2008 IEEE International Conference on
Robotics and Automation
Pasadena, CA, USA, May 19-23, 2008

Programming Modular Robots with Locally Distributed Predicates

Michael De Rosa
Seth 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.campbell]@intel.com

***Abstract*— We present a high-level language for programming modular robotic systems, based on *locally distributed predicates* (LDP), which are distributed conditions that hold for a connected subensemble of the robotic system. An LDP program is a collection of LDPs with associated actions which are triggered on any subensemble that matches the predicate. The result is a reactive programming language which efficiently and concisely supports ensemble-level programming. We demonstrate the utility of LDP by implementing three common, but diverse, modular robotic tasks.**

I. Introduction

There are a significant number of challenges to programming modular robots. These challenges can broadly be divided into two areas: managing the ensemble and controlling the individual modules. In this paper we present an approach to programming the ensemble based on *locally distributed predicates (LDP)*. LDP lets a programmer specify how the entire ensemble should behave by breaking the problem down into how small groups of robots should interact. In this manner LDP significantly reduces the disparate problems of inter-robot timing and concurrency, resource management, and the lack of global knowledge at any individual robot.

Traditional imperative programming languages, such as C/C++, Java, do little to address the ensemble-level issues involved in programming modular robots. These languages are inherently oriented towards a single processing node, and require significant additional effort when used in a distributed setting. In addition to creating a representation of the data needed for an algorithm, the programmer must determine what information is available locally and what must be obtained from remote nodes, the messages and protocol used to transfer this data, mechanisms to route or propagate information through multiple hops as needed, and a means to ensure the consistency of this data. Furthermore, in algorithms to control ensembles, it is often necessary to express and test conditions that span multiple modules. Languages that constrain the programmer to the perspective of a single node make such algorithms difficult to implement.

Related work in modular robot programming can be roughly divided into three categories: logical declarative languages for programming distributed systems, reactive programming techniques for robots, and functional approaches with roots in sensor network research. In the first category

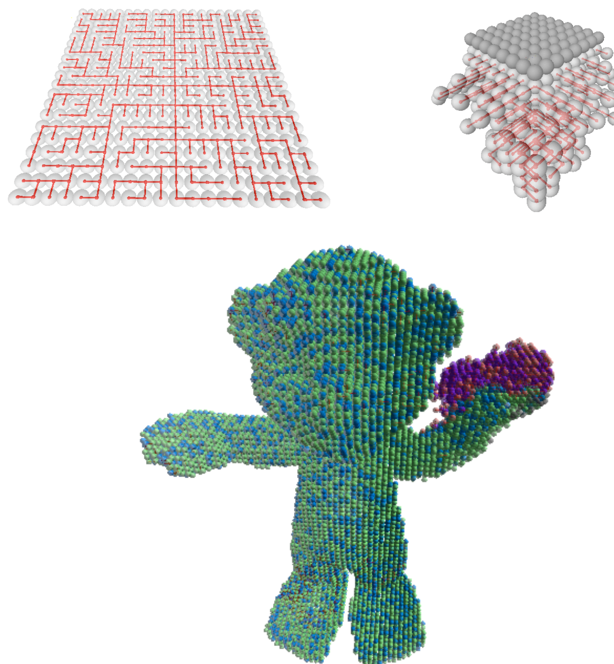

Fig. 1. Visualizations of distributed programs written in LDP. Top-left: Data aggregation (400 modules). Top-right: Metamodule shape planner (450 metamodules). Bottom: Metamodule shape planner (22,000 metamodules), red area is slated for deletion.

we have such tools as P2 [1] and Meld [2], which provide a logic programming facility for distributed systems (and modular robots in particular, in the case of Meld). These tools are powerful, in that programs written in them have certain provable properties, but this provability limits the expressive range of the languages. Subsumption architectures [3] and reactive programming languages [4] provide other expressive formats for programming single robots, but are not specialized for ensemble-level multi-robot programming. Functional approaches include such languages as Regiment [5] and Proto [6]. These approaches also raise the abstraction level, but are less concerned with geometry (i.e., the neighbor relationships) and actuation.

We extend prior work on distributed watchpoints [7] to arrive at a reactive programming approach aimed at modular

978-1-4244-1647-9/08/$25.00 ©2008 IEEE.

TABLE I
LDP OPERATORS AND PRIMITIVES

<table>
<tr><td>Boolean</td><td>& | !</td></tr>
<tr><td>Mathematical</td><td>+ - * / %</td></tr>
<tr><td>Comparison</td><td>> < >= <= == !=</td></tr>
<tr><td>Temporal</td><td>prev() next()</td></tr>
<tr><td>Topological</td><td>neighbors()</td></tr>
<tr><td>Set</td><td>union size any ···</td></tr>
</table>

robots. The resulting language, LDP, is used to specify lists of actions that are predicated on local distributed conditions. LDP can express conditions that span multiple modules, can incorporate both temporal and spatial relations, and use various computational and logical operations. This mechanism breaks free from the node-centric paradigm enforced by C-like languages, and moves towards implementing algorithms from an ensemble perspective. Furthermore, LDP relieves the programmer from having to coordinate data distribution, as the system automates the distribution and coordination of all data needed to evaluate the conditions and carry out any triggered actions.

In the next section we give an overview of LDP. We follow this description with detailed examples of how LDP can be used to efficiently implement problems from three different domains. In Section III we demonstrate how LDP can implement a snake-gait in a chain-style modular robot. Section IV describes a typical data aggregation example by creating a spanning tree and then combining values from each robot's sensors. Finally, in Section V we demonstrate how LDP can be used in a metamodule system, implementing a complete planner in 9 lines of LDP. Overall, we show that LDP is effective for concisely expressing some real-world modular robotic programs, can greatly reduce programmer effort in implementing a complex distributed algorithm, and can do so efficiently.

II. LOCALLY DISTRIBUTED PREDICATES

Locally distributed predicates are useful for describing and detecting distributed state configurations in subsets of an ensemble. In contrast to classical global predicate evaluation [8], [9], which attempts to detect conditions over entire distributed systems, LDP operates on fixed-size, connected subgroups of modules. The advantages of such an approach are twofold. First, searching in fixed-sized, connected subgroups is a significantly less expensive operation than searching the entire ensemble, allowing us to execute more searches more frequently. Second, the notion of small, connected groups of modules reflects the natural structure of distributed programs written for large modular robots, where global decisions are expensive and rare.

A. LDP Syntax

An LDP program consists of data declarations and a series of statements, each of which has a predicate clause and a collection of action clauses. When a predicate matches on a particular sub-ensemble, the actions are carried out on that sub-ensemble. LDP has no explicit control structures, such as looping or function calls, though these can be emulated with the use of flag and counter variables.

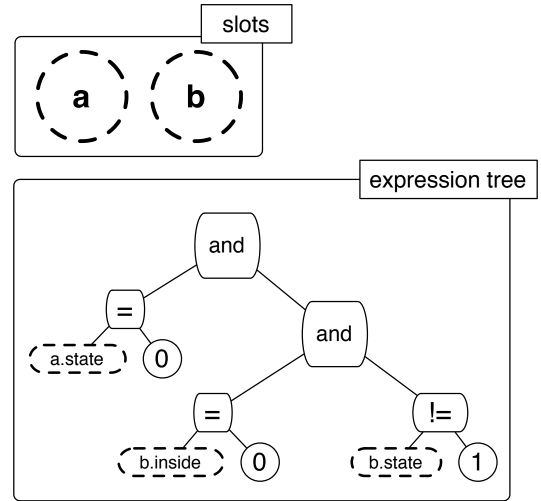

Fig. 2. Schematic representation of a PatternMatcher object, the fundamental LDP data structure used for both data sharing and condition detection. To continually find all possible instances of a condition, PatternMatchers are generated at every timestep on every module, and are passed between modules. 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.

Each predicate begins with a declaration for each module involved in the statement. These modules are searched for in the order listed and, most importantly, there must be a path between all modules in a matching subensemble. The condition itself is composed of numeric state variables (expressed as `module.variableName`, temporal offsets (the operators `prev()` and `next()`), and topology restrictions (via the neighbor relation `neighbors(moduleA,moduleB)`). These primitives can be linked together with the mathematical, boolean, and comparison operators summarized in Table I.

The core language of LDP extends the condition grammar for distributed watchpoints with the addition of set variables (variables prefixed with a `$` are set variables, as in `moduleName.$setVar`) and the requisite operators for manipulating these variables. Specifically, we have implemented `intersection()`, `union()`, `size()`, `any()`, `add()`, and `remove()`.

B. Distributed Predicate Detection

The core of the LDP execution model is the *Pattern-Matcher* (Figure 2). A PatternMatcher is a mobile data structure that encapsulates one distributed search attempt for a particular statement. This object migrates around the sub-ensemble until either it fails to match or it matches. In addition to the active PatternMatchers, each robot has a collection of continually running threads (one for each statement in the program) which creates new PatternMatchers at every event of importance, e.g., a clock tick, a new sensor

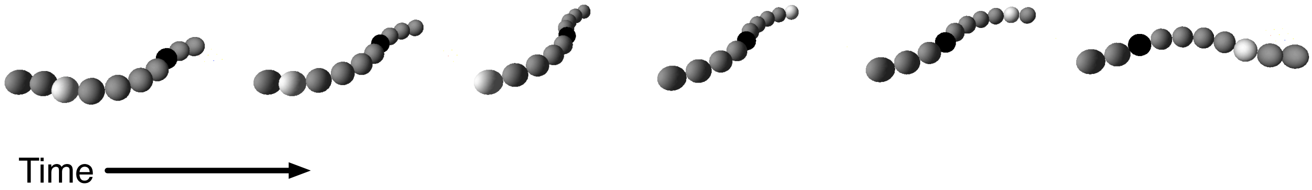

Fig. 3. Snake gait in chain-style modules. Black modules are actuating negative joint angle, white modules are actuating a positive joint angle.

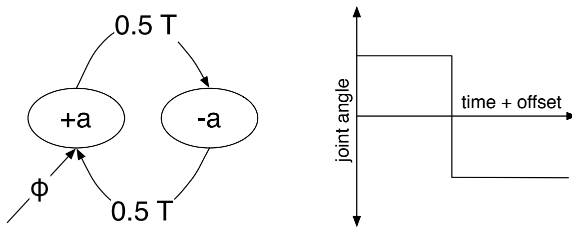

Fig. 4. a) Snake gait phase automaton, b) Joint angle vs. time graph

reading, etc. In its simplest form, a new PatternMatcher is created for each statement on each robot at every time tick.

The PatternMatcher is an object which encapsulates a search attempt for a particular predicate. Every PatternMatcher contains an expression tree, which encodes the boolean condition that the LDP is attempting to match. This expression tree contains storage for state variable values, to 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. Triggering Actions

By themselves, distributed watchpoints [7] were insufficient to serve as a programming language, as they could not trigger arbitrary actions on predicate matches. For LDP, we add a final clause to the predicate—the trigger. We define three types of triggers: (1) setting a state variable to a value, (2) changing the topology of the system, and (3) calling an arbitrary function implemented by the robot's runtime. Any predicate may have more than one trigger action, however we require that all the actions must be executed on the same module. This eliminates the need for locking or synchronization across multiple actions and/or modules. Notice that an LDP predicate can specify trigger actions to execute on any one module in the matching subensemble, and because that module lies within the matching subensemble we can use the route information gathered during the corresponding PatternMatcher's journey to notify the acting module of the predicate's match. This avoids the need for a standalone multihop communications infrastructure.

D. Implementing LDP

Using LDP in any given modular robotic system is straightforward. The system must call an LDP initialization function to set up various data structures. The runtime requires the implementation of three basic routines which (1) enumerate a module's current neighbors, (2) transmit PatternMatchers between neighboring robots, and (3) invoke the statement threads at appropriate intervals, e.g., `tick()` function. Finally, the system must ensure that incoming LDP messages trigger the appropriate callback.

Each application that uses LDP must additionally implement variable initialization, access, and modification for any state variables used in the program. The programmer must also implement any custom library functions that will be called from LDP actions.

III. Example Program: Phase Automata For Snake-Style Gait

Phase automata [11] are a technique for scalably describing cyclic gaits in chain-style modular robots, such as Polypod [12] and Superbot [13]. A phase automaton consists of a set of multiple states with associated actions, whose transitions are governed either by external events or an internal globally-synchronized clock. A phase automaton additionally possesses an initial time offset ϕ, which can vary from module to module.

A simple phase automaton for a snake-like robot is shown in Figure 4a. In this automaton, the joint angle of a particular module is set to either $+\alpha$ or $-\alpha$ in a cyclic manner, with period T. The initial phase offset ϕ is determined by a module's position in the chain, and increases by a constant $\triangle\phi$ at each module. The resulting gait is shown in Fig.3.

To implement this automaton in a modular robotic system, there are two fundamental tasks: distributing the correct phase offset to each module, and setting the joint angle to

```
    // per-module state variables
    int id;              // the id number of module. Read-only.
    int parent = -1;     // the id number of the previous module in the chain
    float time;          // the current time at the module. Updated by runtime.
    float offset = 0;    // the module's phase offset
    float angle = 0;     // the joint angle of the module's central joint. Changing this value actuates the module's motor.

    // this predicate causes module 1 to recognize itself as the leader
1   forall (a) where (a.id == 1) do a.parent = a.id;
    // link successive neighboring modules, forming a chain from head to tail and setting phase offsets
2   forall (a,b) where (a.prev(1).parent != a.parent) & (a.id < b.id)
        do b.parent = a.id, b.phase = a.phase + 0.1;
    // set joints to bend positively or negatively at the indicated phase offsets
3   forall (a) where ((a.time + a.phase) % 1.0 == 0.5) & (a.parent != -1) do a.angle = 15.0;
4   forall (a) where ((a.time + a.phase) % 1.0 == 0.0) & (a.parent != -1) do a.angle = -15.0;
```

Fig. 5. Complete Source Code for Snake Gait Example, $\triangle\phi = 0.1, \alpha = 15.0, T = 1.0$

the correct value based on the current time and offset. Figure 5 shows the complete code for these two steps. This example program assumes that the modules have unique id numbers that are ordered in increasing fashion from the "head" of the chain, which has id 1.

The first statement sets the `parent` of the head module to be its own id. The second statement triggers only when a module has changed its parent variable, and traverses one link of the chain at a time, setting the parent and phase variables of successive modules.

The next two lines implement the actual snake gait of the phase automaton. For each module, these statements check to see if the current time, modified by the phase ϕ and period T, corresponds to one of the transition points of the automaton. If so, the joint angle of the module is set appropriately. It is interesting to note that the first two statements are broadly applicable to any chain-style robot, and that they may be reused for different gaits.

There are several interesting features of the phase automaton code which bear closer examination. As all of the statements in the program run simultaneously and concurrently, it is necessary to enable and disable the various steps of the algorithm by the use of *gating subpredicates*. These are predicates which match only once, or only for a certain period of time. There are two such subpredicates in Figure 5. The first, `a.prev(1).parent != a.parent` in line 2, ensures that the predicate matches only on the tick after the `parent` variable has changed. This prevents the continual (and unnecessary) reassignment of parents to the modules in the chain. The second subpredicate `a.parent != -1`, in lines 3 and 4, prevents the predicate from matching (and motion from occurring) until a parent and phase offset have been assigned to the module.

We evaluated the snake gait program on chains of 5 to 20 modules, and found that the resulting gait appeared visually similar to that presented in the original paper . The additional gaits (rolling and centipede) described in [11] could also be implemented using similar LDP programs.

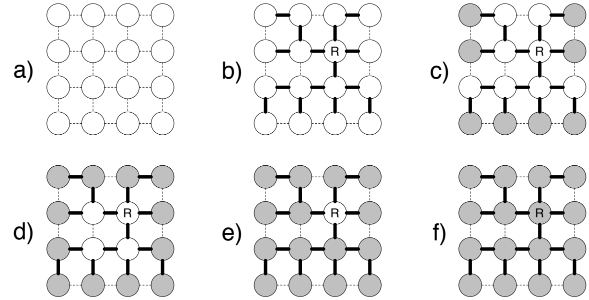

Fig. 6. Data Aggregation Algorithm: a) Available communications links b) Shaded links show establishment of spanning tree rooted at R c) Leaves propagate data upwards (shaded circles) d-e) Additional levels of propagation f)Data aggregation complete

IV. Example Program: Data Aggregation

A common task in modular robots, and distributed systems in general, is the aggregation of a distributed set of values at a central point. In this example program, we implement distributed averaging of a scalar variable over the entire ensemble. This is useful for such tasks as distributed sensing, localization, and center of mass estimation.

To obtain the average of a variable over all modules, we use a technique popular in sensor networks. We begin by designating one module as the root of a spanning tree, and having all modules transmit their value up the hierarchy of the tree to the root, where it is accumulated (Fig.6). The naïve implementation of such an algorithm would be for each module to transmit its variable's value, and for that value to be propagated all the way to the root of the tree, where the root module would add it to a running total.

Propagating each value independently is clearly inefficient, and so instead we implement summing and averaging at each level of the tree, so that only one data value must be passed up to a module's parent. The difficulty with this technique lies in knowing when all of a module's children have sent it information, so that the module can propagate the sum to a higher level of the tree. To solve this, we have each module maintain two set variables. One tracks immediate neighbors that are known not to be its children. The other

```
int isSeed;                     // set to 1 on the spanning tree's root, 0 otherwise. Read-only.
int id;                         // the id number of module. Read-only.
int parent = -1;                // the id number of the parent module in the tree
set<int> $notChildren = {};     // the set of module ids of neighbors who are not children
set<int> $children = {};        // the set of module ids of neighbors who are children, and have provided data
set<int> $neighbors;            // the set of a module's neighbors' ids. Updated by runtime.
int isComplete = 0;             // set to 1 is a module has completed aggregation
int sensor;                     // the variable to average over. Read-only.
int sum = 0;                    // sum of all sensor values received
int count = 0;                  // number of modules that have transmitted data
int average = 0;                // average of the sensor value over all modules

  // build spanning tree from seed outwards. Note each module's parent field is initialized to -1 above.
1 forall (a) where (a.isSeed == 1) do a.parent = a.id;
2 forall (a,b) where (a.parent != -1) & (a.parent != a.prev(1).parent) & (b.parent == -1)
    do b.parent = a.id;
  // build notChildren sets (all of the neighboring modules that are not b's children)
3 forall (a,b) where (a.parent != -1) & (a.parent != a.prev(1).parent) & (a.parent != b.id)
    do b.$notChildren.add(a.id);
  // start propagation at leaves
4 forall (a) where (size(a.$neighbors) == size(a.$children) + size(a.$notChildren))
    do a. isComplete = 1;
  // propagate data up parent links
5 forall (a,b) where (a. isComplete != a.prev(1). isComplete) & (b.id == a.parent)
    do b.sum = b.sum + a.sum + a.sensor,
    b.count = a.count + b.count + 1,
    b.$children.add(a.id);
  // compute average
6 forall (a) where (a.count > 0) do a.average = a.sum / a.count;
```

Fig. 7. Complete Source Code for Data Aggregation Example

tracks immediate neighbors that are its children and have already provided it with data. When the size of these two sets sums to the total number of neighbors that a module has, it can transmit its own information up the tree. This algorithm is not the most efficient or robust choice [14], [15], but it serves to illustrate how one might implement such a task.

The code in Figure 7 is an implementation of this averaging algorithm. It has 6 statements, spread over 5 different phases. Note that these phases are sequenced by explicit conditions in the predicates — the order in which they are listed is unimportant. The first two statements establish a spanning tree rooted at the designated module. The next statement adds all of module `a`'s neighbors who are in the tree but not children of `a` to the set `a.$notChildren`. The next statement begins propagation at each level by setting the `isComplete` variable to 1 once all children have provided data. If a module becomes a leaf, the fifth statement adds its running `count` and `total` to that of its parent, and adds it to the parent's `$children` set. Finally, the sixth statement continually sets the known average to be the `total` of all reported values divided by the `count` of reporting modules. The code as presented computes the average only once, but the addition of a "reset" mechanism based on epochs or changing sensor values is a simple change.

We evaluated the performance of the data aggregation algorithm on a simulated robot ensemble. The modules were arranged in a square lattice, in flat planes with sizes ranging from 5 by 5 (25 modules) to 20 by 20 (400 modules). In all cases, the number of messages required was exactly three times the number of discrete communications links between the modules. Each phase of the algorithm (spanning tree construction, not-child set construction, and data aggregation) required that each adjacent pair of modules exchange one message. In terms of time complexity, the algorithm required time linear in the depth of the spanning tree to complete.

V. Example Program: Metamodule Planner

As our final example, we explore the problem of distributed shape planning for an ensemble of lattice-style modular robots. We use an extension of the shape change algorithm described in [16]. 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. 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.8.

A. The planning algorithm

The basic planner in [16] finds a sequence of rearrangements to go from a starting configuration to reach a target shape while maintaining global connectivity. The algorithm operates on metamodules, i.e., particular structures of modules, which are assumed to provide an abstraction where one metamodule can spawn a new metamodule in an adjoining empty spot, or absorb an adjacent metamodule to create an

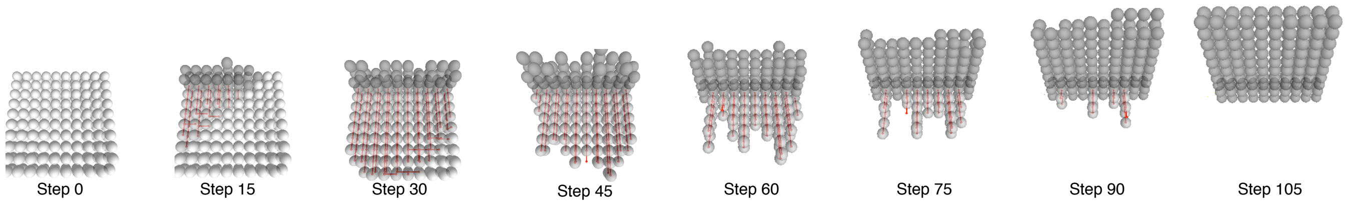

Fig. 8. Metamodule-based Shape Planner. Grey metamodules at top of structure are being created, while those at the bottom are generating deletion trees (red arrows) and being destroyed.

```
int isSeed;                      // set to 1 on the metamodule which initiates the motion planner, 0 otherwise. Read-only.
int id;                          // the id number of metamodule. Read-only.
int parent = -1;                 // the id number of the parent metamodule in the deletion tree
int state = NEUTRAL;             // the role of the metamodule: NEUTRAL,PATH, or FINAL.
int inside;                      // set to 1 if the metamodule is inside the target shape, 0 otherwise. Read-only.
set<int> $notChildren = {};      // the set of metamodule ids of neighbors who are not children
set<int> $spaces;                // the set of adjacent free locations where an additional metamodule should be created,
                                 // updated by runtime
set<int> $neighbors;             // the set of a metamodule's neighbors' ids. Updated by runtime.
function create(int);            // creates a new metamodule adjacent to the calling one at a given space
function destroy();              // destroys the current metamodule by dispersing it into adjoining metamodules
```

```
  // propagate FINAL state from seed outward to all modules already in target shape
1 forall (a) where (a.isSeed == 1) do a.state = FINAL;
2 forall (a,b) where (a.state == FINAL) & (b.inside == 1) do b.state = FINAL;
  // create new metamodules at edges of start shape
3 forall (a) where (a.state == FINAL) & (size(a.$spaces) > 0) do a.create(a.$spaces.any());
  // propagate PATH state to all metamodules outside start shape
4 forall (a,b) where (a.state == FINAL) & (b.inside == 0) &
     (b.state == NEUTRAL) do b.state = PATH;
5 forall (a,b) where (a.state == PATH) & (b.state == NEUTRAL) do b.state = PATH;
  // build deletion trees from FINAL out through PATH metamodules
6 forall (a,b) where (b.state == PATH) & (b.parent == -1) &
      ((a.parent != -1) | (a.state == FINAL)) do b.parent = a.id;
  // build notChildren sets (all of the neighboring modules which do not have b as a parent)
7 forall (a,b) where (a.parent != -1) & (a.parent != b.id); b.$notChildren.add(a.id);
8 forall (a,b) where (a.state == FINAL) do b.$notChildren.add(a.id);
  // delete PATH metamodules with no children
9 forall (a) where (a.state == PATH) &
     (size(a.$neighbors) == size(intersect(a.$notChildren,a.$neighbors))) do a.destroy();
```

Fig. 9. Complete Source Code for the Basic Metamodule Planner Example

empty spot. Initially, the ensemble of metamodules is in the start shape. During execution, metamodules are created and destroyed to reach the target shape.

The planing algorithm starts with a seed metamodule, which must lie in the intersection of the start and goal shapes. Each metamodule is in one of three states at any given time, NEUTRAL (the initial state for all modules), PATH, and FINAL. The seed metamodule marks itself as being in the FINAL state, then recruits every neighboring metamodule in the goal shape (and every neighbor's neighbor, recursing as long as possible) to also enter the FINAL state. It recruits a similar marking of every NEUTRAL neighbor not in the goal shape as being a candidate for removal. To do it marks these metamodules as being in the PATH state.

Preserving global connectivity when removing metamodules is one of the primary objectives of the planner. By successively expanding this initial set of PATH-state metamodules, the planner creates PATH-state tree structures that will sequence deletion operations later on. Every metamodule in these PATH state trees has a link to its parent, and as long as the link remains, the module will remain connected to the goal shape. Eventually, the trees have no further space to expand, at which point, the leaves can be safely trimmed without risking loss of connectivity. In Fig.8, the start shape is indicated by the lighter colored metamodules, the goal shape by the darker colored ones, and PATH-state trees are indicated by (red) arrows.

B. Implementation

The implementation of the motion planner runs at the *metamodule* level, on structured subgroups of modules. This allows for the creation and destruction of metamodules, as their constituent modules can be absorbed or provided by other nearby metamodules. To implement this application,

we "ported" the LDP runtime to the metamodule level, which required implementing communication and state variable storage across multiple modules. The (complete) code for the planning algorithm is shown in Fig.9. The state variables `isSeed`, `inside`, and `$spaces` are dependent on the start and goal shapes of the specific plan, and are initialized and managed by the low-level support code.

The first two lines spread the FINAL state to the seed, and then to every contiguous metamodule which is inside the target shape. The third line causes new metamodules to be created at empty locations in the goal region which adjoin metamodules already in the FINAL state. The fourth and fifth lines propagate the PATH state outwards from the edges of the FINAL region to all metamodules that are outside the goal shape. The sixth line creates a forest of trees rooted at the edges of the FINAL region, and spreading throughout any metamodules outside the target region. Statements seven and eight create `notChildren` sets, in a similar fashion to the data aggregation of Section IV. Finally, the ninth statement deletes PATH-state metamodules with no remaining children. (i.e., the leaves of each PATH tree)

VI. Discussion & Conclusions

We have demonstrated the utility of LDP by implementing three common classes of modular robotic algorithm. LDP provides a concise abstraction of distributed state, and helps to separate the actions of the algorithm from the support code necessary in traditional imperative languages.

As with any language, there are certain tasks which are more or less difficult to express in LDP. In particular, distributed state comparison and simple temporal relationships are quite naturally written in LDP. The lack of any explicit control structures or ordering makes certain other tasks more difficult. In particular, imposing an execution sequence (e.g., different "phases") on an LDP program, requires gating subpredicates to bound the times when certain statements can be active. This is especially important for statements which we want executed exactly once on each module. Also, the use of snapshot consistency complicates multiple read and writes to the same variable during a single tick, as the read value will be taken from the snapshot (and not the current value of the variable). Further, the issue of race conditions and concurrent operation is thus still an area of active research, with new language constructs currently under development.

Finally, we note that the design of the LDP language and runtime is deliberately amenable to extension with new primitives and operators. The addition of new primitives (such as per-edge variables or subexpression quantification) allow LDP to address more specialized application domains. As LDP is derived from the distributed watchpoint language, it is also possible to debug LDP programs in a distributed fashion within the language itself.

Acknowledgment

This research was sponsored by the National Science Foundation (NSF) under grant no. CNS-0428738. The views and conclusions contained in this document are those of the author 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. Additional funding provided by Intel Research Pittsburgh. The authors would like to thank Siddhartha Srinivasa and Daniel Dewey for their assistance with the metamodule planning algorithm, and Michael Ashley-Rollman for invaluable insights into distributed declarative programming.

References

[1] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[2] M. Ashley-Rollman, S. Goldstein, P. Lee, T. Mowry, and P. Pillai, "Meld: A declarative approach to programming ensembles," in *Proceedings of the IEEE International Conference on Robots and Systems IROS '07*, 2007.

[3] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal Of Robotics And Automation, RA-2*, pp. 14–23, April 1986.

[4] G. Berry, "The esterel v5 language primer," Centre de Mathematiques Appliquees, Ecole des Mines and INRIA, Tech. Rep., 1999. [Online]. Available: http://www-sop.inria.fr/meije/esterel/esterel-eng.html

[5] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*. New York, NY, USA: ACM Press, 2007, pp. 489–498.

[6] J. Beal and J. Bachrach, "Infrastructure for engineered emergence on sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, no. 2, pp. 10–19, 2006.

[7] M. DeRosa, S. Goldstein, P.Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging large multi-robot systems," in *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '07*, 2007.

[8] C. M. Chase and V. K. Garg, "Detection of global predicates: Techniques and their limitations," *Distributed Computing*, vol. 11, no. 4, pp. 191–201, 1998.

[9] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, vol. 26, 1991, pp. 167–174.

[10] M. DeRosa, S. C. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging large modular robotic systems," *International Journal of Robotics Research (special issue, to appear)*, 2007.

[11] Z. Ying, M. Yim, C. Eldershaw, D. Duff, and K. Roufas, "Phase automata: a programming model of locomotion gaits for scalable chain-type modular robots," in *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, (IROS 2003)*, 2003.

[12] D. Duff, M. Yim, and K. Roufas, "Evolution of polybot: A modular reconfigurable robot," in *Proc. of COE/Super-Mechano-Systems Workshop*, 2001.

[13] B. Salemi, M. Moll, and W.-M. Shen, "SUPERBOT: A deployable, multi-functional, and modular self-reconfigurable robotic system," in *Proceedings of the IEEE International Conference on Intelligent Robots and Systems IROS '06*, 2006.

[14] A. Manjhi, S. Nath, and P. B. Gibbons, "Tributaries and deltas: efficient and robust aggregation in sensor network streams," in *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2005, pp. 287–298.

[15] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson, "Synopsis diffusion for robust aggregation in sensor networks," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM Press, 2004, pp. 250–262.

[16] M. Ashley-Rollman, M. DeRosa, S. Srinivasa, P. Pillai, S. Goldstein, and J. Campbell, "Declarative programming for modular robots," in *IROS 2007 Workshop on Modular Robots*, 2007.