# A Tale of Two Planners: Modular Robotic Planning with LDP

Michael De Rosa
Seth Copen Goldstein
Peter Lee
School of Computer Science
Carnegie Mellon University
[mderosa,seth,petel]@cs.cmu.edu

Padmanabhan Pillai
Jason Campbell
Intel Research Pittsburgh
[padmanabhan.s.pillai,jason.d.campbell]@intel.com

***Abstract*— LDP (Locally Distributed Predicates) is a distributed, high-level language for programming modular reconfigurable robot systems (MRRs). In this paper we present the implementation of two motion-planning algorithms in LDP, and analyze both their performance and ease of implementation. We present multiple variations of one planner, including a novel resource allocation algorithm. We then draw conclusions about both the utility of the motion-planning algorithms and the suitability of LDP to the problem space. Our experiments suggest that metamodule-based planning approaches have a cost in time and/or energy terms, but that the cost can be worth paying in exchange for the additional generality and separation-of-concerns offered by these techniques. The particular tradeoff for a given system will depend upon its goals and the details of the underlying modules.**

## I. MRRs and Motion Planning

The problem of reconfiguration / shape planning for modular robotic systems (MRRs) presents several challenges over and above those found in non-modular robots. The large number of discrete modules results in a correspondingly large number of degrees of freedom, and creates a very large state space that a planner may have to explore. Furthermore, the motions of individual modules are often restricted in non-trivial ways, e.g., moving a module to an adjacent empty space may require many other modules to move out of the way due to blocking constraints. Essentially, two configurations that are similar in state space, may actually be separated by a long reconfiguration path.

Two recent approaches attempt to overcome blocking constraints and allow scalable, disconnection-free, stochastic planning in large MRRs. A scaffold-based technique [1] restricts modules to a specific grid structure that allows other modules to pass through unhindered. More recent work [2] groups modules together into metamodules and provides a high-level set of primitives that are not subject to blocking constraints. Both allow greedy or stochastic shape planning to succeed. However, the metamodule system is more general, and can be implemented on a variety of different MRR designs. The question we seek to answer is what price does one pay, if any, for this added generality? In this paper, we compare these approaches to scalable shape change, using LDP [3], a system for concisely representing distributed programs for MRRs. As part of this study we illustrate the capability of LDP to enable rapid and concise expression of MRR algorithms.

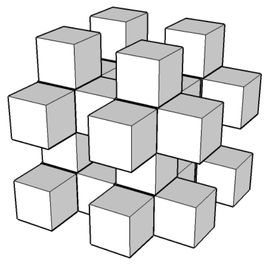
Fig. 1. Scaffold-based shape change: 4x4x4 subunit of scaffolding.

### A. The Shape Change Problem

The shape change problem considered in this paper is the reconfiguration of a large lattice-style MRR from a starting shape to a target shape. We assume that the system is provided with a target shape (e.g., list of desired module positions, variably-sized blocks [1], or isosurface equations). Furthermore, we assume that the modules are aware of their locations in the initial shape, either through *a-priori* knowledge (e.g. from the structure of the lattice), or from a localization algorithm [4], [5].

All communication within large MRRs is performed using neighbor-to-neighbor links between adjacent modules. Local communications are relatively inexpensive, while broadcast and multi-hop communication scale in cost with the size of the ensemble. The complexity of planning and expense of global communication make online centralized planning impractical. However, any distributed solution must contend with network topology changes as the ensemble shape changes, creating the potential for unwanted disconnections and asynchronous operation of the individual modules.

Finally, the modules are restricted to a particular lattice arrangement, and limited motion primitives. In this paper, we use the *corner-turning cubes* motion model [6], [7], which uses a cubic lattice and provides two basic movements: an axis-aligned slide, and a 2D corner turn (see Figure 3). This motion model is assumed by the scaffold planner, and is one of many models supported by the metamodule planner.

### B. Scaffold-Based Shape Change Algorithm

The first planning algorithm that we evaluate is the scaffold-based self-repair and self-reconfiguration algorithm

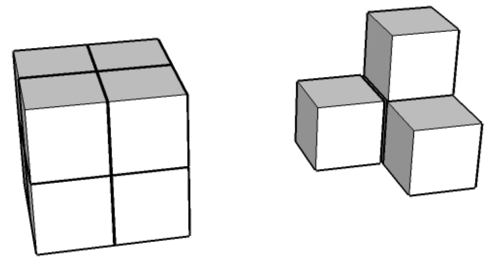
Fig. 2. Metamodule shape planner: Full and empty 2x2x2 metamodules.

developed by Støy and Nagpal [1]. The algorithm begins with the modules in an arbitrary configuration, and maintains an internal scaffold (Figure 1) to allow for unrestricted module movement within the shape.

Shape change is accomplished through the use of vector gradients, emitted by modules adjacent to locations that require additional modules to complete the target shape. The gradients propagate from module to module (decaying at each step), and modules that are not in the target shape move in the direction of the strongest gradient. Once the open locations have been filled, the gradient is deleted. Disconnection of the ensemble is prevented through the use of simple local rules based on the gradient values and states of a module's immediate neighbors, as discussed in [8].

## C. Metamodule Shape Planner

The second planner we evaluate is the generalized metamodule shape planner described by Dewey et. al. [2]. This system uses particular arrangements of modules, called *metamodules* as the basis for constructing shapes and planning. This abstraction allows metamodules to be created and destroyed, like pixels being turned on and off. In contrast to movement of modules, this approach eliminates nonholonomic constraints, greatly simplifying planning.

Of course, the modules that are used to build metamodules cannot just appear and disappear. Metamodules are designed to contain empty space that can hold additional modules, or *resources*. A metamodule is either *empty* or *full* depending on whether it contains resources. When a metamodule is destroyed, its modules are treated as resources, and transferred to $k$ empty metamodules. Likewise, the resources from $k$ full metamodules are used to construct a new empty metamodule in an unoccupied location. The value of $k$ depends on the specific metamodule implementation.

The metamodule system permits the separation of concerns between three software components: a low-level metamodule controller, a high-level shape planner, and a resource manager. The low-level metamodule controller actually implements the metamodule create, destroy, and resource transfer operations as a sequence of module movements. This is the only component that is tied to the specifics of the particular MRR hardware, the lattice used, and the metamodule structure. Only this component needs to be modified to use this planning framework with different hardware or lattice types, making the metamodule approach a general and broadly applicable shape planning framework.

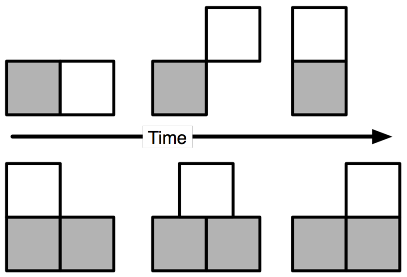

Fig. 3. Motion primitives for the corner-turning cubes model. Top: Corner-turning move. Bottom: linear slide.

The high-level shape change planner is conceptually simple. It uses resources held by existing metamodules to construct new ones in adjacent, unoccupied spaces that are in the target shape. Similarly, it attempts to destroy metamodules that are not in the target shape. The planner maintains connectivity during deletion by constructing logical trees that extend from the target shape into the deletion region. By destroying metamodules only at the leaves of these trees, the planner provably ensures connectivity.

The final component is the resource manager, whose purpose is to transfer resources, moving them from the deletion region where they are produced, to the creation region where they are consumed.

In this paper, we use a metamodule composed of 4 cubic modules, occupying half of the locations in a 2x2x2 cube (Fig. 2). This metamodule has space for 4 additional modules as resources, so a single metamodule can hold all of the resources to create a new metamodule (i.e., $k = 1$). This metamodule design mimics the scale and geometry of the scaffold-based planner; a shape constructed from empty metamodules creates a structure identical to that of the scaffold used in Støy's algorithm. The difference between the two algorithms is thus primarily the choice of atomic motion units for the planner: for Støy's algorithm it is individual modules, which the metamodule planner operates on structured groups.

For our resource management algorithm, we used a naïve random swap manager. At each timestep, a metamodule with a resource will attempt to transfer the resource to an empty, randomly selected neighboring metamodule. As a trivial optimization, we forbid resources from traveling toward the leaves of the deletion tree.

## II. Locally Distributed Predicates

The shape planners studied in this paper have been implemented using Locally Distributed Predicates (LDP). LDP, first described in [9], is a technique for concisely and efficiently representing distributed programs that can be describe in terms of fixed-sized, connected subgroups of a larger ensemble. By abstracting such concerns as variable storage, consistency, and messaging, LDP allows programmers to concentrate on the details of their particular distributed algorithm, rather than on support code. A brief overview of LDP's syntax and operation is presented beginning in Section II-A, with more extensive treatment of the language and runtime having been presented in [3].

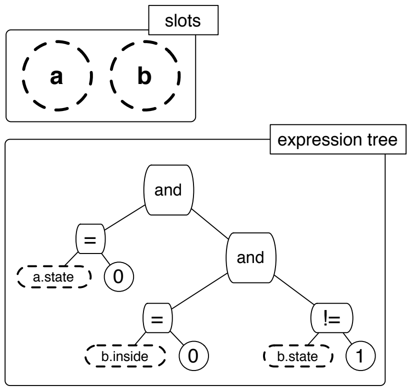

Fig. 4. Schematic representation of a PatternMatcher object, the fundamental LDP data structure used for both data sharing and condition detection. To continually find all possible instances of a condition, PatternMatchers are generated at every timestep on every module, and are passed between modules. At each module that the PatternMatcher visits, the next available slot is filled with the module's id, and corresponding data is used to populate the expression tree. Partially filled PatternMatchers are propagated to all neighbors, until the expression tree is decidable.

### A. LDP Syntax

An LDP program consists of two components: variable declarations and action predicates. Each module is assumed to contain some set of named member variables and functions, and these must be declared before use (Figure 5). Variables may be managed by the runtime system, or they may be *pass-through*, and rely on a user-supplied implementation for read and write. Pass-through variables allow for the easy abstraction of such values as sensor readings and joint angles (in the case of MRRs) into simple scalar variables. Variables may be either scalar or set-valued.

Action predicates can be thought of as distributed if-then clauses. Predicates define a distributed condition that is monitored at runtime, and an action to be executed on each successful match of the predicate. Each predicate begins by declaring the named, ordered set of modules participating in the predicate. As an example, `forall(a,b,c)` declares a predicate over all size-3 connected subgroups, whose members are (in order) `a,b,c`.

After the declaration follows a boolean predicate over the state of the participating modules. LDP provides access to the current snapshot value of each state variable through C-style variable access (e.x. `module.variableName`). Predicates can also access previous values of a variable (via `prev()`) and the immediate value of a variable (via `current`). The utility of these temporal access modes is described more fully in Section II-C. LDP supports a full set of mathematical, boolean, and set operators for use in predicates.

The final component of an action predicate is one or more action clauses. LDP supports three types of action clauses: function calls, variable assignments, and topology manipulations. All action clauses for any single predicate must take place on the same module, to avoid having to implicitly synchronize distributed actions.

### B. LDP Operation

The core of the LDP execution model is the *Pattern-Matcher* (Figure 4). A PatternMatcher is a mobile data structure that encapsulates an instance of a distributed search attempt for a particular statement. This object migrates around the sub-ensemble until either it fails to match or it matches. A new PatternMatcher is created for each predicate on each robot at every time tick. The PatternMatcher is an object that encapsulates a search attempt for a particular predicate, including an expression tree encoding the predicate, and storage for state variable values that allow for comparison of state between multiple modules.

When a PatternMatcher is created, the current module id is bound to the first slot and the values of its state variables populate the expression tree. The expression tree is then examined for success or failure of the boolean predicate. If the expression tree is successful, then the action clauses of the statement are executed. If the tree is unsuccessful, the PatternMatcher is discarded. If no determination can be made, the PatternMatcher is forwarded to all of the module's neighbors, where the above process is repeated.

PatternMatchers provide numerous opportunities for optimization, allowing for boolean short-circuiting, as well as more intelligent search strategies than spreading to all neighbors. Additionally, PatternMatchers allow for backtracking in search paths, allowing for the detection of nonlinear configurations of matching modules. These extensions, as well as a full description of the distributed predicate detection algorithm, are presented in detail in [10].

### C. Snapshot Consistency Model

LDP utilizes a virtual timestep model of execution: program execution at each module is divided into timesteps, where the module can take a snapshot of the local state, process received messages, perform computation and actuation, and then send outgoing messages. Any LDP statements that execute during a timestep will read state variables only from the local snapshot, thus preventing action clauses from creating inconsistent state. Access to previous local snapshots is available via temporal operators, which allows for reasoning over the state history of a group of modules.

LDP programs capture the state of multiple robots using *distributed snapshot* semantics. The data used in a particular LDP search attempt is equivalent to that obtained by a distributed snapshot of the participating modules. As we place no restrictions upon the length of a module's timestep, or the variance of timesteps between modules, there arises a natural question: how can a predicate that uses state variables from multiple modules ensure that it has obtained consistent state?

In a fully asynchronous distributed system, there is in general no means to obtain state from multiple modules *at the same instant in time*. There is, however, a weaker guarantee that can be satisfied: that distributed state is *consistent*. Consistency is the property that the gathered state reflects some valid linearization of the distributed events occurring at the modules. In particular, if module A sends a

```
scalar parent = INVALID_ID;
scalar id;
scalar is_root;
scalar depth = MAX_INT;

forall (a,b) where (a.is_root == 1) do b.parent = a.id & b.depth = 1;
forall (a,b) where (b.depth > a.depth + 1)
       do b.parent = a.id & b.depth = a.depth + 1;
```

Fig. 5. A Simple LDP Example: Spanning Tree Formation

message M to module B, all events that occur on B after M is received must occur after any event on A that occurs before the transmission of M. This *happens-before* relationship was formally defined by Chandy and Lamport [11]. As we show in [10], the PatternMatcher mechanism of the LDP runtime preserves consistency.

### *D. A Simple LDP Example*

Figure 5 demonstrates the various features of LDP using a simple example that constructs a depth-minimizing spanning tree with a fixed root. The spanning tree declares 4 scalar state variables: `id,is_root,parent,depth`. `id` and `is_root` are pass-through variables, and are assumed to have been set by some other software component. The first predicate begins propagation of the spanning tree by detecting all connected pairs that include the root. The non-root member of the pair then has its parent and depth set appropriately. Construction of the tree proceeds iteratively in the second predicate, with each module attempting to minimize its depth in the tree.

## III. Implementation Lessons

An important result of the exercise of implementing the two planners in LDP was the discovery of several common design patterns and language requirements for implementing distributed algorithms. These design patterns, combined with other implementation lessons from the two planners, are a valuable resource for developing distributed algorithms using LDP in the future.

### *A. LDP Enhancements*

A key challenge in the implementation of the metamodule planner is integrating the low-level metamodule controller and high-level planner operations, and providing an abstraction that makes multi-step actions of the lower layer appear atomic to the high-level planner. Our solution relies on a hierarchical implementation that completely isolates the variables and statements of the planner from those of the low-level controller. To mediate between the two levels of the hierarchy, we add a lock variable, which specifies the currently active layer. When the planner initiates a low-level operation, the lock is acquired by the low-level controller, blocking any further planner-level activity by the affected metamodules. Once the low-level action has completed or aborted, the lock is released, and planner-level activity resumes.

Under the strict consistency model used by LDP (as described in Section II-C), it is impossible to ensure reliable two-party locking with a simple lock variable. This stems from the use of timestep-driven snapshots to gather data to evaluate predicates — values from the previous snapshot are used to predicate any write actions, the results of which are not visible until the next timestep. This means that if two competing write actions are executed in the same timestep, one will overwrite the other, potentially leading to inconsistent state. To allow correct implementation of lock variables, we introduce the `current` temporal quantifier. The `current` quantifier instructs the runtime to use the actual value of a particular variable, as opposed to the most recent snapshot. This allows writes to the lock variable to be immediately visible to other potential writers within the same timestep. The use of `current` does potentially violate the snapshot consistency model, but in practice does not have any noticeable effect on the operation of programs.

### *B. Design Patterns in LDP*

Several common design patterns emerged repeatedly in the implementations of the two planners in LDP. These repeated structures shed light on both the challenges inherent in MRR programming, and the suitability of LDP for the task.

**Dissemination Trees**: The first frequently occurring pattern was the use of communication trees for dissemination of data. At the ensemble level, these trees were used to propagate vector gradients for the scaffold planner, and to construct deletion trees for the metamodule planner, which ensured the ensemble remained connected during reconfiguration. At a smaller scale, distribution trees were used within a metamodule to provide movement commands to the constituent modules.

**Aggregation Sets**: A second repeated design pattern we observed was was the use of set variables (specifically, sets of module id numbers) to ensure reliable aggregation and parent/child relationships. To use sets in this manner, we first create a set containing all of the module's children or neighbors (as appropriate). When a child reports information back to the parent, its id number is added to a second set. By using set intersection, it is trivial to determine when all children have reported back. We use this technique to decide which modules can move (both in the metamodule and scaffold planners) and to determine when a low-level operation has finished (in the metamodule planner).

**Hybrid Coding**: A final design decision which we observed to be of great help was the use of hybrid (multi-langauge) coding. LDP supports the execution of arbitrary C++ functions, both in action clauses and as part of variable

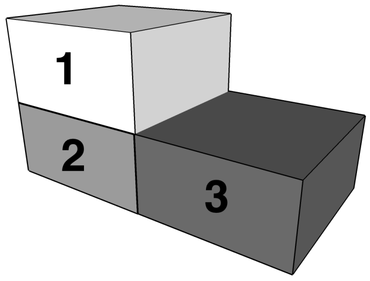

Fig. 6. Experimental configuration for shape change evaluation. Start configuration (regions 2 and 3) is 8:4:2, end configuration (regions 1 and 2) is 4:4:4.

reads. We found this functionality invaluable, as it made implementing table-based lookups for low-level metamodule operations trivial. Hybrid coding greatly reduces the set of functionality that must be present in the high-level language (LDP), allowing specialized operations to instead be implemented in C++ rather than cluttering the base syntax of LDP. In this way, LDP works as a coordination language, in the same vein as Delirium [12]. Hybrid coding allows us to reduce the amount of code necessary to implement the planning algorithms. The original implementation of the scaffold planner required approximately 700 lines of Java code, while the implementation in LDP required only 10 statements, plus 50 lines of C++. Similarly, the implementation of the more complex metamodule planner was also concise: only 40 lines of LDP and 650 lines of C++, with the latter primarily used to encode metamodule movement lookup tables in the metamodule controller.

## IV. Experimental Method

We evaluated both the scaffold and metamodule planners on ensembles of simulated robots using DPRsim [13]. The modules start out arranged as a rectangular solid, with an aspect ratio of 8:4:2, and are tasked to form a rectangular solid of aspect ratio 4:4:4. Figure 6 illustrates these starting and final configurations, as well as their overlap. Note that 50% of the ensemble begins in the intersection of the start and final shapes, there are the exact number of modules needed to form the desired shape, and that the initial packing density of both the metamodule and scaffold shapes is 50%.

Trial runs of the two planners were conducted on 8x4x2 (32 module), 16x8x4 (256 module), and 24x12x6 (864 module) ensembles. Experiments were run until the target shape was completed.

## V. Results and Discussion

### A. Metamodule vs. Scaffold Planner Comparison

We compared the performance of the scaffold planner against that of the metamodule planner (with random resource movement), measuring the total number of messages, moves, and simulated timesteps before completion (see Table I). In the smallest experiments, with only 32 modules, the metamodule planner took 45% more timesteps to complete, and almost double the number of messages. As the scale of the experiments increased, we saw a widening of the gap between the scaffold planner and the metamodule algorithm. This was especially noticeable in the number of timesteps for the metamodule planner to complete. The number of moves required by the two planners also diverged as the scale of the experiment grew.

We observed three key reasons for this performance gap:

1) **Planner decision quality differences**—The random (rather than gradient-directed) resource movement used by the basic metamodule planner is less efficient, particularly when measured by completion time. We explore this quality factor in some detail below.

2) **Metamodule implementation overhead**—Both the distributed locks required during metamodule operations and the messaging latency associated with low-level resource transfer and metamodule creation/deletion routines add delay. Our metamodule operations are implemented optimally, but our simulation parameters likely dramatically overstate the cost of these overheads (see next).

3) **Measurement bias**—In order to fairly schedule communication, DPRsim routes messages on a tick-to-tick basis and thereby exacts the same latency penalty for communication as for motion. Although communication in a distributed system is never free, parity with motion is almost certainly too high a cost estimate. We are presently working on an extension to DPRsim which would allow us to parameterize the relationship between messaging cost and movement cost.

### B. Beyond Random Resource Allocation

In an effort to improve the performance of the metamodule planner, we implemented two variant resource managers, which use gradient techniques similar to those found in the scaffold planner. The first resource manager is a straightforward port of the scaffold planner's gradient algorithm to the metamodule environment, a planner which we designate MM+Gradient.

Our second variant on metamodule resource management is based on the observation that the scaffold planner's gradient has a uniform start value, which ignores the number of resources needed at a particular growth site. By making the starting value of the gradient proportional to the distance form the creation site to the boundary of the target shape, the gradient can potentially provide guidance as to both the need for resources, as well as the number of resources needed. We designate this second planner MM+Gradient2.

Both of these planners improve on the results seen with the random resource manager, and while neither is close to the performance of the scaffold planner in time, both are well within a factor of 2 in terms of the amount of energy required (moves commanded).

### C. Analyzing Progress over Time

Each planner exhibited an abrupt slowdown in forward progress. This can be seen most clearly by plotting moves or completion percentage versus time. See Figures 7 and 8 for the corresponding plots for the largest of the simulation experiments. The knee (slowdown) in each curves occurs at

TABLE I

PERFORMANCE OF PLANNING ALGORITHMS (AND VARIANTS).
$Q_{path}$ IS PATH LENGTH QUALITY, $D_A$ IS TOTAL ASSIGNMENT DISTANCE.

| Experiment | # modules | # timesteps | # moves | # messages | $Q_{path}$ | $D_a$ |
|---|---|---|---|---|---|---|
| Scaffold 8x4x2 | 32 | 124 | 109 | 32 545 | 0.607 | 152.3 |
| MM+Random 8x4x2 | 32 | 180 | 110 | 57 733 | 0.577 | 146.3 |
| MM+Gradient 8x4x2 | 32 | 180 | 110 | 65 742 | 0.577 | 146.3 |
| MM+Gradient2 8x4x2 | 32 | 180 | 110 | 65 742 | 0.577 | 146.3 |
| Scaffold 16x8x4 | 256 | 287 | 1 802 | 746 601 | 0.555 | 2 434.9 |
| MM+Random 16x8x4 | 256 | 2 581 | 4 106 | 7 935 802 | 0.362 | 2 481.6 |
| MM+Gradient 16x8x4 | 256 | 2 094 | 2 936 | 7 922 922 | 0.426 | 2 494.5 |
| MM+Gradient2 16x8x4 | 256 | 1 924 | 3 522 | 7 591 609 | 0.362 | 2 458.2 |
| Scaffold 24x12x6 | 864 | 1 973 | 13 508 | 18 746 742 | 0.533 | 11 958.4 |
| MM+Random 24x12x6 | 864 | >6 000 | >34 000 | >72 000 000 | <0.272 | 12 119.4 |
| MM+Gradient 24x12x6 | 864 | 5 230 | 19 332 | 74 571 924 | 0.426 | 12 226.5 |
| MM+Gradient2 24x12x6 | 864 | 4 168 | 21 754 | 63 386 489 | 0.311 | 12 351.8 |

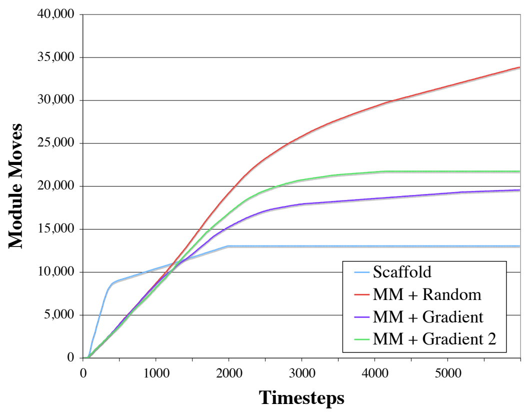

Fig. 7. Planner performance: Module moves over time.

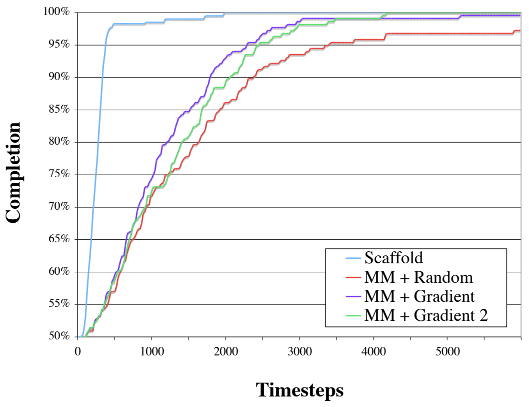

Fig. 8. Planner performance: Completion percentage over time.

the point where physical contention and resource exhaustion limit the rate at which target destinations can be filled.

Below the knee, each algorithm proceeds at a rate set by the degree of parallelism it can command. Maintaining the metamodule structures limits the number of simultaneous motions possible and results in a shallower initial curve. Above the knee, Metamodules+random resource management exhibits a particularly long and slow climb to completion as the number of free resources in circulation dwindles and the time it takes for each resource to find an available target location on its random walk lengthens. The gradient-based approaches, both metamodule and scaffold, waste no time/motion on random walks and so complete faster. Again, the degree of potential parallelism is higher for the scaffold approach than for the metamodule+gradient approaches, meaning the scaffold case has a steeper slope after the knee on its curve and completes more rapidly.

When we consider completion percentage as measured by the proportion of target locations filled the picture changes somewhat. The metamodule based planners again complete more slowly than the scaffold planner, but performance among the three metamodule planners is very similar. That is, the three fill target locations with similar rapidity, but random resource allocation in particular expends a large amount of effort moving the last few free resources around (randomly) in search of the few remaining empty target locations.

## *D. Path Quality and Planner Performance*

To more explicitly quantify the performance impact of planner decision quality, we examined two more statistics: *total assignment distance* ($D_A$) and *path length quality* ($Q_{path}$). $D_A$ is the sum of Cartesian distances between starting and ending module locations for each final module location. This estimates the minimum amount of work an optimum path planner algorithm might require to effect the same mapping between module initial and final positions as selected by each planner. $Q_{path}$ is defined for each module as the total distance moved, divided by the distance between start and end positions. $Q_{path}$ measures the relative efficiency of each algorithm in selecting a path for each module from start to destination. Neither of these is an exact measure as the direct Cartesian path may not be feasible due to missing support or interfering modules, and in any case the set of all per-module optimal paths may not result in a globally optimal plan. Nonetheless, both provide useful approximations for comparing the different planners.

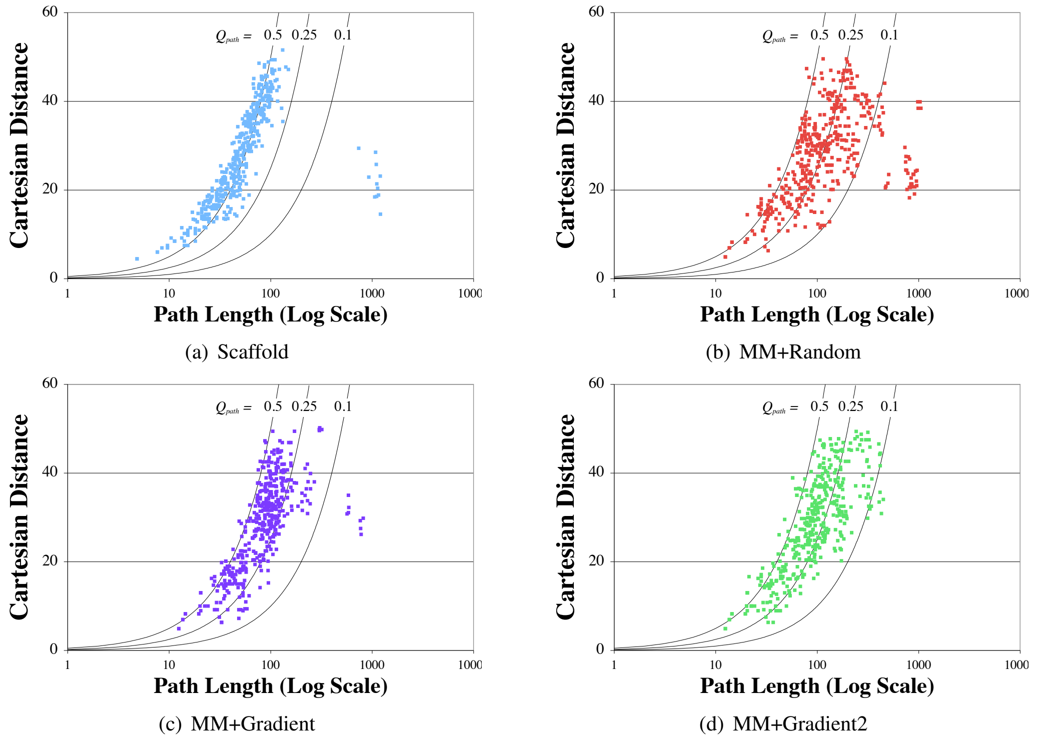

Fig. 9. Path length traversed vs. Cartesian distance between start and end positions for individual modules for the different planners

When we consider the total assignment distance for each algorithm we see a high degree of similarity across all the trials at each scale (Table I, rightmost column). This means there is very little difference in algorithm performance in terms of assigning modules from the initial shape to module locations in the target shape. (Note that without further information we cannot say whether all the algorithms do well or all do poorly, we can only say their performance is comparable to one another.)

When we compare the $Q_{path}$ figures we see a different story. At the smallest scale there is little difference because motion in such a small ensemble is highly constrained, but as scale increases the scaffold algorithm clearly outperforms the random-walk behavior of Metamodules+Random. This shouldn't come as a surprise, but since one of the great strengths of the metamodule planner is its modularity we can readily insert other resource management policies to study their impact. We see that with a gradient-based resource manager $Q_{path}$ is substantially improved and falls somewhere between that of the scaffold planner and a random walk. Interestingly, MM+Gradient2 completes in fewer timesteps than MM+Gradient, but at the cost of additional moves, which is indicative of a more aggressive algorithm.

By plotting the components of $Q_{path}$ against one another for each module in each scenario (Figure 9), we can understand the performance differences in greater detail. The scaffold planner achieves very tight clustering around its mean $Q_{path}$ value, indicating a very controlled, homogeneous strategy. However a few outliers far to the right consume substantial energy. These outliers appear consistently in particular experimental scenarios at several scales and are similar to those observed in the original implementation of the algorithm [14].

When we look at the scatter plots for the metamodule-based planners we see a wider spread of $Q_{path}$ values, with lower mean. One subtle difference to note is the somewhat wider distribution for MM+Gradient2 versus MM+Gradient. We hypothesize the the additional gain factor introduced with Gradient2's scaling of gradient messages by the number of modules needed at the target location induces additional oscillation of free resources/modules between multiple target destinations. The additional gain drives somewhat faster completion time but at a cost of somewhat greater overshoot and hence energy consumption. We can also clearly see in the plots for MM+Random and MM+Gradient a few modules which struggle to reach their final locations and hence appear isolated as outliers to the right of the distribution. These are the reason for the long tails seen for these algorithms in Figure 8.

## *E. Implications for Module and Metamodule Design*

The differences between scaffold and metamodules can be seen as the cost of generality vs. specialization. The scaffold planner can be seen as a version of the metamodule algorithm, with a basic gradient resource manager, and size-1 metamodules that obey a particular set of movement constraints. The cost of moving from single modules to 2x2x2 metamodules is clearly seen in the path quality measures, and makes a compelling argument for developing MRR systems that do not require large metamodules in order to avoid motion constraints. Additionally, we see that a random-walk resource manager is totally unsuitable for systems without an overabundance of resources, as the resulting starvation of creation sites dramatically slows completion of the shape-change.

Still, we note that more intelligent resource managers, either with more expressive gradients or other mechanisms, can potentially increase the performance of metamodule-based planners to a level that approaches that of the scaffold algorithm. Combined with the potential for a metamodule-based approach to permit more design freedom at the module level, hence enabling faster, lower-power, more reliable, or smaller modules, it is certainly possible (though not inevitable) that metamodules could be a net benefit at the MRR system level.

## VI. Conclusions

Metamodule-based approaches greatly simplify motion planner construction for modular reconfigurable robots due to metamodules' ability to remove holonomic constraints. This can improve the generality of the planning units (metamodules) when compared with a planner for the underlying modular system (modules). In this paper we have shown that this generality comes at a price in both energy and time: Effectively, metamodules drive up the cost of planner mistakes because each "primitive" operation at the planner level actually requires execution of a sequence of operations at the underlying module level. This disparity is likely to increase as the metamodule size increases.

In making this comparison we chose to evaluate systems where both the metamodules and the underlying modules offer the same motion capabilities. This makes our comparison "fair" in one sense, but also removes a key advantage of metamodules from the picture, namely, their potential to make a highly-non-holonomic module design tractable from a planning perspective. Thus, despite our finding that the particular metamodule planner we tested required more time and energy than a non-metamodule planner in the test scenarios used, a metamodule-based approach may nonetheless be the best choice for a given MRR system – for instance, in cases where a non-metamodule planner is intractable, is too complex, proves too difficult to write, or where use of metamodules can allow the underlying modules to be simplified such that they move more quickly, more reliably, or using less energy.

Ultimately, the performance of any motion planner is limited by the quality of its decisions. A still-open question is how metamodule and non-metamodule planners might compare given optimal decision criteria for each. Because optimal planners are, in general, impractical (and often intractable) for the numbers of degrees of freedom presented by even small MRR systems, we chose to compare real, published planners in this study.

Our results also demonstrate that all planner decisions are *not* of equal importance. In particular, we have observed that the choice of resource manager (random walk, gradient, or otherwise) plays a large role in a metamodule-based planner's performance, whereas the assignment from starting to final location for each module seems to play a much smaller role (at least at the scales we have tested).

Finally, implementing both planners in LDP has shown that the use of a high-level language can result in dramatically shorter code, with a corresponding increase in clarity. As a consequence we have readily been able to explore the impact on planner performance of a variety of resource manager strategies. Our work has highlighted the utility of such a hybrid coding approach, while illustrating some shortcomings of the LDP language.

## VII. Acknowledgements

This work was partially funded by the National Science Foundation (NSF) under grant no. CNS-0428738, Intel Labs Pittsburgh, Carnegie Mellon University, and a generous contribution from the Microsoft Corporation. Special thanks to Kasper Støy for the use of his existing gradient planner implementation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

## References

[1] K. Støy and R. Nagpal, "Self-repair through scale independent self-reconfiguration," in *Proceedings of IEEE/RSJ International Conference on Robots and Systems, (IROS)*, 2004, pp. 2062–2067.

[2] D. Dewey and S. Srinivasa, "A metamodule shape planner for modular robots," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '08 (in submission)*, 2008.

[3] M. De Rosa, S. C. Goldstein, P. Lee, P. S. Pillai, and J. Campbell, "Programming modular robots with locally distributed predicates," in *Proceedings of the IEEE ICRA*, 2008.

[4] S. Funiak, P. Pillai, M. P. Ashley-Rollman, J. D. Campbell, and S. C. Goldstein, "Distributed localization of modular robot ensembles," in *Proceedings of Robotics: Science and Systems*, June 2008.

[5] G. Reshko, "Localization techniques for synthetic reality," Master's thesis, Carnegie Mellon University, August 2004. [Online]. Available: http://www-2.cs.cmu.edu/ reshko/thesis/

[6] Z. Butler, K. Kotay, D. Rus, and K. Tomita, "Generic decentralized control for a class of self-reconfigurable robots," in *Intl Conf on Robotics and Automation (ICRA)*. IEEE, 2002, pp. 809–816.

[7] R. Fitch, Z. Butler, and D. Rus, "Reconfiguration planning for heterogeneous self-reconfiguring robots," in *Intelligent Robots and Systems (IROS)*. IEEE, 2003, pp. 2460–2467.

[8] K. Støy, "Controlling self-reconfiguration using cellular automata and gradients," in *Proceedings of the 8th international conference on intelligent autonomous systems (IAS-8)*, March 2004, pp. 693–702. [Online]. Available: http://www.mip.sdu.dk/ kaspers/publications.php

[9] M. De Rosa, S. Goldstein, P.Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging very large ensembles of robots (extended abstract)," in *RSS'06 Workshop on Self-reconfigurable Modular Robotics*, August 2006.

[10] M. De Rosa, S. C. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging large modular robotic systems (in preparation)," *International Journal of Robotics Research*, vol. 27, no. 3, Special Issue on Modular Robotics 2008.

[11] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[12] S. Lucco and O. Sharp, "Delirium: an embedding coordination language," in *Supercomputing '90: Proceedings of the 1990 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 515–524.

[13] Dprsim. [Online]. Available: http://www.pittsburgh.intel-research.net/dprweb/

[14] K. Støy, "Emergent control of self-reconfigurable robots," Ph.D. dissertation, AdapTronics Group, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, 2003.