Carnegie Mellon University

Computer Science Department

Thesis Proposal

Doctor of Philosophy

Title:

Meld: A Logical Approach to Programming Ensembles

Date of Submission: ????

Submitted by: Michael Ashley-Rollman

Supervisors: Professor Seth Goldstein
School of Computer Science, Carnegie Mellon University

Professor Frank Pfenning
School of Computer Science, Carnegie Mellon University

Committee Members: Professor David Andersen
School of Computer Science, Carnegie Mellon University

Professor Peter Lee
School of Computer Science, Carnegie Mellon University

Professor Radhika Nagpal
School of Engineering and Applied Sciences
& Wyss Institute for Bioinspired Engineering, Harvard University

Contents

1 Introduction **5**

2 Meld by Example **7**

2.1 Walk Example 7

2.2 Routing Example 13

2.3 Locality 14

2.4 Parallelism 14

2.5 Fault Tolerance 16

3 Literature Review **16**

3.1 Ensemble Languages 16

3.2 Other Concurrent Languages 19

4 Meld v1.0 Semantics: Language and Meaning **20**

4.1 Structure of a Meld v1.0 Program 21

4.2 Meaning of a Meld v1.0 Program 23

5 Distributed Implementation of Meld v1.0 programs **25**

5.1 Basic Distribution/Implementation Approach 25

5.2 Triggered Derivations 27

5.3 Deletion 27

5.4 Concerning Deletion and Actions 30

5.5 X-Y Stratification 31

5.6 Implementation Targets of Meld v1.0 32

5.6.1 DPRSim/DPRSim2 32

5.6.2 Blinky Blocks 33

5.6.3 Multi-core Machines 33

6 Analysis of Meld v1.0 **33**

6.1 Fault Tolerance 35

6.2 Provability 36

6.3 Messaging Efficiency 36

6.4 Memory Efficiency 37

7 Shortcomings of Meld v1.0 **38**

8 Proposed New Research for Meld v2.0 **40**

9 Timeline **42**

9.1 Already Done 42

9.2 Summer 2011 42

9.3 Fall 2011 42

9.4 Spring 2012 43

10 Summary **43**

Abstract

Concurrent systems are notoriously difficult to program. They suffer from such problems as race conditions, deadlocks, and livelocks in addition to all the bugs encountered in sequential programs. Better methods are necessary to write correct programs for concurrent systems. In this thesis I will focus on a class of concurrent systems call 'ensembles'. An ensemble is a massively distributed system in which the topology varies over time. Modular robotic systems and sensor networks are examples of ensembles.

I posit that the logic programming paradigm is a good match for programming ensembles. A logic program is an inherently parallel representation. Each rule of a logic program expresses a single unit of computation. Rules can be evaluated in any order that preserves the dependencies between them. This implicit parallelism allows the programmer to focus on the program while the compiler and run-time distribute it across an ensemble.

Thesis Statement: Logic programming is well-suited for ensemble programming. It permits us to express algorithms in a way that can be executed efficiently and reasoned about easily.

In support of this thesis, I have designed and implemented Meld v1.0 [1], an expressive logic programming language for concurrent systems. Meld programs are distributed across an ensemble by the compiler and run-time, making use of the implicit parallelism in the logic programming representation. Meld programs are concise and amenable to proof, as demonstrated in the proofs of correctness of Dewey et al.'s meta-module planner [2], which showed the planner to be free of race conditions, deadlocks, and livelocks. Meld v1.0 programs already run on multi-million-robot simulations of Claytronics [3], Blinky Blocks [4], and multi-core machines.

In this thesis, I propose Meld v2.0 for programming ensembles. I will extend Meld v1.0 to be more expressive (details in proposal document). I will explore the logical foundations of Meld, providing a clear definition of Meld v2.0's semantics and an understanding of Meld's basis in logic. I will prove correctness properties of other Meld programs. And I will analyze the fault tolerance of Meld programs.

1 Introduction

Tremendous advances have been made in networks and distributed hardware. Personal computers are no longer isolated devices, but now maintain a constant connection to other devices via local area networks and the Internet. Other devices like cellular phones, televisions, and game consoles are increasingly joining PCs on networks. Applications are being hosted in the cloud, permitting people access to their documents from any computer. Cars and airplanes contain many computers connected via an internal network. Sensor networks are being researched to gather and aggregate data. Robots are being developed in a modular fashion to allow them to be reconfigured for different applications and purposes. Taken to the extreme, they may form programmable matter [3], a material that can be programmed to change it's shape and color. Distributed systems, in various forms, are becoming increasingly prevalent. Furthermore, multi-core systems, previously confined to expensive machines for hosting servers and performing scientific computing are now common place.

While concurrent systems have multiplied, our programming models for them have failed to keep pace. Most programs are still written with the threading model for expressing parallel computation. This model is fraught with peril as a programmer must be wary of such concurrency bugs as race conditions, deadlock, and livelock. These concurrency bugs occur erratically depending upon the particular order in which concurrent code happens to execute, making them notoriously difficult to catch and eliminate. Any programming model that can alleviate or eliminate these sorts of errors greatly simplifies the problem of programming distributed systems. Indeed, a variety of approaches have been presented for some classes of problems, especially those of data-parallel algorithms. Some of these models are discussed in §3.

One of the biggest problems with the threading model is that it requires explicit parallel constructs in an artificially sequentialized version of an algorithm. Typical imperative and functional programming languages require that a set of instructions be given along with a precise ordering for those instructions to execute. The threading model, along with many other proposed models, provides a mechanism for teasing apart this sequential code into a set of independent sequential blocks to be executed simultaneously. This works well only when the blocks are really independent, such as for data-parallel algorithms. For other algorithms, when the data dependencies follow more

arbitrary graph structures, this approach deteriorates.

The logic programming paradigm, however, allows a programmer to easily specify parallelism in the form of an arbitrary graph of data dependencies. Logic programs are specified as rules which are small pieces of computation that can be run in arbitrary order so long as the implicit data dependencies are satisfied. This mechanism is discussed in greater detail in §2, but the high level point is that logic programs naturally express the parallelism present in a program and are adept at handling arbitrary dependency graphs. They are not limited to largely independent threads of control. Furthermore, the logic program representation is one that has been greatly studied and can be reasoned about via the rules of logic. This allows us to show that particular individual logic programs are correct, free of any of the concurrency bugs we discussed before.

The question, then, is if logic programs are the right representation to express parallel algorithms and show that they are correct, can we efficiently make use of their parallelism? The answer to this question turns out to be 'yes', for a class of distributed systems called 'ensembles'. An ensemble is a distributed system in which the network topology can change dynamically and the nodes are relevant to the computation. Examples include sensor networks, modular robotic systems, and Claytronics [3] (a particular modular robotic system containing millions of robots).

Thesis Statement: Logic programming is well-suited for ensemble programming. It permits us to express algorithms in a way that can be run efficiently and reasoned about easily.

In the remainder of this document I discuss a logic programming language, Meld v2.0, which is designed for this purpose. I provide examples of what logic programs for ensembles look like in §2. In §4 I define the existing Meld v1.0 language and then explain how to implement it on an ensemble in §5. Next I evaluate Meld v1.0, in §6, discussing both its successes and its shortcomings. In §8, I propose a plan for addressing these shortcomings in Meld v2.0.

A point of clarification for the reader: there are two versions of Meld discussed throughout this proposal. The term "Meld v1.0" is used in reference to the current published implementation of Meld while "Meld v2.0" is used for the new version proposed here. "Meld" is reserved for statements that apply equally to either version and do not need to be disambiguated.

RULE1:
```
dist(S,D_S):-
    at(S,P),
    destination(P_d),
    D_S = |P - P_d|,
    D_S > module radius.
```

RULE2:
```
farther(S,T):-
    neighbor(S,T),
    dist(S,D_S),
    dist(T,D_T),
    D_S ≥ D_T.
```

RULE3:
```
moveAround(S,T,P):-
    farther(S,T),
    vacant(T,P),
    dist(S,D_S),
    destination(P_d),
    D_S > |P - P_d|.
```

Figure 1: A first cut of the walk program.

2 Meld by Example

"Logic programming" is a broad term that can be used to describe any use of mathematical logic for programming computers. There is a plethora of different logics that one might use, varying on such components as what structural rules are permitted and rejected, what modalities are included, etc. In logic programming this is further complicated by the potential addition of super-logical operations (such as aggregation) and implementation details. Here I clarify what is meant by "logic programming" in this context by first walking through a few examples. The full details and semantics of the Meld v1.0 logic are discussed in §4.

2.1 Walk Example

The first example is Walk. Walk is a program for moving an ensemble of modular robots, such as those envisioned by the Claytronics [3] project. The Claytronics model assumes that each module is a sphere which is capable of rolling over or around an adjacent module. In this model, each module is able to communicate directly with exactly those other modules it is in physical contact with (modulo hardware failures).

A Meld program consists of a set of rules for generating new facts from an existing set of true

Initial facts:

```
destination(<0,5>)
at(s, <0,0>)
at(t, <0,1>)
neighbor(s,t)
neighbor(t,s)
vacant(t, <0,2>)
vacant(t, <1,1>)
...
```

Facts derived by RULE1

```
dist(s, 5)
dist(t, 4)
```

Facts derived by RULE2

```
farther(s, t)
```

Actions derived by RULE3

```
moveAround(s, t, <0,2>)
moveAround(s, t, <1,1>)
...
```

New initial facts after `moveAround(s,t,<0,2>)`

```
destination(<0,5>)
at(s, <0,2>)
at(t, <0,1>)
neighbor(s,t)
neighbor(t,s)
vacant(t, <1,1>)
vacant(s, <0,3>)
...
```

Figure 2: Initial facts and those derived as the first cut of Walk is run.

```
RULE1: dist(S,D_S):-
           at(S,P),
           destination(P_d),
           D_S = |P - P_d|,
           D_S > module radius.

RULE2: farther(S,T):-
           neighbor(S,T),
           dist(S,D_S),
           dist(T,D_T),
           D_S ≥ D_T.

RULE3: moveAround(S,T,P):-
           farther(S,T),
           bestVacant(T,P).

RULE4: vacantDist(S,P,D) :-
           vacant(S,P),
           destination(P_d),
           D = |P - P_d|.

DECL1: type bestVacantDist(module, min <float>).

RULE5: bestVacantDist(S,D) :-
           vacant(S,P,D).

RULE6: bestVacant(S,P) :-
           vacantDist(S,P,D),
           bestVacantDist(S,D).
```

Figure 3: A second cut of the walk program.

facts. The set of facts is initially seeded based upon sensor data and program inputs. An example set of initial facts for a two module ensemble is shown in Figure 2. In the Claytronics model, `neighbor(s,t)` is defined to be true when the modules `s` and `t` are in physical contact and able to communicate. Similarly, `at(s,<0,0>)` is defined to be true when the module `s` is at the point `<0,0>` in space. Finally, `vacant(t, <0,2>)` indicates that the point `<0,2>` adjacent to module `t` is available for another module to enter. A first cut of the Walk program, shown in Figure 1, uses these facts along with an input `destination(<0,5>)` to derive instructions for a module to move closer to the target point `<0,5>`. The program is then reset and run again to take an additional step. This process repeats until the modules reach the destination.

There are two common approaches for executing logic programs. The first approach, top-down execution, consists of a query for a particular fact and a proof search is carried out to look for a derivation of that fact using the supplied rules and set of initial facts. This approach, used by Prolog [5], is not a good match for ensemble programming because of the requirement for a specific query. Instead we employ bottom-up execution, as used in Datalog [6], which takes the initial set of

RULE1:
```
dist(S,D_S):-
    at(S,P),
    destination(P_d),
    D_S = |P - P_d|,
    D_S > module radius.
```

RULE2:
```
farther(S,T):-
    neighbor(S,T),
    dist(S,D_S),
    dist(T,D_T),
    D_S ≥ D_T.
```

RULE3:
```
moveAround(S,T,P):-
    forall neighbor(S,U) [farther(S,U)],
    neighbor(S,T).
    bestVacant(T,P).
```

RULE4:
```
vacantDist(S,P,D) :-
    vacant(S, P),
    destination(P_d),
    D = |P - P_d|.
```

DECL1:
```
type bestVacantDist(module, min <float>).
```

RULE5:
```
bestVacantDist(S,D) :-
    vacant(S,P,D).
```

RULE6:
```
bestVacant(S,P) :-
    vacantDist(S,P,D),
    bestVacantDist(S,D).
```

Figure 4: Final cut of the walk program.

facts and continuously derives whatever new facts can be derived using the rules until all derivable facts have been found.

The Walk program works by determining the distance of each module from the destination, determining which module is farther from the destination, and then instructing the farthest module to move closer. These tasks are accomplished by RULE1, RULE2, and RULE3, respectively. RULE1 depends upon the `at` fact and the `destination` input to calculate the distance (`dist`) of a module from the destination. `dist` is a new derived fact, created by the programmer to represent internal program state, in this case the distance to the destination. The rule is written with variables so it can match on any particular `at` and `destination` facts and produce a corresponding `dist` fact. This produces `dist(s,5)` from `destination(<0,5>)` and `at(s,<0,0>)` and produces `dist(t,4)` from `destination(<0,5>)` and `at(t,<0,1>)`. These are the only pairs of facts that match the rule, so these are all the facts that can be derived from this rule. These derived facts are shown in Figure 2.

The second rule (RULE2) uses the `dist` facts derived by the RULE1 to determine which of two adjacent modules is farther away from the destination. The usage of the `neighbor` fact restricts this comparison to adjacent modules, so we will only be able to determine whether a module is farthest amongst its immediate neighbors, not globally. Notice that when we start with `neighbor(s,t)` we are able to derive `farther(s,t)` since the module `s` is farther from the destination than module `t`, but would do nothing if module `t` were farther from the destination than module `s`. The reader might think another rule is required to cover the second case where module `t` is farther. This, however, is unnecessary. Observe that the `neighbor` fact is symmetric. That is, if `neighbor(s,t)` is true then so is `neighbor(t,s)`, just as in the initial facts in Figure 2. RULE2 covers both cases for us as it can match whichever module happens to be farther away.

The third rule (RULE3) uses the `farther` fact derived by RULE2 along with the `vacant` fact to determine which module should move and where it should move to. It uses the `destination` input fact and the `dist` fact to make sure the location given by the `vacant` fact is closer to the destination. This rule does not derive a new fact like the previous rules, but derives an action instead. An action is a special predicate recognized by the system that causes something to occur. In this case, the `moveAround(s,t,<0,2>)` action causes module `s` to move to the location `<0,2>` by rotating around module `t`. Once this occurs, the program resets and is run again with a new set of initial facts to determine the next step it needs to take. This process continues until the modules reach the destination. If multiple actions are derivable, such as `moveAround(s,t,<0,2>)` and `moveAround(s,t,<1,1>)` then one is arbitrarily chosen to be performed.

The first cut of the Walk program, as presented in Figure 1, illustrates the basic workings of the Meld logic, but has two shortcomings that require us to refine the program using more advanced features of the language. The first of these is that the program moves the farther away module to some closer location adjacent to a closer module. This can be improved upon by selected the closest location adjacent to the closer module. This is done in our second cut of the Walk program. The second short-coming is that the group of modules may become disconnected if some middle module moves closer. Disconnection is undesirable in the Claytronics model because the modules can only communicate with directly connected neighbors, so a break in connectivity breaks the ensemble of modules into two new independent ensembles. This problem will be addressed in the final version of the Walk program.

The second cut of the Walk program (shown in Figure 3) makes use of the `min` *aggregate* to select the closest vacancy to move into. An aggregate is a function that takes a list of values and computes a single scalar from the list. In Meld we take the set of derivable facts that match on all fields other than the one being aggregated over and then compute the aggregate of all values derived for the aggregated field. To do this accurately Meld must derive all the facts that will be aggregated together before calculating and using the aggregate value. Meld stages the firing of the rules using stratification, discussed in §5.5. In this new version of the program, RULE1 and RULE2 remain the same as in Figure 1, but RULE3 is updated to use a new `bestVacant` predicate to choose a vacancy. To find the best vacancy, we calculate the distance of each vacancy to the destination and select the one with the minimum distance.

RULE4 computes the distance of each vacancy from the destination just as RULE1 computed the distance of each module from the destination. RULE5 makes use of an aggregate to select the minimum distance from all of those introduced as `vacantDist` facts by RULE4. Observe the typing declaration for `bestVacantDist` in DECL1. It declares that instead of producing all possible `bestVacantDist` facts, we instead want only the one with the smallest second field for each module. That is to say, we want only the `bestVacantDist` with the smallest distance for each module. RULE6 then uses the `bestVacantDist` fact to pick out the actual vacant point that has that distance which is closest to the destination. Meld supports a wide variety of aggregates, some of which pick out a particular optimal value such as the minimum value or the maximum value and others which combine the values such as a summation or an average. These are discussed further in §4.

The second short-coming of the Walk program is addressed by adding additional constraints to prevent a module from being left behind. In particular, a module will not be permitted to move unless it is farther away from the destination than all neighboring modules.[1] The features of the language demonstrated so far are sufficient to add this constraint, but not in an intuitive manner. Instead, we will make use of universal quantification. We update RULE3 in Figure 4 to check that for every module `u` that is a neighbor of module `s`, module `s` is farther away than module `u`.

[1]This is not sufficient to guarantee an ensemble will not become disconnected unless it starts in a 'good' state. A 'good' state is one in which the ensemble is tightly connected in a canonball packing. It is sufficient to maintain a 'good' state once the invariant has been established.

```
DECL1: type dist(module,module,min<int>).

RULE1: dist(S,S,0).

RULE2: dist(S,T,D+1) :-
           neighbor(S,U),
           dist(U,T,D).

RULE3: ?msg(U,T,M) :-
           ?msg(S,T,M),
           dist(S,T,D),
           neighbor(S,U),
           dist(U,T,D-1).
```

Figure 5: A distance vector routing program.

2.2 Routing Example

The next example, shown in Figure 5, is an algorithm for sending messages between arbitrary nodes in any type of connected ensemble. All we assume here is that a `neighbor(s,t)` fact exists for every pair of connected modules `s` and `t`. These could be modular robots, sensor nodes, computers on a network, etc. This program implements a distance vector routing protocol via the `dist` predicate. A `dist(s,t,d)` fact means that module `s` is distance `d` from module `t`. This is seeded by RULE1 which sets the distance between any module and itself to 0. This rule has no preconditions and contains an unbound variable, something disallowed in Datalog as it seemingly spawns an infinite number of initial facts. The `S` here ranges over only the finite set modules in the ensemble and is implicitly bound as discussed in §2.3, allowing us to support this rule. RULE2 then builds up the distances via routing through all possible neighbors and uses the minimum aggregate to select the shortest route.

Once the routes are populated, RULE3 can be used to forward messages between nodes. RULE3 the linear facts from Linear Logic [7] to represent messages. Linear facts, denoted with a question mark in Meld v2.0, have the property that they can only be used once. Their usage here implies that a message is to be deleted after it has been forwarded along to the next node on the route. Without the use of linearity, these facts would accumulate at each node along the route.

2.3 Locality

A particularly astute reader may have observed that the first field of every fact is a module. In the distributed setting, the facts are spread throughout the ensemble with each fact located at the module referenced in the first field. An investigation of our examples shows that this is exactly where we would expect these facts to be located. Initial facts like `at(s,p)` and `neighbor(s,t)` are observed by `s` and stored at `s`. Derived facts like `?msg(s,t,d)` are stored at `s`, fitting with the model that `s` currently has a message that is destined for `t`. Derived actions like `moveAround(s,t,p)` are located at `s`, the module which needs to perform the action.

A consequence of this method of localizing the facts is that a rule like `Rule1` from the routing example can be implemented. Each module in the ensemble can add an instance of the fact `dist(S,S,0)` to its local database with its own identity substituted for `S`. Thus, `S` is implicitly bound in this rule to the identities of each of the modules in the ensemble. Note that the rule `dist(S,T,0).` is not supported as `T` remains ungrounded.

2.4 Parallelism

The logic programming paradigm provides a natural way to express the parallelism present in an algorithm. Recall the final Walk program (shown in Figure 4). The program does not specify any particular serial ordering in which the rules must be executed. The only ordering information provided are the dependencies between the rules. Rule1 and Rule4 can be run in any order or even simultaneously for a single module. They can be run in different orders for different modules. Rule1 and Rule4 can be run simultaneously for different modules. Since there is no dependency between them, the order in which they are run is irrelevant. On the other hand, Rule3 can only be run for a particular module after Rule2 has been run for all adjacent neighbors and Rule6 has been run for some adjacent neighbor.

Many methods of expressing algorithms require that the programmer produce a serialization of the algorithm or use explicit parallel constructs to express some subset of the available parallelism. The logic programming paradigm permits the programmer to implicitly specify the parallelism inherent in the algorithm and provides the compiler/runtime with the opportunity to take advantage of that parallelism without needing to reconstruct it from some arbitrary serialization. As a result,

it is trivial to extract parallelism from a logic program and distribute it across an ensemble. The tricky part comes in finding the right way to assign computation to processors such that the performance of the system is good and the program state is distributed.

Various techniques can be used to group and assign computation to the various processors, three of which are discussed here:

1. A work list can be maintained, letting each processor pull work off the queue.
2. Rules can be partitioned, giving each processor the responsibility of performing some of the rules. Data is distributed accordingly.
3. Data can be partitioned, making each processor responsibly for some of the data. Rules are distributed accordingly.

The first approach of maintaining a common work queue between the different processors might seem promissing at first, but the size of a single computation is so tiny that a means of combining the computations would be required in order to make the resulting system efficient. Furthermore, each processor would seem to require arbitrary pieces of memory, requiring a shared memory machine. This is an impractical requirement for ensembles.

The other two approachs are two different ways of looking at the same idea - namely partitioning the database and the computation on the database into sets and making each processor responsible for a given set. Approach (2) is a computation-centric view on this idea while approach (3) is data-centric. In either case the idea could work well, provided a good partitioning is used. But how can we find a good partitioning? In an ensemble there is a natural partitioning and assignment of the data and rules to the processors! The input data is already distributed to the different nodes in the system and derived data is associated with the existing nodes. Thus the database can be partitioned such that the data about a node is stored on that node. The partitioning of the rules follows such that each processor is running instances of all of the rules with some variables preassigned to the name of the processor they are running on.

In this way, a program's state and execution are distributed across the nodes that make up the ensemble. Each node in the ensemble runs its local rules and implicitly communicates with other nodes in the ensemble when rules transcend multiple modules. These rules can be automatically

split into multiple pieces with explicit communication between them. Thus the programmer is not required to write any explicit communication as it is implied by the distribution method. This approach specifies a means to run a logic program on an ensemble.

2.5 Fault Tolerance

The way in which logic programs are expressed provides enough information to allow a limited kind of fault tolerance. In the case of the routing example, for instance, if a module in the system or a link between modules should fail the `dist` facts can be discarded and recomputed to create a new set of `dist` facts that are representative of the new ensemble without the failed module or link. This approach can be improved upon to remove only the affected `dist` facts from the database while leaving all other routes intact. This is discussed further in §5.3.

3 Literature Review

A plethora of programming languages have been developed over the years, exploring various methods of expressing and executing programs and applying these ideas to different domains. Here I discuss languages targeted to towards ensembles as well as those targeted towards more general distributed systems. Afterwards, I discuss much of the work on logic programming which is the base upon which Meld is built.

3.1 Ensemble Languages

One of the most interesting abstractions explored for ensemble programming language is that of folding paper as demonstrated in Origami Shape Language [8] (OSL). OSL is designed for programming an ensemble of smart paper to fold itself into some particular shape. The neat part about OSL is that a program is writen to think about lines, creases, and points and not about the modules that make up the physical system. In this way it truly provides an abstraction where one writes a program for a piece of paper and the cells that make up that piece of paper implement it. The programmer need never think about the actual cells or even be aware that they exist. The success of this approach is inspiring, but, unfortunately, it's not clear how it could be adapted for

general ensembles.

Regiment [9] is another high level programming language, targeted towards ensembles of sensor networks. Regiment is geared for the common sensor network task of aggregating data across an ensemble to a particular destination. The language allows one to create groups (such as k-nearest neighbors) and talk about aggregate data over such a group. The data can also be easily forwarded to some control system that is connected to the sensor network. The abstraction used provides no mechanism to talk about individual modules or particular locations. This approach works very well for aggregating data, but leaves no clear means to add features such as locomation for modular robots, keeping it in the domain of sensor networks and not general ensembles.

Hood [10] is similarly based upon the idea that programs are written around neighborhoods, with each module communicating via broadcast to all nearby modules. As a nesC library it is lower level than Regiment, but targets providing a similar type of abstraction. Hood presents a simple approach to simplifying programs by allowing a node to advertise a value to its neighbors and permitting the neighbors to filter incoming values and cache them as desired. They show that this simplifies the implementation of a variety of sensor network programs and appears to be a valuable contribution on top of nesC. While Hood is successful in reducing the amount of code required to implement many sensor network algorithms, it adds no benefit on top of traditional message passing when individual communication is required and it does nothing to facilitate distributing the program across many nodes.

Proto [11] is another high level language following the functional programming paradigm. It permits one to write programs for sensor networks or similar systems. Proto abstracts away the exact system and lets nodes interact with other nodes only via a `foldl` like command. This abstraction could likely be abused to provide direct neighbor communication, but doing so would be expensive on some systems as each message sent would go to all neighbors, only to be ignored by most of them. The same functional program is executed at each node in the system and while the authors make a big deal about 3 levels of abstraction, the 'Amorphous Medium', and the idea that one should be able to talk about a chunk of the system, they get no closer to this goal than other languages. Finally, they treat all data as time-sychronized streams and require some external entity to maintain a clock. It's not clear how one would adapt this to a modular robotic system like Claytronics, particular given the requirement for a time oracle. The stream approach to variable

values is would likely result in limitations on state, similar to those of DataLog like languages.

Protoswarm [12] is an attempt to adapt Proto to devices that actuate, but it is not particularly useful for systems like Claytronics. There are serious problems in using the language for this purpose which are not addressed by Protoswarm. In particular, the lack of direct communication makes it extremely challenging (impossible?) to produce a spanning tree, implement any known method for maintaining connectivity, or implement any known method for localization.

Pleiades [13], another low-level language, is an imperative programming language based on C with the addition of a concurrent for loop that iterates over various nodes and lets each node perform the body of the loop in parallel. It is much like an OpenMP [14] implementation, but distributed over a sensor network and with only a particular node able to run each iteration of the loop. It additionally provides extra consistency guarantees that the result of execution is equivalent to some serial ordering of the loop. This is a convenient way of writing some programs (particularly for wireless networks) as one can easily do tasks like computing aggregates over the values at the different nodes. Attempting to write a shape change algorithm for module robots is awkward and challenging, probably even harder from this perspective of a global sequential system than from the perspective of individual nodes.

TinyDB [15] provides an SQL-like interface for extracting data from an ensemble, treating it as a database. It is designed for extracting and aggregating data via resolving SQL queries. The query is sent out through the ensemble, data is gathered and aggregated to form a response. This approach presumes an external entity exists to formulate the queries. This is an effective means of using a sensor network as a sorce of data, but is not applicable to general ensemble programming.

LDP [16] is another high level language for programming ensembles, targeted at modular robots. LDP expresses programs as a set of predicates which match on the configuration of modules and the internal state of the system. The predicates can then change the internal state and/or perform actions such as causing robots to move. This approach produces a similar expression of programs to that of first order logic, with the exception that program state can be mutated resulting in consistency problems. An attempt is made to address these problems by taking snapshots of state at modules, but there is no guarantee that these snapshots are consistent beyond a single module, sometimes producing unexpected behavior. Additionally, LDP requires active polling for matching

predicates. This may work well in a rapidly changing ensemble, but can create repetative queries in a relatively static ensemble. These queries continually failing to match on the same criteria waste message bandwidth, cpu cycles, and energy. The ideas behind this work are solid, but faithfully following the logic programming model will allow Meld to resolve these consistency and performance concerns while allowing similarly concise programs at a similar level of abstraction.

The work on Meld was inspired by the ensemble programming langues P2 [17] and SNLog [18] for declarative networks and sensor networks, respectively. These works pioneered the idea of using logic programming for distributed systems. They view the state of the system as a database with the contents distributed across the nodes of the system. First order logic programs, a la DataLog, can be used to query the database and generate new facts for it. This allows for programs to be implicitly distributed across a number of nodes for execution, but suffers from the restrictions on expressivity inherent in first order logic. These works form a basis for developing more expressiving logic programming languages for use on ensembles such as the one I propose here.

3.2 Other Concurrent Languages

Looking beyond languages targeted at various ensembles, there are many languages designed around concurrent programming. Perhaps the most famous among these is the π-Calculus [19] [20], which expresses parallel programs via a set of concurrent processes. At any time any of these processes may be evaluated a single step. The processes share channels through which they can send data between them. The π-Calculus is a generalization of the λ-Calculus to support parallel processes. The π-calculus is primarily intended for describing concurrent computations rather than implemention.

Another well known, but more recent, language is Map-Reduce [21]. Map-Reduce provides a very simple abstraction for parallelizing programs via defining a task for processing a chunk of data and a means of aggregating the processed chunks once an entire set has been processed. This simple mechanism has had a great impact on simplifying the amount of effort required to perform data-parallel tasks, but its applicability to other tasks is minimal. While it is possible to encode some graph algorithms in the Map-Reduce paradigm, doing so is awkward and clunky, producing inefficient results.

Dryad [22] is a generalization of Map-Reduce that allows the programmer to create a DAG of

programs for processing the data. It focuses on data processing, but is useful because it allows the programmer to divide up the work both as chunks of data and as chunks of computation and lets the system worry about matching this up to the physical resources available. It is more general than Map-Reduce because it allows arbitrary DAGs of computation. It is intended as a mechanism for utilizing large clusters to perform computations on fixed datasets. Ultimately, it is still limited to data-parallel applications.

The Spider Calculus [23] is a functional programming language for creating and reasoning about graphs. The idea consists of having an initial graph (called a web) with one or more computational threads. Each thread (called a spider) runs some program (which may not be related to what the other threads are running) and can both move about and modify the graph. This results in an interesting method of creating, modifying, and computing on graphs. This approach has interesting potential, although it is not clear how it relates to ensemble programming and whether it might be adapted to work with modular robotics or sensor networks,

ML5 [24] is an ML-like language extended with a type system based on modal logic. This allows ML5's type system to understand where different computations occur and to produce appropriate error messages at compile time if a given node in the system doesn't provide the necessary features. This allows for a system where different nodes have different capabilities which the programmer can take advantage of rather than writing separate programs for every type of node in the system. The examples given are for writing webservers which have a very different set of capabilities than web browsers. The ideas in ML5 are very interesting, but are orthogonal to what is being considered here.

4 Meld v1.0 Semantics: Language and Meaning

A running Meld v1.0 program consists of a database of *facts* and a set of production rules for operating on and generating new facts. A Meld v1.0 fact is a predicate and a tuple of values; the predicate denotes a particular relation for which the tuple is an element. Facts represent the state of the world based on observations and inputs (e.g., sensor readings, connectivity or topology information, runtime parameters, etc.), or they reflect the internal state of the program. Starting from an initial set of axioms, new facts are derived and added to the database as the program runs.

Known Facts	$\Gamma ::=$	$\cdot \Vert \Gamma, f(\hat{t})$	Facts	$F ::=$	$f(\hat{x})$
Accumulated Actions	$\Psi ::=$	$\cdot \Vert \Psi, a(\hat{t})$	Constraints	$C ::=$	$c(\hat{x})$
Set of Rules	$\Sigma ::=$	$\cdot \Vert \Sigma, R$	Expression	$E ::=$	$E \wedge E \Vert F \Vert \forall F.E \Vert C$
Actions	$A ::=$	$a(\hat{x})$	Rule	$R ::=$	$E \Rightarrow F \Vert E \Rightarrow A$
					$\Vert agg(F, g, y) \Rightarrow F$

Figure 6: Abstract syntax for Meld v1.0 programs

In addition to facts, *actions* are also generated. They are syntactically similar to facts but cause side effects that change the state of the world rather than the derivations of new facts. In a robotics application, for example, actions are used to initiate motion or control devices. Meld v1.0 rules can use a variety of arithmetic, logical, and set-based expressions, as well as aggregation operations.

4.1 Structure of a Meld v1.0 Program

Figure 6 shows the abstract syntax for states, rules, expressions, and constraints in Meld v1.0. A Meld v1.0 program consists of a set of production rules. A rule may contain variables, the scope of which is the entire rule. Each rule has a head that specifies a fact to be generated and a body of prerequisites to be satisfied. If all prerequisites are satisfied, then the new fact is added to the database. Each prerequisite expression in the body of the rule can either match a fact or specify a constraint. Matching is achieved by finding a consistent substitution for the rule's variables such that one or more facts in the database are matched. A constraint is a boolean expression evaluated on facts in the database; these can use arithmetic, logical, and set-based subexpressions. Finally, `forall` statements iterate over all matching facts in the database and ensure that for each one, a specified expression is satisfied.

Meld v1.0 rules may also derive actions, rather than facts. Action rules have the same syntax as rules, but have a different effect. When the body of this rule is proved true, its head is not inserted into the database. Instead, it causes an action to be carried out in the physical world. The action, much like a fact, has a predicate and a tuple, which can be assigned values by the expressions in the rule.

An important concept in Meld v1.0 is the *aggregate*. The purpose of an aggregate is to define a

```
(a) type logical neighbor parent(module, first module).
(b) type maxTemp(module, max float).

(c) parent(A, A) :- root(A).
(d) parent(A, B) :- neighbor(A, B), parent(B, _).
(e) maxTemp(A, T) :- temperature(A, T).
(f) maxTemp(B, T) :- parent(A, B), maxTemp(A, T).

(g) type globalMax(module, float).
(h) globalMax(A, T) :- maxTemp(A, T), root(A).
(i) globalMax(B, T) :- neighbor(A, B), globalMax(A, T).

(j) type localMax(module).
(k) localMax(A) :- temperature(A, T),
                   forall neighbor(A, B) temperature(B, T') T > T'.
```

Figure 7: A data aggregation example coded in Meld v1.0Å spanning tree is built across the ensemble and used to aggregate the max temperatures of all nodes to the root. The result is flood broadcast back to all nodes. Each node also determines whether it is a local maximum.

type of predicate that combines the values in a collection of facts. As a simple example, consider the program shown in Figure 7, for computing the maximum temperature across all the nodes in an ensemble. The `parent` rules, (c) and (d), build a spanning tree across the ensemble, and then the `maxTemp` rules, (e) and (f), use this tree to compute the maximum temperature. Considering first the rules for calculating the maximum, rule (e) sets the base case; rule (f) then propagates the maximum temperature (T) of the subtree rooted at one node (A) to its parent (B). Applied across the ensemble, this has the effect of producing the maximum temperature at the root of the tree. To accomplish this, the rule prototype given in (b) specifies that when `maxTemp` is matched, the `max` function should be used to aggregate all values in the second field for those cases where the first field matches. In the case of the `parent` rules, the prototype given in (a) indicates the use of the `first` aggregate, limiting each node to a single parent. The `first` aggregate keeps only the first value encountered in any match on the rest of the tuple. The meaning of `logical neighbor` is explained in §5.1.

In general, aggregates may use arbitrary functions to calculate the aggregate value. In the abstract syntax, this is given as a function g that calculates the value of the aggregate given all of the individual values. The result of applying g is then substituted for y in F. In practice, as described by LDL++[25], the programmer implements this as three functions: two to create an aggregate and one to retrieve the final value. The first two functions consist of one to create an aggregate from a single value and a second to update the value of an existing aggregate given another value. The third function, which produces the final value of the aggregate, permits the aggregate to keep additional state necessary to compute the aggregate. For example, an aggregate to compute the average would keep around the sum of all values and the number of values seen. When the final value of the aggregate is requested, the current value of sum is divided by the total number of values seen to produce the requested average.

4.2 Meaning of a Meld v1.0 Program

The state of an ensemble running a Meld v1.0 program consists of two parts: derived facts and derived actions. Γ is the set of facts that have been derived in the current world. Γ is a list of facts that are known to be true. Initially, Γ is populated with observations that modules make about the world. Ψ, is the set of actions derived in the current world. These are much like the derived

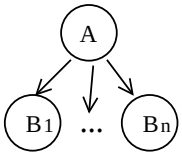
facts that make up Γ, except that they are intended to have an effect upon the ensemble rather than be used to derive further facts.

As a Meld v1.0 program runs, new facts and actions are derived from existing facts which are then added to Γ and Ψ. Once one or more actions have been derived, they can be applied to bring about a change in the physical world. When the actions have been applied to the world, all derived facts are discarded and replaced with a set of new observations about the world. The program then restarts execution in the new world.

The evaluation rules for Meld v1.0 allow for significant uncertainty with respect to actions and their effects. This underspecification has two purposes. First, it does not make assumptions about the type of ensemble nor the kinds of actions which can be triggered by a Meld v1.0 program. Second, it admits the possibility of noisy sensors and faulty actuators. In the case of modular robotics, for instance, a derived action may request that a robot move to a particular location. External constraints, however, may prevent the robot from moving to the desired location. It is, therefore, important that Γ end up containing the actual position of the robot rather than the location it desired to reach.

This result is achieved by discarding Γ when an action is applied and starting fresh. By doing this, we erase all history from the system, removing any dependencies on the intended effect of the action. This interpretation also accounts for the fact that sensors may fail, be noisy, and even in the best case that observations of the real world that are known to the ensemble are only a subset of those that are available in the real world. To account for changes that arise due to external forces we also allow the program to restart even when Ψ is empty.

This interpretation permits us to give Meld v1.0 programs a well-defined meaning even when actuators fail, external forces change the ensemble, or sensors are noisy. In turn, this imbues Meld v1.0 with an inherent form of fault tolerance. The greatest limitation of this approach, however, is in our ability to reason about programs. By allowing the ensemble to enter a state other than the one intended by the action, we eliminate the ability to directly reason about what a program does. To circumvent this, it is necessary to make assumptions about how likely an action is to go awry and in what ways it's possible for it to go awry. This is discussed further in §6.2.



original rule from the temperature example:

```
localMax(A) :- temperature(A, T),
               forall neighbor(A, B)
                      [temperature(B, T'),
                       T > T'].
```

send rule after splitting:

```
__remote_LM(A, B, T') :- neighbor(B, A),
                         temperature(B, T').
```

local rule after splitting:

```
localMax(A) :- temperature(A, T),
               forall neighbor(A, B)
                      [__remote_LM(A, B, T'),
                       T > T'].
```

Figure 8: Example of splitting a rule into its local and send parts. On the left, the spanning tree for home nodes is shown. On the right is a rule from the program in Figure 7 along with the two rules that result from localizing it.

5 Distributed Implementation of Meld v1.0 programs

In this section we describe how Meld v1.0 programs can be run as forward-chaining logic programs across an ensemble. We first explain the basic ideas that make this possible. We then describe the additional techniques of deletion and X-Y stratification that are required to make this feasible and efficient.

5.1 Basic Distribution/Implementation Approach

Meld v1.0 is an ensemble programming language; efficient and scalable execution requires Meld v1.0 programs to be distributed across the nodes of the ensemble. To facilitate this, we require the first variable of a fact, called the *home variable*, to refer to the node where the fact will be stored. This convention permits the compiler to distribute the facts in a Meld v1.0 program to the correct nodes in the ensemble. It also permits the runtime to introduce facts representing the state of the world at the right nodes, i.e., facts that result from local observations are available at the

corresponding module, e.g., `A` in the `temperature` predicate of Figure 7 refers to the temperature observed at node `A`.

Just as the data is distributed to nodes in the ensemble, the rules need to be transformed to run on individual modules. Extending a technique from the P2 compiler, the rules of a program are *localized* — split into rules with local bodies — such that two kinds of rules exist. The first of these are *local rules* in which every fact in the body and head of the rule share the same home node. The second kind of rule is a *send rule* for which the entire body of the rule resides on one module while the head of the rule is instantiated on another module.

To support communication for the send rules, the compiler requires a means of determining what routes will be available at runtime. This is facilitated by special facts, called *logical neighbor facts*, which indicate runtime connectivity between pairs of modules, and potentially multi-hop routes between them. Among the axioms introduced by the runtime system are logical neighbor facts called `neighbor` facts, which indicate a node's direct communication partners. Beyond an ability to communicate (assumed to be symmetric), any meaning attributed to these facts are implementation-dependent (e.g. for Claytronics, these indicate physically neighboring modules; for sensor networks, these indicate motes within wireless range). Additional logical neighbor facts (e.g. `parent`) can be derived transitively from existing ones (e.g. two `neighbor` facts) with the route automatically generated by concatenation. Symmetry is preserved automatically by the creation of a new predicate to support the inverted version of the fact (which contains the reverse route at runtime).

Using the connectivity relations guaranteed by logical neighbor facts, the compiler is able to localize the rules and ensure that routes will be known for all send rules. The compiler considers the graph of the home nodes for all facts involved in the a rule, using the connectivity relations supplied by logical neighbor facts as edges. A spanning tree, rooted at the home node of the head of the rule, is generated (as shown in Figure 8).

For each leaf in the tree, the compiler generates a new predicate (e.g. `__remote_LM`), which will reside on the parent node, and creates a send rule for deriving this predicate based on all of the relevant facts that reside on the leaf node. The new predicate is added as a requirement in the parent, replacing the facts from the leaf node, and the leaf node is removed from the graph. This

is repeated until only the root node remains at which point we are left with a local rule. Note that this process may add dependencies on symmetric versions of logical neighbor facts, such as `neighbor(B, A)` in Figure 8.

Constraints from the original rule can be placed in the local rule's body to produce a correct implementation of the program. A better, more efficient alternative, however, places the constraints in the send rules. This way, if a constraint does not hold, then a message is not sent, effectively short-circuiting the original rule's evaluation. To this end, constraints are pushed as far down the spanning tree as possible to short-circut the process as early as possible.

The techniques of assigning home nodes, generating logical neighbors for multi-hop communications, and automaticly tranforming rules into local and send parts allow Meld v1.0 to execute a program on a distributed set of communicating nodes.

5.2 Triggered Derivations

A Meld v1.0 program, as a bottom-up logic, executes by deriving new facts from existing facts via application of rules. Efficient execution requires applying rules that are likely to find new derivations. Meld v1.0 accomplishes this by ensuring that a new fact is used in every attempt at finding a derivation. Meld v1.0 maintains a *message queue* which contains all the new facts. As a Meld v1.0 program executes, a fact is pulled out of the queue. Then, all the rules that use the fact in their body are selected as candidates rules. For each candidate, the rest of its rule body is matched against the database and, if the candidate can be proven, the head of the rule is instantiated and added to the message queue. This triggered activation of rules by newly derived facts is essential to make Meld v1.0 efficient.

5.3 Deletion

One of the largest hurdles to efficiently implementing Meld v1.0 is that whenever the world changes we must discard all known facts and start the program over from the beginning, as described in §4.2. Fortunately, we can more selectively handle such changes by borrowing the notion of *deletion* from P2. P2 was designed for programming network overlays and uses deletion to correctly handle occasional link failures. Although the ensembles we consider may experience more frequent changes

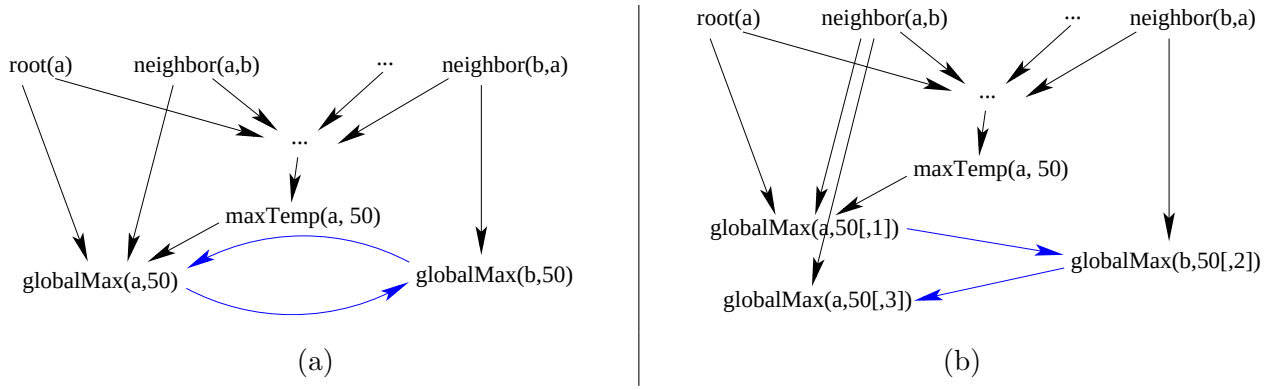


Figure 9: Partial derivation graph for the program in Figure 7. The graph on the left shows the derivation graph for this program using the simple reference counting approach. Note the cycle in the graph which prevents this approach from working correctly. The graph on the right shows how the cycle is eliminated through the usage of the derivation counting approach.

in their world, these can be handled effectively with a local, efficient implementation of deletion.

Deletion avoids the problem of simultaneously discarding every fact at every node and restarting the program by carefully removing only those facts from the system which can no longer be derived. Deletion works by considering a deleted fact and matching the rules in exactly the same way as derivations are done to determine which other facts depend on the deleted one. Each of these facts is then, in turn, deleted. Strictly following this approach will result in a "conservative" approach that deletes too many facts, e.g., ones with alternative derivations that do not depend on the previously deleted facts. This approach would be correct if at each step all possible derivations were tried again, but produces a problem given our triggered application of rules. In other words, a derivable fact that is "conservatively" deleted may never be re-derived, even though an alternate derivation may exist. Therefore, it is necessary to have an exact solution to deletion in order to use our triggered approach to derivation.

P2 addresses this issue by using reference counting techniques similar to those used in garbage collection. Instead of keeping track of the number of objects that point to an object, it keeps track of the number of derivations that can prove a particular fact. When a fact is deleted, this count is decremented. If the count reaches zero, then the fact is removed from the database and facts derived from it are recursively deleted. This approach works for simple cases, but suffers from the cyclic "pointer" problem. In Meld v1.0 a fact is often used to derive another instance of itself,

(a) Initial facts with ref counts:

`neighbor(a,b)` *(×1)*	`root(a)` *(×1)*
`neighbor(b,a)` *(×1)*	`maxTemp(a,50)` *(×1)*

(b) Facts after application of rules with reference counts:

`neighbor(a,b)` *(×1)*	`root(a)` *(×1)*	`globalMax(b,50)` *(×1)*
`neighbor(b,a)` *(×1)*	`maxTemp(a,50)` *(×1)*	`globalMax(a,50)` *(×2)*

(c) Facts after deletion of maxTemp(a,50) using basic reference counts:

`neighbor(a,b)` *(×1)*	`root(a)` *(×1)*	`globalMax(b,50)` *(×1)*
`neighbor(b,a)` *(×1)*	`globalMax(a,50)` *(×1)*	

(d) Facts after application of rules with reference counts with depths:

`neighbor(a,b)` *(×1)*	`root(a)` *(×1)*	`globalMax(b,50)` *(×1@2)*
`neighbor(b,a)` *(×1)*	`globalMax(a,50)` *(×1@1; ×1@3)*	

(e) Facts after deletion of maxTemp(a,50) using reference counts with depths:

`neighbor(a,b)` *(×1)*	`neighbor(b,a)` *(×1)*	`root(a)` *(×1)*

Figure 10: Example of deletion with reference counts, and derivation counts with depth (counts and depths shown in parentheses after each fact). Based on the program from Figure 7, the `globalMax(a,50)` fact can be cyclically derived from itself through `globalMax(b,50)`. Derivation counts that consider depth allow deletions to occur correctly, while simple reference counts fail. Facts leading up to `maxTemp(a,50)` are omitted for brevity and clarity.

leading to cyclic derivation graphs (shown in Figure 9(a)). In this case, simple reference counting fails to properly delete the fact, as illustrated in parts a–c of Figure 10.

In the case of Meld v1.0, and unlike a reference counting garbage collector, we can resolve this problem by tracking the depth of each derivation. For facts that can be involved in a cyclic derivation, we keep a reference count for each existing derivation depth. When a fact with a simple reference count is deleted, we proceed as before. When a fact with reference counts for each derivation depth is deleted, we decrement the reference count for that derivation depth. If the smallest derivation depth is decremented to zero, then we delete the fact and everything derived from it. If one or more derivations still exist after this process completes, then we reinstantiate the fact with the new derivation depth. This process serves to delete any other derivations of the fact that depended upon the fact and eliminates the possibility of producing an infinite cyclic derivation with no start. This solution is illustrated in Figure 9(b) and parts d–e of Figure 10.

5.4 Concerning Deletion and Actions

Since the message queue contains both newly derived facts and the deletion of facts, an opportunity for optimization presents itself. If a new fact (F) and the deletion of that fact ($\not{F}$) both exist in the message queue, one might think that both F and $\not{F}$ can be silently removed from the queue as they cancel one another out. This would be true if all derived rules had no side-effects. However, the possibility of deriving an action requires caution.

The key difference between facts and actions is that for facts we need to know only whether it is true or not, while for an action we must act each time it is derived. The semantics of Meld v1.0 require that deletions be completed "instantly," taking priority over any derivations of new facts. Thus, when F comes before $\not{F}$, then silently removing both from the queue is safe since $\not{F}$ undoes the derivation of any fact that might be derived from F.

If, however, $\not{F}$ comes before F, then canceling them is not safe. In this case, processing them in the order required by the semantics could result in deleting and rederiving an action, causing it to be correctly performed. Had we silently deleted both F and $\not{F}$, the action would not occur. Thus, this optimization breaks correctness when $\not{F}$ occurs before F in the queue. As a result, we only cancel out facts in the queue when the fact occurs before the deletion of the fact.

5.5 X-Y Stratification

A naïve way to implement aggregates (and `forall` statements which require similar considerations) is to assume that all values for the predicate are known, and calculate the aggregate accordingly. If a new value arrives, one can delete the old value, recompute, and instantiate the new one. At first glance, this appears to be a perfectly valid approach, though somewhat inefficient due to the additional work to clean up and update aggregate values that were based on partial data. Unfortunately, however, this is not the case, as the additional work may be arbitrarily expensive. For example, an aggregate computed with partial data early in the program may cause the entire program to execute with the wrong value; an update to the aggregate effectively entails discarding and deleting all facts produced, and rerunning the program. As this can happen multiple, times, this is clearly neither efficient nor scalable, particularly for aggregates that depend on other aggregates. Finally, there is a potential for incorrect behavior—any actions based on the wrong aggregate values may be incorrect and cannot be undone.

Rather than relying on deletion, we ensure the correctness and efficiency of aggregates by using *X-Y stratification*. X-Y stratification, used by LDL++[25], is a method for ensuring that all of the contributing values are known before calculating the value of an aggregate. This is done by imposing a global ordering on the processing of facts to ensure that all possible derivations for the relevant facts have been explored before applying an aggregate. This guarantees that the correct value of an aggregate will be calculated and eliminates the need for expensive or impossible corrections via deletion.

Unfortunately, ensuring a global ordering on facts for X-Y Stratification as described for LDL++ requires global synchronization, an expensive, inefficient process for an ensemble. We propose a safe relaxation of X-Y Stratification that requires only local synchronization and leverages an understanding of the communications paths in Meld v1.0 programs. Because Meld v1.0 has a notion of local rules and send rules (described in §5.1), the compiler can determine whether a fact derivation depends on facts from only the local module, the neighboring modules, or some module far away in the ensemble. Aggregation of facts that originate locally can safely proceed once all such facts have been derived locally. If a fact can come only from a neighboring module, then it is sufficient to know that all of the neighboring modules have derived all such facts and will produce no

more. In these two cases, only local synchronization between a module and its immediate neighbors is necessary to ensure stratification.

Therefore, locally on each node, we impose an ordering on fact derivations. This is precisely the ordering that is provided via X-Y stratification, but it is only enforced within a node's neighborhood, i.e., between a single node and its direct neighbors. An aggregation of facts that can only be derived locally on a single node is handled in the usual way. Aggregation of facts that might come from a direct neighbor is deferred until each neighbor has promised not to send any additional facts of that type. Thus, to ensure that all the facts contributing to an aggregate are derived beforehand, some nodes are allowed to idle, even though they may be able to produce new facts based on aggregates of partial sets of facts. For the rare program that aggregates facts which can originate from an arbitrary module in the ensemble, it may be necessary to synchronize the entire ensemble. The compiler, therefore, disallows aggregates that depend upon such facts. To date we have not needed such an aggregate, but intend to investigate this further in the future.

5.6 Implementation Targets of Meld v1.0

The Meld v1.0 compiler produces byte codes which are interpretted by an virtual machine. Meld v1.0 programs can run on various systems by implementing a virtual machine on a target system and providing the Meld v1.0 compiler with a model for the system. The system model consists of the axiom predicates and action predicates for the target system, allowing the compiler to perform type-checking for the system and link the axioms and actions correctly with the runtime. The Meld v1.0 virtual machine and runtime is currently implemented on the three different systems described below.

5.6.1 DPRSim/DPRSim2

DPRSim [26] and DPRSim2 [27] are a pair of simulators which simulate programmable matter. They simulate robots which are able to perform computation, store data locally, and locomote around one another. DPRSim runs on a single machine and is capable of simulating tens of thousands of robots. DPRSim2 runs across many nodes in a cluster and is capable of simulating tens of millions of robots. These simulators are capable of running arbitrary code on each simulated

robot and, therefore, an implementation of the Meld v1.0 VM. Using this simulator demonstrates the utility of Meld v1.0's actions as well as the ability to create and execute Meld v1.0 programs scaled across millions of modules.

5.6.2 Blinky Blocks

The Blinky Blocks are "human-actuated" modular robots. Each block has a small processor and small amount of memory and is capable of communicating with adjacent robots. They feature accelerometers, allowing them to determine the direction of gravity and to detect taps. They also feature bright LEDs, permitting them to change color. The blocks are fault-prone, power-cycling any time they are moved or bumped too violently. This platform demonstrates that Meld v1.0 programs can be run on resources-limited nodes and are tolerant of abundant faults.

5.6.3 Multi-core Machines

Finally, a version of the VM has been ported to multi-core machines, exploring the utility of Meld v1.0 in this environment. As previously mentioned, much prior work has focused on implementing concurrent versions of logic programming languages. These works focused on splitting queries to be performed by multiple nodes whereas Meld v1.0 partitions the database, leading to a different level of parallelism. Exploring multi-core will allow us to test this method of parallizing logic programs, but is not a focus of this thesis.

6 Analysis of Meld v1.0

In this section we discuss some of the advantages and disadvantages of writing programs in Meld v1.0. To facilitate this, we consider two real programs for modular robots that have been implemented in Meld v1.0 in addition to the temperature averaging program for sensor networks shown in Figure 7. These programs implement a shape change algorithm as provided by Dewey et. al. [2] (a simplified version is shown in Figure 11) and a localization algorithm provided by Funiak et. al. [28]. The localization algorithm generates a coordinate system for an ensemble by estimating node positions from local sensor data and then iteratively refining the estimation.

```
// Choose only best state:
// FINAL=0, PATH=1, NEUTRAL=2
type state(module, min int).
type parent(module, first module).
type notChild(module, module).

// generate PATH state next to FINAL
state(B, PATH) :-
   neighbor(A, B),
   state(A, FINAL),
   position(B, Spot),
   0 = inTargetShape(Spot).

// propagate PATH/FINAL state
state(B, PATH) :-
   neighbor(A, B),
   state(A, PATH).

state(B, FINAL) :-
   neighbor(A, B),
   state(A, FINAL),
   position(B, Spot),
   1 = inTargetShape(Spot).

// construct deletion tree from FINAL
parent(B, A) :-
   neighbor(A, B),
   state(B, PATH),
   state(A, FINAL).

// extend deletion tree along PATH
parent(B, A) :-
   neighbor(A, B),
   state(B, PATH),
   parent(A, _).
```

```
// B is not a child of A
notChild(A, B) :-
   neighbor(A, B),
   parent(B, C), A != C.

notChild(A, B) :-
   neighbor(A, B),
   state(B, FINAL).


// action to destroy A, give resources to B
// can apply if A is a leaf in deletion tree
destroy(A, B) :-
   state(A, PATH),
   neighbor(A, B),
   resources(A, DESTROY),
   resources(B, DESTROY),
   forall neighbor(A, N)
      notChild(A, N).


// action to transfer resource from A to B
give(A, B) :-
   neighbor(A, B),
   resources(A, CREATE),
   resources(B, DESTROY),
   parent(A, B).


// action to create new module
create(A, Spot) :-
   state(A, FINAL),
   vacant(A, Spot),
   1 = inTargetShape(Spot),
   resources(A, CREATE).
```

Figure 11: A metamodule-based shape planner based on [2] implemented in Meld v1.0. It uses an abstraction that provides metamodule creation, destruction, and resource transfer as basic operations. The code ensures the ensemble stays connected by forming trees and deleting only leaf nodes. This code has been tested in simulations with up to 1 million metamodules, demonstrating the scalability of the distributed Meld v1.0 implementation.

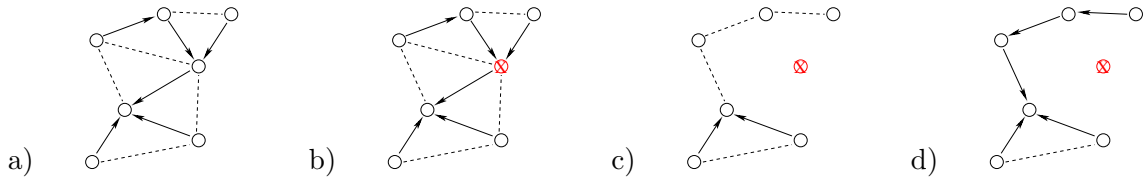


Figure 12: (The max temperature program (in Figure 7) (a) creates a tree. When (b) a node fails, the Meld v1.0 runtime is able to (c) destroy the subtree rooted at the failed node via deletion and (d) reconnect the tree.

The shape change algorithm is a motion planner for modular robots. Planning for individual modules is plagued by non-holomonic constraints, however planning can be done for groups, called *metamodules*, with only holonomic constraints. Dewey's algorithm runs on this metamodule abstraction rather than on individual modules. These metamodules are not capable of motion themselves. Instead they can be absorbed into (destroyed by) or extruded out of (created by) an adjacent metamodule. An absorbed metamodule can be transfered from one metamodule to an adjacent one, allowing it to travel throughout the ensemble as a resource. The planner makes local decisions on where to create new metamodules, destroy existing ones, and how to move resources.

6.1 Fault Tolerance

As evident from the discussion in §5, Meld v1.0 inherently provides a degree of fault tolerance to programs. The operational semantics of Meld v1.0 allows for arbitrary changes in the physical world; any visible change causes removal of facts that are no longer supported by the derivation rules. In the event that a module ceases to function (fail-dead), every fact that is derived from axioms about that module is deleted. New axioms, representing the new state of the world, are introduced and affected portions of the algorithm are rerun. This allows the program to run as though the failed module had never been present, modulo actions that have already occurred. As long as the program has no special dependence on this module, it continues to run and tolerates the failure. Other failures can also be tolerated as long as the program can proceed without the lost functionality.

For the temperature averaging program, this feature of Meld v1.0 is very effective. If, for instance, a module fails then a break occurs in the constructed tree. In a naïve implementation in

another language, this could result in a failure to complete execution or a failure to include observations from the subtree rooted at the failed node. An implementation that can tolerate such a fault and reconstruct the tree (assuming the ensemble is still connected) requires significant additional code, foresight, and effort from the programmer. The Meld v1.0 implementation, however, requires nothing additional. When a module fails, Meld v1.0 automatically deletes the subtree rooted at the failed node and, if the network is still connected, adds these modules back into the tree, as shown in Figure 12.

6.2 Provability

As Meld v1.0 is a logic programming language, Meld v1.0 programs are generally well-suited for use in correctness proofs. In particular, the structure and semantics let one directly reason about and apply proof methods to Meld v1.0 program implementations, rather than on just the specifications or translated pseudo-code representations. Furthermore, Meld v1.0 code is amenable to mechanized analysis via theorem checkers such as Twelf [29]. Twelf is designed for analyzing program logics, but can be used for analyzing logic program implementations as well.

Proofs of correctness for programs involving actions, however, may need to make assumptions about what happens when an action is attempted. For the planner example, a proof of correctness has been carried out with the assumption that actions are always performed exactly as specified. The planner has been proven to achieve a correct target shape in finite time while maintaining the connectivity of the ensemble.[2] These simplifying assumptions, however, prevent any formal reasoning about fault tolerance, as discussed in §6.1. Although empirical evidence shows that the Meld v1.0 implementation is indeed tolerant to some faults, a good fault model will be required to formally analyze this aspect of the program.

6.3 Messaging Efficiency

The distributed implementation of Meld v1.0 is effective at propagating just the information needed for making forward progress on the program. As a result, a Meld v1.0 program's message complexity can be competitive with hand-crafted messaging written in other languages. This was

[2] A sketch of the proofs is available in [2] and the full proofs on the Meld v1.0 source code are available in [30].

demonstrated in [31] for small programs and our enhancements carry this through for more complex programs that use aggregates. In particular, the use of aggregates can cause high message complexity. Before adding X-Y stratification, aggregates that depend on data received from neighbors, such as those used in the iterative refinement steps of the localization algorithm, could cause multiple re-evaluations of the aggregate as data trickled in. In the worst case, this could cause an avalanche of facts with intermediate values to be sent throughout the ensemble, each of which is then deleted and replaced with another partial result. For localization, this resulted in a lack of progress due to an explosion of messages on all but trivially small examples in the original implementation of Meld v1.0. Our addition of X-Y-stratification to Meld v1.0 alleviates this issue: the result of an aggregate is not generated or propagated until all supporting facts have been seen, limiting both messaging and computation overheads. With X-Y stratification, localization has been demonstrated on ensembles of up to 10,000 nodes, with a message complexity logarithmic in the number of modules, exactly as one would expect from a high level description of the algorithm.

6.4 Memory Efficiency

Although the Meld v1.0 compiler is not fully optimized for memory, many Meld v1.0 programs have small memory footprints and can, therefore, fit into the limited memory available on sensor network motes and on modular robots. To test this, we measure the maximum memory used among all the modules in an ensemble executing the example Meld v1.0 programs. Both the temperature aggregation program and the shape change algorithm prove to have very small memory footprints, requiring at most only 488 and 932 bytes per module, respectively. The aggregation program is sensitive to neighborhood size; this was assumed to be 6, and the memory required grows by 38 bytes for each additional neighbor. Furthermore, these numbers assume 32-bit module identifiers and temperature readings; 16-bit module identifiers and data would halve the maximum memory footprint. Both of these programs fit comfortably into the memory available on a mote or a modular robot.

The localization algorithm, on the other hand, requires tens to hundreds of kilobytes of memory depending on the ensemble size. This is due to the lack of support within Meld v1.0 for dynamic state. Because of this limitation, the localization algorithm is written such that it produces a new (static) estimated position fact for each step of iterative refinement. Furthermore, as the old

estimates are used in the derivation of the new ones, these are not discarded and they quickly accumulate. As the ensemble grows, more steps of iterative refinement are required, generating even larger quantities of outdated facts that only serve to establish a long chain of derivation from the axioms. Thus, programs that require dynamic state (such as algorithms involving iterative refinement) can not currently be efficiently run in Meld v1.0.

7 Shortcomings of Meld v1.0

Meld v1.0 has proven to be effective for writing some algorithms, such as the meta-module shape change algorithm depicted in Figure 11. Unfortunately, other algorithms are difficult or impossible to present faithfully in Meld v1.0. Funiak et al.'s localization algorithm [28], for instance, runs into two problems when implemented in Meld v1.0. The two problems appear to be different, but they stem from the same source.

The localization algorithm described by Funiak et al. uses sensor readings between closely packed modules to build an accurate coordinate system. The algorithm assumes that sensor readings indicate physical contact between the modules in the system and that the reading provides a direction to a module with some degree of inaccuracy. The algorithm proceeds by building small groups of modules (chosen via normalized cut) and localizing each group. A gradient decent step is performed on each group to minimize any errors. The groups are then combined to form larger groups via a rigid transformation, followed by additional gradient decent in order to use the new information present in the larger group to further minimize the error. This process repeats until all the groups have been combined and the ensemble has a single coordinate system.

Two problems arise in implementing this algorithm in Meld v1.0 - one with ease of representation and efficiency of running and the other with trying to accurately represent the algorithm. The first shortcoming is highlighted by the gradient descent step. As the algorithm runs, it continually produces better estimates as to the location of each module in the ensemble. In an imperative language we might represent the location of a module with some particular variable which we update continuously as the algorithm is run. In Meld v1.0 we lack the ability to change a value once assigned. We cannot have some position fact which we continuously update as we run the algorithm. Instead, we must keep every estimate of our position. This can be made to work by

tagging position facts with a version number, but this solution is unsatisfying. Firstly, it places greater burden on the program and inhibits the programmer's ability to correctly implement the algorithm. In particular, it is necessary to devise a method of tagging the position facts in such a way that the current value can be selected. This is non-trivial because there is no way to inspect the facts to determine which tag is the best one, rather the algorithm must be able to determine a priori which tag is appropriate for the step. This problem is certainly tractable (as evidenced by the completed implementation of the localization algorithm), but unnecessarily creates extra work and a greater mental burden on the programmer.

Secondly, this solution is unsatisfying because it is wildly inefficient. The algorithm passes through hundreds of estimates for each module. Keeping all of these estimates requires hundreds of times more memory than keeping just the current estimate. This is problematic, to say the least, on a small embedded module with minimal resources. Furthermore, this potentially increases the runtime of the algorithm as the database now contains extra facts that we might try to match against using the various rules of our program. Successful or not, attempting to match these will waste CPU cycles and power.

The other concern arises when we consider changes to the positions of modules after successfully building a coordinate system. What should we do if a module is removed from the ensemble? Nothing. The positions of all robots still present have not changed, so we should keep them. Meld v1.0 will helpfully erase many or all of these positions as it goes through the deletion process and then generate a new coordinate system for us. This is a time consuming, a waste of resources, and potential wrong as the new coordinate system might be substantially different from the prior one (perhaps a different module will be chosen for the origin). What should we do if a module is added to the ensemble? It should receive a coordinate and then gradient decent should be performed locally to minimize the error. Meld v1.0, however, will again go through the deletion process and produce a new coordinate system. We would like to write the localization algorithm in two stages, one where an initial coordinate system is generated and one where it goes into an update mode, handling minor changes in topology. Unfortunately, this is not possible in Meld v1.0.

Thus we see that Meld v1.0 has two major shortcomings. First, there is no way for the programmer to remove old stale facts. Meld v1.0 determines when facts are stale based on whether they are still derivable, but a particular algorithm may have no further use for them and they may be

in the way. Second, there is no way for the programmer to prevent Meld v1.0 from removing good facts. Meld v1.0 removes facts because they are no longer derivable, but they may still be useful for the algorithm. Therefore, Meld requires some principled means of allowing the programmer to determine when some facts are stale.

8 Proposed New Research for Meld v2.0

As shown in §7, some algorithms cannot be expressed accurately and easily in Meld v1.0. Any approach to addressing these expressivity problems will require an extension of the Meld v1.0 language. I express here the ideas for addressing these problems in Meld v2.0.

In Meld v1.0, the compiler and runtime are responsible for managing all facts, determining when they are stale and removing them via deletion. In many instances this functionality is convenient for the programmer, but in some cases it prevents a program, such as localization, from being easily or correctly expressed in Meld v1.0. In Meld v2.0, I intend to address this issue by splitting facts into two classes - those managed by the Meld v2.0 runtime and those managed by the programmer.

To this end I will extend Meld v1.0 with a modality for user-managed facts. To support allowing the user to add, modify, and remove facts in a disciplined way modal facts may be linear (in the sense of linear logic). When considering how Meld v1.0 programs execute, I observe that the existing facts are automatically updated to accurately reflect the state of the ensemble at all times. The user-managed modality will sustain no automatic changes when a module moves or fails or any other change in observable state occurs. These linear facts can, however, be used and influenced by the observable facts.

Linear facts are permitted to retain knowledge about the history of the ensemble rather than just the current physical state. This would allow for simple representation of time-dependent state, program status, etc., while eliminating the need for and problems with the aforementioned embeddings. The biggest concern about this approach is that it may (partially) eliminate some of the nice features of Meld v1.0, such as fault tolerance. It will be necessary to understand what these consequences will be and whether there are ways in which we can ameliorate the impact on fault tolerance.

To this end, I will produce a formal operational semantics for Meld v2.0, just as was done for Meld v1.0. To show that these additions are principled, I will need to uncover the logical foundations for Meld v2.0. In the event that they prove elusive, we will still have a formal semantics for the language along with an explanation of the problems encountered in uncovering the logical underpinnings. Regardless, I will use the formal semantics to prove correctness properties of a few example programs, like those already done for the Meld v1.0 implementation of the meta-module shape change algorithm.

Another necessary challenge will be producing an efficient implementation of Meld v2.0. Meld v1.0 programs have been shown to execute within the hardware constraints of the Blinky Blocks and with asymptotic runtime and memory requirements comparable to equivalent C programs. The number and size of messages exchanged as a Meld v1.0 program executes is comparable to that of equivalent C program execution. I anticipate that Meld v2.0 programs will be able to boast similar performance claims. In particular, Meld v2.0 programs should be able to match theoretical asymptotic runtime and memory usage and interesting programs should fit within the hardware available on the Blinky Blocks. The goal here is to demonstrate that the language is usable, not to produce an optimizing compiler.

Implementing this system in a distributed fashion will require some changes from the Meld v1.0 compilation and execution model. The Meld v1.0 implementation localizes rules, breaking them into sub-rules which run independently. As the execution of a rule in Meld v2.0 (potentially) requires removing facts from the database in addition to inserting them, it will be necessary to ensure that rules are executed atomically. It should be sufficient to augment the existing execution model with the usage of transactions to atomically execute each original rule across the relevent nodes of the system. A concern here is that the usage of transactions could have a substantial negative impact on performance and may make performance more difficult to evaluate.

This new work will demonstrate my thesis. The new linear modality in Meld v2.0 shows sufficient expressivity to write interesting programs for a variety of ensembles. The implementation of Meld v2.0 will demonstrate efficient execution of these programs on various platforms. The operational semantics and proofs of program properties will demonstrate that Meld v2.0 programs can be reasoned about. Thus, this work will show that logic programming, as exemplified by Meld v2.0, is well-suited for ensemble programming.

9 Timeline

9.1 Already Done

I have an implementation of the Meld v1.0 compiler and runtime. The runtime runs on three different systems - the blinky blocks and blinky block simulator, the DPRSim and DPRSim2 Claytronics simulators, and a parallel machine. The global and local semantics for Meld v1.0 are formally defined in Twelf and I have shown that they are equivalent. I have also proven the correctness and termination (assuming adequate resources) of Dewey et al.'s meta-module shape change algorithm implemented in Meld v1.0 using Twelf.

The performance of some simple Meld v1.0 programs for Claytronics has been analyzed in comparison to C programs, showing that Meld v1.0 programs are comparable in messages sent and memory/cpu scaling, but lag behind by an order of magnitude in actual memory/cpu usage. The programs evaluated are an order magnitude more concise than their C counterparts. This performance data is obsolete and does not reflect the current version of Meld v1.0.

A preliminary implementation of Meld v2.0 exists and is running on all three systems.

9.2 Summer 2011

This summer I intend to focus on the Meld v2.0 semantics and understanding their logical underpinnings. Presently I have a draft version of the semantics, but the logic behind them is unclear at this time as the choice of modalities is novel. The modalities in use have many similaties to those of the linear and persistent modalities and those of necessitation and truth. This will primarily be a matter of understanding exactly what the relation is between the Meld v2.0 managed facts and the user managed facts and relating that to logic. The Meld v2.0 semantics will be formalized in either Twelf or Celf as appropriate.

9.3 Fall 2011

The fall semester will be spent ensuring that the Meld v2.0 implementation is consistent with the formal semantics and analyzing the implementation. This will include analyzing the performance

of various example programs as well as proving properties about some of these programs. These programs are to including Dewey et al.'s meta-module shape change algorithm, Funiak et at.'s localization algorithm as reimplemented in Meld v2.0, and Blinky Block games.

9.4 Spring 2012

The spring semester is reserved for writing the thesis document. I expect that all necessary research will be completed by this time.

10 Summary

I propose to design and implement a language, Meld v2.0, for the purpose of programming ensembles. Preliminary work on Meld v1.0 has shown the logic programming paradigm to be a good match for programming ensembles. I will enhance the expressivity of Meld v1.0, eliminating the weaknesses that hinder its present use for some classes of applications. Meld v1.0 programs are automatically distributed across an ensemble and are inherently fault tolerant. Meld v2.0 will continue to boast these features. Furthermore, I will demonstrate proving properties of Meldprograms, ensuring that implementations behave as expected.

References

[1] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell, "A language for large ensembles of independently executing nodes," in *Int'l Conf. on Logic Programming*, 2009.

[2] Daniel Dewey, Siddhartha Srinivasa, Michael P. Ashley-Rollman, Michael De Rosa, Padmanabhan Pillai, Todd C. Mowry, Jason D. Campbell, and Seth Copen Goldstein, "Generalizing metamodules to simplify planning in modular robotic systems," in *Proc. of Int'l Conf. on Intelligent Robots and Systems*, Nice, France, Sept. 2008.

[3] Seth Goldstein, Jason Campbell, and Todd Mowry, "Programmable matter," *IEEE Computer*, June 2005.

[4] Brian Kirby, Michael Ashley-Rollman, and Seth Copen Goldstein, "Blinky blocks: A physical ensemble programming platform," in *CHI*, 2011.

[5] International Organization for Standardization, "ISO/IEC 13211: Information technology – Programming languages – Prolog," Geneva, 1995.

[6] The MITRE Corporation, "Datalog user manual," `http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html`, 2004.

[7] Jean-Yves Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–101, 1987.

[8] Radhika Nagpal, *Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics*, Ph.D. thesis, MIT, 2001, MIT AI Lab Technical Memo 2001-008.

[9] Ryan Newton, Greg Morrisett, and Matt Welsh, "The Regiment macroprogramming system," in *Proc. of the Int'l conf. on Information Processing in Sensor Networks (IPSN'07)*, April 2007.

[10] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proc. of the 2nd int'l conf. on Mobile systems, applications, and services*, New York, NY, USA, 2004, pp. 99–110, ACM Press.

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

[12] Jonathan Bachrach, James McLurkin, and Anthony Grue, "Protoswarm: A language for programming multi-robot systems using the amorphous medium abstraction," in *Int'l Conf. in Autonomous Agents and Multiagent Systems (AAMAS)*, May 2008.

[13] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," in *PLDI '07: Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2007, pp. 200–210, ACM.

[14] The OpenMP Architecture Review Board, "OpenMP Application Program Interface," `http://www.openmp.org/mp-documents/spec30.pdf`, 2008.

[15] 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.*, vol. 30, no. 1, pp. 122–173, 2005.

[16] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason D. Campbell, and Padmanabhan Pillai, "Programming modular robots with locally distributed predicates," in *Proc. of the IEEE Int'l Conf. on Robotics and Automation*, 2008.

[17] 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 *Proc. of the 2006 ACM SIGMOD int'l conf. on Management of data*, New York, NY, USA, 2006, pp. 97–108, ACM Press.

[18] David Chu, Arsalan Tavakoli, Lucian Popa, and Joseph Hellerstein, "Entirely declarative sensor network systems," 2006.

[19] Robin Milner, Joachim Parrow, and David Walker, "A calculus of mobile processes, part i," *I AND II. INFORMATION AND COMPUTATION*, vol. 100, 1989.

[20] Robin Milner, Joachim Parrow, and David Walker, "A calculus of mobile processes, part ii," 1989.

[21] Jeffrey Dean, Sanjay Ghemawat, and Google Inc, "Mapreduce: simplified data processing on large clusters," in *In OSDI '04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. 2004, USENIX Association.

[22] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, New York, NY, USA, 2007, pp. 59–72, ACM.

[23] Benjamin C. Pierce, Alessandro Romanel, and Daniel Wagner, "The Spider Calculus: Computing in active graphs," 2010.

[24] Tom Murphy VII, "Modal types for mobile code," 2008.

[25] Carlo Zaniolo, Natraj Arni, and KayLiang Ong, "Negation and aggregates in recursive rules: the LDL++ approach," in *DOOD*, 1993, pp. 204–221.

[26] Intel Corporation and Carnegie Mellon University, "Dprsim: The dynamic physical rendering simulator," `http://www.pittsburgh.intel-research.net/dprweb/`, 2006.

[27] Michael P. Ashley-Rollman, Padmanabhan Pillai, and Michelle Goodstein, "Simulating multi-million-robot ensembles," in *International Conference on Robotics and Automation*, May 2011.

[28] Stano Funiak, Michael P. Ashley-Rollman, Padmanabhan Pillai, Jason D. Campbell, and Seth Copen Goldstein, "Distributed localization of modular robot ensembles," in *Proc. of the 3rd Robotics Science and Systems*, 2008.

[29] Frank Pfenning and Carsten Schürmann, "System description: Twelf - a meta-logical framework for deductive systems," in *Proc. of Int'l Conf. on Automated Deduction*, 1999, pp. 202–206.

[30] Daniel Dewey, Siddhartha Srinivasa, Michael P. Ashley-Rollman, Michael De Rosa, Padmanabhan Pillai, Todd Mowry, Jason D. Campbell, and Seth Copen Goldstein, "Generalizing metamodules to simplify planning in modular robotic systems," Tech. Rep. CMU-CS-08-139, Carnegie Mellon University, 2008.

[31] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai, "Meld: A declarative approach to programming ensembles," in *Proc. of the IEEE Int'l Conf. on Intelligent Robots and Systems*, Oct. 2007.