Locally Distributed Predicates: A Technique for Distributed Programming

Michael De Rosa

CMU-CS-10-118
May 2010

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Seth Goldstein, Co-chair
Peter Lee, Co-chair
Garth Gibson
Maurice Herlihy, Brown University

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2010 Michael De Rosa

This work was partially funded by the National Science Foundation (NSF) under grant no. CNS-0428738, Intel Labs Pittsburgh, and a generous contribution from the Microsoft Corporation.

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 the Microsoft Corporation, of Intel Corporation, or the U.S. Government.

Keywords: Distributed Programming, Modular Robotics, Distributed Predicates, Coordination Languages, Distributed Snapshots

For Mom.

Abstract

New research in wireless networks, sensor networks, and modular robotics has spurred renewed interest in distributed programming techniques. Distributed programming is inherently more difficult than its single-threaded equivalent, due to the need for an executing thread of a distributed program located at one computation node to access state located at a different node. Traditionally, remote state has been aggregated or summarized using distributed snapshots or global predicate evaluation.

Traditional techniques for gathering state in distributed systems may be inappropriate for large-scale, sparsely-connected systems, as they are bandwidth intensive and scale poorly. We have identified a novel class of distributed predicates in such systems that we term locally distributed predicates (LDPs). These are predicates over the local neighborhood of a particular node, bounded to a finite number of communication hops. These locally distributed predicates allow a programmer to describe the state configuration of a bounded subgraph of a distributed system.

In this work, we formalize locally distributed predicates, and present two algorithms for detecting stable subclasses of LDPs. We then show how to perform common distributed detection tasks with LDPs, and how, with simple extensions, LDPs can serve as the foundation for a reactive programming language for distributed systems. We iteratively develop the language $\mathcal{L}$, showing how the addition of various language features extends the expressiveness of the language. We present implementations of many distributed algorithms in $\mathcal{L}$ from multiple application domains. We then implement $\mathcal{L}$ twice, as a naive interpreted runtime, and as an efficient bytecode execution engine. Using our implementations, we proceed to characterize the performance and scalability for $\mathcal{L}$, and its suitability for various distributed programming tasks.

Acknowledgments

Many thanks go, first and foremost, to the love of my life: my wife Jennifer. Without her constant love and support, none of this would have been possible. My greatest thanks also to my parents and family for their support, and for raising me with the curiosity and wonder that science requires.

To my advisors, Peter and Seth, thank you for the guidance in all things research-related, a sense of humor in assigning me "impossible" problems, and the freedom to let me follow my own path, even when you disagreed with me. My thanks to the committee members, for volunteering their time to help shepherd this work. My thanks to my undergraduate minion Bobby Prochnow, for his timely help with porting $\mathcal{L}$ to embedded hardware. To the researchers at Intel Labs Pittsburgh, especially Jason Campbell and Padmanabhan Pillai, I wish I could make you co-authors. Much of the work within these pages is the result of years of fruitful collaboration, random brainstorming, and late-night deadline scrambles. My thanks.

Finally, my thanks to the many friends who've helped make the last five and a half years fly by. To Michael Ashley-Rollman, Nels Beckman, Andrew Bernard, Deepark Garg, Daniel Golovin, Elise Dunphe, Stano Funiak, Allison Giordano, Emre Karagozler, Brian and Rachel Kirby, Ivan Kourtev, Rahmi Marasli, Amar Phanishayee, the staff of PotA, Manos Renieris, Ben Rister, Alison Schmauch, Michael Weller, and the many others I'm sure I'm forgetting.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Overview

With the rise of many-core systems, peer-to-peer applications, and sensor networks, the problem of distributed programming has taken on new relevance. While previous generations of programmers were mostly concerned with single-processor, or at most single-machine, programming it is now common to find research results targeting systems composed of hundreds or thousands of cooperating, yet still independent, nodes. Distributed programming brings with it a specific set of challenges for developers, as it represents a fusion of concurrent programming and scalable networking techniques.

Three of the main challenges facing developers of distributed applications are: *distributed state*, *concurrency*, and *asynchronicity*. The challenge of distributed state occurs when data resides on multiple computation nodes, and more than one of them must be accessed in order to perform an operation. Alternatively, state can be distributed if it is resident on one node but must be accessed by a different node. Accessing remote state is qualitatively different than accessing local state, as the programmer must contend with communications latency and failure rates. Distributed state is typically addressed via caching and mirroring protocols, which ensure that distributed state is available locally, and that it remains consistent when remote changes are applied.

Concurrency refers to the presence of multiple simultaneously executing threads of execution, which must potentially share processors, memory, communications, and storage resources. Concurrent programs pose difficulties when multiple processes compete for a shared resource, causing such problems as data races and deadlocks. Concurrency is usually managed by placing locking or transaction mechanisms around resources which do not support multiple simultaneous mutators, thereby serializing execution flow around the resource.

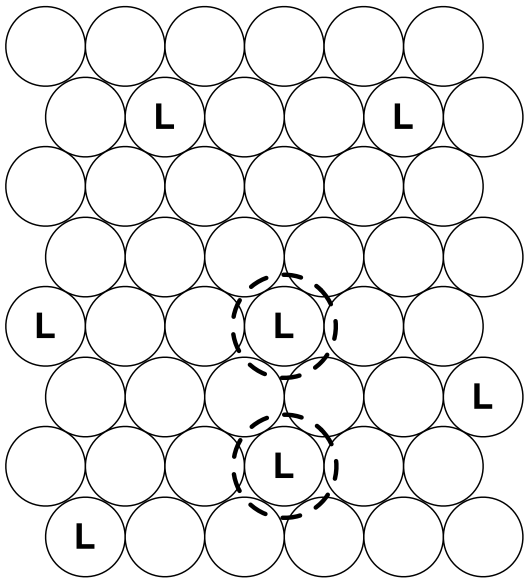


Figure 1.1: Faulty Radius-2 Leader Election.

Finally, asynchronicity refers to the lack of a globally-accessible clock or other synchronization object, leading to different programs executing at different rates. Asynchronicity causes a breakdown in causality, as it sometimes becomes impossible to tell which of two actions occurred before the other. Asynchonicity is countered through the construction of distributed clocks, or the use of alternate channels of causality (such as message delivery or interaction with the environment).

It is our intuition that both asynchronicity and concurrency are subordinate phenomena to distributed state, and that it is distributed state that accounts for many of the difficulties in programming distributed systems. The major problems of concurrency (data races, resource ownership, deadlock) are all caused by the interactions of multiple processes with some piece of distributed state. Take data races as an example: races occur because of multiple processes reading/writing a piece of state, without adequate protections on ordering or atomicity. Similarly, asynchronicity creates problems mainly when it interacts with program state, as writing to state variables is itself a form of serialization.

If we accept the premise that distributed state is a key factor in the difficulty of programming distributed systems, how should that influence our programming tools and approach? Clearly, we must have tools that treat distributed state as a first-class entity, and provide significant support for representing and detecting distributed state configurations. Any tool that allows us to represent distributed state effectively will, by necessity, provide at least limited solutions to asynchronicity and concurrency.

1.2 Motivating Example

As an example of the difficulty of distributed programming, let us take a simple example: radius-k leader election. Briefly, this is the problem of cooperatively electing a leader from among multiple processes while ensuring that there are no two leaders within communication distance k of each other (Fig. 1.1). This form of leader election is used in many distributed systems: e.g. sensor networks (to elect the sensor with the strongest reading), modular robotics (to select the leader of a metamodule), and general-purpose distributed systems (to establish routing topologies or overlays).

There are a number of existing algorithms for distributed leader election (30; 29), using either random selection or a set of election criteria. What if there is a fault in our leader election implementation? How can we efficiently detect the case of multiple overlapping leaders? Traditional approaches for detecting the occurrence of overlapping leaders (14) would require aggregating the state from all processes to a central location.

If the distributed system in question is relatively small, or is densely connected, then its total network diameter is relatively small, and aggregating state at a central location is a reasonable course of action. If the system contains many processes, or those processes are sparsely connected, then such an approach becomes prohibitively expensive. Moreover, the problem of detecting multiple leaders within radius k does not require a centralized approach, as all of the state required to determine such a property is (by definition) within a bounded radius of k communication links.

1.3 Topology-Aware Predicates

In many large-scale applications, the combination of sparse connectivity and expensive communications makes system-wide approaches to property detection impractical. The example in the previous section underscores the need for a way to detect distributed properties that is reflective of the underlying topology of the distributed system. What is needed is a means of detecting *size-constrained* properties, which is *local* to the processes involved.

We call this class of properties *topology-aware predicates*, predicates whose structure reflects system topology. In the case of verifying radius-k leader election, an example of a simple topology aware predicate is: there is no connected chain of discrete processes of length k or less, such that the chain contains two leaders.

We can conceive of many different means of expressing the topologies required by such predicates. At their most general, topology-aware predicates can be constructed over arbitrary network graphs. Unfortunately, arbitrary graph structures complicate our ability to develop efficient, provably consistent algorithms for detecting such predicates. As a simpler first step in this direction, we will examine the class of predicates that can be expressed in terms of fixed-length, linearly connected subgroups of processes. We have named this class of properties *locally distributed predicates*, and the thesis of this dissertation is that:

Thesis statement: Locally distributed predicates are an expressive and efficient means of detecting topology-aware properties in sparsely-connected distributed systems, and as such locally distributed predicates provide a useful tool for implementing and understanding a wide class of distributed algorithms, in a variety of application domains.

1.4 Organization

This goal of this thesis is to develop an expressive and efficient means of representing and detecting distributed properties in large, sparsely-connected distributed systems. To that end, we have developed locally distributed predicates (LDPs), a novel construct for describing distributed properties over connected subgroups of a distributed system. After proving the theoretical soundness of this construct, we proceed to extend the basic predicate grammar with a number of syntactic enhancements, allowing locally distributed predicates to express a large array of distributed algorithms concisely. We show the utility of locally distributed predicates by implementing a number of algorithms from a variety of application domains, including sensor networks and modular robotics. We then create a compiler and runtime which efficiently implement the core algorithms for locally distributed predicate search.

This dissertation is organized as a top-down examination of LDPs, their use, and their implementation. We begin by formally introducing the concept of locally distributed predicates, and setting them in context with other classes of distributed properties. Chapter 2 presents the definition, basic algorithms, and complexity measures of LDPs. These form the essential groundwork for the rest of the dissertation, as they inform and guide the practical implementation and use of LDPs.

Chapter 3 explores the practical uses of LDPs by iteratively developing the programming language $\mathcal{L}$, a reactive programming language for sparse distributed systems. We develop this language through a large number of example programs, which both introduce new language features and demonstrate the utility of LDPs in a large variety of application scenarios.

Chapter 4 presents two implementations of this programming language: a naïve (Section 4.1) runtime that clearly illustrates the basic concepts and algorithms, and a more optimized runtime (Section 4.2) that incorporates an application-specific bytecode engine, and is suitable for use in embedded systems. Chapter 5 evaluates the performance of the $\mathcal{L}$ runtime, beginning with synthetic microbenchmarks, and progressing through comparison against a theoretical lower bound, concluding with a qualitative study of how $\mathcal{L}$ allows for rapid prototyping and modification of distributed algorithms. Chapter 6 recapitulates the major contributions of the dissertation, provides an overview of related distributed programming research, and concludes with an exploration of open questions and future research avenues.

Chapter 2

Theory

The qualities of scale and dynamism that make distributed systems exciting also pose certain challenges to the unwary developer. Fundamentally, the difficulties of programming distributed systems stem from *distributed state*. For a system to be distributed, it must in some way make use of the state of multiple processing nodes, or else it is just a number of disconnected computations. Making use of distributed state requires transferring state information over communications links between nodes, which introduces delay and potential message loss. If the nodes in a distributed system are not centrally coordinated, then there may also be no notion of time shared by all nodes. If there is no common time-base, then what does it mean to say that nodes P_i and P_j are in a given state at the same time? The issue of *asynchronicity* makes reasoning about distributed systems difficult.

Without a global clock, the notion of the state of the entire distributed system becomes ill-defined, as there is no shared time at which all nodes can be examined. In his seminal paper (44), Lamport described the *happens-before* relationship imposed by causality. Briefly, the happens-before relationship states that, for a system to be consistent with causality, the sending of a message must precede its reception (at all observers). This relationship can be used to create a partial-ordering on a set of distributed events, and show whether or not a particular view of a distributed system is *consistent* with causality. This observation by Lamport led to a large number of algorithms for reasoning about distributed state, relying on the use of the happens-before relation to ensure consistency.

Interest in proving properties of distributed systems led to the creation of many *distributed predicate detection* algorithms. Properties such as deadlock (59) and termination (76) can be expressed as predicates over the state of distributed processing nodes. The complete state of the distributed system can be gathered and evaluated, to determine if the predicate is true. However, there is a problem with such a naïve approach. The gathering of state creates an implicit serialization of the parallel event streams in the system. There may be multiple consistent serializations, and the predicate may be true in some of them, but not others. This leads to a situation where predicates can be *possibly-true* (true in at least one consistent serialization) or *definitely-true* (true in all consistent serializations) (17).

To eliminate the need to explore an exponentially-bounded number of serializations, much of the research in distributed predicate detection has been targeted at the class of *stable predicates* (7). Stable predicates are distributed predicates that, once evaluated as true under any consistent serialization, will remain definitely-true for all subsequent serializations. The classical algorithm for detecting such predicates is the Chandy-Lamport snapshot algorithm (14). There have been many subsequent algorithms (40) focused either on improving the performance of distributed snapshots, or on delineating subclasses of stable predicates with more tractable detection properties.

Topology-aware predicates are distinguished from more traditional classes of distributed predicates in that they are concerned with topology, rather than detectability. They form an orthogonal schema to stable properties and their subclasses. Specifically, in systems where there is a sparse communications topology, and topology is important, it is our assertion that locally distributed predicates are a more natural and efficient way to model and detect distributed properties. This is due to the fact that, as global operations are expensive in sparse distributed systems, algorithms for such systems will rely primarily on operations defined over local neighborhoods of nodes, for reasons of both efficiency and comprehensibility. These local neighborhood operations will in turn define properties and predicates with similar topological restrictions.

This chapter begins with a brief review of distributed systems and common distributed predicate classes, before describing the characteristics of LDPs. We will then present two algorithms for detecting important subclasses of locally distributed predicates, including proofs of correctness and complexity measures.

2.1 Preliminaries

A *distributed system* is a graph (P, L), where P is the set of nodes (processes) and L is the set of communication links (edges) connecting them. Let $n = \mid P \mid$ and $l = \mid L \mid$.

The execution history at node $P_i \in P$ is the alternating sequence of states and events $\langle s_i^0, e_i^1, s_i^1, e_i^2, s_i^2, e_i^3, \cdots \rangle$, where s_i^0 is the initial state of the node and e_i^k is the kth event at that node. An *event* at a node can be a message reception, message transmission, or an internal event. A non-initial state s_i^k can be uniquely identified by the event e_i^k which immediately precedes it. We denote an arbitrary state from P_i as s_i^*, and the current state (observed locally by P_i) as s_i.

The state of a node is composed of an application-dependent number of named state variables, with numeric or boolean values. The value of state variable var at the kth state of node P_i is denoted by var_i^k, and the current value of a particular state variable is likewise denoted var_i.

A *channel* C_{ij} is a reliable, unidirectional FIFO link between nodes P_i and P_j, with a finite (but not bounded) delay. For two nodes P_i and P_j, the channel C_{ij} exists if $L_{ij} \in L$. The graph is undirected, thus $L_{ij} \in L \Rightarrow L_{ji} \in L$. We say that two nodes P_i, P_j are *neighbors* if $L_{ij} \in L$. The state of a channel $SC_{ij} = transit(e_i^l, e_j^m)$ is the set of messages sent by P_i up to event e_i^l and not received by P_j up to event e_j^m. We denote the k^{th} message sent along C_{ij} from P_i to P_j as m_{ij}^k.

An *execution* of a distributed system with n nodes is $(E, \prec)$, where $\prec$ is the Lamport happens-before relation on E, the set of events $E = \bigcup_{i \in P} e \in E_i$. E_i is the totally ordered event sequence at node P_i.

For convenience, we define the extended happens-before relation $\prec^*$, which defines happens-before over both states and events, with states being replaced by their immediately preceding event (e.g. $s_i^j \prec^* e_k^l \Rightarrow e_i^j \prec e_k^l$).

A *consistent cut* K is a subset of E such that if $e \in K$ then $(\forall e')\ e' \prec e \Rightarrow e' \in K$. We can define a *consistent sub-cut* of a distributed system as a consistent cut over some subset of P. To show that a sub-cut K is consistent, it is sufficient to show:

1. there is no *orphan* message m_{ij} such that $receive(m_{ij}) \in K \wedge send(m_{ij}) \notin K$
2. if $e_i^k \in K$, then $(\forall j < k) e_i^j \in K$

Less formally, a sub-cut K is consistent if all message reception events in K have matching message transmissions, and the events of P_i included in K form a prefix of the execution history at P_i (67). K_2 is subsequent to K_1iff $\forall e \in K_1, e \in K_2$, and $\exists e_1 \in K_1, \exists e_2 \in K_2$ s.t. $e_2 \notin K_1, e_1 \prec e_2$.

2.2 Classical Distributed Properties

The field of distributed predicate detection is a well-established and active one. There are a large number of algorithms and predicate types described in the literature. We briefly summarize both the definition and detectability results from several of the more important classes of predicates, deferring a more thorough categorization to several excellent survey papers on the topic (40; 8).

***stable*:** A predicate that, once found to be true by a global snapshot, will remain true (14). This is the only class of predicate we discuss that explicitly includes channel state, and it can be used to detect deadlock, termination, and the number of tokens in a token-passing system.

***strongly stable*:** A stable predicate that, if true on some cut (consistent or not), must remain true on all subsequent cuts (42). Deadlock is not strongly stable. Neither is the number of tokens in a token-passing system, as it requires channel state.

***strong stable*:** A stable predicate that, if true on some consistent cut, must remain true on all subsequent consistent cuts (71). Termination and deadlock are strong stable, though distributed garbage collection is not.

***locally stable*:** A stable predicate such that, once it becomes true, none of the variables used by the predicate change in value (54). Termination, deadlock, and global virtual time are locally stable, determining the number of tokens in a token-passing system is not.

***conjunctive stable*:** A predicate formed from the conjunction of local predicates, each of which is individually stable (27; 67).

Strongly stable, strong stable, locally stable, and conjunctive stable are all subclasses of stable predicates. Conjunctive stable predicates are a subclass of strong stable predicates. There is no proven relationship between the classes of strong stable and locally stable, though Schiper and Sandoz speculated that they were similar.

Each of these classes of predicates is differentiated by the requirements they place on detection algorithms: the presence/absence of channel state, the need for consistent cuts, and the ways in which node state may change once a predicate becomes true. Algorithms for detecting various classes of stable predicates are distinguished by the particular class of predicates they detect, the type of communication channel they assume (FIFO, non-FIFO, or causal delivery), and the degree to which they interfere with existing message traffic.

2.3 Locally Distributed Predicates

Locally distributed predicates are distinguished from most classical categories of properties in that they are defined by their topology, rather than their detectability. An LDP is a distributed predicate over a *fixed-size, linearly-connected* group of of nodes. A set of nodes that satisfies the LDP must form a connected chain, with communication channels linking each of them. This formulation allows LDPs to easily express communication distance (number of hops) and other simple topological constraints, such as adjacency (whether of not two nodes share a channel), shared adjacency (whether or not two nodes share a neighbor), etc.

2.3.1 Formal Properties

A *locally distributed predicate* (LDP) is $Q(a_1, a_2, \cdots, a_k)$, a predicate over the state of k abstract nodes a_1 through a_k. Let $\mid Q \mid = k$ be the size of the predicate, and $V = \langle s^*_{i_1}, s^*_{i_2}, \cdots \rangle$ be a vector of states from distinct nodes. V contains at most one state from each abstract node in Q, and thus $\mid V \mid \leq k$. We say that Q is *satisfied* by V if:

1. $L_{ij} \in L \, \forall P_i, P_j$ where s_i^* immediately follows s_j^* in V
2. $Q \leftarrow V = \texttt{true}$, where $\leftarrow$ is element-wise state substitution of V into Q

We will refer to these two properties as *adjacency* and *substitution*, respectively. Adjacency requires that, if s_i^* and s_j^* are adjacent in the state vector V, then their respective nodes P_i and P_j must be neighbors. Taken as a whole, this requires that the nodes represented in V form a linear, connected chain via their communications channels.

2.3.2 State Substitution Operator

Substitution is the second requirement for LDP satisfaction. An LDP is expressed in terms of the state and topology of a finite number of abstract nodes, and must be instantiated with the values of concrete nodes in order to be evaluated. This instantiation is performed via element-wise state substitution. Take the simple LDP:

$$Q(a_1, a_2) = flag_{a_1} \wedge flag_{a_2}$$

This is a predicate over two abstract nodes, a_1 and a_2, which must be neighbors (due to adjacency). Both a_1 and a_2 have a boolean state variable $flag$, whose value must be true. Note that this formulation explicitly omits any reference to consistency or detectability, it is concerned only with topology.

To instantiate this LDP over the state of two concrete nodes P_i and P_j, we first construct a vector with their state:

$$V = \langle s_i^*, s_j^* \rangle$$

Element-wise state substitution yields:

$$(flag_{a_1} \wedge flag_{a_2}) \leftarrow \langle s_i^*, s_j^* \rangle = (flag_i^* \wedge flag_j^*)$$

Where a_1 has been instantiated as P_i and a_2 has been instantiated as P_j. If this predicate is true, and P_i and P_j are neighbors, then Q is satisfied by P_i and P_j.

Once state substitution has been performed, the predicate can be evaluated to produce one of three output values: `true`, `false`, and `undefined`. As $|V| \leq |Q|$ in $Q \leftarrow V$, there may be instances where $|V| < |Q|$, and the predicate can not be definitively evaluated as either `true` or `false`. In this case the predicate's value is `undefined`.

2.3.3 Example LDPs

Using LDPs, it is easy to formulate topology-limited versions of many classical distributed properties. One need merely decide on the number of abstract nodes, and state the property in terms of those nodes.

```
Node $P_i$ begins detection of $Q$:
if $Q \leftarrow \langle s_i \rangle = true$ then
    return $\langle P_i \rangle$ as matching $Q$
else
    $m = \langle s_i \rangle$;
    send $m$ to all neighbors of $P_i$;
end

Node $P_j$ receives message $m = \langle s^*_{i_1}, s^*_{i_2}, s^*_{i_3}, \cdots \rangle$:
$V = \langle s^*_{i_1}, s^*_{i_2}, s^*_{i_3}, \cdots s_j \rangle$;
if $Q \leftarrow V = true$ then
    return $\langle P_{i_1}, P_{i_2}, P_{i_3}, \cdots, P_j \rangle$ to $P_j$ as matching $Q$
else
    if $\mid V \mid < \mid Q \mid$ then
        $m' = V$;
        send $m'$ to all neighbors of $P_j$ not already represented in $V$;
    else
        failed to match;
    end
end
```

Algorithm 1: LDP Detection (Non-Inhibitory)

For example, the termination of two adjacent nodes is represented as:

$$Q(a_1, a_2) = terminated_{a_1} \wedge terminated_{a_2}$$

While the presence of more than 1 token in a 3-node ring is:

$$Q(a_1, a_2, a_3) = token_{a_1} + token_{a_2} + token_{a_3} > 1$$

Finally, mutual deadlock of two neighboring nodes is:

$$Q(a_1, a_2) = (waitingOn_{a_1} == id_{a_2}) \wedge (waitingOn_{a_2} == id_{a_1})$$

It is obvious that the detectability of the above LDPs is consistent with their classical categorization. In other words, though all of the examples are locally distributed predicates, they also fall into categories such as stable, strong stable, etc. We can therefore define a strong stable LDP, for example, as a locally distributed predicate that, once it becomes satisfied for a particular subset of nodes, remains satisfied for that subset.

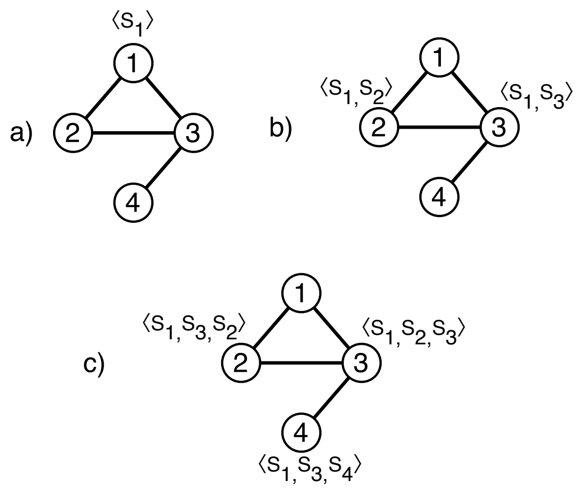


Figure 2.1: Progression of the LDP-Basic algorithm from initiation at P_1, through two phases of spreading.

2.4 The LDP-Basic Algorithm

To detect subgroups that match a given LDP, we have developed a simple distributed search algorithm. This algorithm, which we call LDP-Basic, gathers a state vector to be substituted into a given LDP. We show that the state gathered by LDP-Basic is a consistent sub-cut of the system. From (71), we know that this property allows LDP-Basic to detect properties that are strong stable, an important subclass of stable properties.

2.4.1 Algorithm

Detection begins at the initiator node P_i, where the current local state is substituted into Q. If this substitution evaluates to true, then P_i alone satisfies the LDP. If not, the 1-element vector $\langle s_i \rangle$ is sent to all neighbors of P_i.

When a message containing a state vector is received at node P_j, the node appends its current state s_j to the vector, and performs substitution into Q. If this substitution evaluates to true, then the nodes represented by the state vector satisfy Q. If not, the size of the vector is compared to the number of nodes specified by Q. If $\mid V \mid < \mid Q \mid$, then the state vector V is sent to all neighbors of P_j not already represented in V. In this way, we sequentially gather the current state from P_{i_1}, then P_{i_2}, etc.

Figure 2.1 illustrates the operation of LDP-Basic over a predicate Q of size 3. In Figure 2.1a, node 1 has created a vector containing its current state. In Figure 2.1b, that state has been transmitted to nodes 2 and 3, which have appended their current state to the vector, and so on in Figure 2.1c. If any other these vectors satisfied Q, that condition would have been detected at the last node necessary to fulfill the predicate.

Note that the algorithm can be concurrently initiated at multiple nodes, and will correctly identify multiple overlapping subgroups that satisfy Q, identifying them by their distinct vector of nodes.

2.4.2 Consistency Proof

To show that the state captured by this algorithm is a consistent sub-cut, we must show that all states that are captured obey the Lamport happens-before relation. There are two components to this proof. The first is to show that if $e_i^k \in K$, then $(\forall j < k)e_i^j \in K$. This is trivially true, as s_i^k encompasses the effect of $(\forall j < k)e_i^j$ by definition. Next, we must show that there are no orphan messages within the sub-cut. This requires the use of the following lemma:

Lemma 1 *Under LDP-Basic, if state vector V contains $s_i^{x_i}$ and $s_j^{x_j}$, where $s_i^{x_i}$ appears in V before $s_j^{x_j}$, then $s_i^{x_i} \prec^* s_j^{x_j}$.*

Proof Node state is only appended to V. Each node appends only one element, its current state. If node P_j appends $s_j^{x_j}$ to V while $s_i^{x_i}$ is already a member of V, then P_j must have received V (and thus $s_i^{x_i}$) from another node. Thus, by the causality of message delivery, $s_i^{x_i} \prec^* s_j^{x_j}$.

Note that, while $s_i^{x_i}$ must appear in V before $s_j^{x_j}$, it does not need to appear immediately before $s_j^{x_j}$. Thus, while P_j and P_i do not need to be adjacent nodes, the nodes that connect them in V must form a chain of adjacency.

Theorem 2.4.2.1 *The state vector gathered by LDP-Basic is a consistent sub-cut.*

Proof By contradiction: Take any two states from V, $(s_i^{x_i}, s_j^{x_j})$ where $s_i^{x_i}$ appears in V before $s_j^{x_j}$. Suppose that there is an orphan message m_{ji}, such that $receive(m_{ji})$ is known at $s_i^{x_i}$, but $send(m_{ji})$ is not known at $s_j^{x_j}$. From Lemma 1, $s_i^{x_i} \prec^* s_j^{x_j}$. From the causality of message delivery, $send(m_{ji}) \prec^* receive(m_{ji})$. From the assumption that m_{ji} is an orphan, $receive(m_{ji}) \prec^* s_i^{x_i}$ and $s_j^{x_j} \prec^* send(m_{ji})$. As $\prec^*$ is commutative, $s_j^{x_j} \prec^* receive(m_{ji})$, and thus $s_j^{x_j} \prec^* s_i^{x_i}$. This contradicts $s_i^{x_i} \prec^* s_j^{x_j}$, and thus there can be no such message m_{ji}.

This proof shows that any sub-cut produced by LDP-Basic is a consistent one; from Schiper and Sandoz, we know that this means that LDP-Basic can detect strong properties, including the sub-categories of conjunctive stable and (potentially) locally stable properties. This includes such properties as termination, deadlock, and virtual time, when the properties are restricted to finite chains of nodes. Obviously, when detecting deadlock or termination, the LDP detection process must be separate from the node being observed.

```
Node $P_i$ begins detection of $Q$:
    Obtain an exclusive lock on spanning subtree $T_i$, of depth $\mid Q \mid$;
    Send $\langle Suppress, T_i \rangle$ to $c(T_i, P_i)$;

Node $P_j \in T_i$ receives $\langle Suppress, T_i \rangle$:
    Suppress transmission of any message not related to $P_i$'s detection attempt ;
    Send $\langle Suppress, T_i \rangle$ to all neighboring $P_k \in T_i$;
    Once $P_j$ has received $\langle Suppress, T_i \rangle$ from all neighboring $P_k \in T_i$, send $\langle Ready, P_j \rangle$ to $P_i$;

Node $P_i$ receives $\langle Ready, T_i \rangle$ from all $P_j \in T_i$:
    $P_i$ begins Algorithm 1 ;
    $P_i$ releases lock on subtree $T_i$ ;

Node $P_j \in T_i$ is unlocked from $T_i$:
    Resume transmission of any message not related to $P_i$'s detection attempt ;
```

Algorithm 2: LDP Detection (Inhibitory)

2.5 Detecting Stable Locally Distributed Predicates

The LDP-Basic algorithm is sufficient to detect LDPs in the class of strong stable predicates, but it will not detect the more general stable LDPs. In particular, LDP-Basic cannot detect predicates whose satisfiability may depend on channel state. We therefore introduce a second algorithm, which can detect a broader range of predicates.

2.5.1 Algorithm

Our second algorithm, LDP-Snapshot, is an extension of the basic detection method, and allows for the capture of all stable LDPs. It does so by coercing sub-graphs of the system into configurations where there is no channel state, and then performing the basic detection operation. By ensuring that all channels are empty, any stable LDP must reside purely in node state, and thus be observable via a consistent sub-cut. We model our work on the inhibitory algorithms of (19), who used a similar technique to create consistent cuts in non-FIFO systems.

LDP-Snapshot operates as follows: Let T_i be the spanning subtree of depth $\mid Q \mid$ rooted at P_i. It is obvious that all vectors of nodes that match Q and begin in P_i must be contained within T_i. To obtain a snapshot, P_i begins by obtaining an exclusive lock on all nodes in T_i, to prevent overlapping snapshots. Once the lock has been obtained, P_i distributes the message $\langle Suppress, T_i \rangle$ downwards through the tree, as

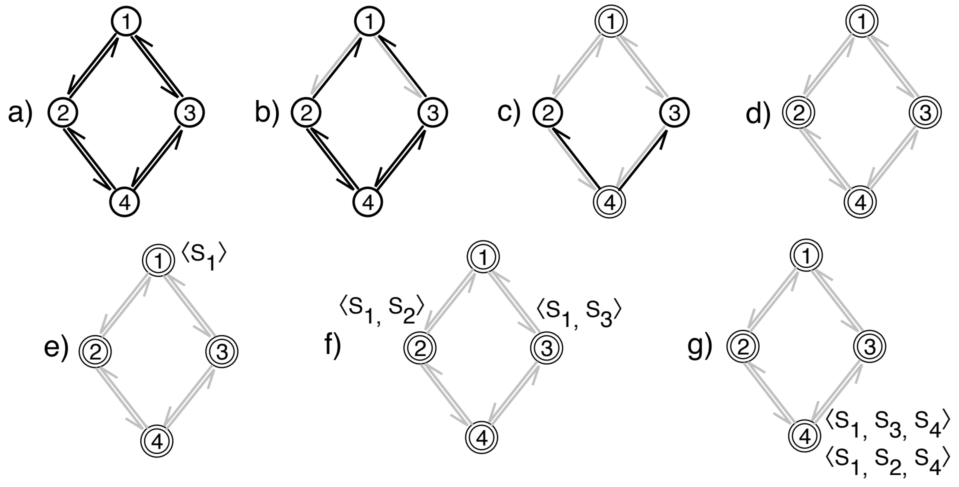


Figure 2.2: Progression of the LDP-Snapshot algorithm from initiation at P_1 (a) Initial state of T_1, with depth 3. (b) P_1 sends $\langle Suppress \rangle$ to its neighbors, and suppresses its outbound channels (represented in grey). (c) P_1 has received a suppression message from all neighbors in T_1, and is now $\langle Ready \rangle$, represented by a double outline. (d) All nodes have sent $\langle Ready \rangle$ messages. (e) Node P_1 initiating LDP-Basic, showing local state vector. (f,g) Continuation of LDP-Basic, showing state vectors growing.

seen in Figure 2.2. Once Suppress has been sent via a channel, transmission via the channel is delayed until after the snapshot is taken. Once a node has received Suppress messages from all neighbors that belong to T_i, it sends $\langle Ready, T_i \rangle$ to P_i via the spanning tree. Once P_i has received Ready messages from all members of T_i, it proceeds with LDP-Basic, and then unlocks the subtree.

2.5.2 Snapshot Consistency Proof

From (14), we know that distributed snapshots require a consistent set of node and channel state. In LDP-Snapshot, we first empty and suppress as communications channels within a subtree, and then take a consistent sub-cut of the node state in that subtree. Consistency of the sub-cut is obtained via the use of LDP-Basic, whose properties are described in Section 2.4. Proving that all channels are empty is as follows:

Theorem 2.5.2.1 *When LDP-Basic is executed by P_i, $SC_{jk} = \emptyset, \forall (P_j, P_k) \in T_i$.*

Proof Once a node P_j receives $\langle Suppress, T_i \rangle$ from P_k, it is guaranteed that $SC_{kj} = \emptyset$, and that all further messages along C_{kj} will be inhibited. A node P_j will not send $\langle Ready, P_j \rangle$ until it has received $\langle Suppress, T_i \rangle$ from all neighbors in T_i. Therefore, sending $\langle Ready, P_j \rangle$ guarantees that $\forall P_k \in T_i, SC_{kj} = \emptyset$. Once the initiator P_i has received $\langle Ready, P_j \rangle$ from all $P_j \in T_i$, then $\forall (P_j, P_k) \in T_i, SC_{jk} = \emptyset$. Therefore, when the basic LDP detection algorithm runs after P_i has received $\langle Ready, P_j \rangle$ from all $P_j \in T_i$, it will be guaranteed that $SC_{jk} = \emptyset, \forall (P_j, P_k) \in T_i$.

As channel state is empty, the state captured by LDP-Basic is a consistent capture of both channel and node state, and thus a distributed snapshot. From Chandy and Lamport, LDP-Snapshot is thus able to detect all properties in the class of stable LDPs.

2.6 Performance and Complexity

In evaluating our algorithms for detecting LDPs, we will show complexity results for three important metrics: overall message traffic, maximum channel bandwidth, and required state storage. We compare our algorithms to the classic Chandy-Lamport distributed snapshot algorithm. While there are more efficient snapshot algorithms (39; 79), the Chandy-Lamport algorithm is both widely known and very succinct.

Briefly, the Chandy-Lamport algorithm gathers a consistent snapshot of an entire distributed system in the following manner: An initiating node begins by saving its local state, and sending a snapshot request to all of its neighbors. Every other node, upon receiving the snapshot request for the first time, will send the initiator its saved state, and propagate the snapshot request on all subsequent outgoing messages. Finally, any node that has received a snapshot request, and subsequent receives a message without such a marker will forward that message to the initiating node. In this manner a complete and consistent copy of both node and channel state can be obtained.

We present upper bounds for systems of bounded degree (Table 2.1), which are reflective of the sparse-topology systems that LDPs were designed for. Additionally, we present upper bounds for general graphs (Table 2.2). Scale-free properties are indicated by bold text in both tables. In all of our analyses, the number of nodes is n, the maximum degree of the system is d, and the size of the relevant LDP is k. We note that it is most appropriate to compare the Chandy-Lamport algorithm's cost to the n initiator cost of the LDP algorithms, as the Chandy-Lamport algorithm is capable of detecting globally distributed predicates.

2.6.1 Messaging Cost

Message complexity is measured by assigning a cost of 1 to each message transmission. We assume that messages are not merged or batched in any way, though they would be in most concrete implementations. In a similar vein, the cost of a message carrying a single boolean value, and one containing the entire state of a node are both 1.

The base-line Chandy-Lamport snapshot algorithm has a message complexity of $O(n^2)$, dominated by two factors. The first is the need to send a marker message along each channel (of which there are potentially

Table 2.1: Complexity Measures of Detection Algorithms (Degree-d Topology)

Metric	C-L	LDP-Basic	LDP-Snapshot
Messages (Single Initiator)	$O(n^2)$	$\boldsymbol{O(d^{k-1})}$	$\boldsymbol{O(d^{2k-2})}$
Messages (n Initiators)	n/a	$O(nd^{k-1})$	$O(nd^{2k-2})$
Bandwidth (messages/channel)	$O(n)$	$\boldsymbol{O(d^{k-2})}$	$\boldsymbol{O(d^{k-1})}$
Storage (state elements/node)	$O(n)$	$\boldsymbol{O(k)}$	$\boldsymbol{O(k)}$

$O(n^2)$). The second is the need to aggregate the state of each node at some distinguished initiator, which involves n nodes sending their recorded state to the initiator, along paths that can be $O(n)$ hops long.

LDP-Basic's messaging complexity is determined by the spread of the state vector from the initiator, to each of its neighbors, to each of their neighbors, etc. for up to k nodes. In a fully connected system, each node has $O(n)$ neighbors, giving a cost of $O(n^{k-1})$ for a detection of a size-k LDP starting at some node P_i. Starting a detection attempt at every node raises the total cost to $O(n^k)$ messages.

The cost of LDP-Snapshot, exclusive of the application-dependent cost of acquiring a lock on T_i, is determined by three factors: the cost to send suppression messages along each channel in T_i, the cost for all members of T_i to send Ready messages back to P_i, and the cost to run LDP-Basic once. These costs are $O(\mid T_i \mid^2)$ (for any spanning tree), $O(\mid T_i \mid^2)$ (for the worst -case spanning tree), and $O(n^{k-1})$. As the maximum size of a depth-k spanning tree is n, these costs are $O(n^2)$, $O(n^2)$, and $O(n^{k-1})$ respectively, for a total complexity of $O(n^{k-1})$ for an attempt rooted at a given P_i. The complexity for each node to initiate the algorithm is $O(n^k)$ messages.

In a system with a maximum degree of d, these metrics change in important ways. The complexity of Chandy-Lamport is still $O(n^2)$, as aggregating state at a central location is an $O(n)$ operation for each node. The cost of LDP-Basic becomes dependent on the maximum degree d, leading to a cost of $O(d^{k-1})$ for a single initiator, or $O(nd^{k-1})$ for all nodes to be initiators. It is important to note that $O(d^{k-1})$ is a constant for a given predicate and maximum degree. Similarly, as the maximum size of a depth-k subtree becomes d^{k-1} in a system with maximum degree d, the cost of LDP-Snapshot becomes $O(d^{2k-2})$ for a single initiator, or $O(nd^{2k-2})$ for n initiators. This makes the LDP family of algorithms significantly cheaper than Chandy-Lamport snapshots for small predicates in sparsely-connected systems.

Table 2.2: Complexity Measures of Detection Algorithms (Arbitrary Topology)

Metric	C-L	LDP-Basic	LDP-Snapshot
Messages (Single Initiator)	$O(n^2)$	$O(n^{k-1})$	$O(n^{k-1})$
Messages (n Initiators)	n/a	$O(n^k)$	$O(n^k)$
Bandwidth (messages/channel)	$O(n)$	$O(n^{k-2})$	$O(n^{k-2})$
Storage (state elements/node)	$O(n)$	$\boldsymbol{O(k)}$	$\boldsymbol{O(k)}$

2.6.2 Bandwidth

We measure bandwidth complexity by observing the number of messages sent along each channel, and taking the maximum over all channels as the complexity. Chandy-Lamport has a bandwidth complexity of $O(n)$, in the case where the initiator node has only a single channel connecting it to the remainder of the system. In that case, all $n-1$ nodes must route their state along this edge.

The bandwidth complexity of LDP-Basic can be determined as follows: assume that the channel C_{ab} is the channel with the highest bandwidth. Furthermore, assume that P_b is located at distance $k-1$ from initiator P_i. The bandwidth through P_{ab} is then the number of distinct paths of length $k-2$ from P_i to P_a, as each of these paths will produce a distinct state vector, which must be forwarded to P_b. The maximum bandwidth is thus $O(n^{k-2})$ for arbitrary systems, or $O(d^{k-2})$ for systems with a maximum degree of d.

The maximum bandwidth needed for LDP-Snapshot is derived from two terms: the cost of every member of T_i sending a Ready message to P_i, and the cost of executing LDP-Basic. Using similar reasoning to Chandy-Lamport, the maximum bandwidth for each member of T_i to send a Ready message to P_i is $O(n)$ in an arbitrary system, or $O(d^{k-1})$ in a degree-d system. The maximum bandwidth for LDP-Snapshot is therefore $O(n^{k-2})$ for arbitrary systems, or $O(d^{k-1})$ for degree-d systems, a constant value for any given predicate and network topology.

2.6.3 State Storage

Once a snapshot or sub-cut has been initiated, the state of the involved nodes must be gathered and processed in some manner, so that the satisfiability of the distributed predicate can be evaluated. In Chandy-Lamport, the distinguished initiator must gather state from every node, requiring $O(n)$ space to store it before the global predicate can be evaluated. In the case of both LDP-Basic and LDP-Snapshot, state is carried in a vector message, and each vector is processed individually, requiring a maximum of $O(k)$ storage at any given node.

2.6.4 Mitigation of Cost Growth

An important shortcoming of the Chandy-Lamport approach is that as the system is scaled up, the costs of running the algorithm grow. As these costs depend only on n, there is nothing a system designer can do to control them. With LDPs, these costs depend instead on the degree of the connectivity graph and the complexity of the predicate and not, in general, on n. Hence, there are parameters available to the system designer that allow some control of the costs of running distributed predicate detection, even as the system size grows. For example, limiting the degree of connectivity between nodes or the length of predicates allowed can keep algorithmic complexity low despite a large value of n.

2.6.5 Reliability

An obvious challenge in this problem domain is the question of reliability in the face of unreliable nodes and channels. Communications channels between wireless devices or robotic modules are subject to significant rates of packet loss (46), over 70% in some cases. There are two potential solutions for such links. The first is the approach taken by TCP/IP (12), which involves creating a reliable channel out of an unreliable one by means of retransmission and acknowledgement protocols. The second is to repeatedly initiate any detection effort. As the predicates to be detected are stable, repeatedly running LDP-Basic over unreliable channels will eventually result in a detection (if one is to be found), as the predicate will remain true once it becomes true.

More difficult is the question of unreliable processes. If a node fails, how would the presented algorithms behave? It is easy to see that LDP-Snapshot, as presented, is unsuitable for use in systems with faulty nodes. If the process holding a subtree lock terminates unexpectedly, there is no way for members of the locked subtree to resume normal operation. To prevent disabling entire groups of nodes, we must either replicate the initiating node, or provide a timeout mechanism to release the lock in the event of node termination. LDP-Basic, as it does not interfere with node operation, is more robust in the face of such failures. If a given node fails during the execution of LDP-Basic, then that node state will not be recorded during construction of state vectors, and the node will not be part of any matching subgroup. It could be argued that this is correct behavior, as the node no longer exists.

Finally, there is the issue of disconnection. If a node or channel fails, it may partition the network graph into two or more pieces. In this case, both LDP algorithms will be unable to detect matching subgroups that involve processes in more than one partitioned subgraph. There is existing work on detecting various predicate classes in faulty environments (31; 45), and we believe that the application of similar techniques will allow for more robust detection of certain classes of LDPs.

Chapter 3

Applications

With the theoretical properties and basic algorithms of LDPs now in hand, we can turn our attention to the practical uses of topology-aware predicates. To do so, we will introduce a simple language for describing LDPs, and show its utility in a wide variety of application contexts.

This chapter iteratively constructs the language $\mathcal{L}$, using a series of 6 different distributed applications to motivate the design of the language. For each application, we will explore the syntactic and semantic choices needed to satisfy the problem. We have chosen algorithms from modular robotics, sensor networks, and general distributed systems to illustrate the broad applicability of $\mathcal{L}_,$ and thus of locally distributed predicates.

The chapter begins with the detection of simple distributed properties, for debugging and analysis purposes (Section 3.1). This application motivates a discussion of the basic syntax of $\mathcal{L}$, along with the initial semantics of the language. This is followed by a large number of example applications, which serve to introduce additional language features. We begin by introducing actions, creating a simple reactive programming language which we use to construct spanning trees (Section 3.2). The language is extended through the introduction of hybrid coding semantics, which we demonstrate by implementing phase automata for modular robotic gaits (Section 3.3). We add support for aggregate variables, which allows for reliable data aggregation, a key operation in many distributed systems (Section 3.4). Temporal operators and history support can be used to greatly reduce unnecessary message traffic, which we illustrate by implementing a minimum-exposure path algorithm for sensor networks (Section 3.5). Finally, we add support for variably-sized statements, a syntactic convenience which makes the implementation of larger programs more tractable. A partial implementation of metamodule-based shape change for modular robotics motivates this addition (Section 3.6). Finally, we conclude with two worked examples: scaffold-based planning for modular robots (Section 3.7.1), and synopsis diffusion for sensor networks (Section 3.7.2).

Table 3.1: LDP Features Used By Example Applications

Application	Actions	Var. Sized Predicates	Set Variables	Temporal Quantifiers	Hybrid Coding
Debugging					
Spanning Tree	X				
Gait Automata	X				X
Aggregation	X		X		
Guided Navigation	X			X	
Scaffold Planner	X		X	X	X
Metamodule Planner	X	X	X	X	X
Synopsis Diffusion	X			X	X

3.1 Basic Language

3.1.1 Application: Distributed Debugging

Designing algorithms for distributed systems is a difficult and error-prone process. Concurrency, non-deterministic timing, and state-space explosions all contribute to the likelihood of bugs in even the most meticulously designed software. These factors also make the detection of such bugs very difficult.

Tools to assist programmers in debugging distributed algorithms are few, and generally inadequate. Most are forced to fall back on standard debugging methods, such as debuggers (e.g. GDB(28)) or logging through console I/O such as printf(). Both debuggers and logging must be used very cautiously, or their file/console I/O can impose unintended serialization (altering the timing behavior of the distributed system) and possibly masking some bug manifestations. Debuggers are useful for detecting errors local to an individual thread or process, but are not effective for errors resulting from the interactions or states of multiple threads of execution that span multiple modules. Console or file logging may be used to detect some of these errors, but this requires collecting all potentially relevant state information at each robot. The information must then be centrally collected, correlated, and post-processed, in order to extract the details of the error condition. This requires significant effort, skill, and even luck on the part of the programmer.

One would like to have a tool that can allow programmers to easily specify and detect distributed conditions with minimal impact to the operation of the existing system. The tool would need to detect such conditions as deadlock, improper state combinations, and termination. The ideal debugger would support detection of these processes over arbitrary subgroups of nodes, but as we observed in Section 2.6, such global predicates are both expensive and intrusive. Thus, we limit the scope of the debugging to those predicates expressible in terms of LDPs.

3.1.2 Concrete Syntax

For a language to be useful for distributed debugging, it must satisfy 3 conditions:

***concise*:** The language must include no unnecessary constructs; this makes the language easy to learn and understand. Programs written in this language must themselves be naturally concise, as they will be revised and rewritten multiple times in the course of most debugging sessions.

***expressive*:** The language must be capable of expressing a wide variety of important and useful predicates, and should ideally support a simple extension mechanism to include additional functionality.

***unambiguous*:** Programs written in the language must the clear and unambiguous, both for human readers and for execution by a debugger/runtime engine.

The language $\mathcal{L}_0$ is a simple concrete syntax for locally distributed predicates. Programs in $\mathcal{L}_0$ are composed of declarations and statements. Declarations can be constants or variables. Variable declarations reference named state variables in the host program to be debugged.

Statements are the core of $\mathcal{L}_0$ programs, and each statement encapsulates one locally distributed predicate. Each statement is composed of a `forall` clause, which names and numbers *slots* for the nodes involved in the LDP, and a `where` clause, which contains the actual predicate. Predicates can be constructed from references to scalar state variables, using a C-style syntax, such as `nodeName.varName`. Variable references can be combined using the expected array of boolean, numeric, and comparative operators. Finally, additional topological constraints can be expressed using the boolean neighbor function, `n(a,b))`. The complete concrete syntax of $\mathcal{L}_0$ is shown in Figure 3.1. The EBNF notation used in this syntax marks non-terminals with angle brackets, keywords and terminal symbols in a typewriter font, and primitive types (integers, floating point, and legal identifier names) in italics.

With this basic syntax in hand, we can now express some of the properties described in previous sections. We begin with the motivating example that opened this dissertation: the detection of incorrect distributed leader election.

Listing 3.1: Leader Election (3-Hop)

```
1 scalar isLeader;                          // set to 1 if node is a leader, 0 otherwise
2
3 forall(a,n1,n2,b) where (a.isLeader == 1) & (b.isLeader == 1);
```

This example illustrates the two components of an $\mathcal{L}_0$ program, a declaration (of `isLeader`), and a statement. The statement on line 3 of Listing 3.1 can be read as follows: find all size-4 linear chains of

$$
\begin{array}{rl}
\langle\text{program}\rangle & \rightarrow \langle\text{declaration}\rangle^{*}\langle\text{statement}\rangle^{+} \\
\langle\text{declaration}\rangle & \rightarrow \langle\text{var declaration}\rangle|\langle\text{const declaration}\rangle \\
\langle\text{statement}\rangle & \rightarrow \texttt{forall (}\langle\text{slot declarations}\rangle\texttt{) where }\langle\text{predicate}\rangle\texttt{;} \\
 & \\
\langle\text{var declaration}\rangle & \rightarrow \texttt{scalar } \textit{name}\ \texttt{;} \\
\langle\text{const declaration}\rangle & \rightarrow \texttt{const scalar } \textit{name}\ \texttt{=}\ (\textit{float}\mid\textit{name})\texttt{;} \\
\langle\text{slot declarations}\rangle & \rightarrow \textit{name}\ (\texttt{,}\ \ \textit{name})^{*} \\
\langle\text{predicate}\rangle & \rightarrow \langle\text{scalar expression}\rangle \\
 & \\
\langle\text{scalar expression}\rangle & \rightarrow \langle\text{logical expression}\rangle \\
\langle\text{logical expression}\rangle & \rightarrow \langle\text{equality expression}\rangle((\texttt{\&}|\texttt{|}|\texttt{\^{}})\langle\text{equality expression}\rangle)^{*} \\
\langle\text{equality expression}\rangle & \rightarrow \langle\text{comparison expression}\rangle((\texttt{==}|\texttt{!=})\langle\text{comparison expression}\rangle)^{*} \\
\langle\text{comparison expression}\rangle & \rightarrow \langle\text{add expression}\rangle((\texttt{>}|\texttt{<}|\texttt{>=}|\texttt{<=})\langle\text{add expression}\rangle)^{*} \\
\langle\text{add expression}\rangle & \rightarrow \langle\text{mult expression}\rangle((\texttt{+}|\texttt{-})\langle\text{mult expression}\rangle)^{*} \\
\langle\text{mult expression}\rangle & \rightarrow \langle\text{expression atom}\rangle((\texttt{*}|\texttt{\%}|\texttt{/})\langle\text{expression atom}\rangle)^{*} \\
\langle\text{expression atom}\rangle & \rightarrow \langle\text{scalar}\rangle \\
 & \mid \texttt{(}\langle\text{scalar expression}\rangle\texttt{)} \\
 & \mid (\texttt{+}|\texttt{-}|\texttt{!})\langle\text{scalar expression}\rangle \\
\langle\text{scalar}\rangle & \rightarrow \textit{float} \\
 & \mid\ \texttt{false}\ \mid\ \texttt{true} \\
 & \mid\ \texttt{n (}\ \textit{name}\ \texttt{,}\ \textit{name}\ \texttt{)} \\
 & \mid \langle\text{variable ref}\rangle \\
 & \mid \textit{name} \\
\langle\text{variable ref}\rangle & \rightarrow \textit{scalar}\ \texttt{.}\ \textit{scalar}
\end{array}
$$

Figure 3.1: Basic $\mathcal{L}_0$ EBNF Syntax

nodes `a,n1,n2,b` where `a.isLeader` is 1 and `b.isLeader` is 1. Any match to this statement will correspond to two leaders within 3 hops each other, and the nodes assigned to `a,n1,n2` and `b` will provide the exemplar. Note that, if there are multiple paths between a given `a` and `b`, it would be entirely reasonable for the detection to occur multiple times.

The second example property is that of 2-party termination. This is the size-limited version of distributed termination detection, a property of significant research interest (76; 35). As in the previous example, there is a single scalar variable with an underlying semantics, and a predicate that is identical to that in Listing 3.1, but for the name of the variable.

Listing 3.2: 2-Party Termination

```
scalar completed;                      // set to 1 if node has terminated, 0 otherwise

forall(a,b) where (a.completed == 1) & (b.completed == 1);
```

Extending the scope of our properties to include three nodes of interest, we now examine the problem of deadlock (59). To detect all instances of deadlock occurring within a ring of three nodes, we must detect two separate conditions: pair-wise deadlock, and 3-party deadlock.

Listing 3.3: 3-Party Deadlock

```
1  scalar id;                                      // the unique id number of the node
2  scalar waitingOn;                   // the id of the node being waited on, otherwise -1
3
4  forall(a,b,c) where (a.waitingOn == b.id) & (b.waitingOn == c.id) &
5                      (c.waitingOn == a.id) & n(a,c);
6  forall(a,b) where (a.waitingOn == b.id) & (b.waitingOn == a.id);
```

The statement in lines 4 and 5 detects the condition where `a` waits on `b`, `b` waits on `c`, and `c` waits on `a`. The statement on line 6 detects pair-wise deadlock. It is obvious that, if we wish to detect larger deadlock cycles, we must include a predicate for each cycle size under consideration. This is due to the limitation that statements must be a fixed size, a limitation we will examine in depth in Section 3.6.

3.1.3 Semantics

Coupled with the introduction of the syntax of $\mathcal{L}_0$ is the semantics of the language. I will briefly describe the semantics of $\mathcal{L}_0$, deferring a formal semantics for the discussion of the complete language $\mathcal{L}$. For each node we wish to inspect, we will attach an $\mathcal{L}_0$ runtime to that node. This runtime will perform three tasks: reading local state variables, processing incoming and outgoing messages, and performing localized detection tasks.

In order to perform predicate detection, we must first choose what algorithm to use. In order to minimize the overhead and impact on the existing distributed system, the $\mathcal{L}_0$ runtime uses the LDP-Basic algorithm, as described in Section 2.4. Every $\mathcal{L}_0$ runtime periodically initiates predicate detection with LDP-Basic, propagating outward from the node. The runtime also responds to incoming state vectors, as specified in the LDP-Basic algorithm. Individual runtimes schedule their activation independently, and we do not require timing that is more than nominally accurate. One can think of each runtime as "ticking" at some frequency, initiating detection locally with each tick.

At every tick, the runtime performs the LDP-Basic algorithm on each statement, in order. Any arriving state vector messages are processed either asynchronously (when received) or as part of the next processing tick. The tick frequency is an application-specific parameter, supplied to the runtime at startup.

Statements, when evaluated, produce a scalar output value. This value can be either numeric or undefined, in the case of missing state variables or illegal operations. For example, if a detection event has been started on node `a` and has not yet propagated to other nodes, then a reference to `b.varName` would be undefined. Undefined values taint the output of operators: any operation that has an undefined value as one of its inputs produces an undefined output. Boolean truth is handled in a manner similar to that of C programs: a scalar value of 0 is false, while all other defined values are true.

Access to underlying state variables is performed via the runtime, and typically requires custom code in the runtime for each named variable. We assume that a variable read is atomic, but that reads of multiple variables are not. A detection attempt that uses a given variable multiple times will read it only once, and cache the value.

Note that, for the purposes of identifying nodes in predicate match attempts, we require that each node have an id number or other identifier. However, as this id must only be unique within the communication radius of the largest statement in the $\mathcal{L}_0$ program, the use of pseudo-unique randomized identifiers, or other similar techniques (62; 69; 24), is acceptable.

3.2 Actions

As presented, the language $\mathcal{L}_0$ has no other function than to detect locally distributed predicates. On detection, it can only signal that the predicate has been matched, and provide the list of matching nodes. This is acceptable (and in some ways preferable) for debugging and observation, as it provides us with a guarantee that the $\mathcal{L}_0$ program will not affect the state of the observed host node. However, if we wish to use the language for more than just passive observation, we must add some mechanism for altering program state or operation.

What if the program could write to state variables, in addition to reading from them? By adding this capability to $\mathcal{L}_0$, we change the character of the language from observational to reactionary. We can now specify preconditions and have actions triggered in response to their observation, in effect creating distributed "if-then" clauses. We call this simple reactive programming language $\mathcal{L}_1$.

$$\langle\text{var declaration}\rangle \rightarrow \texttt{scalar}\ \mathit{name}\ (\ \texttt{=}\ (\ \mathit{float} \mid \mathit{name}\ ))!\ \ \texttt{;}$$

$$\langle\text{statement}\rangle \rightarrow \texttt{forall (}\langle\text{slot declarations}\rangle\texttt{) where}\ \langle\text{predicate}\rangle$$

$$\texttt{do}\ \langle\text{action}\rangle\ (\texttt{,}\ \ \langle\text{action}\rangle)^{*}\texttt{;}$$

$$\langle\text{action}\rangle \rightarrow \mathit{name}\ \texttt{.}\ \mathit{name}\ (\texttt{=}|\texttt{+=}|\texttt{-=})\langle\text{scalar expression}\rangle$$

Figure 3.2: EBNF Action Syntax (for language $\mathcal{L}_1$)

3.2.1 Application: Spanning Tree Construction

Spanning trees are a fundamental construct in distributed systems. They are used in multicast (68) and broadcast (58) messaging, in overlay networks (9), and data aggregation algorithms (77). Construction of a basic spanning tree is a special case of the Distance Vector Routing (DVR) protocol used by Internet routers (63). Construction of a basic spanning tree is typically accomplished via the use of breadth-first search. More complex constructions, such as minimum-weight or degree-constrained trees, require more complicated algorithms, particularly in the distributed case (20).

3.2.2 Action Syntax

The syntactic additions to $\mathcal{L}_1$, as summarized in Figure 3.2, have two major components. The first is the addition of initialization values for state variables. This addition allows us to distinguish between managed and pass-through variable semantics, a difference covered in depth in the semantics section following. The second, more substantial, addition is the inclusion of a `do` clause at the end of the statement body. This clause contains one or more actions, to be triggered when the corresponding predicate is detected. Each action is an assignment to a state variable, either directly or through an increment/decrement operator.

3.2.3 Action Semantics

The ability to execute actions requires a much more complicated semantics than that of $\mathcal{L}_0$, as there are many considerations to addressed. The semantics of $\mathcal{L}_1$ begin with the introduction of two types of state variables: *managed* and *pass-through*. Pass-through variables are those that exist in an underlying host program, which the runtime reads and writes from. These variables do not have initialization values, and their value is free to change asynchronously and independently from the operation of the runtime. Managed variables are allocated and controlled by the runtime, and not shared with the host program. These variables are distinguished (and declared) by the presence of an initialization value. The presence of managed variables makes it possible to execute $\mathcal{L}_1$ programs with no underlying host program.

Action execution proceeds as follows: When a statement's main predicate evaluates to true, all actions are executed sequentially, in the order they are listed. Each action has an explicit location, given by the location of the state variable the action writes to. All actions must be located at the same node. If it is the case that the actions are located on a node other than the one on which the deciding evaluation occurs, a message is routed back to that node in order to execute the actions. For example, in the following statement:

Listing 3.4: Remote Actions

```
forall(a,b,c) where (a.var == 1) & (b.var < 2) & (c.var == 3) do a.var = b.var + c.var;
```

the predicate can not be evaluated as true until the detection attempt reaches `c`. Once at `c`, a message is routed back to `a` (via `b`) in order to initiate actions there. This message will also carry `b.var` and `c.var`, so that the right hand side of the assignment can be evaluated. Any state variables needed by the right hand side of an assignment are gathered in the course of evaluating the predicate, so that they can be available if needed and their recorded state is consistent with that used by the predicate. The presence of undefined values on the right-hand side of an assignment is treated in the same manner as undefined values in the predicate.

The presence of remote actions might mean that the detected predicate is no longer valid when the message to initiate actions is received, but as this can be the case with local actions (due to the fact that pass-through variables can change asynchronously), it is not of major concern.

One final item of possible concern is that, as statements can now change state variables, predicates may no longer be strong stable. An example of this would be

Listing 3.5: Non-Stable Predicate

```
forall(a) where a.var == 1 do a.var = 2;
```

where the predicate is true when detected, but rendered false by the action. In this particular case, the action will trigger only once, barring any external changes to `a.var`. Programmers utilizing actions must therefore be aware of how writing to state variables may make other statements (or even the same statement) no longer stable, and ensure that consistent semantics are maintained.

3.2.4 Spanning Tree Creation

We now demonstrate the use of actions by creating a spanning tree in two ways: a randomly-created tree and a tree where each node attempts to minimize its depth. We assume that the distributed system has exactly one distinguished root, indicated by setting the `isRoot` variable to 1. In Listing 3.6, we define two pass-through variables and a single managed variable. The variables `id` and `isRoot` are pass-through, their semantics defined by the underlying host program. The managed variable `parent` tracks the parent of each node in the spanning tree, and is initially set to an invalid sentinel value.

Listing 3.6: Spanning Tree

```
1 const scalar INVALID_ID = -1;
2 scalar isRoot;            // is this node the distinguished root? 1 if so, 0 otherwise
3 scalar id;                                           // unique identifier for the node
4 scalar parent = INVALID_ID;                                    // parent in spanning tree
5 
6 forall(a) where (a.isRoot == 1) & (a.parent == INVALID_ID) do a.parent = a.id;
7 forall(a,b) where (a.parent != INVALID_ID) & (b.parent == INVALID_ID) do b.parent = a.id;
```

The first statement, on line 6 of Listing 3.6, locates the root and sets the parent of the root to the root's id. This serves to both mark the root (as the only node whose parent is equal to its id), and to trigger the subsequent execution of the statement on line 7. This second statement operates over all pairs of modules and identifies pairs where node `a` has joined the spanning tree, but node `b` has not. Process `b` then joins the tree, setting is parent to be `a`. Once this has happened, `b` will not change its parent on subsequent ticks, as it will fail to match the predicate on line 7. If multiple messages from different prospective parents arrive at `b` during the same tick, `b` will select the last one to arrive as its parent.

While this algorithm will construct a correct spanning tree, the choice of parent for each node is dependent solely upon the arrival order of messages from prospective parents, meaning that locally suboptimal choices will be made. We will therefore implement a simple extension to the basic algorithm: a locally depth-minimizing spanning tree. In this algorithm, each node will track not only its parent in the tree, but also its current depth. If a node `b` has a neighbor with a depth less than `b.depth - 1`, then `b` will set its parent to that node, and update its depth accordingly. As the root has a depth of 0, and the depth of all nodes is a monotonically decreasing function, it is easy to show that this algorithm will always create a valid tree, without loops or disconnections (provided that the network topology is connected).

Listing 3.7: Spanning Tree (Depth-Minimizing)

```
const scalar INVALID_ID = -1;
scalar isRoot;            // is this node the distinguished root? 1 if so, 0 otherwise
scalar id;                                           // unique identifier for the node
scalar parent = INVALID_ID;                                    // parent in spanning tree
scalar depth = INT_MAX;                                // current depth in spanning tree

forall(a) where (a.isRoot == 1) & (a.parent == INVALID_ID)
          do a.parent = a.id, a.depth = 0;
forall(a,b) where (a.depth + 1 < b.depth)
            do b.parent = a.id, b.depth = a.depth + 1;
```

Note the use of multiple actions in this implementation. Executing both actions atomically is important, as it ensures that depth and parent information remains synchronized. Additionally, this implementation will continue to create and propagate detection events at each tick, even after reaching a steady state, an inefficiency that we will address in Section 5.2.

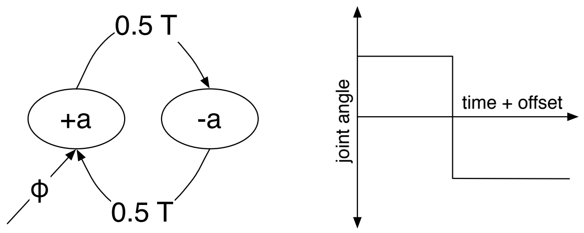


Figure 3.3: a) Snake gait phase automaton, b) Joint angle vs. time graph

3.3 Hybrid Coding

The language $\mathcal{L}_1$ added support for actions that set state variables, but the language is still very limited. Among other considerations, it lacks any means of interfacing with third-party code (a serious deficit in embedded systems), and there is no facility for functions (or any other form of modularity).

To address these deficits, the language $\mathcal{L}_2$ adds support for arbitrary functions, implemented by the host program. These functions may be declared and called directly, or their use may be elided by state variable get/set methods. For the latter case, we redefine the semantics of passthrough variables, so that they provide hooks for getter/setter functions. By adding these capabilities to $\mathcal{L}_2$, we provide a syntactically simple, yet quite powerful, extension semantics to the language. We refer to the use of a low-level language to implement extensions as *hybrid coding*.

3.3.1 Application: Phase Automata

Phase automata (81) are a technique for scalably describing cyclic gaits in chain-style modular robots, such as Polypod (25) and Superbot (70). 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 3.3a. 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 Figure 3.5.

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 the correct value based on the current time and offset.

$$
\begin{aligned}
\langle\text{declaration}\rangle &\rightarrow \langle\text{var declaration}\rangle|\langle\text{const declaration}\rangle|\langle\text{function declaration}\rangle\\
\langle\text{action}\rangle &\rightarrow \textit{name}\ \texttt{.}\ \textit{name}\ (\texttt{=}|\texttt{+=}|\texttt{-=})\langle\text{scalar expression}\rangle\\
&\quad|\ \langle\text{function call}\rangle\\
\langle\text{function declaration}\rangle &\rightarrow \textbf{\texttt{scalar}}\ \textit{name}\,\textbf{\texttt{(}}\ \langle\text{function arg declaration}\rangle!\ \textbf{\texttt{)}}\ \textbf{\texttt{;}}\\
\langle\text{function arg declaration}\rangle &\rightarrow \textbf{\texttt{scalar}}\ \textit{name}\ (\texttt{,}\textbf{\texttt{scalar}}\ \textit{name}\ )^*\\
\langle\text{scalar}\rangle &\rightarrow \textit{float}\\
&\quad|\ \textbf{\texttt{false}}\ |\ \textbf{\texttt{true}}\\
&\quad|\ \textbf{\texttt{n}}\ \textbf{\texttt{(}}\ \textit{name}\ \texttt{,}\ \textit{name}\ \textbf{\texttt{)}}\\
&\quad|\ \langle\text{function call}\rangle\\
&\quad|\ \langle\text{variable ref}\rangle\\
&\quad|\ \textit{name}\\
\langle\text{function call}\rangle &\rightarrow \textit{name}\ \texttt{.}\ \textit{name}\,\textbf{\texttt{(}}\langle\text{function args}\rangle!\textbf{\texttt{)}}\\
\langle\text{function args}\rangle &\rightarrow \langle\text{scalar expression}\rangle(\texttt{,}\ \langle\text{scalar expression}\rangle)^*
\end{aligned}
$$

Figure 3.4: $\mathcal{L}_2$ EBNF Function Syntax

3.3.2 Function Call Syntax

Syntax for supporting function calls begins with function declaration. As with state variables and constants, functions are declared in the preamble of an $\mathcal{L}_2$ program. All functions return a scalar value, and take zero or more scalar arguments. Arguments are named, but these names are for documentary purposes only, and are discarded by the compiler. Variadic functions are not supported. Some examples of valid function declarations:

```
scalar exp(scalar base,scalar e);
```

```
scalar doAction();
```

Functions, once declared, can be used wherever a scalar expression would be appropriate. Explicit calls to functions are prefaced with the location of the call, in a similar manner to state variables. A function call can also be an action, independent of any assignment operation. Function arguments can be any scalar expression, including other function invocations.

3.3.3 Function Call Semantics

The semantics of function calls has three components: declaration, function calls, and elided functions (via passthrough variables). Function declaration is straightforward, registering the function name and arity with the symbol table. Function declarations are deliberately identical to those in C/C++, the language used as a host by our existing LDP implementation. Passthrough variables are now also defined in terms of host program functions. Declaration of a passthrough variable `varName` implicitly declares host functions `varName_set` and `varName_get`, which are used when reading/writing the state variable.

Every function invocation is located at some slot, given in the form `slotName.fcnName()`. This allows the runtime to know when a function should be executed. For purposes of state storage and transfer, an executed function call is considered identical to a state variable reference. For example, in the following predicate fragment:

```
forall(a,b,c) where (b.callFcn(a.callFcn(a.var)) == 1)...
```

The state value `a.var` is evaluated first. This value is then passed to `a.callFcn`, whose return value is stored and transmitted as part of the state vector to `b`, where it is given as a call-by-value argument to `b.callFcn`. As we have just shown, function arguments are evaluated before the function itself, so that nested functions work correctly. Functions with any undefined arguments will return an undefined value *without being called on the host program side*. This may occur if a statement is constructed so that function execution happens before variable access, as is the case with

```
forall(a,b,c) where (b.callFcn(a.var,c.var) == 1)...
```

In this case, the value of the function is `undefined`, as the function call location (`b`) is inaccessible by the time all function arguments are available (at `c`). The function will never be called, which is of concern only when `callFcn` has side effects.

Function calls can appear in actions, either in the right hand side of an assignment, or by themselves. In both cases, these functions are not executed unless the predicate matches, meaning that they must be located at the action location for the statement. Function calls that appear alone as actions have their return value implicitly discarded.

3.3.4 Implementing Phase Automata

With the ability to call arbitrary functions in $\mathcal{L}_2$, we can now implement phase automata, tying the $\mathcal{L}_2$ program to low-level movement routines on each robotic module. We present two implementations; one using an explicit function to actuate the modules, and one using elided function calls via passthrough variables.

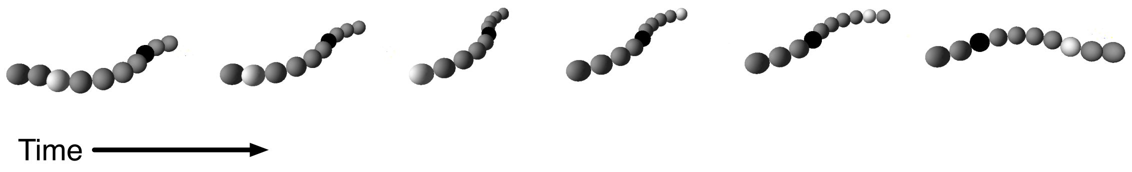


Figure 3.5: Snake gait in chain-style modules. Black modules are actuating negative joint angle, white modules are actuating a positive joint angle.

Both implementations share the same structure. Each module has four state variables: the module id, the parent of the module in the chain, the current time, and the phase offset. We assume that modules are arranged in a single chain, with the front-most module having an id of 1. The first two statements initialize the robots, propagating the parent relationship and phase information to each module. The second two statements perform the actual gait operation, varying the joint angle on each module as a function of time plus offset. The timing of the gait cycle is controlled by the system tick frequency and the constants used in statements 3 and 4. Note the use of the parent state variable to ensure that the module has received a valid phase offset before beginning actuation.

The first implementation (Listing 3.8) uses an explicit function, `setAngle(scalar angle)`, to set the joint angle of the module. This function call is the only action in statements 3 and 4 (lines 12-15), where its return value is ignored.

Listing 3.8: Phase Automata (Function Based)

```
1  scalar id;
2  scalar parent = -1;
3  scalar time;
4  scalar offset = 0;
5
6  scalar setAngle(scalar angle);
7
8  forall (a) where (a.id == 1)
9             do a.parent = a.id;
10 forall (a,b) where (a.parent != -1) & (b.parent == -1)
11              do b.parent = a.id, b.phase = a.phase + 0.1;
12 forall (a) where ((a.time + a.phase) % 1.0 == 0.5) & (a.parent != -1)
13            do a.setAngle(15.0);
14 forall (a) where ((a.time + a.phase) % 1.0 == 0.0) & (a.parent != -1)
15            do a.setAngle(15.0);
```

An alternative implementation is the use of a passthrough variable, `angle` to substitute for an explicit function call to the host code. The runtime cost and semantics of these two approaches are nearly identical, and thus it becomes a question of programmer preference as to which approach should be used. While the explicit function call style clearly demarcates which actions have additional semantics, the use of passthrough variables makes the $\mathcal{L}_2$ code cleaner and easier to read, especially if it were to include numerous function calls.

Listing 3.9: Phase Automata (Hybrid Coding)

```
scalar id;
scalar parent = -1;
scalar time;
scalar offset = 0;
scalar angle = 0;

forall (a) where (a.id == 1)
           do a.parent = a.id;
forall (a,b) where (a.parent != -1) & (b.parent == -1)
             do b.parent = a.id, b.phase = a.phase + 0.1;
forall (a) where ((a.time + a.phase) % 1.0 == 0.5) & (a.parent != -1)
           do a.angle = 15.0;
forall (a) where ((a.time + a.phase) % 1.0 == 0.0) & (a.parent != -1)
           do a.angle = -15.0;
```

3.4 Set Variables

$\mathcal{L}_2$ supports only scalar variables, which limits the data that programs can easily manipulate to a compile-time determined list of singleton state variables. The addition of an aggregate variable type would allow for more complicated programs which manipulate larger amounts of state. For the intermediate language $\mathcal{L}_3$, we have chosen to add support for variables that are sets of scalar values.

There are a number of different aggregates we could have selected: arrays, vectors, dictionaries, etc. Why choose sets? While any aggregate storage type would be useful in $\mathcal{L}_,$ set variables have several interesting properties: they are one of the few aggregates without an indexing operator, they require dynamic storage, and they support a rich range of operations beyond insert and remove. All of these properties pose implementation and design challenges for the language beyond those of simple vectors/arrays.

3.4.1 Application: Data Aggregation

A common task in sensor networks, 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 (Figure 3.6). The naïve implementation

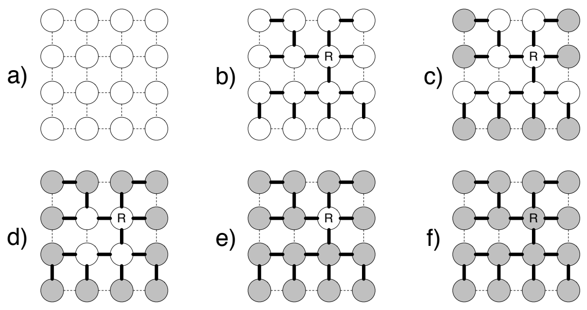


Figure 3.6: Data Aggregation Algorithm: a) Available communications links b) Shaded links show establishment of spanning tree rooted at R c) Leaves propagate data upwards (shaded circles) d-e) Additional levels of propagation f) Data aggregation complete

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 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 (53; 55), but it serves to illustrate how one might implement such a task.

3.4.2 Set Variable Syntax

Syntactically, set variables are a straightforward addition to $\mathcal{L}_3$. We add the capability to declare sets as state variables, constants, function arguments, and function return types. Set constants can be declared, either as `null`, or as a comma-separated list of scalar constants (either explicity or as references to other constants) surrounded by curly-braces. Set expressions are a distinct class from scalar expressions, and we explicitly do not support operator overloading between set and scalar types, with the exception of assignment/insert/remove operations in actions.

Set operations are performed almost exclusively via functions (which are the only way to convert between sets and scalars at runtime). Set operations are declared as standard functions, and we assume that any reasonable $\mathcal{L}_3$ implementation will include such operations as `union()`, `intersect()`, and `size()`.

$$
\begin{aligned}
\langle\text{var declaration}\rangle &\rightarrow \texttt{\textbf{scalar}}\ \textit{name}\ (\ \texttt{\textbf{=}}\ (\ \textit{float} \mid \textit{name}\ ))!\ \ \texttt{\textbf{;}}\\
&\mid \texttt{\textbf{set}}\ \textit{name}\ (\ \texttt{\textbf{=}}\ \langle\text{set constant}\rangle)!\ \ \texttt{\textbf{;}}\\
\langle\text{function declaration}\rangle &\rightarrow (\texttt{\textbf{scalar}} \mid \texttt{\textbf{set}})\ \textit{name}\ \texttt{\textbf{(}}\ \langle\text{function arg declaration}\rangle!\ \texttt{\textbf{)}}\ \ \texttt{\textbf{;}}\\
\langle\text{const declaration}\rangle &\rightarrow \texttt{\textbf{const scalar}}\ \textit{name}\ \ \texttt{\textbf{=}}\ \langle\text{scalar constant}\rangle\texttt{\textbf{;}}\\
&\mid \texttt{\textbf{const set}}\ \textit{name}\ \ \texttt{\textbf{=}}\ \langle\text{set constant}\rangle\ \texttt{\textbf{;}}\\
\langle\text{function arg declaration}\rangle &\rightarrow (\texttt{\textbf{scalar}} \mid \texttt{\textbf{set}})\ \textit{name}\ (\texttt{\textbf{,}}(\texttt{\textbf{scalar}} \mid \texttt{\textbf{set}})\ \textit{name}\ )^{*}\\
\langle\text{function args}\rangle &\rightarrow (\langle\text{scalar expression}\ \rangle|\langle\ \text{set expression}\rangle)\\
&\quad\ (\texttt{\textbf{,}}(\langle\text{scalar expression}\ \rangle|\langle\ \text{set expression}\rangle))^{*}\\
\langle\text{action}\rangle &\rightarrow \textit{name}\ \texttt{\textbf{.}}\ \textit{name}\ (\texttt{\textbf{=}}|\texttt{\textbf{+=}}|\texttt{\textbf{-=}})(\langle\text{scalar expression}\ \rangle|\langle\ \text{set expression}\rangle)\\
&\mid \langle\text{function call}\rangle\\
\langle\text{scalar constant}\rangle &\rightarrow \textit{float} \mid \textit{name}\\
\langle\text{set constant}\rangle &\rightarrow \texttt{\textbf{null}}\\
&\mid \{\langle\text{scalar constant}\rangle(\texttt{\textbf{,}}\ \langle\text{scalar constant}\rangle)^{*}\}\\
\langle\text{set expression}\rangle &\rightarrow \langle\text{set constant}\rangle \mid \langle\text{function call}\rangle \mid \langle\text{variable ref}\rangle
\end{aligned}
$$

Figure 3.7: EBNF Set Variable Syntax (for $\mathcal{L}_3$)

Table 3.2: Variable Assignment Actions

Type		Operator		
LHS	RHS	$=$	$+=$	$-=$
Set	Set	✓		
Scalar	Set			
Set	Scalar		✓	✓
Scalar	Scalar	✓	✓	✓

3.4.3 Set Variable Semantics

The addition of set variables introduces a primitive type system to $\mathcal{L}_3$. As there are now several instances (function returns, constant references, variable references) where the returned type of the expression is variable, and we must use the type information in the symbol tables to ensure type safety. The simple rule used by the $\mathcal{L}_3$ compiler is that sets and scalars are completely disparate entities, and that the type of an expression such as a variable reference or function call is determined explicitly by the declared type (stored in the appropriate symbol table). This rule applies to variable/constant declarations, variable/constant references, function declarations and invocations (both return type and arguments), and actions.

Actions involving assignment, increment, or decrement operators are the only case in the language where potential overloading occurs. Both the right and left-hand sides may be either set or scalar expressions. We summarize the valid combinations in Table 3.2.

In all other respects, set variables are treated in the same manner as scalars. They must be declared in the preamble of the $\mathcal{L}_3$ program. Their storage is allocated and managed by the $\mathcal{L}_3$ runtime (if they are managed variables) or via a function call interface (if they are passthrough). Set variables are transmitted in their entirety when part of a state vector.

3.4.4 Data Aggregation

The code in Listing 3.10 is an implementation of the 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 (lines 12-14) establish a spanning tree rooted at the designated node, as in Section 3.2. The next statement adds all of node `a`'s neighbors who are in the spanning tree but not children of `a` to the set `a.notChildren`. The next statement (line 16) 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 (lines 18-22) 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, requiring merely the tree-based distribution of a reset message, and a corresponding action that resets `sum, count,` and `average` to 0 when that message is received.

Listing 3.10: Aggregation

```
scalar isSeed;                    // is this node the root of the aggregation tree?
scalar id;                                               // unique id for each node
scalar parent = -1;                           // parent's id in the aggregation tree
set notChildren = null;                  // set of neighbors who are not our children
set children = null;                 // set of children that have reported data to us
set neighbors;                                         // set of neighboring nodes
scalar isComplete = 0;                // is the aggregation complete on this node?
scalar sensor;                           // sensor providing the scalar field values
scalar sum = 0;
scalar count = 0;
scalar average = 0;

forall (a) where (a.isSeed == 1) do a.parent = a.id;
forall (a,b) where (a.parent != -1) & (b.parent == -1)  do b.parent = a.id;
forall (a,b) where (a.parent != -1) & (a.parent != b.id)  do b.notChildren += a.id;
forall (a) where (a.size(a.neighbors) == a.size(a.children) + a.size(a.notChildren))
          do a. isComplete = 1;
forall (a,b) where (a. isComplete == 1) & (b.id == a.parent) &
                  (!b.contains(b.children,a.id))
            do b.sum = b.sum + a.sum + a.sensor,
               b.count = a.count + b.count + 1,
               b.children += a.id;
forall (a) where (a.count > 0) do a.average = a.sum / a.count;
```

Note the use of multiple set variables to control aggregation, and determine when all children have reported data to their parent. A node that is a root will have a `notChildren` set the same size as its `neighbor` set, and will thus report immediately to its parent. A node with children will have to wait until the sum of its non-children and reporting children is equal to the number of neighbors it possesses, before reporting to its parent. The more straightforward approach of having a `children` set and a `reportedChildren` set does not work, as there is no way to distinguish between a leaf in the tree, and a node which has not yet received registration requests from any children.

This is not the only way to perform aggregation. If we had instead chosen to support hash tables (dictionaries) instead of set aggregates, we could use a hash table of neighbor names to status values in order to track both child membership and child responses.

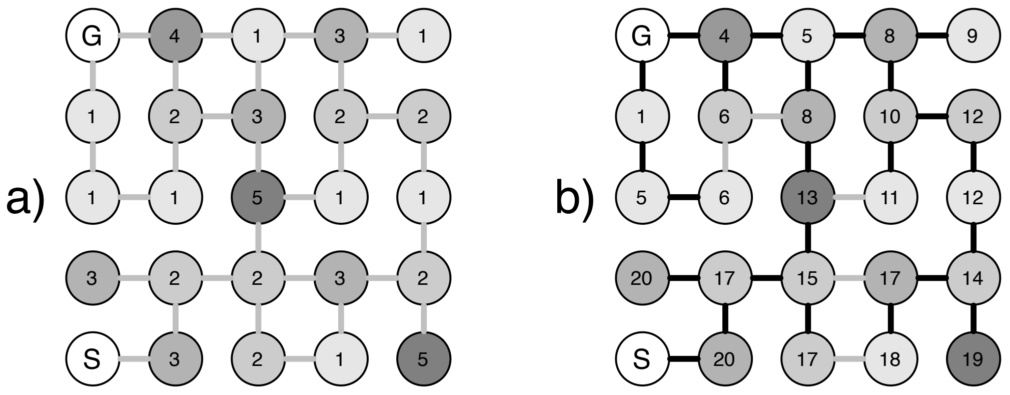


Figure 3.8: Guided Navigation Example: a) Sensor Field (with goal, start and sensor values marked), b) Path integral (with optimal paths highighted)

3.5 Temporal Quantifiers

We note that in $\mathcal{L}_3$, it is difficult to disable statements at their source nodes (to reduce extraneous messaging), as that would require an action on the initiator node, precluding an action on any other node. An example of this would be the continuous messaging behavior seen in the spanning tree implementation presented in Section 3.2. How can we address this issue? One solution is to allow satisfaction of a statement to depend on a transient event, rather than a constant state value.

In order to accomplish this, we add the notion of time to $\mathcal{L}_4$. Temporal operations allow us to reason about transitions in state variables, creating the possibility for edge-triggered (rather than state-triggered) operation. Temporal operations also allow for concise reasoning about sequences of states, a useful ability when encoding distributed state machines or debugging a series of actions.

Processes already trigger LDP search in a discrete manner (via ticks), this provides a timebase for a discrete history of state values. With $\mathcal{L}_4$, we introduce a natural extension to ticks that provides support for temporal quantifiers.

3.5.1 Application: Sensor Field Guidance

Data aggregation and monitoring are the traditional tasks associated with sensor networks, but they are not the only uses for such systems. A more reactive application is that of user guidance: providing a user (either human or machine) with a path that satisfies certain criteria. In cases where the sensor network is monitoring a harmful (or helpful) property, we may wish to provide a path that minimizes danger to the user (in the case of radiation or excess temperature), or maximizes exposure (in the case of sunlight and

$$\langle\text{variable ref}\rangle \rightarrow \textit{name}\ \texttt{.}\ (\langle\text{temporal mod}\rangle\,\texttt{.})!\ \textit{name}$$
$$\langle\text{temporal mod}\rangle \rightarrow (\texttt{\textbf{prev}} \mid \texttt{\textbf{next}})\,\texttt{\textbf{(}}\ \textit{int}\ \texttt{\textbf{)}}$$
$$\mid \texttt{\textbf{prev}} \mid \texttt{\textbf{next}} \mid \texttt{\textbf{curr}}$$

Figure 3.9: $\mathcal{L}_4$ EBNF Temporal Quantifier Syntax

solar-powered robots).

This is the *minimum exposure path* problem, which we solve using a modified version of the algorithm presented by Li et. al. (47). We begin with a sensor network which samples the value of a scalar field (eg. temperature, gas levels, radiation). The user designates a goal sensor, and that node acts as the root for a path-wise integral of the scalar field (see Figure 3.8). The user can then query adjacent nodes in the sensor network, to locate the one with the lowest integral value. The user moves to (or towards) this sensor, and repeats the query. This cycle iterates until the user reaches the goal sensor. The use of a path-based integral ensures that the created path is both globally optimal, and that it contains no loops, disconnections, or local minima.

There are two main goals in the implementation of this algorithm. First, we must ensure that the computer path is accurate, and remains accurate in the face of sensor value changes. Second, the algorithm should minimize communication as much as possible, to reduce battery drain and increase the useful lifespan of the deployed network.

3.5.2 Temporal Quantifier Syntax

The syntax of temporal quantifiers is modeled after a fragment of linear temporal logic (65) with finite extents. Temporal quantifiers provide the ability to request the current value of a state variable, or its value at any specified timestep. Temporal quantifiers are attached to variable references, and must carry a compile-time constant offset. The use of a compile-time value is important, as it allows the compiler to easily and accurately bound the state history required for each variable.

There are three temporal quantifiers: `prev`, `curr`, and `next`. Previous and next quantifiers appear both with an explicit argument (denoting the number of timesteps to offset) and as a no-argument form (with an implicit offset of 1). The previous quantifier specifies an offset into the past, as one would expect. Rather than specifying state from the future, the `next` quantifier implicitly shifts all other variable references into the past. This is useful when reasoning about state sequences in distributed automata. Finally, the `curr` quantifier bypasses variable snapshotting (described below) and directly accesses the value of the state variable.

3.5.3 Temporal Quantifier Semantics

In order to describe the semantics of temporal quantifiers in $\mathcal{L}_4$, we must first address the notion of time. Without access to a global clock, time can only be measured locally, by each individual node. We construct an artificial timebase from the $\mathcal{L}_4$ runtime's ticks, using them as the discrete time intervals for the system. At each tick, we take a snapshot of the current state variables. We assume that the $\mathcal{L}_4$ runtime has the ability to do this atomically, either by pausing any executing host code, or through the use of mutexes/locks. We take the snapshot before any other action in a particular tick. In the case of a passthrough variable, we invoke the variable read function associated with it.

Snapshots are pushed on to the beginning of a vector, where they can be accessed in order (i.e. the snapshot at index 0 is the current snapshot, the one at index 1 is the previous snapshot, etc.). The snapshot vector is constrained in size, with entries at the end being culled once the vector reaches its maximum length. The limit on vector length limits the memory usage of an otherwise potentially memory- and processor-intensive task.

When an $\mathcal{L}_4$ statement accesses a variable, through an unqualified statement like `a.varName`, the $\mathcal{L}_4$ runtime will provide the value of the variable from the current snapshot, rather than the immediate value, as is the case with $\mathcal{L}_0$ through $\mathcal{L}_3$. This provides two main advantages: The first is that all state for a given snapshot is consistent, as there is no question of passthrough variables mutating during runtime execution. The second is that state will remain consistent between multiple statements executed on the same tick (as variable write actions will not be reflected until the next tick). Function call return values are not snapshotted, as function calls may involve state from other nodes, requiring that we snapshot each distinct invocation of the function.

With the notion of time and snapshots in hand, the behavior of the temporal quantifiers is as expected. Unqualified access to a state variable accesses the current snapshot of that variable, while `prev` accesses previous snapshots. The `next` quantifier is convenient shorthand for shifting all other references into the past. For example, the following statement:

```
forall(a,b) where (a.next.varName == a.varName2)...
```

will be rewritten as

```
forall(a,b) where (a.varName == a.prev.varName2)...
```

There are certain cases where the use of snapshots might produce inconsistencies or data races, as when two statements utilize a particular variable both in their predicate and as part of a variable-write action. One example of this is the depth-minimizing spanning tree construction mechanism presented in Listing 3.2. If a node receives two new parent messages in the same tick, it will select the new parent whose message arrived later, rather than the parent whose depth is lowest (as the comparison between local depth and new parent depth will use the snapshotted value).To prevent these data races, we include support for the `curr` temporal quantifier, which bypasses the snapshot mechanism and returns the immediate value of the variable. This temporal quantifier can be used both in predicates and in actions.

3.5.4 Alternative Semantics

Many of the choices we made with regard to the semantics of temporal quantifiers are, at some level, matters of choice. There are a number of alternative temporal quantifier semantics that provide equally valid operation. We will briefly discuss some of them.

As an alternative to taking a snapshot of the entire node at each tick, we can snapshot each variable independently, as the value changes. This is easily done in the case of managed variables, but would require support from the host code to notify the $\mathcal{L}_4$ runtime that a passthrough variable had been modified asynchronously. This semantics would allow for a more intuitive notion of previous variable values (as "previousness" would be defined by per-variable transitions), but would reintroduce the issue of inconsistencies between variables.

Instead of snapshotting at each tick, we could consider snapshotting at some defined rate, independent of ticks. This would allow the user to control the snapshot frequency independently of the execution frequency, for cases when state history must be more or less fine-grained than computation.

Finally, we could initiate a complete snapshot at every variable change. This would provide the more natural semantics of variable history, while preventing cross-variable inconsistency. The overhead of snapshotting this frequently might however be too high.

3.5.5 Sensor Field Guidance

The operation of our sensor field guidance implementation proceeds as follows: We begin at the target sensor, which sets its pathwise integral (the variable `sum`) to be the value from its sensor, and its `parent` to its own id value. Initial formation of the tree proceeds as in previous spanning tree examples, with every node attempting to find a parent that minimizes its pathwise integral.

Once the initial tree has been formed, we use the `prev` quantifier to detect when the sensor or sum on a sensor has changed. In the case of the former, we update the sum to reflect the new sensor reading (lines 13-15). This triggers the final statement (lines 16-18), which invalidates the sum on child sensors (and transitively, on all descendants), leading to a recreation of the tree via statement 2.

Listing 3.11: Field Guidance

```
scalar isTarget;
scalar parent = INV_ID;
scalar id;
scalar sensor;
scalar sum = INT_MAX;

// initialize routing tree
forall (a) where (a.isTarget == 1) & (a.sum == INT_MAX)
           do a.sum = a.sensor, a.parent = a.id;
// initial tree formation
forall (a,b) where ((a.parent == INV_ID) | (a.sum > b.sum + a.sensor)) & (b.sum != INT_MAX)
             do a.parent = b.id, a.sum = a.sensor + b.sum;
// change our personal sum when sensor changes
forall(a) where (a.prev.sensor != a.sensor) & (a.sum != INT_MAX)
          do a.sum = a.sum - a.prev.sensor + a.sensor;
// if our sum has changed, invalidate children
forall (a,b) where (a.sum != a.prev.sum) & (b.parent == a.id)
             do b.parent = INV_ID, b.sum = INT_MAX;
```

3.6 Variably-Sized Predicates

Our language is currently limited to expressing statements over fixed-sized groups of nodes. If we wish to detect conditions that ranges over groups of varying sizes, we must either write multiple statements (as in the case of deadlock detection in Listing 3.3), or write a complicated predicate that expresses the condition in terms of a maximum size, but contains subpredicates for each discrete size we might encounter.

While we do not wish to expand the language to include unbounded groups of nodes, the addition of *quantified expansion* to $\mathcal{L}_5$ is a great convenience to programmers. Though the addition is purely syntactic, quantified expansion is both a useful feature, as well as being an excellent example of how to perform statement transformations in an $\mathcal{L}$ compiler.

3.6.1 Application: Metamodule Planning

As a motivating example, we explore the problem of distributed shape planning for the ensemble. We use the shape change algorithm described in (5). 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 Figure 3.10.

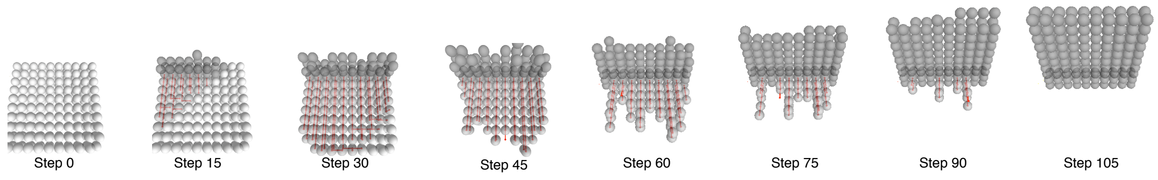


Figure 3.10: Metamodule-based Shape Planner

3.6.2 The Planning Algorithm

We define the abstract model for the robotic ensemble as a collection of states on a compact workspace $\mathcal{W}$ embedded in a lattice in $\mathbb{R}^k$, where each state U is a labeling function of the aggregate using an alphabet of labels $\mathcal{A}$, *i.e.*, $U : \mathcal{W} \rightarrow \mathcal{A}$. For example, the alphabet could be composed of robotic modules R and empty space E. Two adjacent modules would be expressed as RR, while a module next to an empty space would be RE.

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

$$\mathsf{RE} \Leftrightarrow \mathsf{RR} \tag{3.1}$$

This rule states that any module in the abstract model has the ability to create or destroy its neighbor. Dewey and Srinivasa(23) 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. This allows us to perform planning in this abstract model, while guaranteeing that the resulting plan is executable on any well-behaved modular robotic ensemble.

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

The plan starts with a seed module labeled F(for final), 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, ensuring that every connected module in the intersecting region has label F. The algorithm then marks every non-final neighbor in the start state ($\check{\mathsf{N}}$) as P, a candidate for removal. 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 modules to recruit, at which point, the leaves can be trimmed without loss of connectivity. In

Figure 3.10, 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 expressed with the following transformation rules:

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

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

3.6.3 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 Equation 3.1 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

$$\text{num}(\mathsf{C} + \mathsf{E}) = \text{num}(\mathsf{D} + \mathsf{D}) \tag{3.2}$$

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, as the required labels will eventually be available at every creation or deletion site.

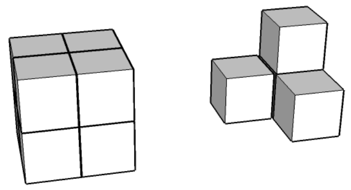


Figure 3.11: Metamodule shape planner: Full and empty 2x2x2 metamodules.

3.6.4 Micro-Plans for Module Movement

While the high-level planner considers such actions as delete, create, and resource transfer to be atomic actions, the reality is more complex. The metamodule planner is instantiated over some concrete metamodule design, representing a predetermined number and configuration of robotic modules. In this implementation, we use a simple 2x2x2 metamodule, shown in figure 3.11 in both its full (resource-carrying) and empty configurations. In addition to a metamodule design, we must also implement a series of micro-plans: predefined movement sequences that implement the high-level create,delete, and transfer operations. When the planner requests such an operation, these micro-plans are distributed to the constituent modules in each involved metamodule, which then execute the necessary movements for the action.

3.6.5 Variably-Sized Predicate Syntax

Syntactic changes in $\mathcal{L}_5$ are limited to the declaration and usage of variably-sized slots. Slots may be declared with a size larger than one, either with a constant size (as in `forall(a[3])`) or with a size range (as in `forall(a[3,5])`). Declared sizes must be non-negative integers, and the second size in a variably-sized slot declaration must be greater than or equal to the first.

Concurrent with the new slot declaration syntax is new syntax for referring to slots. All uses of bare names to refer to slots are replaced with the $\langle$slot name$\rangle$ construct. This allows for the inclusion of an optional index expression, which can be an integer, an index variable, a scalar constant, or some additive/subtractive expression using those components. Index variables are implicitly defined in each statement where they are used. Index expressions that use variables are legal in all variable and function references, with the exception of actions, where they may not be used in stand-alone function calls, or in the left hand size of a variable assignment. These restrictions prevent the occurrence of actions on multiple nodes in the same statement.

3.6.6 Variably-Sized Predicate Semantics

As we previously stated, variably-sized statements are, at their root, no more expressive than their fixed size equivalents. The equivalent fixed size statements for a given variably-sized one are significantly more

$$
\begin{aligned}
\langle\text{slot declarations}\rangle &\rightarrow \langle\text{slot declaration}\rangle(\texttt{,}\ \ \langle\text{slot declaration}\rangle)^{*}\\
\langle\text{index exp}\rangle &\rightarrow (\textit{name} \mid \textit{int})((\texttt{+} \mid \texttt{-})(\textit{name} \mid \textit{int}))^{*}\\
\langle\text{slot declaration}\rangle &\rightarrow \textit{name}\ (\texttt{[}\textit{int}\ (\texttt{,}\textit{int})!\texttt{]})!\\
\langle\text{scalar}\rangle &\rightarrow \textit{float}\\
&\mid\ \texttt{false} \mid \texttt{true}\\
&\mid\ \texttt{n (}\langle\text{slot name}\rangle\texttt{,}\langle\text{slot name}\rangle\texttt{)}\\
&\mid \langle\text{function call}\rangle\\
&\mid \langle\text{variable ref}\rangle\\
&\mid \textit{name}\\
\langle\text{variable ref}\rangle &\rightarrow \langle\text{slot name}\rangle\ \texttt{.}\ (\langle\text{temporal mod}\rangle\texttt{.})!\ \textit{name}\\
\langle\text{slot name}\rangle &\rightarrow \textit{name}\ (\texttt{[}\langle\text{index expression}\rangle\texttt{]})!\\
\langle\text{function call}\rangle &\rightarrow \langle\text{slot name}\rangle\ \texttt{.}\ \textit{name}\,\texttt{(}\langle\text{function args}\rangle!\texttt{)}\\
\langle\text{action}\rangle &\rightarrow \langle\text{slot name}\rangle\ \texttt{.}\ \textit{name}\ (\texttt{=}\mid\texttt{+=}\mid\texttt{-=})(\langle\text{scalar expression}\ \rangle\mid\langle\ \text{set expression}\rangle)\\
&\mid \langle\text{function call}\rangle
\end{aligned}
$$

Figure 3.12: $\mathcal{L}_5$ EBNF Syntax Modifications (Variably-Sized Statements)

complex, as we will see by examining the algorithm that generates such statements.

Variably-sized statements are converted to one or more fixed size statements through the expansion of variadic terms. There are two locations where variadic terms may occur: in slot declaration, and in variable/-function locations. When declaring slots, the user may specify a fixed size (such as `forall(a[2])...`) or a variable size (as in `forall(a[0,3])...`) for the slot. Specifying a fixed size is equivalent to specifying some number of size-1 slots, so `forall(a[2])...` would expand to `forall(a_0,a_1)....` A variably-sized slot expands into multiple statements, one for each possible size of the slot. If there are multiple variadic slots, then the total number of expansions is equal to the product of their valid ranges. As an example, the statement beginning `forall(a[0,2],b[1,2])...` would have 6 valid expansions, with 0-2 `a`'s and 1-2 `b`'s. Note that there must always be at least one slot in a valid statement.

For each of these expanded statements, we must then expand the variadic function and variable expressions within it. Each expression has some index expression associated with it. If this index expression is a constant, we can simply rewrite the expression to point to the correct expanded slot. In the case of negative constants, we use the size of the variadic slot, and work backwards. Thus the statement

```
forall(a[3]) where (a[0].varName > a[-1].varName)...
```

would be converted to

```
forall(a_0,a_1,a_2) where (a_0.varName > a_2.varName)...
```

All other index expressions must include exactly one index variable. Index variables are implicitly defined, and serve to express a predicate ranging over multiple slots. Note that each index variable creates a different range, and so the statement

```
forall(a[3]) where (a[i].varName > a[i].varName2)...
```

ranges over pairs of variables on the same node (of which there are three), while

```
forall(a[3]) where (a[i].varName > a[k].varName2)...
```

ranges over all pairs of variables on three nodes (of which there are nine).

Index expressions can also include additive or subtractive expressions involving integers or integer constants. These expressions constrain the index variable to range over only valid non-negative indexes for the given slot. As an example, the statement `forall(a[5]) where (a[i+1] > 0)...` would constrain `i` to range from 0 to 3 (and thus the index would range from 1 to 4). When the same index variable is used in multiple different index expressions, the valid range of the index variable is the intersection of the valid ranges from all uses of the variable. With this semantics, we can express relationships between elements in chains of varying length. For example, the statement

```
forall(a[2,10]) where (a[i].size < a[i+1].size)...
```

states that, in a chain of length 2 through 10, the size of each node must be strictly less than that of the node that immediately follows it.

In order to perform the expansion of an index expression, we first expand the statement's slots. This gives us a statement with a constant number of slots. For each index expression in that statement, we find the *deepest enclosing boolean expression* for that index expression. As we do not have formal boolean types, we assume that any input to a boolean operator is itself boolean, as is the top level of the predicate. The deepest enclosing boolean expression may be shared by multiple index expressions, and there may be several such booleans in any given statement.

For each enclosing boolean expression, we find every index expression contained within it. For each contained index expression, we determine the valid span of values for the index variable. We then take the intersection of all uses of each individual index variable, to get the valid range. We will now have a valid range for each index variable enclosed by the boolean. These valid ranges define the valid set of expansions through their multiplicative union. We expand the boolean by creating a copy of the expression for each expansion, and joining them all under the boolean and operator.

To clarify this process, let us examine two examples. Take the statement

```
forall(a[1,2]) where (a[i].var > 0)...
```

This statement is first expanded into two statements

```
forall(a_0) where (a[i].var > 0)...

forall(a_0,a_1) where (a[i].var > 0)...
```

We then find the innermost enclosing boolean for the index expression (the greater-than operator in this case), and apply the expression-level expansion operation:

```
forall(a_0) where (a_0.var > 0)...

forall(a_0,a_1) where ((a_0.var > 0) & (a_1.var > 0))...
```

Now let us examine the use of multiple index variables. Take the statement

```
forall(a[2]) where (a[i].var == a[k].var) & (a[i].var2 != a[i+1].var2)...
```

There are two enclosing boolean expressions in this statement (denoted by the equals and not-equals operators). The first expression expands over two index expressions, giving 4 different expansions. When joined, these expressions become

```
(a_0.var == a_0.var) & (a_0.var == a_1.var)

& (a_1.var == a_0.var) & (a_1.var == a_1.var)
```

The second enclosing boolean expression (not-equals) has a restricted range of `i`, due to the additive expression. There is thus only one expansion: `a_0.var2 != a_1.var2`.

3.6.7 Metamodule Planning

The metamodule shape change algorithm is quite complex in its implementation, and we defer presentation of the entire source code for Appendix B. Instead, we examine the use of variably-sized statements in a key area of the planner: distributing and coordinating micro-plans. As discussed in Section 3.6.4, once the high-level planner makes a decision to start a particular operation (create, delete, or resource transfer) on one or two metamodules, the component modules in the metamodule must execute a micro-plan in order to perform the high-level operation.

Within each metamodule, one module is selected as a leader. This leader module distributes the micro-plan to each slave module, and waits for their confirmation that the micro-plan has completed, before continuing with the high-level planner. In the 2x2x2 module configuration we have selected, slave modules may be as far as 5 hops away from their leader (in the case of metamodule creation), or as close as 1 hop. Thus, in order to distribute or confirm micro-plans, the leader would need to execute up to 5 differently-sized statements.

Listing 3.12: Metamodule Shape Planner (Modified Snippet)

```
// variables and functions declared in full source code

// ---------------------------- Low Level move ------
// propagate action to children
forall (a,n[0,2],b) where (a.role == 0) & (a.llStepNum == -1) &
                          (b.llParent == a.id) & (n[i].llParent == a.id)
                    do b.llAction = a.llAction, b.llActionDir = a.llActionDir;

// start action on children. result: step set to 0, moveDir set appropriately
forall (a) where (a.llAction != 0) & (a.llStepNum == -1) do a.setUpAction();

// if move succeeds, step incremented and moveDir is set appropriately
// (this will keep looping until a's motion is done)
forall (a) where (a.llMoveDir != 0) & (a.llStepNum != -1) do a.tryToMove(a.llMoveDir);

// ------------------------------- Move Complete Notification ------
// move is done, inform parent
forall (a) where (a.llMoveDir == 0) & (a.llStepNum != -1) & (a.role == 0)
           do a.childDone += a.id;
forall (b,n[0,4],a) where (b.llMoveDir == 0) & (b.llStepNum != -1) &
                          (b.role != 0) & (b.llParent == a.id)
                    do a.childDone += b.id;

```

```
24 // unlock and add resource to other party (for delete)
25 forall (a) where (a.role == 0) & (a.isPassive == 0) & (a.llAction == DELETE) &
26                  (a.size(a.llChildrenDone) == 4)
27            do a.setNewRole(a.llNextParent);
28 forall (a,b[1,3],c) where (a.role == 0)  & (a.llAction == DELETE) &
29                  (c.curr.llOperationLocked == a.id) & (a.llOperationLocked == c.id)
30             do c.rsc = 1, c.llOperationLocked = 0, c.llProposedOp = NOP;
```

As shown in the modified code snippet above, the use of variably-sized predicates greatly simplifies the code required for micro-plans. The ability to declare variable numbers of modules is used wherever there are two modules of interest, separated from each other by other modules, whose number varies within known bounds. Note the use of an index expression (line 6) to ensure that all intermediate modules are inside the metamodule, which prevents extraneous messages from propagating more than 1 hop from the metamodule.

3.7 Complete $\mathcal{L}$ Syntax

We have now introduced all of the core language features that make up the language $\mathcal{L}$. With this language, we can concisely and unambiguously implement a large number of distributed algorithms. The full syntax of $\mathcal{L}$ is shown in Figure 3.13, and consists of the following important features:

- state variables (declaration, storage, read, write)
- boolean, mathematical, and comparative operators
- action support (variable read/write, function calls)
- arbitrary function support (including elided function via passthrough variables)
- history support (snapshots, temporal quantifiers)
- variably-sized statements

We will now use these features together, to implement two algorithms from current distributed systems research: scaffold-based motion planning and synopsis diffusion.

3.7.1 Application: Scaffold-Based Motion Planning

The first algorithm that we will implement is the scaffold-based self-repair and self-reconfiguration algorithm developed by Støy and Nagpal (75). The algorithm begins with a lattice-based modular robot, whose modules are in an arbitrary configuration, and maintains an internal scaffold (Figure 3.14) to allow for unrestricted module movement within the shape.

```
                 ⟨program⟩ → ⟨declaration⟩* ⟨statement⟩+
             ⟨declaration⟩ → ⟨var declaration⟩ | ⟨const declaration⟩ | ⟨function declaration⟩
               ⟨statement⟩ → forall (⟨slot declarations⟩) where ⟨predicate⟩
                               do ⟨action⟩ (, ⟨action⟩)* ;

         ⟨var declaration⟩ → scalar name ( = ( float | name ))! ;
                             | set name ( = ⟨set constant⟩)! ;
       ⟨const declaration⟩ → const scalar name = (float | name);
                             | const set name = ⟨set constant⟩ ;
    ⟨function declaration⟩ → (scalar | set) name( ⟨function arg declaration⟩! ) ;
⟨function arg declaration⟩ → (scalar | set) name (,(scalar | set) name )*

       ⟨slot declarations⟩ → ⟨slot declaration⟩(, ⟨slot declaration⟩)*
        ⟨slot declaration⟩ → name ([int (,int)!])!
               ⟨predicate⟩ → ⟨scalar expression⟩
            ⟨set constant⟩ → null
                             | {⟨scalar constant⟩(,⟨scalar constant⟩)*}

       ⟨scalar expression⟩ → ⟨logical expression⟩
      ⟨logical expression⟩ → ⟨equality expression⟩((&|||^)⟨equality expression⟩)*
     ⟨equality expression⟩ → ⟨comparison expression⟩((==|!=)⟨comparison expression⟩)*
   ⟨comparison expression⟩ → ⟨add expression⟩((>|<|>=|<=)⟨add expression⟩)*
          ⟨add expression⟩ → ⟨mult expression⟩((+|-)⟨mult expression⟩)*
         ⟨mult expression⟩ → ⟨expression atom⟩((*|%|/)⟨expression atom⟩)*
         ⟨expression atom⟩ → ⟨scalar⟩
                             | (⟨scalar expression⟩)
                             | (+|-|!)⟨scalar expression⟩
                  ⟨scalar⟩ → float
                             | false | true
                             | n (⟨slot name⟩,⟨slot name⟩)
                             | ⟨no one will ever read this⟩
                             | ⟨function call⟩
                             | ⟨variable ref⟩
                             | name
            ⟨variable ref⟩ → ⟨slot name⟩ . (⟨temporal mod⟩ .)! name
               ⟨slot name⟩ → name ([⟨index expression⟩])!
           ⟨function call⟩ → ⟨slot name⟩ . name(⟨function args⟩!)
                  ⟨action⟩ → ⟨slot name⟩ . name (=|+=|-=)(⟨scalar expression ⟩|⟨ set expression⟩)
                             | ⟨function call⟩
            ⟨temporal mod⟩ → (prev | next)( int )
                             | prev | next | curr
          ⟨set expression⟩ → ⟨set constant⟩ | ⟨function call⟩ | ⟨variable ref⟩
               ⟨index exp⟩ → (name | int)((+ | -)(name | int))*
           ⟨function args⟩ → (⟨scalar expression ⟩|⟨ set expression⟩)
                               (,(⟨scalar expression ⟩|⟨ set expression⟩))*
```

Figure 3.13: Complete EBNF Syntax

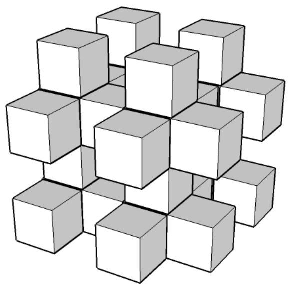


Figure 3.14: Scaffold-Based Planner Subunit

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 (74).

Listing 3.13: Scaffold Planner

```
// state types
const scalar FINAL = 0;
const scalar MOBILE = 1;

const scalar GRAD_MAX = 2048; // max gradient extent

scalar id;
set neighbors;
scalar isSeed; //is the module the seed for the shape?
scalar state = MOBILE;
set freeSpaces; //ids for neighboring lattice pts. that are empty and part of final shape
scalar gradient = 0.0;
scalar inside; //is module inside target shape's scaffold?
scalar parent = INV_ID;
set notChildren = null;
scalar gradientX = 0.0; scalar gradientY = 0.0; scalar gradientZ = 0.0;
scalar gradOrigin = 0;

scalar extendGradient(scalar origin); // extend gradient,gradX,gradY,gradZ towards origin
scalar moveTowards(scalar target); // move in an unblocked direction towards target

// seed module is inside the target shape
forall (a) where (a.isSeed == 1) do a.state = FINAL;
// propagate FINAL to all modules in target shape
forall (a,b) where (a.state == FINAL) & (b.inside == 1) do b.state = FINAL;

// if module is FINAL, and has open space to fill, propagate gradient starting here
forall (a) where (a.state == FINAL) & (a.size(a.freeSpaces) > 0)
           do a.gradient = GRAD_MAX, a.parent = a.id, a.notChildren = null,
              a.gradOrigin = a.any(a.freeSpaces),
31               a.gradientX = 0, a.gradientY = 0, a.gradientZ = 0,
32               a.extendGradient(a.gradOrigin);
33 
34 // gradient propagates w/ subtractive decay
35 forall (a,b) where (a.gradient - 1 > b.gradient) &
36                    (a.gradOrigin != b.id) & (a.parent != b.id)
37              do b.gradient = a.gradient - 1, b.parent = a.id,
38                 b.notChildren = null, b.gradOrigin = a.gradOrigin,
39                 b.gradientX = a.gradientX, b.gradientY = a.gradientY,
40                 b.gradientZ = a.gradientZ, b.extendGradient(a.id);
41 
42 // track the set of modules that are not our children in the gradient
43 forall (a,b) where (a.gradient > 0) & (((a.parent != b.id) &
44                    (a.gradient >= b.gradient -1)) | (a.state == FINAL)) &
45                    (b.gradient > 0)
46              do b.notChildren += a.id;
47 forall (a,b) where (a.gradient > 0) & (a.state != FINAL) & (a.parent == b.id)
48              do b.notChildren -= a.id;
49 
50 // if there are are no children to disconnect, the module can move
51 forall (a) where (a.state == MOBILE) & (a.size(a.curr.neighbors) <=
52                  a.size(a.intersect(a.curr.notChildren, a.curr.neighbors))) &
53                  (a.curr.gradOrigin != -1)
54            do a.moveTowards(a.curr.gradOrigin);
55 
56 // once a module has reached the gradient source, delete the gradient
57 forall (a,b) where (a.state == FINAL) & (b.curr.gradOrigin == a.id)
58              do b.gradient = 0, b.parent = -1, b.gradOrigin = -1;
59 forall (a,b) where (a.gradOrigin != a.prev.gradOrigin) & (b.parent == a.id)
60              do b.gradient = 0, b.parent = -1, b.gradOrigin = -1;
```

Implementation of the scaffold planner algorithm is divided into six component operations. First, in lines 22-25, the modules already within the scaffold of the target shape set their state to `FINAL`, indicating that they will not be mobile during the execution of the algorithm. Any `FINAL` module adjacent to an empty space which should be filled will begin emitting a gradient, originating from that empty space. The module will use the `extendGradient()` helper function, which calculates the direction of the gradient based on the origin module's location and any existing gradient direction.

Once a gradient has been created, it propagates to other modules, updating its value and direction as it progresses (lines 34-40). Each module will attempt to select the gradient with the highest value. Modules that are part of a gradient, and not `FINAL`, will query all of their neighbors, to determine which of them are not their direct children in the gradient (lines 42-48). This is important, as a moving robotic module can disconnect its children from the ensemble.

If a module in the `MOBILE` state has verified that all of its neighbors are not its children (line 52), it is free to move in the direction of the gradient, using the `moveTowards()` function. Finally, once a module reaches its destination, it will terminate the gradient for that location (line 56-58), causing a cascading deletion of the gradient (lines 59-60).

3.7.2 Application: Synopsis Diffusion

The final application we will implement in $\mathcal{L}$ is synopsis diffusion (55). Synopsis diffusion is a sensor-networking technique for obtaining order- and duplicate-insensitive (ODI) aggregates over a scalar sensor field. Such aggregates include mean, maximum, and count. Synopsis diffusion is interesting, as the use of ODI synopses allows for the decoupling of aggregation and routing. To efficiently aggregate information, synopsis diffusion takes advantage of both the inherent broadcast topology of sensor networks, along with the duplicate insensitivity of the synopses. The resulting *rings* topology is both efficient and robust.

From an implementation perspective, synopsis diffusion differs from previous algorithms we have examined, as it implicitly assumes a broadcast-only model of communication. While the initial search phase of $\mathcal{L}$ operation is implicitly broadcast-only, execution of an action on a anything other than the final slot requires hop-by-hop routing to individual nodes in order to trigger actions. By ensuring that all actions occur on the final slot, we can guarantee that all messages in the synopsis diffusion implementation are broadcast-only.

Our implementation presents a generic framework for synopsis diffusion, permitting any aggregate which can be expressed using a set-based synopsis. If this is not the case, there are trivial modifications that can be applied on a per-aggregate basis. Synopses are generated and managed using three helper functions: `SG()`, `SF()`, and `SE()`, which respectively generate, fuse, and extract synopses.

Listing 3.14: Synopsis Diffusion

```
scalar isRoot;  // is this node the distinguished root?
scalar ring = INT_MAX;  // ring level
scalar rxTime;  // 1 if it is the node's tx period, 0 otherwise
scalar txTime;  // 1 if it is the node's rx period, 0 otherwise
set synopsis = null;  // ODI synopsis in progress
set answer = null; // final answer

set SG(); // initialize local synopsis
set SF(set a, set b); // fuse two synopses
set SE(set a); // extract a complete result from a synopsis
scalar initializeTimeIntervals(); // initialize timers for send/receive intervals

forall(a) where (a.isRoot == 1) & (a.ring == INT_MAX) do a.ring = 0;
forall (a,b) where (a.ring + 1 < b.ring)
            do b.ring = a.ring + 1, a.initializeTimeIntervals();
forall (a) where (a.rxTime == 1) & (a.prev.rxTime != 1)
           do a.synopsis = a.SG();
forall (a,b) where (a.txTime == 1) & (a.prev.txTime != 1) &
                   (b.rxTime == 1) & (a.ring = b.ring +1)
            do b.synopsis = b.SF(a.synopsis, b.curr.synopsis);
forall (a) where (a.isRoot == 1) & (a.rxTime == 0) & (a.prev.rxTime != 0)
           do a.answer = a.SE(a.synopsis);
```

The implementation begins with determination of a sensor node's ring level. This construction is similar to that of a depth-minimizing spanning tree, without explicit parent information. When a node sets its ring level, it also calls `initializeTimeIntervals()`, a helper function which manages pre-arranged send and receive intervals for each ring. The definition of this function guarantees that these intervals are non-overlapping, and that reception always precedes transmission.

When it is a node's reception period, it constructs a synopsis and begins waiting for messages. When it receives incoming synopses from higher-ring nodes, it fuses them with the existing synopsis and continues waiting. When the reception time expires, it broadcasts this fused synopsis to all neighbors. This process continues until the root node's reception time ends, at which point it extracts the final answer from the synopsis.

Chapter 4

Implementation & Optimization

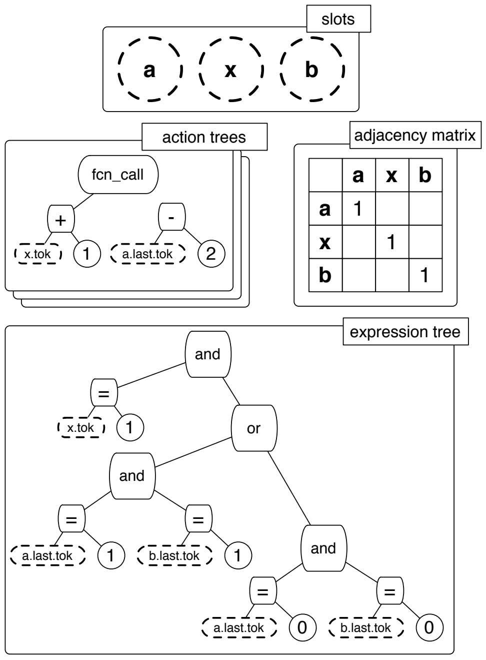
4.1 Naive Runtime

We have implemented the full $\mathcal{L}$ language in the framework of DPRSim (64), a simulator for large modular robotic ensembles. Our initial runtime is simple in concept and execution, but has several glaring inefficiencies. We will present this naive runtime first, as its simple structure allows us to easily explain the key concepts and algorithms used in implementing $\mathcal{L}$ as an interpreted language.

We begin the discussion of the naive runtime by presenting PatternMatchers, the key data structure for distributed predicate search in $\mathcal{L}$. We then present the core predicate search algorithm for detecting matching statement predicates. This centralized algorithm illustrates the basic concepts behind predicate search, but is impractical to implement. We briefly discuss adding the ability to detect certain non-linear ordered subgroups using multihop routing, and then present a distributed algorithm for predicate detection, based on the LDP-Basic algorithm presented in Section 2.4. We then illustrate the operation of the algorithm with an extended example. The section concludes with a brief discussion of the inefficiencies inherent in the naive runtime, which we address in Section 4.2 by presenting the design of an optimized runtime.

4.1.1 Essential Data Structures

The primary data structure used in the initial implementation of $\mathcal{L}$ is the PatternMatcher object (Figure 4.1). Each PatternMatcher represents a distributed search attempt for a particular $\mathcal{L}$ statement. A PatternMatcher of size n consists of four components:



	a	**x**	**b**
a	1		
x		1	
b			1

Figure 4.1: PatternMatcher object. Variables are shown with dotted outlines.

- an ordered and named list of n node id slots (where n is the number of nodes in the statement)
- an $n \times n$ bit adjacency matrix
- an expression tree that both represents the statement predicate and stores any accumulated state variables
- a vector of expression trees, one for each action to be performed

A PatternMatcher may be empty (with none of its slots filled), partially filled (with some slots and state variables filled), or completely filled. A given PatternMatcher may be in one of three states: matched, failed, or indeterminate. The indeterminate state occurs when there is insufficient information in a partially filled PatternMatcher to decide whether its expression is satisfied. Each PatternMatcher is a mobile data structure which represents one particular search to satisfy the statement predicate. A partially filled PatternMatcher is a search which is still in progress, where the empty slots are the incomplete portion of the search. A given node can host many PatternMatchers, representing various searches which are passing through the node, collecting state via the algorithm described below.

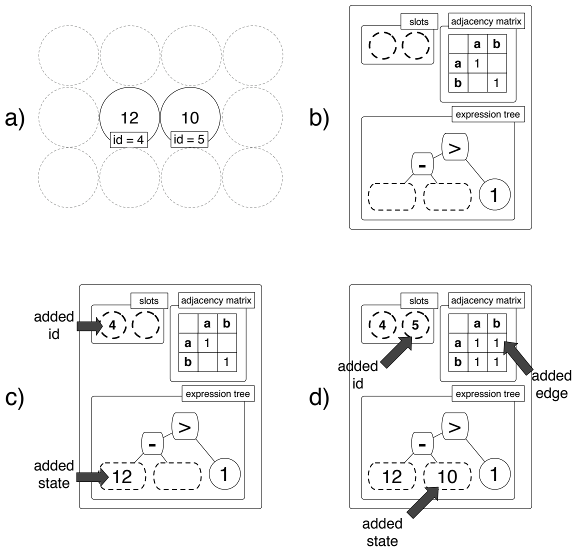


Figure 4.2: The population algorithm, with arrows indicating where data is added. a) System configuration, showing gradient value and node id. b) Initial (empty) PatternMatcher for the statement in Sec. 4.1.2. c) PatternMatcher after being filled by node id 4. d) PatternMatcher after being filled by node id 5.

4.1.2 Algorithms

In order to use PatternMatchers to determine whether statement predicates have been satisfied, we *fill* the components of the PatternMatchers with information from different nodes. This algorithm has three stages that occur at each tick: i) filling the next available node id slot, ii) filling the expression trees, and iii) updating the adjacency matrix. When filling slots, we proceed in order: find the first slot that has not been assigned a node id, then copy the local node's unique id number to that slot. The name of the slot is then used in the second phase, where we traverse the expression trees and instantiate any references to the current slot's state variables or functions with the values from the current node. Finally, we query the node for its current list of neighbors, and update the adjacency matrix with any connections between the current node and any nodes in previously filled slots. The PatternMatcher can then be transmitted to other nodes, where these steps are repeated for every node that the PatternMatcher acquires state information from.

As an example, we illustrate the action of the populating process on the following statement:

```
forall(a, b) where (a.gradient - b.gradient > 1) do b.alert();
```

We examine a single PatternMatcher as it is filled by two nodes in succession. The two nodes have unique id's 4 and 5, and gradient values of 12 and 10 respectively (Fig 4.2a). The PatternMatcher is first filled by node 4. This places the value 4 in slot `a`, and the value 12 in the variable `a.gradient`, as indicated by

```
$S = \emptyset$;
forall modules $m$ do
    create new PatternMatcher $p$ from the statement text;
    fill first open slot of $p$ with $m$;
    $S = S \cup p$;
end
while $S \neq \emptyset$ do
    $T = \emptyset$;
    forall PatternMatchers $p \in S$ do
        if $p$ matches then
            execute actions for $p$;
        else if $p$ is indeterminate then
            forall neighbors $n$ of modules in $p$'s slots do
                $p_1$ = clone($p$) ;
                fill first open slot of $p_1$ with $n$;
                $T = T \cup p_1$;
            end
        else
            // failure:  PatternMatcher is ignored
        end
        $S = S - p$;
    end
    $S = T$;
end
```
Algorithm 3: Centralized PatternMatcher Algorithm

the arrows in (Fig 4.2c). The PatternMatcher is then filled by node 5. This fills the slot `b` and the variable `b.gradient`. Additionally, as `a` is a neighbor of `b`, the relevant entries in the adjacency matrix are updated. The expression tree is now satisfied, and the PatternMatcher has matched the condition described by the statement predicate. The `alert()` function is now called on node `b`.

Centralized Algorithm

Our initial implementation relied on a single centralized procedure to update all PatternMatchers across an entire (simulated) ensemble of robots. A set of vectors maintained each robot's state variables. At each timestep, the current values of all state variables used by active statements were appended to the vectors, providing state history for the variables. The simulator also maintained a single set of PatternMatchers (S), which are updated and processed every timestep as described in Algorithm 3. This centralized algorithm is

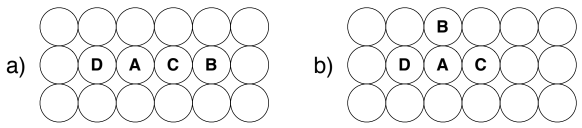


Figure 4.3: Unordered (a) and non-linear (b) subensembles for the set of modules (A B C D).

useful for simulated robot ensembles, but programming actual hardware (or even multithreaded simulations), required the development of a distributed technique for handling $\mathcal{L}$ statements.

Nonlinear Paths

To detect the presence of distributed predicates without a central control entity, we use PatternMatchers as mobile state aggregators. PatternMatchers are transmitted from node to node via single-hop messaging, accumulating state until they either fail or trigger actions. Unfortunately, if we use only single-hop messaging, the subensemble topologies that can be detected are quite limited. Consider the two subensembles in Figure 4.3. Figure 4.3a illustrates a subensemble whose nodes lie in a linear path, but are not ordered sequentially. With only single-hop messaging, there is no way to propagate the slots (A B C D) in order. Figure 4.3b shows a subensemble which has an ordered path, if one is allowed to "backtrack" to previously visited nodes. In order to implement this backtracking (and thus detect this class of subensembles) we introduce an optional multi-hop messaging and rerouting mode.

Multi-hop messaging is used to move a PatternMatcher back through previously visited nodes, so that it can continue to propagate from a different point in the subensemble. PatternMatchers that return to a node that they have already visited in this way are said to have been *rerouted* to the node. Multi-hop messaging and rerouting allow us to detect subensembles of the kind seen in Figure 4.3b, which we term *nonlinear* ensembles, but not to detect those seen in Figure 4.3a, which are *unordered*. To detect unordered subensembles we would have to reorder the slots of the PatternMatcher to correspond to the ordering in the subensemble. To detect all such unordered configurations, we would have to search all permutations of the slots, a potentially expensive proposition (as a size four predicate would have 24 such permutations).

The distributed $\mathcal{L}$ execution algorithm supports non-linear matching as an optional capability, at the cost of some consistency. As we are no longer guaranteed a 1-hop path between successive slots of the statement, we can no longer guarantee that there are no orphan messages found in the sub-cut created by the LDP-Basic algorithm. Faced with this problem, we can either relax the detectability guarantees to those of strongly stable LDPs, or we can employ additional messages to enforce a consistent sub-cut. To do this we can send a numbered beacon message in tandem with the PatternMatcher, that for a k-node statement, propagates to all nodes within radius k of the initiator. The beacon message will cause all nodes to record their current state, and mark that local snapshot with the unique pair of sender id and beacon number. Later,

```
on each node $n$;
create new PatternMatcher $p'$ from the statement text;
$S = S \cup p'$;
forall PatternMatchers $p_1 \in S$ do
    fill first open slot of $p_1$ with $n$;
    if $p_1$ matches then
        execute actions for $p_1$;
    else if $p_1$ is indeterminate then
        message $m_1 = \langle p_1, false, *, *, * \rangle$;
        send $m_1$ to each of $n$'s neighbors;
        forall nodes $d$ in $p_1$'s slots s.t. $d \neq n$ do
            message $m_2 = \langle p_1, true, *, n, d \rangle$ ;
            module $r$ =nextRoutingStep($n$,$d$,$p_1$) ;
            send $m_2$ to $r$ ;
        end
    else
        // failure:  PatternMatcher is ignored
    end
end
forall PatternMatchers$p_2 \in R$ do
    message $m = \langle p_2, false, true, *, * \rangle$;
    send $m$ to each of $n$'s neighbors;
end
$S = \emptyset, R = \emptyset$;
```

Algorithm 4: Distributed Algorithm: PatternMatcher update

when a PatternMatcher arrives at the node, we can use the saved state corresponding to its beacon, rather than the current state of the node. This is a very heavy-weight approach, roughly doubling the number of messages required, and so we opt not to use it, instead accepting the lessened consistency guarantees.

Distributed Algorithm

The distributed implementation of $\mathcal{L}$ involves two components: an update function and a message handler for incoming PatternMatchers. The update function is used at each tick to create, fill, and spread PatternMatchers. Each node maintains three data structures:

- a set S of active PatternMatchers
- a set R of PatternMatchers that have been rerouted to this node

```
message $m$ received by node $n$:
if $m.isMultihop$ then
    if $m.dest = n$ then
        $n.R = n.R \cup m.p$;
    else
        $r$ = nextRoutingStep($n$,$m.dest$,$m.p$);
        $m.source$ = $n$;
        send $m$ to $r$;
    end
else
    if !($m.possibleDup$ & isDuplicate($m, n$)) then
        $n.S = n.S \cup m.p$;
    end
end


isDuplicate(message $m$,node $n$):
  $i_s$ = index of $m.source$ in $m.p$'s slots;
  $i_{max}$ =max index of $n$'s neighbors in $m.p$'s slots;
  return $i_s \geq i_{max}$;

nextRoutingStep(node $s$,node $d$,PatternMatcher $p$):
  calculate BFS over $p$'s adjacency matrix (from $s$);
  $r$ = minimal path from $s$ to $d$ (using BFS distances);
  return the first element of $r$;
```

Algorithm 5: Distributed Algorithm: Message Handler

- a local history for state variables (used by the *fill* routine)

The PatternMatcher update function of the algorithm is executed at each tick, and is composed of five phases. First, a snapshot of local state is taken, and stored in the history variables. In the second phase, a new (empty) PatternMatcher is created for every statement, and added to the set S. Then, for each PatternMatcher in S, its first open slot (and associated expression tree variables/functions) is filled with information from the local node. If a PatternMatcher's expression tree is satisfied, the actions for the statement are executed. If still indeterminate, the PatternMatcher is then spread via two types of messages: local messages sent to each of the node's neighbors, and multihop messages sent to all of the other nodes already identified as being in the PatternMatcher's slots. Every rerouted PatternMatcher in R is then sent via a local message to all of the node's neighbors which are not already in the PatternMatcher's slots. Finally, the sets S and R are cleared.

All messages used in the distributed algorithm carry five data fields:

```
forall(a,b,c) where (a.var = 0) & (b.var = 0) & (c.var = 2) ...
```

Figure 4.4: $\mathcal{L}$ Statement for Distributed Algorithm Example

- the PatternMatcher `p`
- a boolean flag `isMultihop`
- a boolean flag `possibleDup`
- the destination node `dest`
- and the source node `source`

We represent a complete message as <*p, isMultihop, possibleDup, source, dest*>. '$*$' is used to denote field values that we do not care about. Message handling is divided into two subroutines: handling of local messages and handling of multihop messages. Local messages are handled by adding their PatternMatcher p to the local set S of active PatternMatchers.

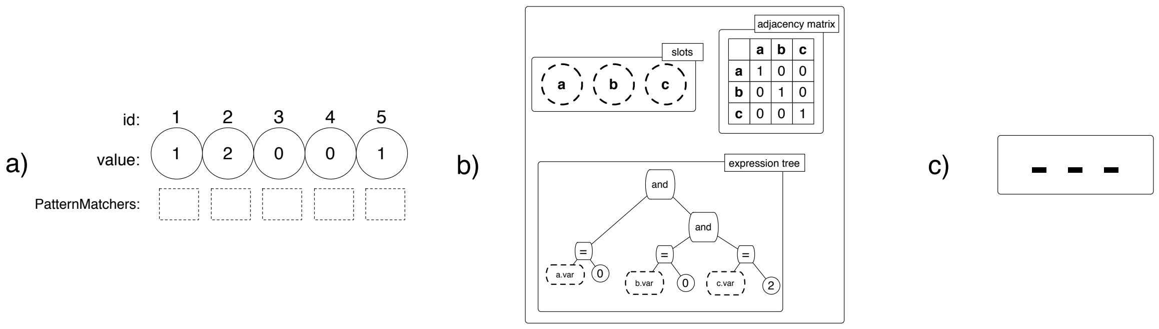
Multihop messages are examined to see if their ultimate destination is the current node. If the message is in transit to another node, a breadth-first search from the current node to the destination is conducted within the adjacency matrix of the message's PatternMatcher. This provides the next step in routing the message to its destination, and the message is forwarded on to this next hop. If the message is meant for this node, its PatternMatcher p is added to the local set R of PatternMatchers rerouted to the node.

Distributed Algorithm: Example

To illustrate how the distributed algorithm functions, we will use the example statement from Figure 4.4, applied to the 5-node ensemble in Figure 4.5a. We note that the two subensembles which satisfy this statement are $[3\ 4\ 2]$ and $[4\ 3\ 2]$. For clarity of explanation, we will assume that all actions at each step occur simultaneously, and that message travel incurs no delay.

The algorithm begins by converting the text of the $\mathcal{L}$ statement into a PatternMatcher (Fig. 4.5b). As the contents of the expression tree and adjacency matrix have been described elsewhere, we abbreviate the PatternMatcher to be the contents of its slots (Fig. 4.5c). Thus, an empty PatternMatcher will be abbreviated $[- - -]$, while one that had been filled by node 3 and then by node 4 would be $[3\ 4\ -]$.

Once an empty PatternMatcher has been created at a node, its first slot is filled by that node (Step a). PatternMatchers which have failed at this point are then discarded (Step b). Likewise, those which succeed activate their respective actions. The remaining PatternMatchers are then spread to the 1-hop neighbors. In this example, the PatternMatcher $[3\ - -]$ at node 3 is spread to nodes 2 and 4 (Step c). All PatternMatchers



	a	b	c
a	1	0	0
b	0	1	0
c	0	0	1

Figure 4.5: Distributed Algorithm Example. a) Process configuration. b) Empty PatternMatcher. c) Abbreviated PatternMatcher.

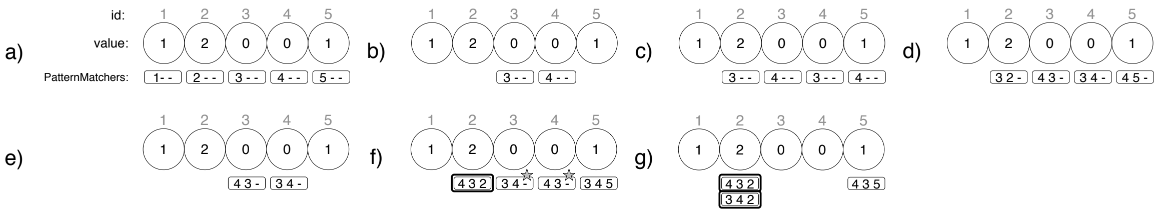


Figure 4.6: Distributed algorithm execution. Rerouted PatternMatchers are indicated with a star, while those that have triggered are marked with a double border.

are then filled by the node that they occupy (Step d), and failed PatternMatchers are again discarded (Step e). As the PatternMatchers have two slots filled, they are now spread both by single-hop messaging (to nodes 2 and 5), and by rerouting (to nodes 3 and 4). Rerouted PatternMatchers are marked with a star, and are not filled by the node they arrive at (Step f). Finally, the rerouted PatternMatchers are spread to the neighbors of the node they occupy (Step g), where they are filled again.

4.1.3 Inefficiencies

The initial $\mathcal{L}$ implementation suffers from a number of inefficiencies, mainly relating to the direct use of PatternMatchers as the primary data structure and message format. To begin with, each node must parse the $\mathcal{L}$ program individually, either at each tick, or at the beginning of program execution. While the inclusion of the parser as part of the runtime adds flexibility (in that it enables run-time modification of the code), the overhead involved is too high for most embedded applications.

Once a PatternMatcher for a given statement has been constructed, we must repeatedly perform two expensive operations on it: deserialization and cloning. Cloning a tree-based structure can be quite expensive in memory-limited environments, as the tree requires the repeated allocation of many small node structures. Serialization and deserialization of the tree is likewise quite expensive.

4.2 Optimized Runtime

Our second implementation of $\mathcal{L}$ is an optimized runtime, designed for use in resource-constrained embedded environments. As a target platform, we consider a TinyOS Mica2 sensor mote (3; 11). These devices have an 8MHz, 8-bit processor, 4k of program memory (SRAM), and 128k of read-only storage (EEPROM). This represents a resource ceiling 3 to 6 orders of magnitude lower than that of a typical desktop computer. It is therefore imperative that we identify and rearchitect the portions of the naive runtime which use significant compute and storage resources.

There are three main considerations when porting the runtime to an embedded platform: runtime memory, processor utilization, and messaging overhead. The Mica2 has only 4k of available program memory, and that memory must be shared between the host OS, any interrupt service routines, space for incoming/outgoing packets, and the $\mathcal{L}$ runtime. In practice, this means that any executing $\mathcal{L}$ program will have less than 2k of available memory, making such luxuries as `malloc` unaffordable. In a similar vein, the use of an 8MHz 8-bit CPU limits not only numeric performance, but imposes a penalty on scattered memory read/write operations due to the need for absolute addressing instructions.

To fit the $\mathcal{L}$ runtime into such a constrained environment, it is obvious that a number of sacrifices must be made. Chief among these is the flexibility of an on-board interpreter. The parser, AST, and interpretation engine of the original runtime are each too heavyweight to run on a sensor node, and together they are far too large. Another luxury that we must abandon is the presence of unlimited-size set variables. Not only do these variables require dynamic storage allocation on the device itself, they can also cause packet sizes to increase dramatically, potentially leading to a need for packet fragmentation. Similarly, we limit the maximum arity of any user-defined functions, to limit the stack space required to process them. Finally, we remove the capability for non-linear matches, introduced in Section 4.1.2. We have found that nonlinear matches are not required by any of the algorithms we implemented, either for sensor networks or modular robots, and thus have removed that capability from the optimized runtime, in the interests of reducing both the number of active packets and the complexity of routing them.

To create the optimized runtime for $\mathcal{L}$, we turn to an *application-specific bytecode engine*. This is a bytecode execution engine, which is specialized at compile-time for the $\mathcal{L}$ program we wish to run. The generic portion of the bytecode engine handles incoming and outgoing packets, manages state history and PatternMatchers, and performs all the common tasks needed by the runtime. Instead of directly creating

PatternMatchers, the front-end (now a separate program) outputs a custom set of headers, which provide compile-time declarations for state variables, function bindings, and the bytecode to be executed. This is important, as it allows the bytecode (representing the PatternMatchers) to be a compile-time constant, which can be stored in the much larger nonvolatile memory of the sensor node, freeing up space for runtime data and allowing for larger $\mathcal{L}$ programs.

Our optimized runtime then consists of 4 components: the formats for bytecodes and packets, the frontend compiler, the generated header files, and the generic bytecode execution engine. In the remainder of this section, we will explore the design of these components, and their effects on overall performance.

4.2.1 Example Compilation and Execution

To illustrate the operation of the optimized runtime, we will first present the compilation and execution of a simple $\mathcal{L}$ program. For our example program, we will use the basic spanning tree creation algorithm presented in Section 3.2. For ease of reference, we reprint the full source listing here.

Listing 4.1: Spanning Tree (Reprise)

```
1  const scalar INVALID_ID = -1;
2  scalar isRoot;              // is this node the distinguished root? 1 if so, 0 otherwise
3  scalar id;                                              // unique identifier for the node
4  scalar parent = INVALID_ID;                                       // parent in spanning tree
5
6  forall(a) where (a.isRoot == 1) & (a.parent == INVALID_ID) do a.parent = a.id;
7  forall(a,b) where (a.parent != INVALID_ID) & (b.parent == INVALID_ID) do b.parent = a.id;
```

In the compilation phase, the $\mathcal{L}$ compiler converts the input source to an application-specific header, which is compiled with both user-supplied code and a generic runtime. Compilation of the program has two parts: creation of variable handlers and creation of the bytecode array. For each managed variable (the scalar `parent` in this example), the generated header contains declared and initialized storage. For each pass-through variable (the scalars `id` and `isRoot`), read/write function names are declared. Implementations of these functions are supplied by the application programmer. Finally, a function dispatch table is created for all variables. It contains pointers to the read/write functions for `id` and `isRoot`, and pointers to generic read/write functions in the case of `parent`.

The generated header also contains an array of bytecode values for each $\mathcal{L}$ statement. We will examine the generated bytecode for the statement in line 6 of Listing 4.1 (Figure 4.7). The bytecode begins with the id number of the statement (0 in this case), followed by the number of slots in the statement, the number of variables referenced in the statement, and the location of any actions. The bytecode preamble ends with the number and location of any actions (1 action, located at index 34). The next 4 lines of the bytecode (indices 6 through 33) are the main statement predicate, consisting of a top-level boolean-AND operator and two comparisons between constants and state variables. Variable references (such are those at indices 21,29, and

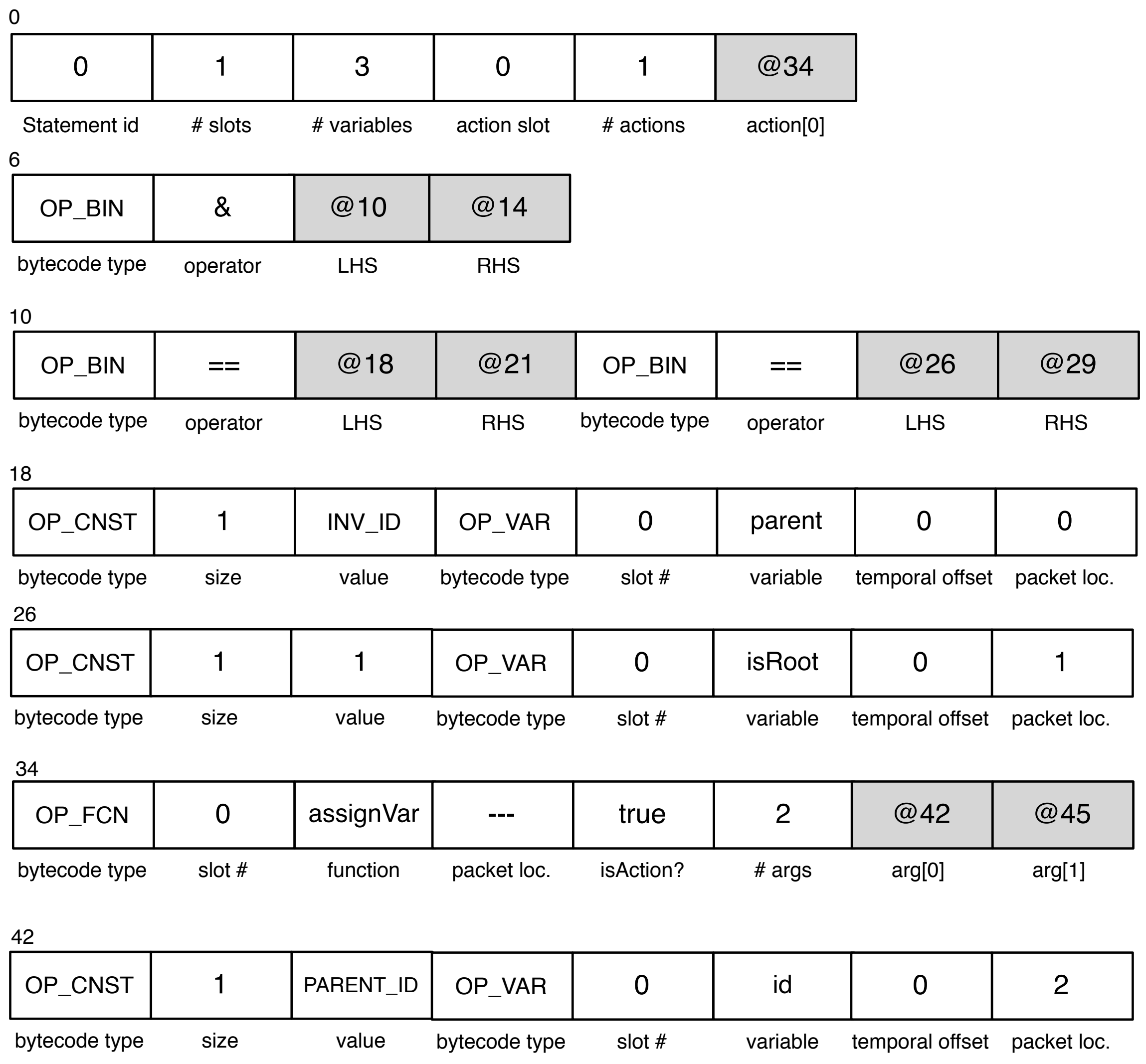

0
0 | 1 | 3 | 0 | 1 | @34
Statement id | # slots | # variables | action slot | # actions | action[0]
6
OP_BIN | & | @10 | @14
bytecode type | operator | LHS | RHS
10
OP_BIN | == | @18 | @21 | OP_BIN | == | @26 | @29
bytecode type | operator | LHS | RHS | bytecode type | operator | LHS | RHS
18
OP_CNST | 1 | INV_ID | OP_VAR | 0 | parent | 0 | 0
bytecode type | size | value | bytecode type | slot # | variable | temporal offset | packet loc.
26
OP_CNST | 1 | 1 | OP_VAR | 0 | isRoot | 0 | 1
bytecode type | size | value | bytecode type | slot # | variable | temporal offset | packet loc.
34
OP_FCN | 0 | assignVar | --- | true | 2 | @42 | @45
bytecode type | slot # | function | packet loc. | isAction? | # args | arg[0] | arg[1]
42
OP_CNST | 1 | PARENT_ID | OP_VAR | 0 | id | 0 | 2
bytecode type | size | value | bytecode type | slot # | variable | temporal offset | packet loc.

Figure 4.7: Example Statement Bytecode.

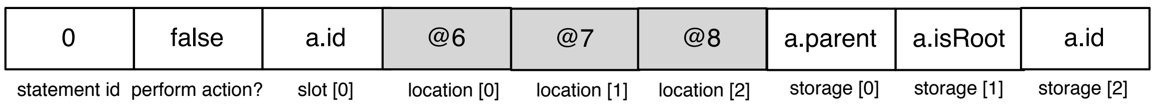

0 | false | a.id | @6 | @7 | @8 | a.parent | a.isRoot | a.id
statement id | perform action? | slot [0] | location [0] | location [1] | location [2] | storage [0] | storage [1] | storage [2]

Figure 4.8: Example Message Format.

45) contain the slot number, variable id, and temporal offset of the relevant variable. Variable references also contain the message storage location for saving and reading state. These storage locations implicitly define a unique packet format for each statement (Figure 4.8). The generated bytecode ends with a function call, the internal `assignVar` function which performs variable assignment. This function takes two arguments: a constant (the id of the variable to be assigned), and a variable reference (the value to be assigned).

Once the application-specific bytecode engine has been compiled with the generated header, the execution phase can begin. At each tick, the runtime will create an empty packet buffer for the first statement. The runtime will then traverse the bytecode trees for the main predicate and any actions, searching for variable references or function calls. In this case, the variable references for `id`, `isRoot`, and `parent` will be discovered, and the corresponding state values will be stored in the packet buffer. The call to `assignVar` will not be executed, as it is an action. Once the *fill* operation has concluded, the statement predicate will be evaluated. If the predicate fails, the buffer is discarded. If the predicate succeeds, then the action (variable assignment) will be triggered. Finally, if the predicate is indeterminate, the packet will be sent to all neighboring nodes. This sequence of operations is repeated for every statement or incoming packet.

4.2.2 Bytecode and Packet Formats

The key component to the bytecode engine is the format of the individual bytecodes, and the associated packet format. These form the representation of the $\mathcal{L}$ program during execution. We begin with the program bytecode format. Program bytecodes are compile-time constant arrays of scalar values, stored in an array ordered by id number. The bytecode for a given statement is constructed in the compiler by traversing the PatternMatcher in infix order, and saving the bytecode to a temporary array. Bytecodes can contain four types of values: scalar constants (either integer or floating point), indices into the bytecode array, and arrays of either scalars or indices. Index values refer to an absolute index within the current statement. Arrays can be of any length, and are prefixed by their size (as a value of type scalar).

There are five different bytecode expressions used by the runtime. The first of these (and the only valid root bytecode) is the statement bytecode. This includes the statement id number, the number of slots in the equivalent PatternMatcher, the number of different variables accessed by the PatternMacher, and the number (and indices) of any actions. This is then followed by the bytecode for the predicate, followed by the bytecode for any actions (in order).

The four remaining bytecode types are: binary expressions, constants, variable references, and function calls (Figure 4.9). During compilation, all numeric, boolean, and comparative expressions are converted to canonical binary expressions. For example, the expression `!a.var` would be converted to `a.var ^ true`. A binary expression bytecode consists of a type constant (B_OP), the operator to apply, and the indices of the left- and right-hand sides. A constant bytecode can represent either a scalar or set constant (as type information is not preserved across compilation). This bytecode consists of a type constant

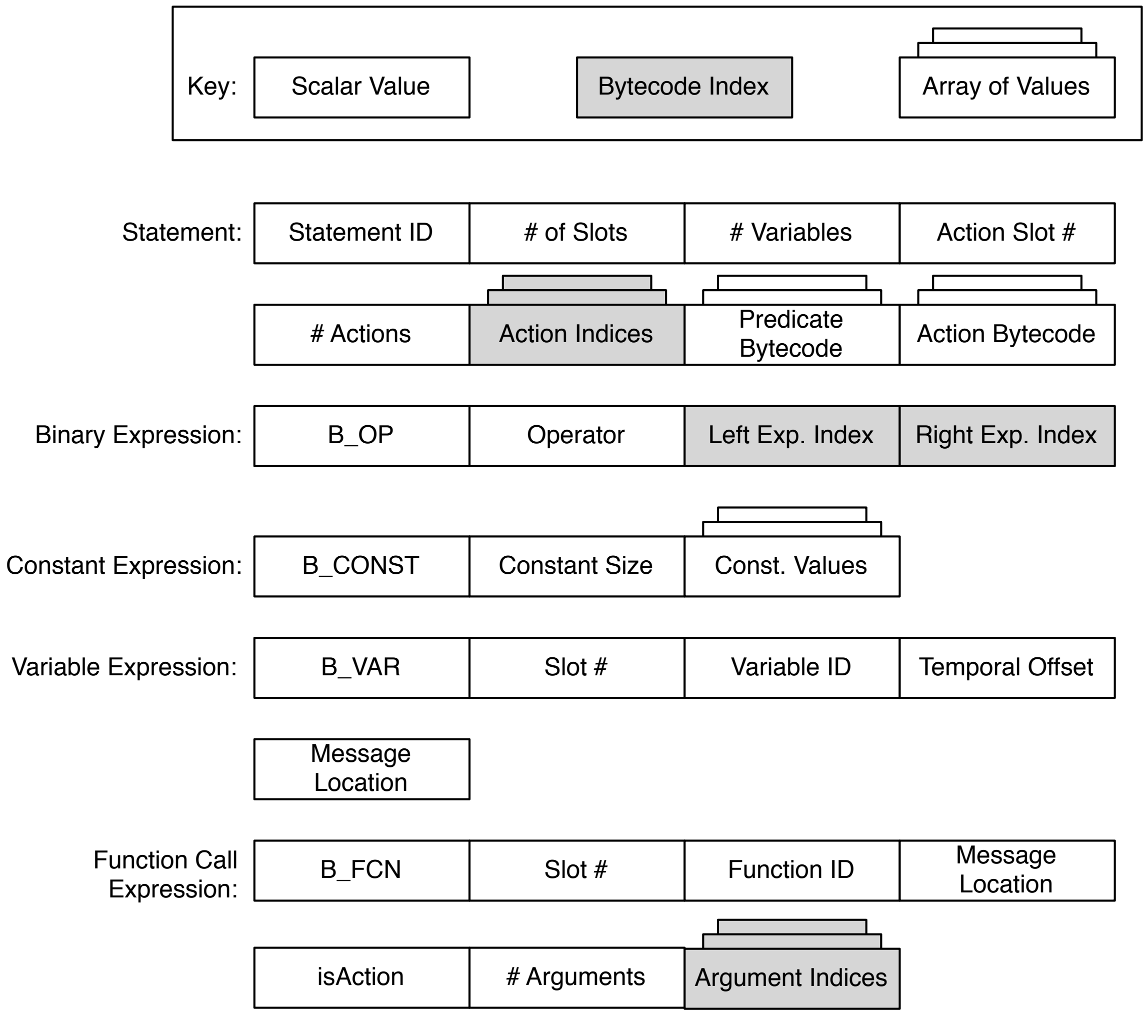


Figure 4.9: Bytecode Format for Optimized Runtime.

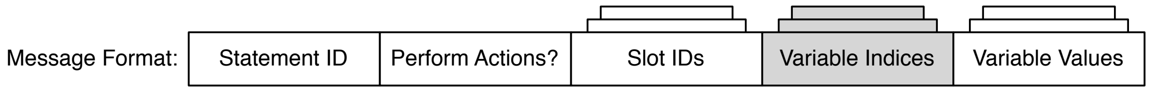


Figure 4.10: Message Format for Optimized Runtime.

(B_CONST), followed by the array of constant values.

A variable reference bytecode provides two types of information: how to read state from a node when filling a packet buffer, and how to later locate that state in the buffer. For the first task, variable references contain the slot number, variable id, and temporal offset of the required state. Once that data is read from the node, it can be stored (and later retrieved) in the message location, which is the last element of the bytecode. For example, the variable reference bytecode for the expression `a.var`, assuming that `a` is the first slot, and `var` has id 42, would have the following form: $\langle B\_VAR, 0, 42, 0, 0\rangle$. Function call bytecodes are structured similarly, with the inclusion of an array of argument indices.

As bytecodes are constant at runtime, all dynamic data for a PatternMatcher must be stored somewhere else during runtime. As this data would by necessity be stored in a packet eventually, we use packet buffers as the primary storage location for PatternMatcher data, completely eliminating the need for any other data structure. There is only one packet format (Fig. 4.10), which contains all of the dynamic information needed for a single statement execution attempt. Each packet begins with the statement id it represents, followed by a flag indicating whether or not it is performing actions (as opposed to merely conducting a predicate search attempt). This is followed by storage for node id numbers, to identify the node for each slot (a size is not necessary, as it is defined in the statement bytecode). This is followed by an array of indices (one for each stored variable or function return value), indicating the start index of that stored value. Finally, there is a variable-length array of scalar values, to store state variable and function return values. The packet format describes the body of the packet only, and does not include OS-specific fields such as source and target addresses, payload length fields, and CRC checksums.

4.2.3 Header Generator

In conjunction with the generated bytecodes, the $\mathcal{L}$ compiler will also produce an application-specific header file. This header contains compile-time constants and declarations for variables, functions, and statements. The components of the header file are summarized in Figure 4.11. When compiled with the bytecode engine and user-supplied functions, the application-specific bytecode engine is completed.

The first section of the header deals with variables. The header defines id numbers for each variable, separating out managed and pass-through variables (Section 3.3). Storage for managed variables is also declared and initialized here. For pass-through variables, the header declares `extern` functions for variable read/write operations.

```
Variables:
    variable storage (for managed variables)
    read/write function externs (for pass-through variables)
    history storage and limits
    variable id numbers
    read/write function table

Functions:
    function externs
    function id numbers
    function call dispatch table

Statements:
    number of statements
    bytecode array
```

Figure 4.11: Contents of Application-Specific Header.

By performing a simple analysis of the program, the compiler can determine the maximum history needed for each variable. The maximum over all variables is the number of whole-program snapshots that the runtime must manage. The header defines a custom datatype for the snapshot (based on the number of variables used by the $\mathcal{L}$ program), and allocates an appropriately sized array.

The final step in supporting state variables is a pair of read/write function dispatch tables. These store the read and write function pointers for each variable (using the variable id number as an index into the array). For managed variables, the runtime uses a generic read/write function, which uses the variable id number to index into the managed variable storage array. The read function therefore has the form `DATATYPE read(id varId)`, while the write function is of the form `void write(id varId, DATATYPE value)`, with the id being both the index into the function table and the id number of the variable.

The second portion of the header file deals with user functions. The header declares `extern` declarations for each user function, which (for efficiency's sake) are normalized to have the same arity. This normalization allows for a single function dispatch table, indexed by assigned function id numbers.

The final component of the generated header is the program bytecode. This is a two-dimensional array of scalar values, representing each statement in sequence. As it is a compile-time constant, it takes up no program memory on the sensor node (it can be placed in the EEPROM), allowing for much larger programs.

4.2.4 Bytecode Engine

The operation of the bytecode engine follows the same general outline as that of the naive runtime, with a number of simplifications and optimizations. We can think of the bytecode engine as having split the original

PatternMatcher data structure into two components, a read-only portion that stores the predicate structure and statement metadata (the bytecode), and a writable component that stores the node ids and process state for each individual attempt (the packet buffer). These two data structures together produce a PatternMatcher.

When a packet arrives, the bytecode engine is notified via an interrupt handler or polling. As packet buffers are among the largest objects in the system, it is imperative that they be be dealt with as promptly and efficiently as possible. An incoming packet has one of two functions: it is either a search attempt for a particular statement, or an action trigger (following a successful search). The "Perform Actions" flag in the packet preamble (Fig. 4.10) allows us to discriminate between these two cases. If the packet is an action trigger, we compare the action slot index (found in the statement bytecode) with the index representing our node. If the current node is the trigger location, the actions are executed in order. If not, the packet is forwarded to the node that appears immediately before the current one in the packet's slot list, where it will be forwarded in turn to its eventual destination.

If the packet is a search attempt, we first check the list of filled slots, to determine if the local node id is already listed. If so, this is a duplicate search (as each node can only be in one slot), and we can discard the packet. Note that this failure mode occurs only in systems with a broadcast topology, such as wireless sensor nodes. If not, and there are still empty slots, we perform the fill operation, in a similar manner to that of the naive runtime. The statement id number in the packet preamble provides an index into the bytecode array. We begin executing the predicate bytecode, recursively traversing the tree structure that the expressions define. Binary expressions evaluate both subexpressions, apply their operator, and return. Constants are trivially handled. Variable expressions and function calls are more complicated, however, and require interaction with the current packet object. For both variable references and function calls, if the referenced state is located on a previously visited node, we read the relevant storage location from the packet and return that value. If the slot number indicates the current node, we read the variable (or execute the function) and store the value in the packet.

Once we have performed the fill operation over the predicate, we evaluate the predicate for satisfaction. If the predicate is definitively false, we can discard it. Otherwise, we perform the fill operation on all action expressions, making sure not to call their highest-level functions (as this would prematurely trigger the action). If the predicate is definitively true, we trigger the actions (if local to the node) or reroute the packet to the action location (if not). If the predicate is indeterminate, we send the packet to all neighbors not already assigned slots, or as a 1-hop broadcast (if available).

The per-tick operation of the runtime builds on the *fill* and *evaluate* operations used in packet handling. At the beginning of each tick, we take a snapshot of the current state. By using modulo-indexing into the snapshot array, we can avoid the need to shift (or otherwise copy/delete) other snapshots in the history array. For each statement in the program, we then create an empty packet, fill in the statement id in the packet, and perform the packet handling node as described above. This begins the distributed search attempt for that statement predicate.

4.3 Optimization

In our implementation of the $\mathcal{L}$ runtime, we identified several opportunities for optimization. Most of these were driven by an observation made while evaluating the runtime: that the spreading of extraneous messages is by far the most expensive cost in the runtime. By reducing the number of "wasted" search attempts, we can greatly reduce the overall cost of the system. We will briefly present the two optimizations that we added to the runtime: boolean short-circuiting and topology culling.

4.3.1 Boolean Short-circuiting

In many languages, the conjunctive and disjunctive boolean operators are implemented in such a way as to allow for *guard conditions*. Take the following fragment of a C program:

```
if (flag && failIfFlagNotSet())...
```

The use of boolean-AND in this case prevents the function from being called unless the variable `flag` has been set to true, a practice known as *short-circuiting*. We implement a similar short-circuiting operation in $\mathcal{L}$, allowing undefined values to be ignored in certain cases. A complete 3-state truth table for both traditional and short-circuiting boolean operators is presented in Table 4.1. To modify LDP-Basic to use short-circuiting, we replace the condition

$$\text{if } \mid V \mid < \mid Q \mid$$

on line 13 of LDP-Basic (Alg.1) with

$$\text{if } (\mid V \mid < \mid Q \mid) \wedge \ (Q \leftarrow V \neq \text{false})$$

By using these short-circuiting operators in both the fill and evaluate phases of PatternMatcher processing, we realize two types of cost savings. The first is the savings in processor time, as the expression trees can often be partially evaluated, rather than being fully traversed. The more important savings is the ability to terminate a PatternMatcher before it is completely filled. Without short-circuiting, it is not possible to discard a PatternMatcher until all of its slots have been filled. If the PatternMatcher was actually unsatisfiable early in the search process, this represents a massive waste of both bandwidth (for all of the descendant PatternMatchers) and computation (for the time required to process them). With short-circuiting, the PatternMatcher is immediately discarded when it fails the predicate, even if not all slots have been filled. This prevents any subsequent descendants of the search attempt from being created or propagated. Note that, while the first form of savings is similar to that found in other languages that support short-circuiting, such as C and Java, the second form is unique to $\mathcal{L}$'s use of hop-by-hop state accumulation.

With short-circuiting, we make a new capability available to the $\mathcal{L}$ programmer: guard conditions. By specifying a statement predicate in conjunctive-normal form, the programmer can add subpredicates that

Table 4.1: Truth Tables for Short-Circuiting

		Regular			Short-Cricuit		
InA	InB	&	|	ˆ	&	|	ˆ
T	T	T	T	F	T	T	F
T	F	F	T	T	F	T	T
T	U	U	U	U	U	**T**	U
F	T	F	T	T	F	T	T
F	F	F	F	F	F	F	F
F	U	U	U	U	**F**	U	U
U	T	U	U	U	U	**T**	U
U	F	U	U	U	**F**	U	U
U	U	U	U	U	U	U	U

will prevent spurious evaluation or spreading, if failure of the guard can be detected in an early slot. Take the following $\mathcal{L}$ statement:

```
forall(a, b) where (a.flag == true) & (b.var == 2)...
```

Assume that `flag` is always false for all nodes. Without short-circuiting, useless PatternMatchers would still be created and sent repeatedly to all nodes, even though the predicate could never match. With short-circuiting, this statement requires zero communication between nodes. We examine the effect of this optimization in greater detail in Section 5.1, where we quantify the impact it provides.

4.3.2 Topology Culling

The second optimization we implement is topology culling: using knowledge of neighboring nodes and statement-level topological restrictions to prevent unnecessary communication. As a reminder, there is a construct in $\mathcal{L}$ for expressing neighbor-to-neighbor connectivity: `n(a, b)`. The neighborhood operator returns true if `a` and `b` are neighbors, false if they are not, and undefined if both `a` and `b` have not been filled.

In statements where the neighborhood operator appears, we can use it in conjunction with both the current node's neighbor set and a PatternMatcher's list of filled slots in order to terminate a search attempt that would yield no matches due to topology restrictions. This can be used to reduce the number of cases where the neighborhood operator is undefined, enabling more short-circuiting (see above). As an example, if a node has been assigned slot `a`, and has no neighbors not already in the PatternMatcher, then `n(a,b)` (assuming slot `a` precedes `b`) can return false immediately, even if `b` has not been assigned to anything.

Chapter 5

Evaluation

With our two implementations of an $\mathcal{L}$ runtime complete, we can turn to the task of evaluating the performance of the language. By evaluating the language over a large set of varying parameters, we can more fully characterize its suitability for various application domains. We divide our evaluation into three main components. First, we evaluate the runtime performance using a set of synthetic microbenchmarks, to derive some intuition into how the system scales (Section 5.1). Second, we evaluate the efficiency of $\mathcal{L}$ when compared to an "ideal" implementation of a distributed algorithm (Section 5.2). In this case, we evaluate the performance of the data aggregation algorithm described in Section 3.4. Finally, we evaluate the performance of $\mathcal{L}$ in a complex, large-scale distributed system (Section 5.3). We evaluate the relative performance of the metamodule (Section 3.6) and scaffold planner (Section 3.7.1) algorithms, to provide a qualitative evaluation of the language and runtime. This evaluation will also shed light on the performance of the respective algorithms, enabling us to make some comments on their overall utility.

Unless otherwise specified, we performed all experiments in simulation using the DPRSim simulator developed by Carnegie Mellon and Intel Labs Pittsburgh (64). DPRSim is a massively multithreaded multi-robot simulator. To more accurately measure the differential overhead of the $\mathcal{L}$ runtime, we disabled the physics and graphic rendering portions of the simulator. Programs were executed on arrays of simulated modular robots, arranged in a cubic lattice, where each module can potentially have 6 neighbors, arranged at 90° intervals. Messaging channels were simulated between physically adjacent robots, and the travel time of a message was set to one virtual timestep. The $\mathcal{L}$ runtime was configured to trigger its main routine once per timestep as well. We tested the naïve runtime, as (with the exception of CPU time), the behavior of the optimized runtime is identical.

5.1 Basic Performance Characterization

When evaluating the performance of the runtime, we noted five significant variables that controlled system performance:

- the size of the robotic ensemble
- the rate at which PatternMatchers were culled at each step
- the predicate detection algorithm (centralized or distributed)
- linearity/non-linearity of predicate detection
- the number of slots in the statement predicate

To vary the size of the simulated robot ensemble, we performed tests on 100 to 1000 robots arranged in a cubic lattice packing in stacked planes of 10 by 10 modules. Simulations were conducted for 100 virtual timesteps, where each timestep allowed for arbitrary computation, including message transmission/reception.

To vary the rate at which PatternMatchers were culled at each step, we used the statement shown in Figure 5.1. x_1 through x_4 are four independent uniformly-distributed integer random variables generated by the host robot. Each variable x_n ranges over the integral values from 0 to $max_{x_n} - 1$. We can thus represent the propagation behavior of PatternMatchers within the system with the tuple $[max_{x_1} : max_{x_2} : max_{x_3} : max_{x_4}]$. For example, the tuple [2:2:2:2] will cause half of all PatternMatchers to be discarded after each slot is filled. In contrast, the tuple [4:1:1:1] will cause $\frac{3}{4}$ of all PatternMatchers to be discarded after filling the first slot, but then all remaining PatternMatchers will survive until they are fully filled, at which point they will match. To vary the number of slots in the statement predicate, we considered variants of the statements in Figure 5.1, utilizing between one and five random variables, and a corresponding number of slots.

To vary the topological restrictions found in the statement, we compared the results of monitoring the expressions in Figure 5.1. Expression 5.1a contains no neighbor restrictions, meaning that it can match any ordered, nonlinear subensemble that passes the other criteria in the statement. Expression 5.1b is the opposite extreme, requiring that all modules in the subensemble lie in an ordered linear path. This is equivalent to enabling or disabling the optional nonlinear matching capability introduced in the naïve runtime (Sec. 4.1).

5.1.1 PatternMatcher Decay Factors

With these test cases we can now examine how variation in PatternMatcher discard rates impacts the number of active PatternMatchers (and thus the execution time). We begin with the "worst case" tuple, [1:1:1:1],

```
a) forall(a,b,c,d) where (a.x1 == 0) & (b.x2 == 0) &
                         (c.x3 == 0) & (d.x4 == 0)
                   do d.logMatch();
b) forall(a,b,c,d) where n(a,b) & n(b,c) & n(c,d) & (a.x1 == 0) &
                         (b.x2 == 0) & (c.x3 == 0) & (d.x4 == 0)
                   do d.logMatch();
```

Figure 5.1: $\mathcal{L}$ statements for evaluating PatternMatcher propagation factors. `x1` through `x4` are per-module independent random variables. a) Nonlinear statement. b) Linear statement.

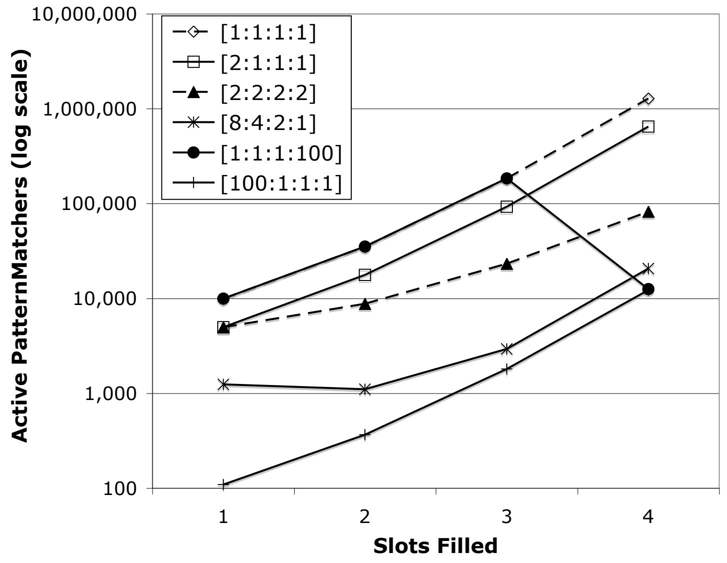


Figure 5.2: Active PatternMatchers vs. Slots Filled (Non-Linear Matching)

Table 5.1: Successful Matches vs. Execution Time

	Linear			Non-linear		
Program tuple	# matches	centralized time(s)	distributed time(s)	# matches	centralized time(s)	distributed time(s)
[1:1:1:1]	253 646	10.6	39.7	1 278 400	102.4	223.5
[2:1:1:1]	127 224	7.0	20.7	635 313	51.6	107.1
[2:2:2:2]	15 616	5.4	9.6	78 320	17.5	30.8
[1:2:4:8]	3 920	7.6	13.3	19 695	21.9	37.8
[8:4:2:1]	3 831	4.0	4.0	19 061	5.0	6.7
[1:1:1:100]	2 622	11.0	39.9	11 628	107.3	223.7
[100:1:1:1]	2 547	3.1	3.5	12 400	4.0	4.7

where all generated PatternMatchers will always survive, leading to an exponential explosion of PatternMatchers as seen at the top of Figure 5.2 (note the log scale). After 100 timesteps on 100 robots, over 250 thousand successful matches have accumulated in the linear case, while over 1.2 million have been found in the nonlinear case. We can easily halve the number of active PatternMatchers by using the tuple [2:1:1:1], which halves the number of PatternMatchers that survive having the first slot filled. Halving the number of active PatternMatchers at each step (as in [2:2:2:2]) results in the expected decrease by a factor of 16. Comparing [1:1:1:100] and [100:1:1:1] is quite instructive. Both eventually generate an almost identical number of successful matches, but over wildly different trajectories, as can be seen in the area under the curve in Figure 5.2. [1:1:1:100], which culls almost all of its PatternMatchers after the last slot has been filled, is much less efficient than [100:1:1:1]. This can be seen at the bottom of Table 5.1, especially in the nonlinear case, where [1:1:1:100] takes over 25 times as long as [100:1:1:1]. Continuing to examine Table 5.1, we note a general linear increase in the amount of time taken by the distributed algorithm as the number of successful matches increases.

These results are a strong testament to the power of the short-circuiting optimization, without which all executions would be identical to the worst-case [1:1:1:1] tuple. They also point to the importance of structuring statement predicates in such a way as to have them fail early in the search process, rather than at a later slot.

Table 5.2: Effect of Varying Network Degree (approx. 1000 modules, [2:2:2:2] program tuple)

Metric	Planar	Cubic	Face-Centered
# of Modules	1 023	1 000	936
Average Degree	3.87	5.4	10.13
Execution Time (s)	288.187	2 215.359	5 072.47
Messages	1 776 726	8 895 050	26 899 725
Matches	403 842	2 692 834	10 634 815
Messages/Match	4.399	3.303	2.529
Time/Match (ms)	0.714	0.823	0.477

5.1.2 Topology and Network Degree

The dependency on decay rates that we have illustrated above is due to a simple fact: there is an exponential number (in the number of slots) of PatternMatchers generated by the spreading of each PatternMatcher to all of a robot's neighbors after each slot is filled. And by shifting the criteria that are least frequently true to early in the watchpoint evaluation, we can dramatically cut execution time, even though the total number of successful matches remains the same.

We can illustrate this with a simple governing equation that will give an upper bound on the number of active PatternMatchers for any number of filled slots. Using the same program tuple formulation as above, we have an m-slot PatternMatcher with an associated program tuple $[x_1{:}x_2{:}\ \cdots\ {:}x_m]$. In an ensemble of n modules, whose connectivity is of degree k, we arrive at the following expression for the number (p) of active matchers with t slots filled:

$$p = \frac{n}{x_1} \prod_{i=2}^{t} \frac{ki}{x_i}$$

For a fully linear watchpoint, the expression becomes:

$$p = \frac{n}{x_1} \prod_{i=2}^{t} \frac{k}{x_i}$$

We note that both expressions are exponential in m (with base k), and linear in n. That is to say, the number of active PatternMatchers is exponential in the number of slots (as each additional slot can be filled with at least k neighbors), but only linear in the number of modules (as each module begins the search process in parallel). Comparing this to the results above, we have an ensemble of size 100, over 100 timesteps, with program tuple [1:1:1:1] and degree 3 (an approximation), giving an expected number of matches of 270,000,

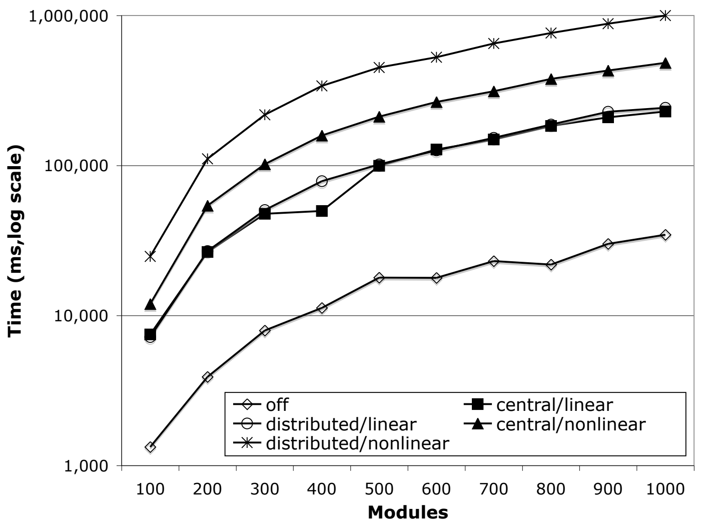


Figure 5.3: Execution Time vs. Ensemble Size

which is quite close to the actual value of 253,646. This number corresponds well to the expected costs predicted by the LDP-Basic formal model presented in Section 2.4.

To further explore this issue, we examine the performance of the $\mathcal{L}$ microbenchmarks in differing packing densities. We executed the [2:2:2:2] program tuple on three ensembles of $\sim$1000 simulated robots, in a 2D rectilinear lattice (maximum of 4 neighbors), a cubic lattice (maximum of 6 neighbors) and a face-centered cubic lattice (maximum of 12 neighbors). Table 5.2 summarizes the results. As the average network degree increases, we see the expected super-linear growth in both messaging and number of successful matches. We note that the number of messages per match decreases with increasing network degree (as the fan-out from the penultimate slot produces proportionately more matches), and that the simulation time per match remains below 1 ms/match.

5.1.3 Computational Overhead

We also analyzed the overall execution time of the runtime, and the scaling behavior as the size of the simulated robot ensemble grows (Figure 5.3). In these tests, we used the same $\mathcal{L}$ statements as above, with the [2:2:2:2] program tuple. Each robot also ran a medium-intensity background tasks (a data aggregation and landmark routing program), to simulate a preexisting workload on the system. Tests were run on ensembles of various sizes, using the centralized and distributed implementations, as well as without the $\mathcal{L}$ runtime enabled (for comparison).

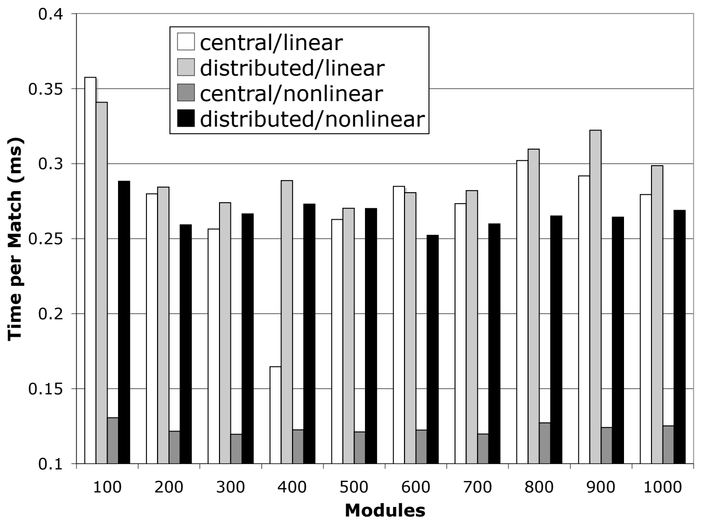


Figure 5.4: Time per Match vs. Ensemble Size

Overhead for linear expressions is quite reasonable, with the centralized algorithm having an average overhead of 115% and the distributed algorithm an average overhead of 304%. The overhead for both algorithms is well within the range of other debugging tools like GDB (28) and Valgrind (56). Unfortunately, the overhead for nonlinear expressions is much higher, being as high as 5182% in the worst case. While this may seem unacceptably high, we must consider the number of successful matches being found via the algorithm. With 1000 modules, the nonlinear statement finds 3.57 million matches over 100 timesteps. If we plot the time expended per match (Figure 5.4), we see that the time required as a function of the number of matches is actually quite low: less than one millisecond per match. If we assume that the system will be used to detect relatively infrequent events, then this level of performance is quite adequate.

To test this hypothesis, we compared a set of experiments on 1000 modules using the [100:1:1:1] program tuple. Results were quite encouraging, with an overhead of only 5% (a 20-fold reduction) for linear subensembles detected via the centralized algorithm, and 43% (over a 100x reduction) for nonlinear subensembles detected with the distributed algorithm. We should emphasize that these CPU overhead numbers are more of a general guideline than an absolute commentary on the performance of $\mathcal{L}$, as there was little attempt made to optimize the naïve runtime for optimal CPU usage.

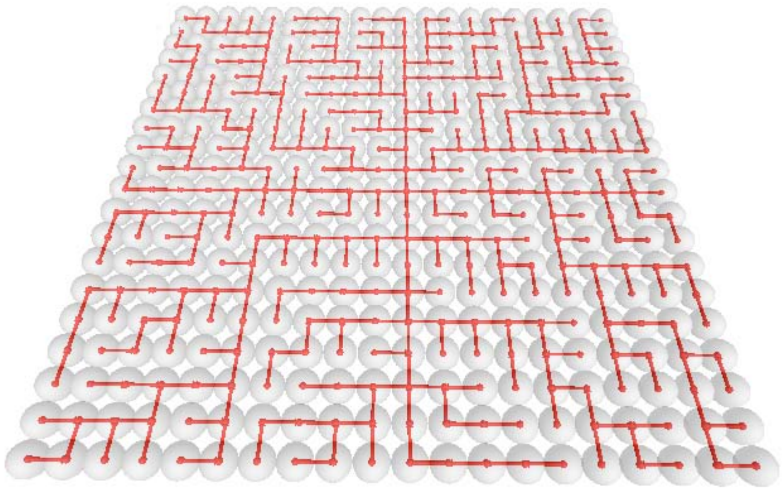


Figure 5.5: Spanning Tree For Data Aggregation

5.2 Spanning Tree and Aggregation Evaluation

To evaluate the performance of an $\mathcal{L}$ program in absolute terms, we must compare the $\mathcal{L}$ runtime with some implementation-independent performance standard. For this purpose, we examine the messaging behavior of $\mathcal{L}$ as messaging is both a significant cost factor in real systems, and a cost that is not as subject to simulator artifacts as time-based metrics.

We evaluate the two phases of the data aggregation algorithm presented in Section 3.4: formation of a spanning tree, and transmission of aggregate data to the root. We evaluate the messaging cost of the implementation on a planar grid of 20x20 simulated robots, with the root of the tree placed near the middle of the ensemble. The ensemble (and resulting spanning tree) can be seen in Figure 5.5. This ensemble has 400 modules, and 1520 different unidirectional communication links, or 760 graph edges.

First, we must determine a proper baseline cost for both spanning tree construction and data aggregation. From Halperin (34), we know that parallel spanning tree formation on a graph of n vertices and m edges can be performed in $2m$ messages, and expected $O(\log n)$ time. For data aggregation, we note that each module need only send its current aggregate value to its parent (once it has collected data from all of its children), at a cost of $n - 1$ messages.

With these baseline numbers in place, let us evaluate the performance of the $\mathcal{L}$ program given in Listing 3.10, as it runs to completion. A sample execution of the program reveals that the spanning tree construction phase used 124,422 messages, while the aggregation phase used 122,898 messages. This is a dramatically higher number than we would expect, so we must investigate the reason behind such phenomena. Examining the individual messages, we immediately notice something: the messages for both spanning tree construction and child set membership continue *even after the trees/sets have been completed*. This is due to the fact that the data aggregation algorithm was implemented in $\mathcal{L}_3$, which does not have temporal quantifiers.

Listing 5.1: Improved Aggregation

```
1  scalar isSeed;                     // is this node the root of the aggregation tree?
2  scalar id;                                                // unique id for each node
3  scalar parent = -1;                             // parent's id in the aggregation tree
4  set notChildren = null;                   // set of neighbors who are not our children
5  set children = null;                    // set of children that have reported data to us
6  set neighbors;                                           // set of neighboring nodes
7  scalar isComplete = 0;                   // is the aggregation complete on this node?
8  scalar sensor;                              // sensor providing the scalar field values
9  scalar sum = 0;
10 scalar count = 0;
11 scalar average = 0;
12
13 forall (a) where (a.isSeed == 1) do a.parent = a.id;
14 forall (a,b) where (a.parent != -1) & (a.prev.parent == -1) & (b.parent == -1)
15              do b.parent = a.id;
16 forall (a,b) where (a.parent != -1) & (a.prev.parent == -1) & (a.parent != b.id)
17              do b.notChildren += a.id;
18 forall (a) where (a.size(a.neighbors) == a.size(a.children) + a.size(a.notChildren))
19             do a. isComplete = 1;
20 forall (a,b) where (a. isComplete == 1) & (b.id == a.parent) &
21                    (!b.contains(b.children,a.id))
22              do b.sum = b.sum + a.sum + a.sensor,
23                 b.count = a.count + b.count + 1,
24                 b.children += a.id;
25 forall (a) where (a.count > 0) do a.average = a.sum / a.count;
```

If we rewrite the aggregation program to take advantage of edge-triggering, we can significantly reduce the number of wasted messages. The changes in Listing 5.1 are confined to lines 14-17. The addition of the `(a.prev.parent == -1)` clause acts as an edge-triggering mechanism, allowing the predicate to match only on the tick immediately following assignment of a parent. As the predicate is designed to fail on slot `a` in this case, we can eliminate many wasted messages. Execution of this improved program requires 1520 messages for tree construction, and 3040 messages for aggregation. Creation of the spanning tree now takes the number of messages required to traverse each link once, the exact number required. Aggregation is still more expensive than we would like, a factor of 7.6 higher than it needs to be.

Can we reduce this even further, or is it a result of some inherent inefficiency in the system? Let us consider the precise costs of the aggregation phase. Creating the child sets requires 1520 messages, and performing the aggregation requires another 1520. These operations could both conceivably be performed in $n-1$ messages each, if the relevant PatternMatcher was sent only to the parent of the module. Unfortunately, the $\mathcal{L}$ runtime is not capable of performing predicate search targeted to only 1 neighbor, if the selection of

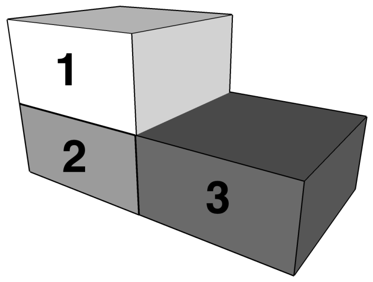


Figure 5.6: Experimental configuration for shape change evaluation. Start configuration (regions 2 and 3) is 8:4:2, end configuration (regions 1 and 2) is 4:4:4.

that neighbor is dependent on a variable value. Propagation of PatternMatchers always proceeds in a pseudo-broadcast fashion, which explains the cost of 1520 messages (based on the number of links) instead of 399 messages (from the number of modules). This is clearly a weakness in our design, and we propose a possible solution in Section 6.2.

5.3 Motion Planning

We evaluated both the scaffold and metamodule planners on ensembles of simulated robots using DPRsim (64). 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 5.6 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 and that there are the exact number of modules needed to form the desired shape. Trial runs of the two planners were conducted on 8x4x2 (32 module), 16x8x4 (256 module), and 24x12x6 (864 module) ensembles, as the initial packing density of both the metamodule and scaffold shapes is 50%. Experiments were run until the target shape was completed.

5.3.1 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 5.3). 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 schedule communication fairly, DPRsim1 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. Ideally, we would modify DPRsim1 in order allow us to parameterize the relationship between messaging cost and movement cost.

5.3.2 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).

5.3.3 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 5.7 and 5.8 for the corresponding plots for the largest of the simulation experiments. The knee (slowdown) in each curves occurs at the point where physical contention and resource exhaustion limit the rate at which target destinations can be filled.

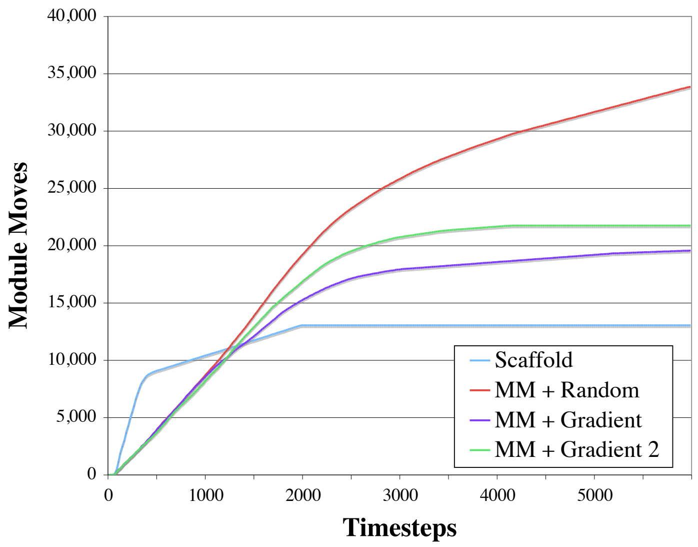


Figure 5.7: Planner performance: Module moves over time.

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.

5.3.4 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

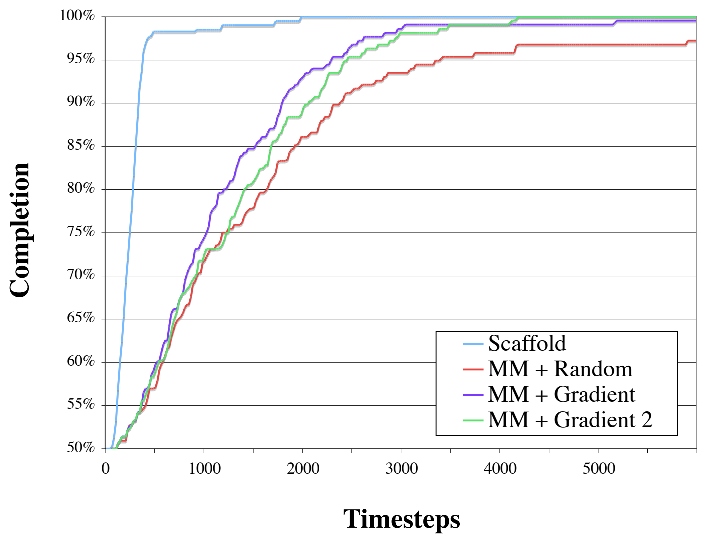


Figure 5.8: Planner performance: Completion percentage over time.

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 distance between start and end positions, divided by the total distance moved. 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.

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 5.3, 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.

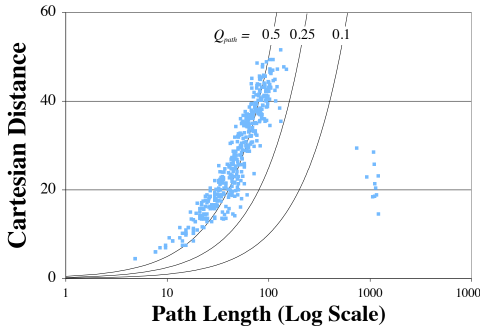
By plotting the components of Q_{path} against one another for each module in each scenario (Figure 5.9), we can understand the performance differences in greater detail. The scaffold planner achieves very tight clustering around its mean Q_{path} value, indicating a very controlled, homogeneous strategy. 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 (73).

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 5.8.

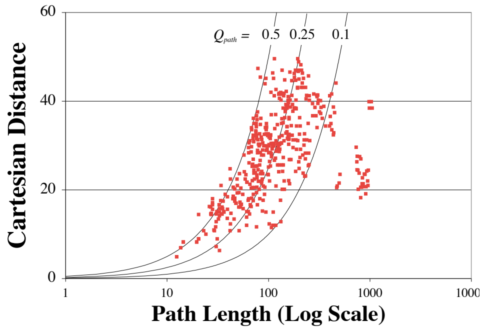
5.3.5 Implications for Module and Metamodule Design

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 modular robotic 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.

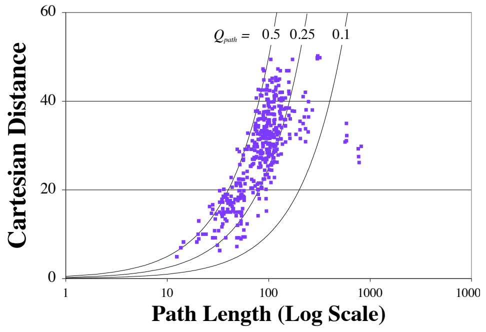
Still, we note that more intelligent resource managers, either with more expressive gradients or other mechanisms, can potentially increase the performance of metamodule-based planners to a level that approaches that of the scaffold algorithm. Combined with the potential for a metamodule-based approach 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 modular robot system level.



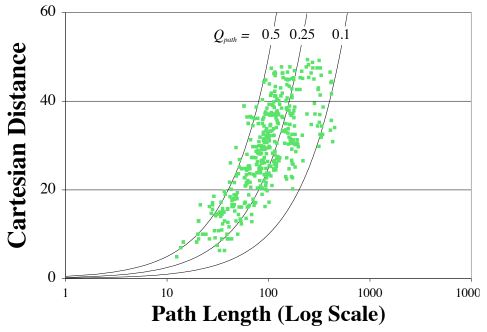
(a) Scaffold



(b) MM+Random



(c) MM+Gradient



(d) MM+Gradient2

Figure 5.9: Path length traversed vs. Cartesian distance between start and end positions for individual modules for the different planners

Table 5.3: Performance of Planning Algorithms (And Variants)

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

Table 5.4: Bytecode Performance on BlinkyBlocks

# operations/ statement	statements / sec	operations / sec	# cycles / operation
3	232.56	697.68	45 866
5	172.69	863.45	37 061
9	113.25	1019.16	31 398
17	67.52	1147.84	27 878

5.4 Evaluation on Embedded Hardware

We ported the optimized $\mathcal{L}$ runtime to the BlinkyBlock platform, a research vehicle for introducing distributed programming to students and artists. BlinkyBlocks are cubical modules which snap together via magnetic connectors, creating a 3-dimensional lattice structure. Connected blocks can share power and data, and each block has computation, storage, and RGB LEDs. The processor for each block is an Atmel ATxMega 32A4, an embedded 8-bit RISC processor running at 32 Mhz, with 32 Kb of program storage (Flash), and 4 Kb of RAM. Each of a block's six bidirectional serial ports operates at 38.4 kbps, exchanging active-message style packets **CITE** with a post-fragmentation payload size of 18 bytes. We evaluated the $\mathcal{L}$ runtime on this platform using both microbenchmarks and full programs.

We used a series of simple microbenchmarks to profile the basic memory, computation, and communications performance characteristics of the $\mathcal{L}$ runtime. To measure the CPU usage of the runtime, we created a simple $\mathcal{L}$ program with one size-one statement, which performed a variable number of bytecode operations. By varying the number of operations in the statement, we could observe the static overhead of the $\mathcal{L}$ runtime, as well as the cost to execute a single bytecode instruction. We ran the system for 30 seconds, and measured the number of statements that the runtime could process when given an infinitely small timestep quanta. Each microbenchmark was repeated 10 times, and we report the mean of all executions in Table 5.4. Total variance during all runs was below 0.1% of the mean. As we can clearly see, the static overhead of the runtime is quite considerable, and as the system processes more bytecode operations as a fraction of its total workload, the cost per bytecode instruction drops dramatically.

We evaluated the sustained message throughput in a similar manner, recording the total number of messages that a pair of blocks was able to exchange in thirty seconds. We varied the number of 16-bit scalar variables that each message carried, and ran the blocks as fast as the runtime was able to process incoming packets. We repeated each experiment 10 times, and report the mean (and derived values) in Table 5.5. Total variance was under 4.0% in all cases. We attained at most 28% of the rated channel bandwidth, leading to the conclusion that the system is CPU-bound, a conclusion supported by the previous microbenchmark. We

Table 5.5: Messaging Performance on BlinkyBlocks

bytes / message	messages / sec	payload bits / sec
12	110.19	10578.2
20	47.68	7628.8
35	26.13	7316.4

Table 5.6: Early Termination on BlinkyBlocks

Termination Rate	Statements / sec
0%	1982.1
25%	2025.2
50%	2093.6
75%	2144.8
100%	2261.1

can also see the dramatic decrease in throughput as message size increases above 18 bytes, and payloads must be fragmented across multiple packets.

We then evaluated the effects of the short-circuiting and early termination optimizations. We used a similar $\mathcal{L}$ program to that in Figure 5.1, but evaluated on only two blocks. By varying the chance that a PatternMatcher would terminate early, we can see how communication- or processor-bound the runtime is. As can be seen in Table 5.6, the sustained statement throughput of the blocks increases by approximately 10% as the early termination rate increases from 0% to 100%. This is reflective of the high overhead of both the $\mathcal{L}$ runtime and the underlying embedded runtime, which consumes more than 50% of the available compute resources.

Our final microbenchmark analysis centered on memory usage, both static program size and runtime RAM allocation. By examining the compiler output, we observed that the base $\mathcal{L}$ runtime required 6804 bytes of program memory on top of the basic system libraries, and that the $\mathcal{L}$ program took approximately 9 bytes of static program memory per bytecode instruction. The runtime took 964 bytes of memory, most of which was the 32 packet buffers reserved for incoming messages. Dynamic memory usage was confined to the execution stack of the bytecode interpreter, which consumed 28 bytes per stack frame.

After completing the microbenchmarks, we implemented a simple $\mathcal{L}$ program to test the usability of the system in an actual distributed programming scenario. We selected a simple Z-stratification algorithm, which colors the blocks according to their vertical position in the ensemble, with the lowest block(s) being red, then those above them orange, etc. This program is interesting in that it must respond to the ensemble's dynamic topology, even in the case where the lowest block is added or removed, and the entire ensemble must recolor.

Listing 5.2: Rainbow Z-Stratification

```
scalar resetCount = -1;                                        // current reset wave
scalar checkDown  = false;                         // check if this block is the bottom-most
scalar color;  // color-table index, in rainbow order, evaluated modulo number of colors

scalar const DOWN = 0;
scalar const UP = 1;

scalar down(); // returns 1 if there is a block below us, 0 otherwise
scalar msgDirection(scalar); // returns the direction that the current msg is moving
scalar max(scalar, scalar); // returns the maximum of two scalars

forall(a) where (a.resetCount < 0) do a.resetCount = 0, a.color = 0;
forall (a) where (a.checkDown == 0 & a.down() != 0 & a.color > 0) do a.checkDown = true;
forall (a) where (a.down() == 0 & a.checkDown == true)
           do a.resetCount = a.resetCount + 1, a.color = 0, a.checkDown = false;
forall (a,b) where (a.resetCount > b.resetCount) do b.resetCount = a.resetCount,
        b.color = 0;

forall (a,b) where (a.resetCount == b.resetCount) & (b.msgDirection() == DOWN) &
                   (a.color + 1 > b.color) do b.color = a.color + 1;
forall (a,b) where (a.resetCount == b.resetCount) & (b.msgDirection() == UP) &
                   (a.color - 1 > b.color) do b.color = a.color - 1;
forall (a,b) where (a.resetCount == b.resetCount) & (b.msgDirection() != UP) &
                   (b.msgDirection() != DOWN) & (a.color > b.color)
            do b.color = a.color;
```

Implementation of this program encountered several minor obstacles, all related to the development environment on the blocks themselves. The blocks do not have unique identifiers, and so the slots of the PatternMatchers are filled with relative directions, instead of identifiers. This functionality is exposed via the msgDirection() function seen in the code listing. The blocks also possess the ability to sense the presence of neighbors in any of the six lattice directions, via a heartbeat message that the underlying operating system propagates at regular intervals. Unfortunately, the links between blocks are prone to temporary failure, leading to cases where a block may falsely sense that a neighbor has disappeared, even while that neighbor remains aware of their relationship. Despite these issues, we were able to successfully implement Rainbow Z-Stratification on an ensemble of 10 blocks, and had the algorithm run to completion within 30 timesteps in all of the configurations that we tested.

Chapter 6

Conclusions

We conclude this dissertation with a comparison of $\mathcal{L}$ and LDPs to existing work in the areas of distributed predicate detection and distributed programming. We then discuss some open problems and future directions for research in topology-aware predicates. Finally, we recapitulate the conclusions of this work and summarize its contributions.

6.1 Related Work

Distributed programming, as a research area, has attracted considerable attention over the years. To summarize all work in the field would require a book-length treatise (41). Nonetheless, there are several specific lines of inquiry that bear great relevance to our work on LDPs and $\mathcal{L}$. In this section, we summarize several of the most relevant works in distributed programming, coordination languages, and distributed predicate detection.

Datalog and Its Derivatives

Datalog (13) is a logic programming language for databases, derived from Prolog (80). Developed in 1978, it served as a major influence on the design of other declarative database languages, such as SQL (16; 1). Datalog programs are composed of two types of entities: *facts* and *rules*. Facts are assertions about the state of the system, and rules are statements which allow for the derivation of new facts from existing ones. Rules are constructed from *variables*, *predicates*, and *constants*. By applying forward-chaining proof search, we can produce complete derivations for any fact obtainable from the set of base facts. Safety restrictions on rules and variables ensure that a given Datalog program can only derive a finite number of facts.

Recent research on sensor networks and distributed hash tables (2; 66) led to the realization that Datalog-style programming was well suited for processing distributed event streams. One of the first major works on declarative networking in this fashion was P2 (48), a Datalog derivative explicitly designed for distributed systems operation. In particular, P2 was used to implement network overlay protocols, such as the Chord distributed hash table (72). P2 differs from Datalog in several key ways, though it shares much of the fundamental runtime architecture. P2 operates over tuples of reference-counted value objects, present either as continuous streams or materialized tables. Data tables have a fixed size, and the user can specify lifetime and eviction policies for these tables. Once a table is full, tuples may be evicted (and deleted), making P2 no longer a pure logic language. P2 also supports the notion of locality: a particular fact (tuple) may be located at one node, and rules may require facts from multiple nodes. The P2 compiler is then responsible for the conversion of the program into a distributed dataflow engine, which produces the desired output facts.

The research group behind P2 continued their work in two different directions: general declarative networking (NDlog) (49), and sensor network programming (Snlog) (15). NDlog is an extension of P2 that allows for *link relations*, external rules that specify the communications topology, which can be used as inputs to other rules. This allows for link-restricted rules, which operate only on pairs of nodes which share a communications link. Research on using NDlog for debugging and monitoring reinforced the utility of logic programming for this application domain, allowing for the unification of specification, implementation, monitoring, and debugging.

Snlog is a specialization of P2 for programming embedded wireless sensor networks. In particular, it targets applications such as TinyDB (52), code dissemination (46), and entity tracking (60). Innovations in Snlog are primarily concerned with reducing the memory, CPU, and network footprint of execution, by providing the programmer the ability to many particulars of program execution, including rule execution frequency and priority. Rules can be configured to ignore subsequent matching predicates once they have been triggered. Facts in Snlog can have side-effects, allowing the logic program to interface with the low-level implementation of the sensor node, in a manner similar to $\mathcal{L}$'s hybrid coding approach.

A final derivative of Datalog/P2/Snlog is Meld (6), a declarative language for programming modular robots. Meld is distinguished from other declarative approaches in that it supports explicit revocation of facts, rather than having facts implicitly garbage-collected due to space concerns. As an example, derivation of the fact `moveTo(x,y,z)` might cause a module to move to a new position, explicitly invalidating any facts relating to the module's position or neighbors, and forcing the re-derivation of any dependent facts. Meld also allows for the declaration of *virtual neighbors*, similar to NDlog's link relations, but without any restriction to neighboring modules. Virtual neighbors can then be used to perform part of a Meld program using the overlay network defined by the virtual neighbors.

This family of declarative languages can be seen as addressing the same problem domain as $\mathcal{L}$, though via a dramatically different approach. Meld et. al. use forward-chaining proof search to derive facts (and thus behaviors) in a distributed system, while $\mathcal{L}$ relies on a snapshot-inspired predicate detection algorithm to drive reactive behaviors. Meld is a logic programming language, which provides a certain level of specification-checking and verification for free. This approach does have some drawbacks, however, as expressing properties that cannot be readily proven is difficult, as in the case of negation or operations with side-effects.

Linear Temporal Logic

Linear temporal logic (LTL) (65) is an extension of first-order logic with modal temporal operators. LTL has been used to model and verify the operation of distributed and parallel systems. Typically, verification is performed by converting a safety or progress property from its expression in LTL to a pair of Büchi automata (32), one for the model and one for the negation of the property of interest. If the intersection of these two automata is empty, the model satisfies the property. Unfortunately, the verification of arbitrary LTL expressions has been shown to be PSPACE-complete, leading to the use of practical fragments of the logic (10). The most widely-used example of LTL in this application is the Spin model checker (36), used in pre-mission verification of a number of NASA missions.

LTL provides the basis for a number of works on multi-robot planning. In (26), LTL formulas are created that express the motion of one or more robots within a discrete space. The negation of these formulas is then model-checked, providing as counterexample (a witness) a computation path that satisfies the desired property. This sequence of discrete actions can then be used as inputs to continuous controllers that drive the desired behaviors. In (43), LTL is used to safety-check the runtime behavior of robots, by continuously verifying robot state (and execution history) against a provided set of LTL properties.

$\mathcal{L}$ does not directly use LTL, but the notion of temporal modal operators was highly influential on the initial design of distributed watchpoints (21), the precursor to work on LDPs and $\mathcal{L}$. The use of finitely-bounded modal operators and discrete timesteps suggests that $\mathcal{L}$ programs might be amenable to analysis with LTL-based model checkers.

Coordination Languages (Delirium, Axum,Reo)

Coordination languages are an approach to distributed programming that treat inter-process coordination as an orthogonal activity to computation, and (in most cases) try to provide high-level tools for managing the former. Coordination languages are differentiated from other distributed programming techniques (such as GPGPU (61) or MapReduce (22)) by their ability to express arbitrary (or widely varying) communication structures.

The first successful coordination language was Linda(33), a programming environment that provided a shared tuple-space that all programs could read from and write to. Through the use of shared names and sequence numbers, the tuple-space could form an ad-hoc implementation of many common distributed communication topologies. Linda inspired a large number of divergent coordination languages (50; 51), though we will choose to focus on four recent works that exemplify current research in the area.

Reo (4) is an exogenous coordination language, meaning that the dataflow topology can be controlled by external entities not involved in processing data. Reo's expressive power comes from two sources: node topology operations and channel types. In Reo, channels are anonymous, bidirectional links between nodes. Channel endpoints may be decoupled from the nodes they are connected to, and both endpoints and nodes may be dynamically migrated between machines or processors. Reo supports a wide variety of channel types, including synchronous, asynchronous, FIFO (bounded or unbounded), and lossy. Lossy channels are particularly interesting, as they allow the channel to discard messages that do not match a specified pattern. Channels are combined using passive aggregation/dissemination nodes, and the developers of Reo show that the available channel types can be used to emulate many common distributed dataflow modalities, such as barrier synchronization and controllable flow gates.

In Dryad (38), a distributed computation is mapped to a DAGs of processes, organized in producer/consumer chains. A central job coordinator and name server are responsible for assigning each process to an execution node. Based on the communication distance between the two endpoints, a channel may be dynamically remapped to use local file storage, shared-memory FIFOs, or network communications. Dataflow graphs are built using library calls and operator overloading in C++, and subgraphs may be saved, parameterized, and instantiated as single units. The Dryad job coordinator is capable of dynamic graph refinement, in topology, number of nodes (input partitioning), and node location in order to efficiently exploit available computing resources.

Regiment (57) is a coordination language designed for sensor networks: programs are expressed in terms of geographic regions and operations, and operations on those regions. Nodes emit streams of data, which are manipulated and filtered in a region-specific manner. Regions can be defined geographically (circles, closest-k), in terms of network diameter (k-hop diameter), or in terms of broadcast radii (gossip-based grouping). Regiment is suitable mainly for apps with a single operation mode, that use only spanning-tree friendly communication patterns (as this is the underlying primitive used to implement regions).

Axum (18) is an agent-based coordination language. Agents share data via channels and shared-object domains. Channels can optionally be order preserving, and can include state machines to enforce communications protocols. The Axum runtime will utilize annotations denoting side-effect-free functions, and can clone/join the computation graph to parallelize purely functional operations. Shared memory domains use implicit reader/writer locks to provide a scoped Linda-style shared tuple-space. Axum supports a wide variety of topology operators, including combine, multiplex, broadcast, and load-balance.

$\mathcal{L}$ is distinguished from previous approaches to coordination languages by its focus on small-scale subgraphs of larger systems. Programs in $\mathcal{L}$ are composed of dynamic, changing cluster of nodes, with nodes belonging to multiple statements at once. Previous coordination languages have focused on creating large-scale, relatively stable, dataflow processing graphs. They describe the program architecture from a macrolevel, while $\mathcal{L}$ operates on the micro-scale.

6.2 Future Work

During the course of our research, we have identified a number of research opportunities and open questions relating to topology-aware predicates. Exploring these unanswered questions would help to increase our understanding of this class of distributed system. We can broadly divide these areas for future research into four categories: more expressive predicates, advanced runtime features, improvements to the $\mathcal{L}$ language, and additional application domains.

$\mathcal{L}$ is based on the notion of locally distributed predicate search, which limits it to operations over fixed-size, linearly connected subgroups of modules. The presence of the neighborhood operator, variably-sized statements, and nonlinear matching slightly improve the expressivity of the system, but the fundamental restrictions of the predicate class remain. We anticipate the development of a more general class of topology-aware predicates than LDPs, capable of representing more complex topologies than just linear arrangements. Such predicates would be able to represent tree structures, DAGs, or arbitrary graphs within a distributed system. This capability would enable programmers to capture more complex distributed relationships, and implement a wider variety of distributed algorithms.

The second area where additional research might prove fruitful is in the implementation of the $\mathcal{L}$ runtime. The first issue of concern is the inefficiency of broadcast-mode predicate search, as described in Section 5.2. The inability of the runtime to confine predicate search to a dynamically chosen subset of neighbors results in significant excess communication. The addition of such capability (along with language support for expressing it) would allow for much more efficient implementation of non-broadcast algorithms. There is currently no explicit support for fault tolerance or recovery in the $\mathcal{L}$ runtime. If a node fails, then any PatternMatchers involving the node may also fail, either during their search phase or during action routing/execution. The current $\mathcal{L}$ implementation provides incidental failure recovery (via retries on the next tick), but this is not a general-purpose solution, as it is incompatible with edge-triggering techniques. A more robust failure-recovery system would allow for explicitly controlling retransmissions, or provide notification support to alert the programmer when communications with a node have failed. A more advanced option would be to cache data from PatternMatchers as they passed through a node, allowing neighboring nodes to form a distributed cache of each other's state variables. This could also have the effect of reducing communications, even if there are no node failures.

Enhancements to the runtime of $\mathcal{L}$ would be complemented by research on enhancements to the language syntax. While the use of sets as the only primitive aggregate makes for a simple language (and implementation), it makes the handling of many forms of ordered or structured data more difficult. Additionally, the language's inability to nest aggregates precludes many complex data structures. The addition of either hashtables (as in Lua (37)) or arrays (as in C) as a primitive aggregate type would greatly ease the implementation of tuple-spaces (33) and other similar constructs. As we have identified many common design patterns in $\mathcal{L}$ programs (edge triggering, one-time triggers, cascading statements, membership sets), it would be helpful to develop and implement syntactic support for these common idioms, so that they are not as cumbersome to use. As mentioned in Section 3.5.4, there are a number of potentially valid local snapshot semantics for $\mathcal{L}$. We anticipate that further research is needed to characterize which of these semantics is most useful for programmers, or if programmers should have the opportunity to explicitly specify the snapshot behavior. Finally, we note that, while the neighborhood operator provides support for describing static topologies in $\mathcal{L}$, there exists no built-in support for reacting to topology changes (the addition/removal of neighbors). We see this lack as a serious omission in the functionality of $\mathcal{L}$, whose rectification would enable such applications as dynamic failure recovery.

The last potential area for future research would be the applicability of these techniques to other application domains. As an example, recent advances in many-core chips have led to processors with 16, 64, or even 80 cores on a single chip (78). Similarly, many current-generation GPUs have upwards of 128 programmable computation units (61). These large computational fabrics are typically connected using ring, grid, torus, or hypertorus geometries. All of these topologies are sufficiently sparse that LDPs might be a viable choice for parallel programming on many-core chips, either as a primary programming language, or as a coordination language to organize many quasi-independent threads of computation. A second application domain where LDPs might be useful is in high performance routing. Routers could exchange information on traffic flows, congestion, etc using $\mathcal{L}$, and could be controlled and configured in a distributed manner in order to maximize network performance.

6.3 Conclusions

This dissertation has explored the consequences of a simple premise: that distributed systems with sparse topologies require tools and techniques capable of expressing topology. We began by describing a novel class of distributed properties: *topology-aware predicates*. These predicates can include not only program state, but also the topology of the system. We then identified a concrete subclass of these predicates, over fixed-length linear subgroups of nodes. This predicate class, which we called *locally distributed predicates*, formed the basis for our exploration of distributed state in sparse topology systems.

We began by formally defining locally distributed predicates, and placing this definition in the context of traditional distributed property classes. It was obvious that LDPs formed an orthogonal categorization axis to predicate classes that were defined in terms of detectability, such as strong or stable. We then developed two algorithms for detecting LDPs: LDP-Basic and LDP-Snapshot. LDP-Basic allowed us to gather a consistent sub-cut of state, permitting the detection of strong stable LDPs. LDP-Snapshot was a more intrusive algorithm, but it allowed for the detection of stable LDPs. We then performed an analysis of the complexity of the LDP algorithms, and showed that, in sparse topology systems, both algorithms scaled significantly better than Chandy-Lamport snapshots in terms of messaging, bandwidth, and state storage.

With the theoretical properties of LDPs well in hand, we began the design of a simple programming language, based on the concept of distributed predicate search. This language, which we called $\mathcal{L}$, allowed for the concise expression of a wide variety of distributed predicates. We extended the basic syntax with support for managed storage, triggered actions, aggregate variables, and temporal quantifiers. In so doing, we created a distributed, reactive programming language capable of responding to state variables, system topology, and historical data.

By implementing a number of distributed algorithms for different problem domains in $\mathcal{L}$, we hoped to show the applicability of the language (and the underlying predicate search technique) to different distributed applications. In so doing, we also highlighted the advantages of $\mathcal{L}$ over traditional implementation choices such as C or Java. One of the chief advantages of $\mathcal{L}$ is its ability to express distributed algorithms in an extremely concise manner. This brevity not only makes understanding and modifying the code easier, it also makes it less intertwined, as there is no brittle scaffolding code for such tasks as message serialization, marshaling, or communication exceptions. The availability of the hybrid coding mechanism in $\mathcal{L}$ allows for a clean separation between distributed code and performance-sensitive local code, and keeps the syntax of $\mathcal{L}$ both simple and expressive.

In order to execute programs in $\mathcal{L}$, we constructed a naïve interpreter runtime for the language. While this runtime was too inefficient for use in any embedded environment, it did provide us with an opportunity to explore the fundamental algorithms and data structures used by $\mathcal{L}$. Chief among these were PatternMatchers, data structures which we used to encapsulate a single predicate search instance within the distributed system. We then created an optimized runtime, based on the idea of an application-specific bytecode engine. This runtime was produced by compiling a partial implementation of a bytecode-based runtime with machine-generated headers which provided both variable storage declarations and the program bytecode stream. The resulting compiled binary executes a given $\mathcal{L}$ program in a much more memory- and CPU-efficient manner than the original runtime. We then implemented two simple optimizations: boolean short-circuiting and topology culling, to further improve the efficiency of the runtime.

To complete the evaluation of LDPs and $\mathcal{L}$, we analyzed the performance of the runtime system using a massively multithreaded robot simulator. We examined the performance of $\mathcal{L}$ over various metrics, and in various experimental setups. We quantified the effect of decay factors and short-circuiting, illustrating the extreme differences in efficiency between early- and late-terminating predicates. We compared the performance of a data aggregation algorithm in $\mathcal{L}$ to the ideal minimum, and determined that, while the broadcast nature of predicate search results in certain unavoidable inefficiencies, the resulting performance is still quite acceptable. Finally, we showed the utility of $\mathcal{L}$ in implementing, comparing, and improving two modular robotic planning algorithms.

LDPs and $\mathcal{L}$ are first steps in the attempt to incorporate topology into distributed programming. It is our hope that the research presented in this dissertation provides some insight into the problems of distributed state, concurrency, and asynchronicity that make distributed programming so difficult.

Appendix A

Tree-Locking Algorithm

```
Each process $P_j$ initializes local variable $max\_nonce_j = 0$;
Each process $P_j$ initializes local variable $locked_j = 0$;

Initiator process $P_i$ :
Create spanning tree $T_i$ of depth $| Q |$, and positive unique nonce $n_i$;
$max\_nonce_i = n + i$ ;
Send $\langle Prepare, T_i, n_i \rangle$ to $c(T_i, P_i)$;

Each internal node $P_j$ of $T_i$, after receiving $\langle Prepare, T_i, n_i \rangle$ :
if $locked_j = 0$ then
    $max\_nonce_j = \max(max\_nonce_j, n_i)$;
end
Send $\langle Prepare, T_i, n_i \rangle$ to $c(T_i, P_j)$;

Each leaf node $P_j$ of $T_i$, after receiving $\langle Prepare, T_i, n_i \rangle$ :
if $locked_j = 0$ then
    $max\_nonce_j = \max(max\_nonce_j, n_i)$;
end
Send $\langle PrepComplete, T_i, n_i \rangle$ to $p(T_i, P_j)$;

Each internal node $P_j$ of $T_i$, after receiving $\langle PrepComplete, T_i, n_i \rangle$ from all $c(T_i, P_j)$ :
Send $\langle PrepComplete, T_i, n_i \rangle$ to $p(T_i, P_j)$;

Initiator process $P_i$ receiving $\langle PrepComplete, T_i, n_i \rangle$ from all $c(T_i, P_i)$ :
Send $\langle Lock, T_i, n_i \rangle$ to $c(T_i, P_i)$;
```

Algorithm 6: Subtree Locking Protocol

```
Each internal node $P_j$ of $T_i$, after receiving $\langle Lock, T_i, n_i\rangle$ :
if $(locked_j = 0) \wedge (max\_nonce_j = n_i)$ then
    $locked_j = 1$;
    Send $\langle Lock, T_i, n_i\rangle$ to $c(T_i, P_j)$;
else
    Send $\langle Fail, T_i, n_i\rangle$ to $p(T_i, P_j)$;
end

Each leaf node $P_j$ of $T_i$, after receiving $\langle Lock, T_i, n_i\rangle$ :
if $(locked_j = 0) \wedge (max\_nonce_j = n_i)$ then
    $locked_j = 1$;
    Send $\langle Pass, T_i, n_i\rangle$ to $p(T_i, P_j)$;
else
    Send $\langle Fail, T_i, n_i\rangle$ to $p(T_i, P_j)$;
end

Each internal node $P_j$ of $T_i$, after receiving $\langle Pass, T_i, n_i\rangle$ from all $c(T_i, P_j)$ :
Send $\langle Pass, T_i, n_i\rangle$ to $p(T_i, P_j)$;

Each internal node $P_j$ of $T_i$, after receiving $\langle Fail, T_i, n_i\rangle$ from any $c(T_i, P_j)$ :
Send $\langle Fail, T_i, n_i\rangle$ to $p(T_i, P_j)$;

Initiator process $P_i$ after receiving $\langle Pass, T_i, n_i\rangle$ from all $c(T_i, P_i)$ :
if $(locked_i = 0) \wedge (max\_nonce_i = n_i)$ then
    $locked_i = 1$;
    Lock acquired, perform any additional operations ;
    $locked_i = 0$;
    $max\_nonce_i = 0$;
    Send $\langle Release, T_i, n_i\rangle$ to $c(T_i, P_i)$;
else
    Send $\langle Release, T_i, n_i\rangle$ to $c(T_i, P_i)$;
    Perform random exponential backoff and restart algorithm;
end
```

Algorithm 7: Subtree Locking Protocol (part 2)

```
Initiator process $P_i$ after receiving $\langle Fail, T_i, n_i \rangle$ from any $c(T_i, P_i)$ :
$max\_nonce_i = 0$;
Send $\langle Release, T_i, n_i \rangle$ to $c(T_i, P_i)$;

Any non-root node $P_j$ of $T_i$ after receiving $\langle Release, T_i, n_i \rangle$ :
if $(locked_j = 1) \wedge (max\_nonce_j = n_i)$ then
    $locked_j = 0$;
    $max\_nonce_j = 0$;
else
    if $(max\_nonce_j = n_i$ then
        $max\_nonce_j = 0$;
    end
end
Send $\langle Release, T_i, n_i \rangle$ to all neighboring $P_k \in (T_i - p(T_i, P_j))$;
```
Algorithm 8: Subtree Locking Protocol (part 3)

Appendix B

Metamodule Planner

Listing B.1: Metamodule Shape Planner

```
// state types
const scalar FINAL = 0;
const scalar PATH = 1;
const scalar NEUTRAL = 2;

// operation types
const scalar NOP = 0;
const scalar CREATE = 1;
const scalar DELETE = 2;
const scalar TRANSFER = 3;

const scalar LEADER = 0;
const scalar INV_ID = -1; // invalid module id number

scalar random;  // random 1-bit number
scalar id; // unique id of the module
scalar time; // current tick number

scalar state = NEUTRAL;
scalar inside; // inside target shape? (provided by shape description)
scalar isSeed; // is this module the seed for the planner? (provided by shape description)

scalar role; // initial role determined by position
// the following variables are valid on leader only
scalar depth = INT_MAX; // depth of this MM in the deletion forest
scalar parent = INV_ID; // parent of this MM in the deletion forest
scalar gradient; // resource movement gradient for this MM
scalar hasRsc = 0; // does this metamodule have a resource?
set hlNeighbors = nil; // set of neighboring metamodule leaders
set notChildOf = nil; // set of neighboring metamodules that are not children
set llChildrenDone = nil; // set of child modules that have completed microplan

// low-level variables for microplans
scalar llParent; // parent in MM, initial parent determined by initial position
scalar llAction = NOP; // microplan being executed
scalar llActionDir = 0; // orientation of the microplan
scalar llMoveDir = 0; // direction for module to move (in microplan)
scalar llStepNum = -1; // current step in microplan
scalar llNextParent = INV_ID; // new MM leader once microplan terminates

// multi-party locking protocol
scalar llOperationLocked = INV_ID; // other party in lock attempt
scalar isPassive = 0; // is this module a lock initiator?
scalar llOpTimestamp = 0; // timestamp for lock attempt
scalar llProposedOp = NOP; // proposed operation to perform

// --------------------- functions ----------------------------------------------------

// attempt to move the module in the specified direction,
// if move succeeds increment llStepNum and update llMoveDir
scalar tryToMove(scalar direction);

// initialize llStepNum, llMoveDir, llActionDir for create operation in given direction
scalar setUpCreate(scalar direction);

// set llActionDir based on direction of target module
scalar setActionDir(scalar targetId);

// initialize llStepNum, llMoveDir for given action, action direction, and role
scalar setUpAction(scalar action, scalar actionDirection, scalar role);

// sets new role and llParent for module once plan is complete
scalar setNewRole(scalar action, scalar actionDirection,
                  scalar currentRole, scalar nextParent);

// ----------------------------------------------------------------------------------

// update the hlNeighbors variable
forall (a,b,c) where (a.role == LEADER) & (c.role == LEADER)
               do c.HLNeighbors += a.id;
forall (a,b,c) where (a.role == LEADER) & (a.llAction == NOP) &
                     (a.llOperationLocked != 0) & (a.llProposedOp == DELETE) &
                     (a.isPassive == 0) & (c.role == LEADER)
               do c.HLNeighbors += a.id;

// if a is the seed, its state becomes FINAL
forall (a) where (a.isSeed == 1) & (a.state != FINAL) do a.state = FINAL, a.depth = 0;

// if a is in state FINAL, any neighbors that are inside the desired shape become FINAL
forall (a,b) where (a.state == FINAL) & (b.inside == 1) & (b.state != FINAL)
             do b.state = FINAL, b.depth = 0;

// if c is outside the desired shape, and next to a MM that is FINAL, it becomes PATH
forall (a,b,c) where (a.role == LEADER) & (a.state == FINAL) & (c.role == 0) &
                     (c.inside == 0) & (c.state == NEUTRAL)
               do c.state = PATH;

// if a is looking for a path, and c is not final, c begins looking for a path
forall (a,b,c) where (a.role == 0) & (a.state == PATH) &
                     (c.role == 0) & (c.state == NEUTRAL)
               do c.state = PATH;


```

```
// if c is looking for a path and has no parent,
// and a either has a parent or is final, a becomes c's parent
forall (a,b,c) where (c.role == 0) & (c.state == PATH) & (c.curr.parent == -1) &
                     (c.curr.llOperationLocked == 0) & (a.role == LEADER) &
                     ((a.parent != -1) | (a.state == FINAL))
               do c.parent = a.id;

// if c has a parent and it is not a, add c to the notChild set of a
forall (c,b,a) where (c.role == LEADER) & (c.parent != -1) &
                     (a.role == LEADER) & (c.parent != a.id)
               do a.notChild += c.id;

// if c is final, add c to the notChild set of a
forall (c,b,a) where (c.role == LEADER) & (c.state == FINAL) & (a.role == LEADER)
               do a.notChild += c.id;

// if our depth just changed, potentially update all nearby depths
forall (a,b,c) where (a.depth < a.prev.depth) & (a.role == LEADER) &
                     (c.role == LEADER) & (c.curr.depth > a.depth + 1)
               do c.depth = a.depth + 1;

// ---------------------------- locking protocol ----------------------------------------

// locking begins by a setting llOperationLocked, llProposedOp,
// and isPassive on remote leader c

// when c becomes locked, it tries to lock a in turn,
// setting llOperationLocked, llProposedOp, and isPassive
forall (c,b,a) where (c.role == LEADER) & (c.isPassive == 1) &
                     (a.curr.llOperationLocked == 0) &
                     (c.llOperationLocked != c.prev.llOperationLocked) &
                     (c.llOperationLocked != 0) &
                     (c.llOperationLocked != c.id) & (c.llOperationLocked == a.id)
               do a.llOperationLocked = c.id, a.llProposedOp = c.llProposedOp,
                  a.isPassive = 0;

// c also broadcasts to all neighbors (n) that it is negotiating with a,
// and that they should release any attempts on c
forall (c,b,n) where (c.role == LEADER) & (c.isPassive == 1) & (c.llOperationLocked != 0) &
                     (c.llOperationLocked != c.id) &
                     (c.llOperationLocked != c.prev.llOperationLocked) &
                     (c.llOperationLocked != n.id) & (n.curr.llOperationLocked == c.id)
               do n.isPassive = 0, n.llOperationLocked = 0, n.llProposedOp = 0;

// if locking fails, c unsets llOperationLocked, llProposedOp, and isPassive
forall (c,b,a) where (c.role == LEADER) & (c.isPassive == 1) &
                     (a.curr.llOperationLocked != c.id) &
                     (c.llOperationLocked != c.prev.llOperationLocked) &
                     (c.llOperationLocked != 0) & (c.llOperationLocked != c.id) &
                     (c.llOperationLocked == a.id) & (a.curr.llOperationLocked != 0) &
               do c.llOperationLocked = 0, c.isPassive = 0, c.llProposedOp = 0;

// locking is complete on a if a.llOperationLocked == c.id,
// a.llProposedOp != 0, and a.isPassive == 0
// ---------------------------- creation protocol ----------------------------------------
// if a has any free locations where it wants to create a new metamodule,
// it locks itself and finds the new parent
forall (a) where (a.role == LEADER) &
                 (a.llOperationLocked == 0) &
                 (a.curr.llOperationLocked == 0) & (a.state == FINAL) &
                 (a.size(a.freeSpaces) > 0) & (a.rsc == 1)
           do a.rsc = 0, a.llOperationLocked = a.id,  a.llProposedOp = CREATE,
              a.setUpCreate(a.any(a.freeSpaces)));

// find the new parent, then start the action
forall (a,b[1,3],c) where (a.role == LEADER) &
                          (a.llOperationLocked != a.prev.llOperationLocked) &
                          (a.curr.llOperationLocked == a.id) & (b[i].llParent == a.id) &
                          (c.llParent == a.id) & (a.newParentRole == c.role) &
                          (a.llProposedOp == CREATE)
                    do a.llAction = 1, a.llStepNum = -1, a.llNextParent = c.id,
                       a.childDone = null, a.llOpTimestamp = a.time;

// ----------------------resource transfer protocol (a wants to give resource to c) ------

// a initiates by locking c to it
forall (a,b,c) where (a.role == LEADER) & (a.llOperationLocked == 0) &
                     (a.curr.llOperationLocked == 0) & (a.rsc == 1) & (c.role == LEADER) &
                     (c.curr.llOperationLocked == 0) & (c.curr.rsc == 0) &
                     (a.random == c.random) &
                     ((c.state == FINAL) | (c.depth < a.depth) | (a.parent == c.id))
               do c.isPassive = 1, c.llOperationLocked = a.id, c.llProposedOp = TRANSFER;

// at this point, locking has succeeded. a can now set its flags to initiate the action
forall (a) where (a.role == LEADER) & (a.llAction == 0) & (a.llOperationLocked != 0) &
                 (a.llProposedOp == TRANSFER) & (a.isPassive == 0)
           do a.llAction = TRANSFER, a.llStepNum = -1,
              a.llNextParent = a.llOperationLocked,
              a.childDone = null, a.setActionDir(a.llOperationLocked),
              a.rsc = 0, a.llOpTimestamp = a.time;

// ------------------- deletion protocol (a wants to delete towards c) -------------------

// a initiates by locking c to it
forall (a,b,c) where (a.role == LEADER) & (a.llOperationLocked == 0) &
                     (a.curr.llOperationLocked == 0) & (a.state == PATH) & (a.rsc == 0) &
                     (a.parent == c.id) &  (a.size(a.hlNeighbors) > 0) &
                     (a.size(a.hlNeighbors) == a.size(
                         a.intersect(a.notChildOf, a.hlNeighbors))) &
                     (a.random == 1) & (c.role == LEADER) &
                     (c.curr.llOperationLocked == 0) & (c.curr.rsc == 0)
               do c.llOperationLocked = a.id, c.isPassive = 1, c.llProposedOp = DELETE;

// at this point, locking has succeeded. a can now set its flags to initiate the action
forall (a) where (a.role == LEADER) & (a.llAction == 0) & (a.llOperationLocked != 0) &
                 (a.llProposedOp == DELETE) & (a.isPassive == 0)
           do a.llAction = DELETE, a.llStepNum = -1, a.llNextParent = a.llOperationLocked,
              a.childDone = null, & a.setActionDir(a.llOperationLocked),
              a.parent = -1, a.llOpTimestamp = a.time;
```

```
// -------------------- low level move ---------------------------------------------------
// propagate action to children
forall (a,n[0,2],b) where (a.role == LEADER) & (a.llAction != a.prev.llAction) &
                          (a.llStepNum == -1) & (b.llParent == a.id) &
                          (n[i].llParent == a.id)
                    do b.llAction = a.llAction, b.llActionDir = a.llActionDir,
                       b.llNextParent = a.llNextParent, b.llOpTimestamp = a.llOpTimestamp;

// start action on children. result: step set to 0, moveDir set appropriately
forall (a) where (a.llAction != 0) & (a.llStepNum == -1)
           do a.setUpAction(a.llAction, a.llActionDir, a.role);

// if move succeeds, step incremented and moveDir is set appropriately
// (this will keep looping until a's motion is done)
forall (a); (a.llMoveDir != 0) & (a.llStepNum != -1) do a.tryToMove(a.llMoveDir);

// -------------------- move complete notification ----------------------------------------

// move is done, inform parent
forall (a) where (a.llMoveDir == 0) & (a.llStepNum != -1) & (a.role == 0)
           do a.childDone += a.id;
forall (b,n[0,4],a) where (b.llMoveDir == 0) & (b.llStepNum != -1) & (b.role != 0) &
                          (b.llParent == a.id) & (a.curr.llOpTimestamp == b.llOpTimestamp)
                    do a.childDone += b.id;

// turn off notification once we've done it, and (potentially) swap to new parent/role
forall (a) where (a.role != 0) & (a.llMoveDir == 0) & (a.llStepNum != -1)
           do a.setNewRole(a.llAction, a.llActionDir, a.role, a.llNextParent),
              a.llAction = NOP, a.llStepNum = -1;

// parent has been notified. If we're still a parent, unlock higher-level functioning.
// (for create/transfer, delete handled below)
forall (a) where (a.role == 0) & (a.size(a.llChildrenDone) == 8)
           do a.llAction = NOP, a.llStepNum = -1, a.llOperationLocked = 0,
              a.isPassive = 0, a.childDone = null, a.llProposedOp = NOP;

// notify and unlock other party of action (for transfer)
forall (a,b,c) where (a.role == 0) & (a.prev.llAction == TRANSFER) &
                     (a.llOperationLocked == 0) &
                     (a.llOperationLocked != a.prev.llOperationLocked) &
                     (a.prev.llOperationLocked == c.id) & (c.curr.llOperationLocked == a.id)
                do c.rsc = 1, c.llOperationLocked = 0, c.isPassive = 0, c.llProposedOp = NOP;

// notify and unlock new master (for creation)
forall (a) where (a.role == 0) & (a.prev.role > 0)
           do a.rsc = 0, a.llOperationLocked = 0, a.isPassive = 0, a.llProposedOp = NOP;

// unlock and add resource to other party (for delete)
forall (a) where (a.role == 0) & (a.isPassive == 0) & (a.llAction == DELETE) &
                 (a.size(a.llChildrenDone) == 4)
           do a.setNewRole(a.llAction, a.llActionDir, a.role, a.llNextParent),
              a.llAction = NOP, a.llStepNum = -1, a.llOperationLocked = 0,
              a.childDone = null, a.isPassive = 0, a.llProposedOp = NOP;
forall (a,b[1,3],c) where (a.role == 0) & (a.isPassive == 0) & (a.llAction == DELETE) &
                          (a.size(a.llChildrenDone) == 4) &
                          (c.curr.llOperationLocked == a.id) &
                          (a.llOperationLocked == c.id)
            do c.rsc = 1, c.isPassive = 0, c.llOperationLocked = 0, c.llProposedOp = NOP;
```

Bibliography

[1] *X3.135-1986 American National Standard Database Language SQL.* American National Standards Institute, January 1986. 6.1

[2] Serge Abiteboul, Zoë Abrams, Stefan Haar, and Tova Milo. Diagnosis of asynchronous discrete event systems: datalog to the rescue! In *PODS '05: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 358–367, New York, NY, USA, 2005. ACM. 6.1

[3] G. Anastasi, A. Falchi, A. Passarella, M. Conti, and E. Gregori. Performance measurements of motes sensor networks. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 174–181, New York, NY, USA, 2004. ACM. 4.2

[4] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, 2004. 6.1

[5] M. Ashley-Rollman, Michael De Rosa, S. Srinivasa, P. Pillai, S. Goldstein, and J. Campbell. Declarative programming for modular robots. In *IROS 2007 Workshop on Modular Robots*, 2007. 3.6.1

[6] 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. 6.1

[7] Özalp Babao, Keith Marzullo, and Keith Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In *Distributed Systems*, pages 55–96. Addison-wesley, 1993. 2

[8] Roberto Baldoni and Michel Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. In *Distributed Systems Online*, February 2002. 2.2

[9] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 205–217, New York, NY, USA, 2002. ACM. 3.2.1

[10] Michael Bauland, Martin Mundhenk, Thomas Schneider, Henning Schnoor, Ilka Schnoor, and Heribert Vollmer. The tractability of model-checking for ltl: The good, the bad, and the ugly fragments. Technical report, Electronic Colloqium on Computational Complexity, 2008. 6.1

[11] U.C. Berkeley, The tinyos project, http://webs.cs.berkeley.edu/tos/. 4.2

[12] R. Braden. Requirements for internet hosts – communication layers. RFC 1122, 1989. 2.6.5

[13] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989. 6.1

[14] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states in distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985. 1.2, 2, 2.2, 2.5.2

[15] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, New York, NY, USA, 2007. ACM. 6.1

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. 6.1

[17] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, volume 26, pages 167–174, 1991. 2

[18] Microsoft Corporation, Axum programmer's guide, http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx. 6.1

[19] Carol Critchlow and Kim Taylor. The inhibition spectrum and the achievement of causal consistency. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 31–42, New York, NY, USA, 1990. ACM. 2.5.1

[20] Y. K. Dalal. A distributed algorithm for constructing minimal spanning trees. *IEEE Trans. Softw. Eng.*, 13(3):398–405, 1987. 3.2.1

[21] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason Campbell, and Padmanabhan Pillai. Distributed watchpoints: Debugging large modular robotic systems. *International Journal of Robotics Research*, 27(3), Special Issue on Modular Robotics 2008. 6.1

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. 6.1

[23] 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. 3.6.2

[24] Stefan Dobrev and Andrzej Pelc. Leader election in rings with nonunique labels. *Fundam. Inf.*, 59(4):333–347, 2003. 3.1.3

[25] D. Duff, M. Yim, and K. Roufas. Evolution of polybot: A modular reconfigurable robot. In *Proc. of COE/Super-Mechano-Systems Workshop*, 2001. 3.3.1

[26] Georgios E. Fainekos and Hadas Kress-gazit. Temporal logic motion planning for mobile robots. In *In Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pages 2020–2025, 2005. 6.1

[27] C. Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug 1991. 2.2

[28] FSF. *GDB: The GNU Project Debugger*. Free Software Foundation, 2007. 3.1.1, 5.1.3

[29] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. In *ACM Transactions on Programming Languages and Systems*, volume 5, pages 66–77, January 1983. 1.2

[30] H. Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, C-31(1):48–59, Jan. 1982. 1.2

[31] V. K. Garg and J. R. Mitchell. Distributed predicate detection in a faulty environment. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 416, Washington, DC, USA, 1998. IEEE Computer Society. 2.6.5

[32] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification*, 2001. 6.1

[33] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985. 6.1, 6.2

[34] Shay Halperin and Uri Zwick. Optimal randomized erew pram algorithms for finding spanning forests. In *J. Algorithms*, pages 438–447, 2000. 5.2

[35] T. A. Henzinger, Z. Manna, and A. Pnueli. An interleaving model for real-time. In *Information Technology, 1990. 'Next Decade in Information Technology', Proceedings of the 5th Jerusalem Conference on (Cat. No.90TH0326-9)*, pages 717–730, 1990. 3.1.2

[36] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003. 6.1

[37] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6):635 – 652, 1999. 6.2

[38] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007. 6.1

[39] A. Kshemkalyani and Bin Wu. Detecting arbitrary stable properties using efficient snapshots. *Software Engineering, IEEE Transactions on*, 33(5):330–346, May 2007. 2.6

[40] A D Kshemkalyani, M Raynal, and M Singhal. An introduction to snapshot algorithms in distributed computing. *Distributed Systems Engineering*, 2(4):224–233, 1995. 2, 2.2

[41] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008. 6.1

[42] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Inf. Process. Lett.*, 25(3):153–158, 1987. 2.2

[43] 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. 6.1

[44] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. 2

[45] P. Lee and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. In *Seventeenth Annual International Computer Software and Applications Conference*, pages 224 – 230, 1993. 2.6.5

[46] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association. 2.6.5, 6.1

[47] Qun Li, Michael De Rosa, and Daniela Rus. Distributed algorithms for guiding navigation across a sensor network. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 313–325, New York, NY, USA, 2003. ACM. 3.5.1

[48] 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. 6.1

[49] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: language, execution and optimization. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, New York, NY, USA, 2006. ACM. 6.1

[50] Steven Lucco and Oliver Sharp. Delirium: an embedding coordination language. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 515–524, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. 6.1

[51] Steven E. Lucco. Parallel programming in a virtual object space. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 26–34, New York, NY, USA, 1987. ACM. 6.1

[52] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005. 6.1

[53] Amit Manjhi, Suman Nath, and Phillip 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*, pages 287–298, New York, NY, USA, 2005. ACM Press. 3.4.1

[54] Keith Marzullo and Laura S. Sabel. Efficient detection of a class of stable properties. *Distrib. Comput.*, 8(2):81–91, 1994. 2.2

[55] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 250–262, New York, NY, USA, 2004. ACM Press. 3.4.1, 3.7.2

[56] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003. 5.1.3

[57] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM Press. 6.1

[58] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu. The broadcast storm problem in a mobile ad hoc network. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162, New York, NY, USA, 1999. ACM. 3.2.1

[59] Ron Obermarck. Distributed deadlock detection algorithm. *ACM Trans. Database Syst.*, 7(2):187–208, 1982. 2, 3.1.2

[60] Songhwai Oh, Phoebus Chen, Michael Manzo, and Shankar Sastry. Instrumenting wireless sensor networks for real-time surveillance, April 2006. Poster given at Trust NSF Site Visit. 6.1

[61] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. 6.1, 6.2

[62] Miguel A. Ortuno Perez, Vicente M. Olivera, Luis Rodero Merino, and Gregorio Robles Martinez. Abbreviated dynamic source routing: Source routing with non-unique network identifiers. *Wireless on Demand Network Systems and Service, International Conference on*, 0:76–82, 2005. 3.1.3

[63] Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. *SIGCOMM Comput. Commun. Rev.*, 15(4):44–53, 1985. 3.2.1

[64] Intel Research Pittsburgh, Dprsim, http://www.pittsburgh.intel-research.net/dprweb/. 4.1, 5, 5.3

[65] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977. 3.5.2, 6.1

[66] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mob. Netw. Appl.*, 8(4):427–442, 2003. 6.1

[67] Michel Raynal. Illustrating the use of vector clocks in property detection: An example and a counter-example. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 806–814, London, UK, 1999. Springer-Verlag. 2.1, 2.2

[68] Georgios Rodolakis, Amina Meraihi Naimi, and Anis Laouiti. Multicast overlay spanning tree protocol for ad hoc networks. In *WWIC '07: Proceedings of the 5th international conference on Wired/Wireless Internet Communications*, pages 290–301, Berlin, Heidelberg, 2007. Springer-Verlag. 3.2.1

[69] Naoshi Sakamoto. Comparison of initial conditions for distributed algorithms on anonymous networks. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 173–179, New York, NY, USA, 1999. ACM. 3.1.3

[70] Behnam Salemi, Mark Moll, and Wei-Min 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. 3.3.1

[71] André Schiper and Alain Sandoz. Strong stable properties in distributed systems. *Distrib. Comput.*, 8(2):93–103, 1994. 2.2, 2.4

[72] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001. 6.1

[73] Kasper Støy. *Emergent Control of Self-Reconfigurable Robots*. PhD thesis, AdapTronics Group, The Maersk Mc-Kinney Moller Institute for Production Technology, University of Southern Denmark, 2003. 5.3.4

[74] Kasper Støy. Controlling self-reconfiguration using cellular automata and gradients. In *Proceedings of the 8th international conference on intelligent autonomous systems (IAS-8)*, pages 693–702, March 2004. 3.7.1

[75] Kasper Støy and R. Nagpal. Self-repair through scale independent self-reconfiguration. In *Proceedings of IEEE/RSJ International Conference on Robots and Systems, (IROS)*, pages 2062–2067, 2004. 3.7.1

[76] Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. In *ACM Transactions on Programming Languages and Systems*, volume 15, pages 1–35. ACM Press, January 1993. 2, 3.1.2

[77] S. Upadhyayula and S. K. S. Gupta. Spanning tree based algorithms for low latency and energy efficient data aggregation enhanced convergecast (dac) in wireless sensor networks. *Ad Hoc Netw.*, 5(5):626–648, 2007. 3.2.1

[78] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65nm cmos. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 98–589, Feb. 2007. 6.2

[79] S. Venkatesan. Message-optimal incremental snapshots. In *Distributed Computing Systems, 1989., 9th International Conference on*, pages 53–60, Jun 1989. 2.6

[80] David H D Warren. Prolog: The language and its implementation compared with lisp. In *ACM SIGPLAN Notices*, pages 109–115, 1977. 6.1

[81] Zhang 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. 3.3.1