the *difference* between two configurations can also be quantified in terms of their respective trees, that is in terms of the difference in the number of levels, number of substructures per level, etc. This measure of difference is used to generate the sequence of precomputed substructure operations in the reconfiguration stage of the algorithms as explained in the next section.

3.4.1 Canonical Form Algorithm

A very simple algorithm results in overall reconfiguration by first transforming into a simple intermediate configuration. It is based on two observations: 1) the symmetry of reconfiguration sequences, that is, a sequence of *rmove*s that takes **I** to **G** can be obtained by reversing the order of the sequence that transforms **G** to **I**; 2) "destructive" sequences that reconfigure a complex configuration into a simpler one are easier to devise than vice versa.

Based on these observations, we find the sequence of *rmove*s to reconfigure both **I** and **G** into an intermediate simple configuration, **C**. The reconfiguration from **I** to **G** is then:

$$I \to C \; + \; \left(G \to C\right)'$$

where $I \to C$ is the sequence of *rmove*s that takes **I** to **C**, and $\left(G \to C\right)'$ is the *reverse* of the sequence taking **G** to **C**, both of which are very simple to obtain. One of the simplest possible configurations is a single *chain*, which could be used as **C**. This is illustrated in Figure 6. Alternatively, different **I** and **G** pairs may have other more efficient intermediate configurations; in some cases it may be obvious what **C** should be while in others a cost must be incurred to search for it.

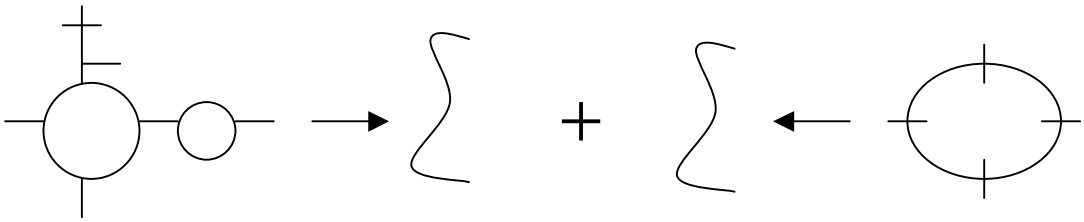

Figure 6: **I** can reconfigure into **G** by two easy reconfigurations into a simple intermediate configuration, **C**

The hierarchical decomposition is used to guide the sequence of operations needed to reconfigure **I** and **G** into **C**. If we select **C** as a single *chain* then the reconfiguration stage is simply a concatenation of *merge* operations (retrieved from a lookup table) resulting in ever decreasing number of levels and substructures.

This algorithm is, in general, sub-optimal as there may be redundant operations involved in going through the intermediate configuration. It is however, a very simple way to obtain otherwise difficult sequences of moves. It is an efficient heuristic solution for cases where the **I**, **G** pair are close to the intermediate configuration. The Canonical Form algorithm is also likely to perform better than the Matching algorithm presented below in cases where the latter requires many *relocation* operations, which may be more costly than the other substructure operations (see Section 3.3.1).

3.4.2 Matching Algorithm

This algorithm works by progressively matching the 5 equivalence conditions in Section 3.2. As a first step, the algorithm matches the **number of levels**. In this case, **G** has a smaller number of levels than **I**. In order to reduce the number of levels in **I** to match **G**'s, the algorithm invokes *merge* operations between the leaf nodes and their parent ($l_I - l_G$) times, where l_G and l_I are the number of levels in **G** and **I** respectively. Figure 7 shows the process.

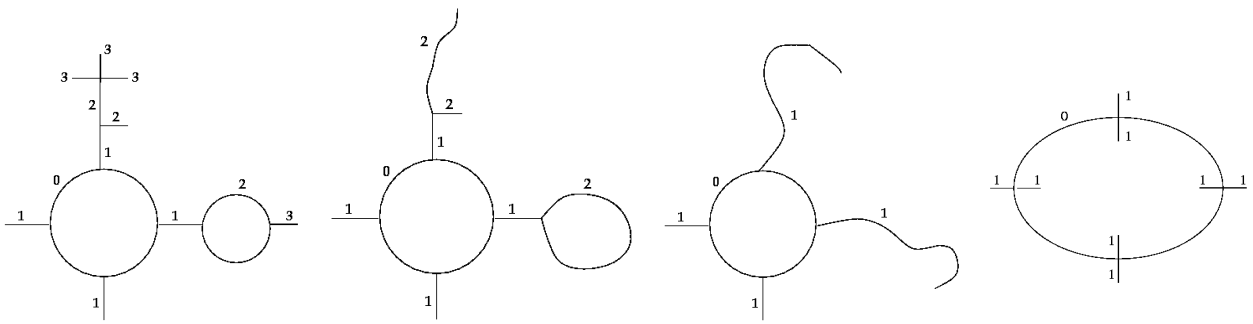

Figure 7: matching the number of levels and corresponding goal configuration, **G**. Numbers indicate the level for each substructure. At the end of the process **I** has two levels, like **G**.

Once the number of levels has been matched, the algorithm proceeds to match the **number of modules at each level**. A series of *merge* and *split* operations is invoked that effectively redistributes modules between levels. One pertinent observation is that the number of modules per level need not be matched exactly. As long as the relative sizes are fairly close (90% was used in our implementation), the functionality will be the same and we can save extra reconfiguration operations. In this particular **I**, **G** pair, once **I** has reconfigured to have only two levels like **G**, the number of modules per level is deemed close to the goal's and no matching is needed.

Next, the **size of the substructures at every level** is matched. Within a level, *merge*s and *split*s occur as needed to achieve the correct sizes. Figure 8 shows the process.

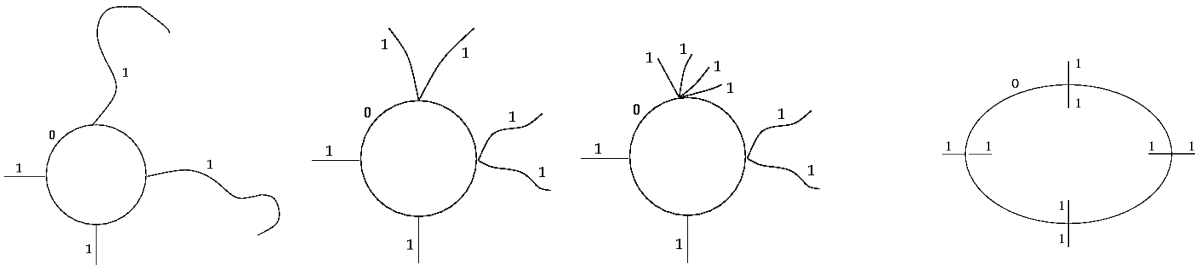

Figure 8: Matching the number and type of substructures at every level in **I** and corresponding goal **G**. At the end of the process, **I** has one substructure at level 0 and eight substructures at level 1, like **G**.

The **type of substructures at every level** is matched by invoking *transformation* operations to change a *loop* into a *chain* or vice versa. In the example shown, the types at every level already match those of the goal **G** and don't require any *transformation* operations.

As a last step, *relocation* operations attach the substructures at their goal **attachment points on their parent**. The *relocation* operations involve, in general, the moving of a branch of the tree relative to the parent substructure. If the branch is several levels deep and/or if the move is over a large distance, motion planning techniques must be brought to bear (see Section 2.4). This may result in these operations being costly or even unrealizable. However, if the *relocation* operations are simple, this algorithm will in general be more efficient than the Canonical Form algorithm, as it compares **I** and **G** for differences and seeks to make only those changes necessary to remove them.

3.5 Choices of Decomposition and Correspondence

It has been previously pointed out that following the definitions of σ, *chain* and *loop* above, there are many different ways in which to group modules into substructures. An example is shown in the Figure 9. The number of different ways of selecting substructures is proportional to the number of ways of partitioning the associated module graph. This number increases with the total number of modules.

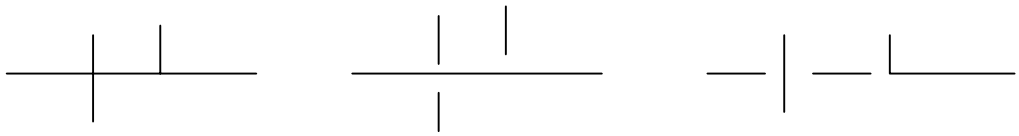

Figure 9: A configuration and two ambiguous ways to partition into substructures.

This initial choice of substructures determines to a large extent the ensuing hierarchical decomposition. This means that for a given **I** and **G** pair, there may be $n_I * n_G$ different ways to partition both configurations into substructures, where n_I and n_G are the number of different ways to select substructures in **I** and **G** respectively. There is, in addition, a choice involved in the creation of the tree, as a particular substructure must be distinguished to be the root.

The simplicity of the reconfiguration sequence depends to a large extent on these choices. We can make use of a concept of "correspondence" (closeness) between **I** and **G** to result in a simpler reconfiguration. In particular, we wish to select decompositions that result in **I** and **G** having trees as similar as possible. For instance, for the Matching algorithm we would like **I** and **G** to have:

- an equal (or as similar as possible) number of levels
- an equal (or as similar as possible) number of modules per level
- the same (or as similar as possible) number of substructures per level

For the Canonical Form algorithm, **I** and **G**'s decomposition should also correspond with **C** as closely as possible.

We can use heuristics to guide the selection of the root. For example, we may select the largest substructure (largest number of modules) to become the root in both **I** and **G**. This partly satisfies one of the correspondence rules above, and is resonant with the macro-mini concept of complex hierarchical systems. Or alternatively, we could balance both trees to try to satisfy the first rule.

Although these criteria do not necessarily yield an optimal solution, they help to guide the decomposition towards choices with good correspondence between **I** and **G**, and so result in simpler reconfigurations. However, searching for the best **I** and **G** pair can be costly, since it implies creating all possible decompositions for both **I** and **G** and then searching for the best pair among all ($n_I * n_G$) choices. It is also possible to simply pick a choice "blindly" for both **I** and **G** and continue with the decomposition, saving the cost of searching for the best correspondence.

3.6 Extensions

HSD analyses a network of substructures as a tree, which is an acyclic graph. By grouping modules in a cycle as a single *loop* substructure that becomes a node in the tree, we are effectively removing cycles from the graph. However, there are still cases where cycles remain. These include cases when a substructure (*loop* or *chain*) connects to more than one point on another substructure, or when it connects two substructures that belong to different levels. For example, this happens when *loop*s share common edges, or when there are cycles of substructures, as illustrated in Figure 10.

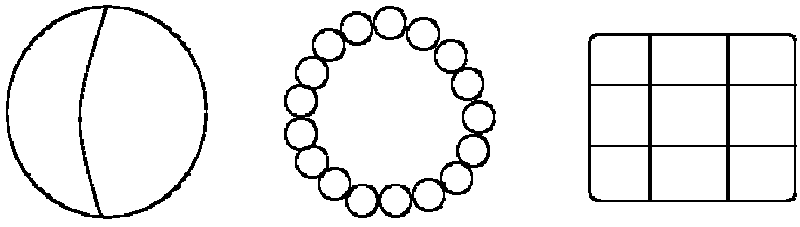

Figure 10: Overconstrained examples.

We call such substructures overconstrained, as an excess of connections causes the problem. The solution involves selectively disconnecting one of the attachment points of the overconstrained substructure so that the cycle is effectively removed and the tree can be constructed. Reverting to a module graph representation of a configuration (Section 3.2),

overconstrained cases occur whenever the graph contains 3-connected minors, i.e. there exist pairs of vertex modules with three or more distinct paths connecting them [26].

A clever choice of σ may in some cases partly, or even entirely, avoid the problem. For example, we may select σ to contain a substructure equal to the first configuration shown in Figure 10. By encapsulating this particular topological grouping of modules as a *single* substructure with its associated operations, we effectively remove any such overconstrained substructures from a global configuration.

One way to handle overconstrained **I** or **G** configurations in the planning is to form a virtual break in the overconstrained substructures as they are discovered in the decomposition. This will produce **I'** or **G**', which are free of overconstrained substructures. Planning can then happen normally between two non-overconstrained configurations. A step is added in the beginning of the reconfiguration plan to transform **I** to **I'** and a step at the end to transform **G'** to **G**, as needed. This process is sub-optimal since it introduces additional *rmoves* and removes overconstrained configurations from the reconfiguration space that may be indeed part of an optimal sequence.

3.7 Experimental Results

A computer simulation was run to test the HSD method. Several **I**,**G** pairs were used, the most complex configurations having up to four levels and twelve substructures. The desired goal configuration is achieved in all cases. One of the **I**,**G** pairs tested is the example pair used to illustrate the method in Section 3. The sequence of *rmove*s output by the method leads to the same changes shown in Figures 5-8, and these figures serve as illustration of the simulation results. Decompositions with overconstrained substructures, and subsequently needing treatment as described in Section 3.6, were also tested.

4. CONCLUSIONS

This paper introduces a taxonomy of reconfiguration classes for modular, self-reconfigurable robots and defines their characteristics. A study of the *closed-chain* reconfiguration class, its topological space and suitable reconfiguration algorithms are also presented. To the authors' knowledge, this is the first time a reconfiguration approach has been published for this class.

This paper introduces a heuristic approach called Hierarchical Substructure Decomposition (HSD). It centers on the partitioning of a robot configuration into small components, called substructures, and their subsequent ordering as a hierarchy. Reconfiguration of the entire system is thus divided into a series of simple, precomputed reconfiguration subsequences happening among the substructures.

Two algorithms based on HSD are described and simulation results discussed. The method presented outputs a sequence of connectivity changes to reconfigure between two arbitrary configurations. For the *closed-chain* class this sequence of connectivity changes must be accompanied by physically realizable trajectories for the moving kinematic chains. We do not address these considerations in the paper. As a next step, we intend to add a path planning stage to the method to produce feasible trajectories.

ACKNOWLEDGEMENTS

This project was partly funded by DARPA contract number MDA972-98-C-0009. We would like to thank Sunil Agrawal and Ying Zhang who reviewed this paper and provided important insight, and Jean-Claude Latombe and Leo Guibas for their guidance.

REFERENCES

1. Yim, M., "Locomotion With A Unit-Modular Reconfigurable Robot", Stanford University Thesis, 1994.
2. Yim, M., Lamping, J, Mao, E., Chase J.G., "Rhombic Dodecahedron Shape for Self-Assembling Robots", Xerox PARC, SPL TechReport P9710777, 1997.
3. Pamecha, A., Chiang, C-J., Stein, D., Chirikjian, G.S., "Design and Implementation of Metamorphic Robots", Proc. of the 1996 ASME Design Engineering Technical Conf. and Computers in Engineering Conference, 1996.
4. Murata S., Kurakawa, H., Kokaji, S., "Self-Assembling Machine", IEEE Proc. of Intl. Conf. On Robotics and Automation 1994.
5. Murata S. , Kurakawa, H., Yoshida, E., Tomita, K., Kokaji, S., "A 3-D Self-Reconfigurable Structure", IEEE Proc. of Intl. Conf. On Robotics and Automation, 1998.
6. Kotay, K., Rus, D., Vona, M., McGray, C., "The Self-reconfiguring Robotic Molecule: Design and Control Algorithms", Algorithmic Foundations of Robotics, 1998.
7. Rus, D., Vona, M., "Self-reconfiguration Planning with Compressible Unit Modules", IEEE Proc. of Intl. Conf. On Robotics and Automation 1999.
8. http://www.parc.xerox.com/spl/projects/modrobots/polybot/polybot.html
9. http://www.isi.edu/conro/index.html
10. http://www.cs.cmu.edu/~aml/research/DRS/index.html#I-cubes
11. http://www.cs.cmu.edu/~aml/research/DRS/index.html#millibots
12. Chen, I-M. Burdick, J.W., "Determining Task Optimal Modular Robot Assembly Configurations", IEEE Proc. of Intl. Conf. On Robotics and Automation 1995.
13. Chen, I-M., Burdick, J.W., "Enumerating the Non-Isomorphic Assembly Configurations of a Modular Robotic System", International Journal of Robotics Research, 1995.
14. Paredis, C. and Khosla, P., "Syntehsis Methodology for Task Based Reconfiguration of Modular Manipulator Systems", Proceedings Int. Symp. Robot Res., Hidden Valley, PA, October 1994.
15. Hamlin, G. and Sanderson A., "TETROBOT Modular Robotics: Prototype and Experiments", Proc. of IROS 1996.
16. Fukuda, T. and Nakagawa,S.,"Dynamically Reconfigurable Robotic Systems", IEEE Proc. of Intl. Conf. On Robotics and Automation 1988.
17. M. Henle, *A Combinatorial Introduction to Topology*, Dover Publications 1994.
18. Chirikjian G.S. et al,"Evaluating Efficiency of Self-Reconfiguration in a class of modular robots", Journal of Robotic Systems, June 1996.
19. A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley and sons, 1987.
20. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization, Algorithms and Complexity*, Prentice Hall, 1982.
21. Pamecha, A., Ebert-Uphoff, I., Chirikjian G.S.,"Useful Metrics for Modular Robot Motion Planning", IEEE Transactions on Robotics and Automation, Vol.**13**, No.4, 1997.
22. C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, 1964.
23. H. Simon, *The Sciences of the Artificial*, MIT Press, 1985.
24. H. Moravec, *Mind Children: the Future of Robot and Human Intelligence*, Harvard University Press, 1990.
25. T. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
26. W. T. Tutte, *Graph Theory*, *Encyclopedia of Mathematics and its Applications*, Volume **21,** Addison-Wesley, 1984.
27. J.C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, 1991.
28. F. Harary, E. M. Palmer, *Graphical Enumeration*, Univ. of Toronto Press, 1973.