A Language for Large Ensembles of Independently Executing Nodes

Michael P. Ashley-Rollman[†], Peter Lee[†], Seth Copen Goldstein[†]
Padmanabhan Pillai[‡], and Jason D. Campbell[‡]

[†]Carnegie Mellon University, Pittsburgh, PA 15213
{mpa,petel,seth}@cs.cmu.edu
[‡]Intel Research Pittsburgh, Pittsburgh, PA 15213
{padmanabhan.s.pillai,jason.d.campbell}@intel.com

Abstract. We address how to write programs for distributed computing systems in which the network topology can change dynamically. Examples of such systems, which we call *ensembles*, include programmable sensor networks (where the network topology can change due to failures in the nodes or links) and modular robotics systems (whose physical configuration can be rearranged under program control). We extend Meld [1], a logic programming language that allows an ensemble to be viewed as a single computing system. In addition to proving some key properties of the language, we have also implemented a complete compiler for Meld. It generates code for TinyOS [14] and for a Claytronics simulator [12]. We have successfully written correct, efficient, and complex programs for ensembles containing over one million nodes.

1 Introduction

Several types of distributed systems have the property that the network topology can change dynamically. For example, in *ad hoc* sensor networks [6], it is common for the network to change due to failures in both the nodes and network links. Modular robotic systems [21] are another common example. Under software control, a modular robotic system can rearrange how its modules are connected, which means that its network topology changes, too. A third example is Claytronics [11], which can be viewed as a type of modular robotic system containing, potentially, millions of unreliable modules. We use the term *ensemble* to refer to any such network-varying distributed system.

How shall we write programs for ensembles? Writing software is hard; writing software for distributed systems is even harder. Add to that the possibility of a dynamic network topology, and it is easy to see that we are faced with a daunting programming problem. Furthermore, real-world ensembles such as robots and sensor nets pose additional challenges such as unreliable communications, imperfect or failing actuators, and soft and hard compute errors. The complexity of writing code to recover from such failures is magnified by the number of potential interactions within an ensemble.

Given these considerations, we have been extending the language Meld [1], that allows ensembles to be programmed as a unified whole and then compiled automatically into fully distributed code. This approach frees programmers from the need to understand how or where in the system the program state is to be maintained or messages sent. Furthermore, Meld programs are written in the logic programming paradigm, leading to clear and concise programs. And as our early experiments have thus far demonstrated,

Meld programs are inherently robust to changes in network topology and provide for simple fault handling. The initial design of Meld was influenced heavily by Datalog [4] and, like Datalog, programs in Meld lend themselves to proofs of correctness.

We have made substantial progress on the design, implementation, and application of Meld [1]. We have adopted X-Y stratification from LDL++ [22] and adapted it to work in a distributed environment. We have addressed problems with the deletion algorithm used in prior work, such as P2 [15]. We have achieved what we believe to be a fully practical language for a range of modular robotics and sensor network applications, and have completed a thorough definition of the operational semantics, much of which has been implemented in the Twelf [19] proof system. A complete compiler for Meld has also been implemented, which generates code for TinyOS [14] and a Claytronics simulator [12]. Using the compiler and simulator, we have written correct, efficient programs for ensembles containing over one million nodes.

2 Related Work

The P2 [15] language and SNLog [5] language, which were designed for programming overlay networks and sensor networks respectively, showed that a logic programming approach could be used to allow an ensemble to be programmed as a unified whole. Meld adds, among other things, support for robot actuation and sensing, and is based on a well-defined formal semantics. In principle, even very large ensembles can be reasoned about formally.

Several languages have been proposed for sensor nets. Hood [20], Tinydb [16], and Regiment [18] provide excellent support for applications such as data collection and aggregation, but do not directly address more dynamic scenarios involving physical reconfiguration. Pleiades [13], also designed for sensor networks, can be used in situations with dynamic network topologies. It adopts a programming style similar to OpenMP, for example allowing one to write parallel loops that run across multiple modules. Of course, this requires the programmer to specify whether a variable is to be stored locally or propagated about the ensemble as the program runs. Thus, the programmer's focus is on the individual modules instead of the whole ensemble.

Origami Shape Language [17] and Proto [3] are effective for programming distributed systems as a whole. They were originally targeted towards stationary wireless modules, rather than ensembles. Recently, Proto has been extended to mobile robots [2]. LDP [7] was derived from a method for distributed debugging. Originally designed for modular robotics, LDP sends condition-matchers around the ensemble. It is based on a tick model, generating a new set of matchers throughout the ensemble on each tick. While this works well in highly dynamic systems, it can lead to excessive messaging in more static environments.

3 Meld: Language and Meaning

A running Meld program consists of a database of *facts* and a set of production rules for operating on and generating new facts. A Meld 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

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

Fig. 1. *Abstract syntax for Meld programs*

and added to the database as the program runs. 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 rules can use a variety of arithmetic, logical, and set-based expressions, as well as aggregation operations.

3.1 Structure of a Meld Program

Fig. 1 shows the abstract syntax for states, rules, expressions, and constraints in Meld. A Meld 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 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 is the *aggregate*. The purpose of an aggregate is to define a type of predicate that combines the values in a collection of facts. As a simple example, consider the program shown in Fig. 2, 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 §4.1.

```
(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'.
```

Fig. 2. *A data aggregation example coded in Meld. A 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.*

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++[22], 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.

3.2 Meaning of a Meld Program

The state of an ensemble running a Meld 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 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 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 trig-

gered by a Meld 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 programs a well-defined meaning even when actuators fail, external forces change the ensemble, or sensors are noisy. In turn, this imbues Meld 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 §5.2.

4 Distributed Implementation of Meld programs

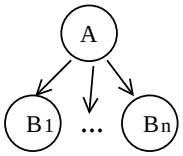
In this section we describe how Meld programs can be run as forward-chaining logic programs across an ensemble. We first explain the basic ideas that make this possible. We then describe the additional techniques of deletion and X-Y stratification that are required to make this feasible and efficient.

4.1 Basic Distribution/Implementation Approach

Meld is an ensemble programming language; efficient and scalable execution requires Meld 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 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 Fig. 2 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



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'].
```

Fig. 3. *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 Fig. 2 along with the two rules that result from localizing it.*

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 Fig. 3).

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 Fig. 3.

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.

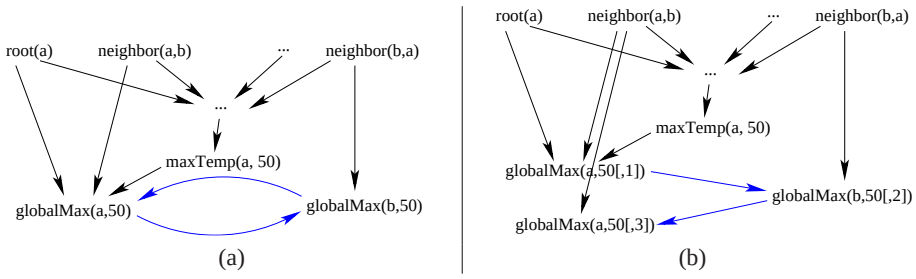


Fig. 4. Partial derivation graph for the program in Fig. 2. The graph on the left shows the derivation graph for this program using the simple reference counting approach. Note the cycle in the graph which prevents this approach from working correctly. The graph on the right shows how the cycle is eliminated through the usage of the derivation counting approach.

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

4.2 Triggered Derivations

A Meld 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 accomplishes this by ensuring that a new fact is used in every attempt at finding a derivation. Meld maintains a *message queue* which contains all the new facts. As a Meld 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 efficient.

4.3 Deletion

One of the largest hurdles to efficiently implementing Meld is that whenever the world changes we must discard all known facts and start the program over from the beginning, as described in §3.2. 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 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

(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)*

Fig. 5. *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 Fig. 2, 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.*

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 a fact is often used to derive another instance of itself, leading to cyclic derivation graphs (shown in Fig. 4(a)). In this case, simple reference counting fails to properly delete the fact, as illustrated in parts a–c of Fig. 5.

In the case of Meld, 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 Fig. 4(b) and parts d–e of Fig. 5.

4.4 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 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.

4.5 X-Y Stratification

A naïve way to implement aggregates (and `foralll` 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++[22], 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 programs. Because Meld has a notion of local rules and send rules (described in

§4.1), 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 Analysis and Discussion

In this section we discuss some of the advantages and disadvantages of writing programs in Meld. To facilitate this, we consider two real programs for modular robots that have been implemented in Meld in addition to the temperature averaging program for sensor networks shown in Fig. 2. These programs implement a shape change algorithm as provided by Dewey et. al. [8] (a simplified version is shown in Fig. 6) and a localization algorithm provided by Funiak et. al. [10]. The localization algorithm generates a coordinate system for an ensemble by estimating node positions from local sensor data and then iteratively refining the estimation.

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.

5.1 Fault Tolerance

As evident from the discussion in §4, Meld inherently provides a degree of fault tolerance to programs. The operational semantics of Meld 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

```
// 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).
```

Fig. 6. *A metamodule-based shape planner based on [8] implemented in Meld. 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 implementation.*

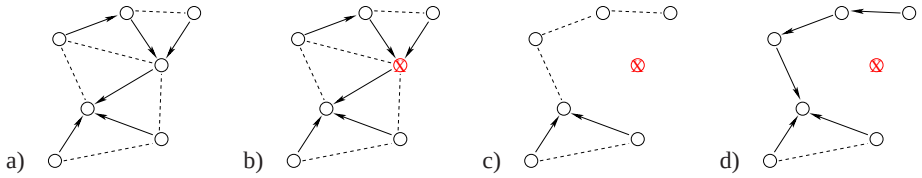


Fig. 7. *(The max temperature program (in Fig. 2) (a) creates a tree. When (b) a node fails, the Meld runtime is able to (c) destroy the subtree rooted at the failed node via deletion and (d) reconnect the tree.*

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 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 implementation, however, requires nothing additional. When a module fails, Meld 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 Fig. 7.

5.2 Provability

As Meld is a logic programming language, Meld 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 program implementations, rather than on just the specifications or translated pseudo-code representations. Furthermore, Meld code is amenable to mechanized analysis via theorem checkers such as Twelf [19]. 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.[1] These simplifying assumptions, however, prevent any formal reasoning about fault tolerance, as discussed in §5.1. Although empirical evidence shows that the Meld implementation is indeed tolerant to some faults, a good fault model will be required to formally analyze this aspect of the program.

[1] A sketch of the proofs is available in [8] and the full proofs on the Meld source code are available in [9].

5.3 Messaging Efficiency

The distributed implementation of Meld is effective at propagating just the information needed for making forward progress on the program. As a result, a Meld program's message complexity can be competitive with hand-crafted messaging written in other languages. This was demonstrated in [1] 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. Our addition of X-Y-stratification to Meld 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.

5.4 Memory Efficiency

Although the Meld compiler is not fully optimized for memory, many Meld 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 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 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.

6 Conclusions and Future Work

Meld has grown into a substantially more effective language for programming ensembles of independently executing nodes. Our early experiments have shown that concise

and efficient programs involving very large numbers of nodes can be developed successfully. Both of the example programs in this paper (for calculating max temperature in a sensor network and for achieving a desired 3D shape in a modular robotics system) were shown to be concise and efficient in our extended version of Meld.

The Meld programs we have written thus far are, to a surprising degree, tolerant of node failure. Such robust behavior in the face of individual node failures is, we believe, an important property, especially as ensemble size grows. We also showed that Meld programs are amenable to formal analysis and proof. In particular, because of Meld's logic-programming roots, programs written in Meld can be used directly in proofs of correctness, e.g., the shape-change planner has been proven correct in this manner.

We have extended Meld in ways that enable better efficiency on larger ensembles, and believe that large ensembles are precisely where the advantages of Meld become most valuable. We described results from simulations of Meld programs running on up to 1 million nodes. For systems of this scale, we found Meld's ability to generate all of the needed messaging and distribution of state across the nodes to be a great aid in helping the programmer to understand, control, and reason about the program.

Despite all of this progress, Meld is still not an ideal language language for certain problem domains. For instance, problems requiring the maintenance of dynamic state, as demonstrated via the iterative gradient decent in the localization algorithm, are not efficiently executable in Meld. While such state can be encoded in Meld, the lack of direct support leads to suboptimal behavior. In particular, such encodings can require unbounded quantities of memory and may fall apart in the event of a fault. This issue of dynamic state will need to be addressed for Meld to become an ideal language for writing a more general class of ensemble programs. In the meantime, Meld offers distinct advantages for implementing many classes of distributed algorithms for execution on a variety of ensemble types.

Acknowledgements We thank Robert J. Simmons and Frank Pfenning for providing feedback on and insights into Meld and this paper. We thank the anonymous reviewers for their helpful comments. This work was supported by Intel Corporation, Microsoft Corporation, and the National Science Foundation under Grant #CNS-0428738.

References

1. 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.
2. 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.
3. Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.
4. Stefand Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.
5. David Chu, Arsalan Tavakoli, Lucian Popa, and Joseph Hellerstein. Entirely declarative sensor network systems. 2006.

6. David Culler, Deborah Estrin, and Mani Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37(8):41–49, Aug. 2004.
7. 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.
8. 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.
9. 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. Technical Report CMU-CS-08-139, Carnegie Mellon University, 2008.
10. 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.
11. Seth Goldstein, Jason Campbell, and Todd Mowry. Programmable matter. *IEEE Computer*, June 2005.
12. Intel Corporation and Carnegie Mellon University. Dprsim: The dynamic physical rendering simulator. `http://www.pittsburgh.intel-research.net/dprweb/`, 2006.
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*, pages 200–210, New York, NY, USA, 2007. ACM.
14. Philip Levis. *TinyOS Programming*. UC - Berkeley, 2006.
15. 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*, pages 97–108, New York, NY, USA, 2006. ACM Press.
16. Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
17. Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001. MIT AI Lab Technical Memo 2001-008.
18. 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.
19. Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proc. of Int'l Conf. on Automated Deduction*, pages 202–206, 1999.
20. 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*, pages 99–110, New York, NY, USA, 2004. ACM Press.
21. Mark Yim, Wei-Min Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory S. Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *Robotics and Automation Magazine, IEEE*, 14(1):43–52, March 2007.
22. Carlo Zaniolo, Natraj Arni, and KayLiang Ong. Negation and aggregates in recursive rules: the LDL++ approach. In *DOOD*, pages 204–221, 1993.