Declarative Programming for Modular Robots

Michael P. Ashley-Rollman, Michael De Rosa, Siddhartha S. Srinivasa,
Padmanabhan Pillai, Seth Copen Goldstein, Jason Campbell

***Abstract*— Because of the timing, complexity, and asynchronicity challenges common in modular robot software we have recently begun to explore new programming models for modular robot ensembles. In this paper we apply two of those models to a metamodule-based shape planning algorithm and comment on the differences between the two approaches. Our results suggest that declarative programming can provide several advantages over more traditional imperative approaches, and that the differences between declarative programming styles can themselves contribute leverage to different parts of the problem domain.**

I. Introduction

Modular robot programming can be substantially more challenging than normal robot programming due to:

- scale / number of modules
- concurrency and asynchronicity, both in physical interactions and potentially at the software level
- the local scope of information naturally available at each module

Recent declarative approaches such as P2[6] and SAPHIRA[5] have shown promise in other domains that share some of these characteristics. Inspired by those results we have been developing two modular-robot-specific declarative programming languages, *Meld* and *LDP*. Both languages provide the illusion of executing a single program across an ensemble, while the runtime system of each language automatically distributes the computation and applies a variety of optimizations to reduce the net computational and messaging overhead.

We have previously described a hole-motion based shape planning algorithm that exhibits constant planning complexity per module and requires information linear in the complexity of the target shape (regardless of the number of modules involved in the ensemble) [3]. Subsequently we generalized this approach to extend its functioning to other local metamorphic systems [4]. The latter, generalized algorithm operates on subensembles (metamodules) to accomplish both shape control and resource allocation while maintaining global connectivity of the ensemble.

In this paper we describe our progress implementing this algorithm using both Meld and LDP. So far we have found these declarative implementations to be substantially more concise ($\tilde{2}$0x) and simultaneously more amenable to optimization than was the case with an earlier imperative implementation. Meld has proven more effective at many of the global coordination aspects of this algorithm, at efficiently tracking persistent operating conditions, and at coping with (non-local) nonlinear topologies. LDP has proven more effective at local coordination, sophisticated temporal conditions, detecting local topological configurations, and more generally, at expressing variable-length conditional expressions.

We have chosen to present the planning algorithm (in Section II) before the language descriptions (Meld in Section III and LDP in Section IV), but some readers may prefer to read the sections about the languages first. We compare the languages using their respective implementation of the planner in Section V.

II. Shape Planning

As a motivating example, we explore the problem of distributed shape planning for the ensemble. We use an extension of the shape change algorithm described in [4]. The algorithm produces a distributed asynchronous plan for a group of modules to transform from a feasible start state to a feasible goal state, while maintaining global connectivity throughout the execution of the plan. Furthermore, the algorithm provides provable guarantees of completeness: if there exists a globally connected path, it will be found. A film strip of the planner in action is shown in Fig.1.

A. The planning algorithm

We define the system as a collection of states on a compact workspace $\mathcal{W}$ embedded in a lattice $\mathcal{L} \in \mathbb{R}^k$, where each state U is a labeling function of the aggregate using an alphabet of labels $\mathcal{A}$, *i.e.*, $U : \mathcal{W} \to \mathcal{A}$. For example, the alphabet could comprise of homogeneous modules R and empty space E.

States are modified by a rearrangement of their labels. The algorithm uses the following rearrangement rule:

$$\mathsf{RE} \Leftrightarrow \mathsf{RR} \tag{1}$$

The rule states that any module has the ability to create or destroy its neighbor. Dewey and Srinivasa[4] describe a metamodule-based algorithm for combining the rules of any local metamorphic system to produce a new metamodule system that obeys the above rearrangement rule.

The planner produces a sequence of rearrangements to reach the target shape while maintaining global connectivity. At the beginning, all modules are in the start shape. During the plan, modules that are not in the goal shape must be removed and empty spaces in the goal shape must be filled with modules. Adding modules cannot break global connectivity, removing modules can. The planner removes modules only after ensuring that their removal does not affect global connectivity. It achieves this by growing trees out of

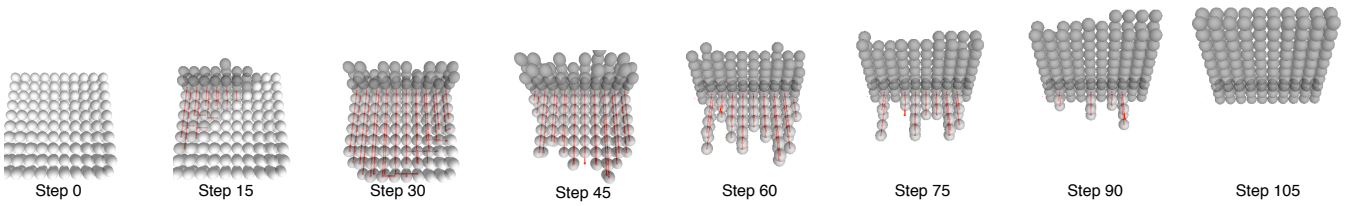

Fig. 1. Metamodule-based Shape Planner

connected sections of the ensemble and deleting modules only at the leaves.

The plan starts with a seed module labeled F, which is in the intersection of the start and goal shape. In our notation, labels in the goal shape are marked with ($\hat{}$) and those not in the goal shape are marked with ($\check{}$). The module F recruits every neighboring slot in the goal shape $\hat{\mathsf{X}}$ to become F. It marks every neutral neighbor in the start state $\check{\mathsf{N}}$ as a candidate for removal called P. Every P has a link $\rightarrow$ from its parent and as long as the link is not broken, the P will remain connected to the goal shape. Eventually, the P trees will have no further space to expand, at which point, the leaves can be trimmed without loss of connectivity. In Fig.1, the start shape is indicated by the lighter colored modules, the goal shape by the darker colored ones, and trees are indicated by red arrows.

The above plan may be symbolized as follows:

$$\begin{aligned}
\mathsf{F}\hat{\mathsf{X}} &\Rightarrow \mathsf{FF}, \quad \mathsf{X} \in \{\mathsf{E}, \mathsf{N}, \mathsf{P}\} \\
\mathsf{F}\check{\mathsf{N}} &\Rightarrow \mathsf{F} \rightarrow \mathsf{P} \\
\mathsf{PN} &\Rightarrow \mathsf{P} \rightarrow \mathsf{P} \\
\mathsf{P} &\Rightarrow \mathsf{E}, \quad \text{if } \nexists \mathsf{X} \in \text{nbr}(\mathsf{P}) : \mathsf{N}(\mathsf{X}) \text{ or childof}(\mathsf{P}, \mathsf{X})
\end{aligned}$$

where 'nbr' returns the neighbors of a module and 'childof' returns true if P is the parent of X.

B. Resource allocation

While provably complete, the above planning algorithm is oblivious to global constraints on the labels, which arise, in this case, since the total number of physical modules needs to be conserved. A resource allocator can be overlaid on top of the algorithm to enforce label constraints, allowing and disallowing creations and deletions of neighbors based on availability, as well as distributing resources to where they are needed. We modify Eqn.1 into two label conserving rules, one for resource transportation and one for module creation and deletion:

$$\begin{aligned}
\mathsf{CD} &\Leftrightarrow \mathsf{DC} \\
\mathsf{CE} &\Leftrightarrow \mathsf{DD}
\end{aligned}$$

satisfying

$$\text{num}(\mathsf{C} + \mathsf{E}) = \text{num}(\mathsf{D} + \mathsf{D}) \tag{2}$$

where 'num' is a measure of resource.

A creator C has the ability to exchange resources with a destroyer D. The creator can also produce a destroyer in an empty neighboring space, turning itself into a destroyer in the process. Likewise, two neighboring destroyers can coalesce into a creator, leaving behind an empty space.

Since there is a global constraint on the total number of C and D labels, a good resource allocator must distribute them well, sending the D to regions of anticipated deletion and the C to regions of anticipated creation. A simple, highly suboptimal allocator is a randomizer which transports resources by randomly switching adjacent C and D labels. Note that no matter which resource allocator is used, the algorithm is provably complete, but the better the allocator, the faster the time to completion.

C. Optimizations

We describe three optimizations to the above algorithm. While the optimizations do require some computational and messaging overhead, they provide up to an order of magnitude speedup.

The first optimization exploits two observations: that deletions occur at the leaves of trees, and that it takes two neighboring D to execute a deletion. As a result, we funnel the D down trees to their leaves:

$$\mathsf{DC} \Rightarrow \mathsf{CD}, \text{only if } \mathsf{D} \rightarrow \mathsf{C} \tag{3}$$

The second optimization attempts to shorten tree length as this will result in less total movement for the D to reach leaves and cause deletion. Each P in the tree stores the distance d to its root. Whenever a P encounters a neighboring P with a larger d, it reparents the neighbor to be part of its own tree and updates the d accordingly. The root F also performs a similar operation. The optimization can be summarized as:

$$\begin{aligned}
\mathsf{F}\check{\mathsf{N}} &\Rightarrow \mathsf{F} \rightarrow \mathsf{P}^1 \\
\mathsf{P}^d\mathsf{N} &\Rightarrow \mathsf{P}^d \rightarrow \mathsf{P}^{d+1} \\
\mathsf{P}^d\mathsf{P}^e &\Rightarrow \mathsf{P}^d \rightarrow \mathsf{P}^{d+1}, \quad \text{only if } d+1 < e \\
\mathsf{F}\check{\mathsf{P}}^{d>1} &\Rightarrow \mathsf{F} \rightarrow \mathsf{P}^1
\end{aligned}$$

where superscripts denote distance from the root.

While the first optimization moves resources purposefully once they are in the trees, the third optimization attempts to attract resources to the roots of the trees by spreading a gradient from the root:

$$\begin{aligned}
\mathsf{P} &\Rightarrow \mathsf{P}_0 \\
\mathsf{FP} &\Rightarrow \mathsf{F}_1\mathsf{P} \\
\mathsf{X}_a\mathsf{X}_b &\Rightarrow \mathsf{X}_{\min(a,b+1)}\mathsf{X}_b
\end{aligned}$$

```
// gradient: min aggregate over N
gradient(Module, N) :-
     state(Module, Path), 1 = Path,N = 0.

gradient(Module1, N) :-
     neighbor(Module1, Module2, _),
     gradient(Module2, M),
     N = M + 1,N <= 5.
```

Fig. 2. Example Meld code producing a bounded gradient for use in optimization 3

where subscripts denote the strength of the gradient and X is any module.

Destroyers descend the gradient:

$$\mathsf{D}_a\mathsf{C}_b \Rightarrow \mathsf{C}_a\mathsf{D}_b, \text{only if } a > b \tag{4}$$

Local gradients usually suffer from two problems: local minima and excessive messaging. However, our local gradients have a nice self-correcting property. Once a tree is completely deleted, its gradient dies, alleviating both the above problems. The ease of cleanup of the local gradient, described in the next section, illustrates both the advantages of declarative programming and the differences between Meld and LDP.

III. MELD

Meld [**?**] is a logic-based declarative programming language that operates on "facts" using a collection of production rules. A Meld program is a collection of rules for deriving, or "proving," new facts by combining existing ones. Using a process called *forward chaining*, Meld starts with a set of base facts (i.e., the execution environment), checks them against the rules, and sees if any new facts can be generated. These are then added to the collection of facts and the process continues iteratively until all provable facts under the given system of rules and base facts are generated. This forward chaining and generation of facts constitutes the execution of a Meld program.

The Meld logic itself makes no presumptions on the meanings of facts, leaving this to the programmer. However, in practice, it is useful to maintain certain conventions. For example, a NEIGHBOR(M1,M2) base fact generally indicates that modules M1 and M2 are adjacent and are capable of communicating with one another. Furthermore, to make the language useful in a robotics context, the generation of some facts can have side effects, permitting robots to move, perform actions, or otherwise affect the physical world. For example, the generation of a MOVE(M1,M2,A) fact can cause module M1 to rotate about module M2 by the specified angle A. Base facts reflect physical state, and facts with side effects correspond to the sensing and actuation primitives available on the system. Changes to base facts, due to actuation, for example, will trigger the generation of new facts as well as the deletion of old facts that can no longer be proved.

Rules are of the form NEWFACT(X,Y,Z) :- FACT1(...), FACT2(...), ... This defines an instance of NEWFACT derived from the existing facts FACT1, FACT2, ... All combinations of existing facts that can match the right hand side of the statement are applied to create the new facts. Expressions such as X=A+B (see example in Figure 2) in the right hand side of the statement serve to compute values for the new fact from pieces of the existing facts, as well as to relate and restrict the sets of facts that satisfy the rule. All standard mathematical operations are supported in the expressions. In addition, the language supports the concept of an aggregation over all facts of a particular type, permitting min, max, and summation operation across all instances of a fact type.

The Meld paradigm permits the writing of code from a global perspective, rather than the from the view of the individual robotic modules. This allows the programmer to focus solely on the logic of the algorithms. Under the hood, the Meld compiler distributes the data and the computation, allocating some of the facts to each module. It automatically generates the necessary messages to communicate just the facts that are useful for the generation of new facts at neighboring modules, as well as the messages to revoke them upon deletion.

Meld need not run just at the level of individual modules; it can readily run on top of abstract aggregates of modules such as metamodules, given that the base facts and facts with side-effects are matched to the characteristics of the metamodule system. For the metamodule system used in the rest of this paper, we define the following base facts:

- NEIGHBOR(M1, M2) indicates that M1 and M2 are adjacent metamodules capable of communicating with one another.
- POSITION(M, L) indicates metamodule M is at location L in the globally consistent coordinate system.
- VACANT(M, L) says that location L is not occupied by a metamodule
- RESOURCES(M, R) says that metamodule M has R resources.

This assumes that the metamodule system provides location information in some globally consistent coordinate system, in addition to metamodule-level communications and adjacency information. For this system, the set of facts with physical side-effects are:

- CREATE(M1, L) creates a new metamodule at location L using resources from M1.
- DESTROY(M1, M2) causes M2 to be destroyed and the resources to be absorbed by M1.
- GIVE(M1, M2) transfers resources from M1 to M2.

These side-effects are the basic operations available in the metamodule system described and used in the shape change algorithm in Section II.

IV. LOCALLY DISTRIBUTED PREDICATES

Tools such as global predicate evaluation [1] allow a programmer to encode queries over the state of an entire distributed system, in distributed systems with a communications topology there exists another class of distributed predicates, which we call *locally distributed predicates* (LDPs).

These are predicates over the local neighborhood of a particular node, bounded to a finite number of communication hops.

Locally distributed predicates differ from global predicates in two important respects. First, because LDPs do not encompass the entire distributed system, there may be multiple matching subgraphs for a particular predicate. Second, LDPs can describe not only the logical state of the entities in a distributed system, but also their topological configuration – a property that is inherently ignored by global predicates. This topological sensitivity can be especially useful in writing programs for modular robots.

Using such locally distributed predicates we are developing a programming environment for modular robots, based in part on our prior work on distributed watchpoints for modular robot debugging [2]. LDPs allow a programmer to describe the state configuration of a bounded subgraph of a distributed system, and to specify actions that operate on portions of that subgraph.

A. Program Structure

An LDP program consists of a number of predicates, each of which includes in turn three components:

1) **Named node list:** Each predicate begins with a named list of node variables. This list defines the size of the matching subensemble (the connected subgroup whose state satisfies the predicate's logical expression).
2) **Expression:** Following the node list, each predicate includes a single logical expression, composed from a language including standard boolean and grouping primitives, basic mathematical operators, topological restrictions, and state variable comparisons. This expression is implicitly quantified over all connected subgroups of length equal to the node list length.
3) **Action(s):** One or more action clauses, optionally parametrized using named nodes or variables.

Topological restrictions in the expression take the form of the function `neighbor(a b)`, and indicate that the two specified nodes are neighbors (that they share a communications link). When multiple neighbor expressions are combined using boolean operators, arbitrary node topologies can be expressed. State variable comparisons allow for the comparison of named state variables in one node against constants, other local variables, or remote variables on other nodes. Additionally, state variable comparisons may include arbitrarily nested uses of the `last` and `next` temporal modal operators, which provide access to the past and future states of the node's state variables.

Action clauses specify the action which will be taken when the predicate matches. Actions can include such tasks as setting state variables, moving modules, or calling custom low-level routines. Named nodes from the nodelist, and the corresponding state variables (including temporal modal operators) can be passed as arguments in functions called from action clauses. To prevent issues with locking and synchronization, all actions for a particular instance of a predicate must be executed on the same module. A full formal syntax of an early version of this language, and several examples of its use in distributed debugging, are provided in [2].

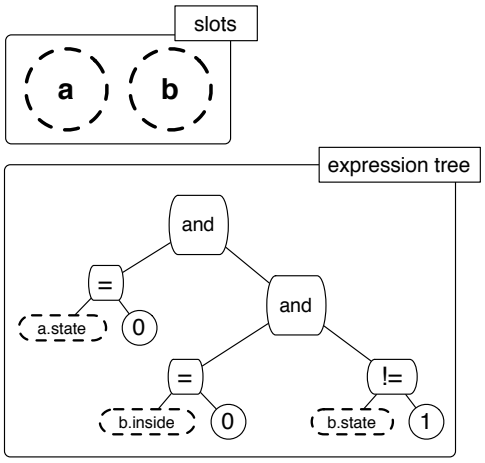

Fig. 3. PatternMatcher Object

```
(a b);(a.state == 0) and (b.inside == 0)
and (b.state != 1);b.state = 1;
```

Fig. 4. Simple LDP Conditional for Propagating Path Search State

B. Implementation

Predicate detection is accomplished through the use of PatternMatcher objects similar to those we describe in [2](see Fig.3). These mobile data structures encode a distributed search attempt and travel between modules, collecting state information until the expression they encode either matches or fails. A match will trigger any actions attached to the predicate, potentially requiring routing the successful PatternMatcher back to the node where the action is to be performed.

C. Example of a Locally Distributed Predicate

In Figure 4 we show a simple LDP conditional, which propagates the "looking for path" state. This conditional finds all size-2 connected subensembles such that `a` is in the desired shape, and `b` is outside the final shape and not already looking for a path. Module `b` is then set to be looking for a path.

V. Comparing Meld and LDP

We implemented the metamodule planner using both Meld and LDP, and noticed that, even under the general umbrella of declarative programming, the implementation of the planner was quite different for the two languages. The differing syntax, semantic, and runtime support provided by the two systems led to two very different sets of implementation challenges. Below, we highlight a few of those challenges, to give some flavor of the differences between LDP and Meld.

```
// if b has a parent and it is is not a,
// add b to the notChild set of a
(b,a); (b.parent != -1) & (b.parent != a.id);
              a.$notChild.add(b.id);

// if b is final, add b to aś notChild set
(b,a); (b.state == 0) ;
              a.$notChild.add(b.id);

// if a is in a path, and all of its
// neighbors are not its children, it can
// delete itself
(a); (a.state == 1) &
     (a.$neighbors->size ==
      intersect(a.$neighbors,a.$notChild)
       ->size);
              a.destroy(a.id);
```

```
// if we are supposed to destroy ourselves (on a path) and we
// have no children, then itś safe to do so and we should go
// ahead and do so.
destroy(Module1, Module2) :-
     state(Module1, Path), Path = 1,
     neighbor(Module1, Module2, _),
     resources(Module1, Destroy),
     resources(Module2, Destroy),
     Destroy = 0,
     forall neighbor(Module1, N, _)
               notChild(Module1, N).
```

Fig. 5. LDP (top) and Meld (bottom) code to evaluate a predicate on all neighbors.

A. Evaluating Predicates Over All Neighbors

Meld and LDP approach the problem of evaluating a predicate over all neighbors quite differently, as shown in Figure 5. This code implements the test to see if the particular module is a leaf, by checking that none of its neighbors are its child, to see if it can be safely destroyed. The LDP syntax does not have a construct to express a predicate on all neighbors, so the LDP implementation searches from the initiating module outwards, adding each neighbor that satisfies the condition to a named set variable. The size of this set is then compared with the number of neighbors the module has. If these are the same, then the predicate must hold for all neighbors. Although not needed in this example, in general LDP code will need to remove neighbors from the set when the condition no longer holds for them, or arrange to empty the set regularly. Meld, on the other hand, takes the list of neighbors and checks that the fact holds for each one. Both methods accomplish the same result of checking that the condition holds on every neighbor. Both methods are also vulnerable to receiving stale data from neighbors and thus inaccurately calculating the state of the predicate.

B. Selecting Unique Optimal Parents

Another difference between Meld and LDP is the presence of classical state. In LDP there are globally accessible state variables that can be read or written arbitrarily by the rules involved. This permits the programmer more flexibility and makes it easy to maintain information such as the parent of a node in a generated tree. Meld gives up this flexibility in exchange for an event-based architecture and automated fact deletion, the benefits of which are discussed in the next two sections.

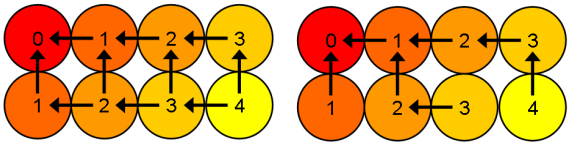

Fig. 6. Example DAG and one of several trees that can be constructed from it

```
// parentDist: min aggregate over distance
type parentDist(module, min int).
parentDist(Module, Dist) :-
     possibleParent(Module, _, Dist, _).

// parentID: min aggregate over ID
type parentID(module, int, min int).
parentID(Module, Dist, ID) :-
     possibleParent(Module, _, Dist, ID).

parent(Mod1, Mod2) :-
     possibleParent(Mod1, Mod2, Dist, ID),
     parentID(Mod1, Dist, ID),
     parentDist(Mod1, Dist).
```

Fig. 7. Meld code to select a single nearest parent

Since Meld gives up explicit state variables, an alternative mechanism is required in situations where we would typically use one, such as selecting a single unique parent when generating the P trees in the shape change algorithm, i.e., constructing a tree from a set of relations that form a DAG (see Figure 6). Without explicit state, we must make use of aggregates to pick a single parent. As shown in Figure 7, we first use an aggregate to find the minimum distance of any possibleParent such that we create the flattest possible tree. This aggregate gives us a parentDist fact which we can combine with possibleParent facts to give us parents at minimum distance, but if we have two or more parents that are equidistant from the final module we will end up with multiple parents. We must, therefore, use a second minimization aggregate over the IDs of the parents to uniquely select one parents at each distance, providing a set of parentID facts.

After both aggregates have been applied, the parentDist fact specifies a single optimal distance and the parentID facts specify a single possible parent for each distance. These can, therefore, be combined to specify a single unique parent. This method of identifying a single parent is clunky and work is being done to find a better solution.

C. Polled Predicates and State Change

LDP detects conditionals using a polled approach, where new searches are started at every “tick”. This causes LDP to repeatedly proceed with searches even if no state has changed. For example, a simple predicate that spreads the final label to neutral neighbors in the target region will wastefully propagate pattern matchers from all final state

```
// predicate with no guard
(a,b); (a.state == 0) & (b.inside == 1);
            b.state = 0;

// predicate with previous-state guard
(a,b); (a.state == 0) & (b.inside == 1)
    & (a.state != a.last(1).state);
            b.state = 0;
```

Fig. 8. Preventing needless execution of LDP conditionals with state guards

TABLE I
MESSAGING OVERHEAD FOR LDP AND MELD PROGRAMS (200 METAMODULES, 100 TIMESTEPS)

Program	LDP		Meld
	No Guards	Guards	
Single Condition	13311	152	172
Full Planner	349514	193204	202117

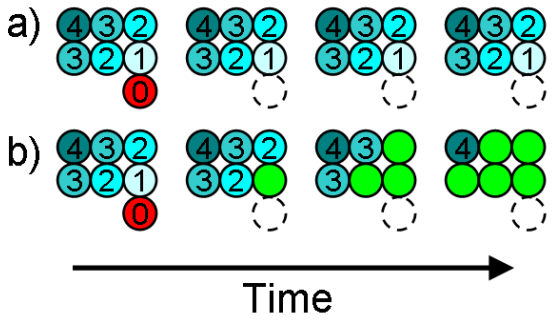

Fig. 9. A gradient after the root has been deleted. With explicit state, as in LDP, stale gradient information remains until explicitly removed (a). With Meld (b), the gradient information is neatly deleted, i.e., unproved, automatically.

modules every tick. To prevent this, we can add additional clauses which stop a search from propagating if there has been no change in local state since the last timestep (Figure 8). The statistics in Table I show that adding the guard condition shown in Figure 8 reduces messaging for that particular condition by 98.8%, and adding guard conditions to all relevant conditions in the metamodule planner reduces message overhead by 42.3%. Meld uses event-driven processing only when some facts change, and thus does not suffer this problem.

D. Retracting Expired Information

As LDP operates by setting state variables, we must explicitly delete any state which is no longer valid. An example of such state is the deletion gradient which moves destroyer metamodules away from the ensemble's surface, described in Subsection II-C. This gradient must be removed after deletion finishes, or destroyers may be incorrectly routed (see Figure 9). We do this by propagating the originator of the gradient along with the gradient value, and having the originator broadcast a deletion message once the gradient is no longer needed. In contrast, Meld manages the deletion of facts automatically, and thus the gradient is deleted at all points merely by retracting the fact that the originator exists.

VI. DISCUSSIONS

We have described the implementation of a new shape planning algorithm in two new declarative programming languages, MELD and LDP. Both languages allow for a natural expression of our new shape planning algorithm, though the example also serves to highlight some of the similarities and differences between the two.

In terms of similarities, Meld and LDP are both well suited to tracking persistent operating conditions, evaluating predicates over all neighbors, and expressing variable-length conditional expressions. In terms of differences, LDP's implicit state model lends itself well to managing intermediate state (e.g., first-seen or any-of clauses), whereas Meld's explicit state approach is better suited to managing changing state (e.g., parent relationships in an ensemble with moving components).

In either language, alternative implementations of a given primitive can have widely varying efficiency results. For instance, some Meld programs may needlessly rederive facts, whereas other programs may produce the same end result with less rederivation. Our future work will explore how to maximize the efficiency of commonly encountered distributed primitives in either language.

Given Meld and LDP's similarities and differences, a logical suggestion would be to combine the two in a hybrid, ideally preserving the strengths from both. Unfortunately the divergent state models make this a challenging task. For instance, Meld's deletion unproves facts which depend upon what is being deleted, but it is not obvious how to undo changes to state variables. Further future work may find alternative approaches that combine the advantages of both languages. In the meantime, we are beginning to use both together at a coarser granularity, implementing entire modules in one or the other and interfacing those modules.

REFERENCES

[1] Bernadette Charron-Bost, Carole Delporte-Gallet, and Hugues Fauconnier. Local and temporal predicates in distributed systems. *ACM Trans. Program. Lang. Syst.*, 17(1):157–179, 1995.

[2] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason Campbell, and Padmanabhan Pillai. Distributed watchpoints: Debugging large multi-robot systems (in preparation). *International Journal of Robotics Research*, 2007.

[3] M. DeRosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai. Scalable shape sculpting via hole motion: Motion planning in lattice constrained modular robots. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '06*, 2006.

[4] Daniel Dewey and Siddhartha S. Srinivasa. A planning framework for local metamorphic systems. Technical Report CMU-RI-TR-XX, The Robotics Institute, Carnegie Mellon University, 2007.

[5] K. B. Lamine and L. Kabanza. Reasoning about robot actions: A model checking approach. *Advances in Plan-Based Control of Robotic Agents*, pages 123–139, 2002.

[6] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Helle rstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM Press.