Meld: A Declarative Approach to Programming Ensembles

Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, Padmanabhan Pillai

Abstract— **This paper presents Meld, a programming language for modular robots, i.e., for independently executing robots where inter-robot communication is limited to immediate neighbors. Meld is a declarative language, based on P2, a logic-programming language originally designed for programming overlay networks. By using logic programming, the code for an ensemble of robots can be written from a global perspective, as opposed to a large collection of independent robot views. This greatly simplifies the thought process needed for programming large ensembles. Initial experience shows that this also leads to a considerable reduction in code size and complexity.**

An initial implementation of Meld has been completed and has been used to demonstrate its effectiveness in the Claytronics simulator. Early results indicate that Meld programs are considerably more concise (more than 20x shorter) than programs written in C++, while running nearly as efficiently.

I. INTRODUCTION

Modular robotics is based on the concept of a (typically homogenous) collection of robots working together to carry out a task. Most of the designs proposed to-date involve the robots attaching together to form larger robotic structures. These connected ensembles gain capabilities beyond those of the individual robots, including: greater strength from concerted, parallel efforts; flexible, reconfigurable form to better deal with the environment and obstacles; and greater capacity to redundantly sense the surroundings. However, planning and control for the ensemble increases in complexity due to the large number of robots and exponentially large configuration space. Compounding the problem is the fact that computation, sensing, and actuation are distributed among the relatively limited robots—in general, no one robot is strong enough to move all of the other robots, nor is it possible, typically, for any one robot to determine the positions of every other robot or where they should be. These limitations in strength and knowledge multiply the number of possible failure modes. For example, a robot might move to the wrong location or physically impede the desired movement of other robots. The ensemble as a whole might move into an unstable structure and risk collapse. To succeed, the individual robots in a modular robotics system must work cooperatively.

Modular robots must, as a collective, compute the overall goal, move into the right positions, and know when they have achieved the goal. All of this must be achieved in a fault-tolerant manner. These difficulties raise significant questions about how a programmer should think about programming modular robot systems. We believe it is natural to think about what the robot ensemble as a whole should do, but programming the robots requires thinking about what the individual robots should do. This creates a problem of perspective, or of understanding the relationship between local and global behavior. When considering the desired global behavior it is frequently unclear how to translate it into local rules that can be run on individual robots.

For example, to have a set of robots share some sensor information and cooperatively decide on a simple next goal may require a complex program that manages dozens of possible states, several different message types, and hundreds of state transitions. Part of this complexity and burden on the programmer is due to the low-level nature of many common programming languages (particularly C/C++ and similar languages) that force thinking in terms of local data structures, low-level operations, and communication primitives. Often, it is more natural to think at a higher level of abstraction, in terms of the bits of information known, what can be deduced from these, and how to make a decision or select a goal based on these. This leads us to consider a declarative programming paradigm, in which a programmer specifies the high-level logic of *what* is to be decided or achieved, and leaves it to the implementation of the programming language to work out the low-level details of data manipulation and communication, i.e., *how* to achieve it.

There has been some precedence in using declarative languages in situations where a large number of computing nodes must work cooperatively. A notable example is the use of a language called P2 [5] in the context of programming overlay networks. The experience of using P2 for networks demonstrates that logic programming is well suited for cooperative, distributed computing environments. The P2 language allows common network routing protocols to be written in just a few lines of P2 code rather than pages of C++, and often with competitive run-time performance. In essence, P2 shows that independent network nodes can work collectively and efficiently to prove that a global goal has or has not been attained. Agreement among the many nodes is achieved when they all arrive at the same irrefutable proof.

Although actuation and motion in a robotics system causes continuous changes in network topology and provides a more dynamic environment than is the case for network programming, we claim that the same level of conciseness, simplicity, and efficiency in programming is achievable for modular robotics. To support this claim, we present Meld, a logic-programming language that is inspired by P2, and designed for modular robotics. More specifically, Meld addresses

This work was supported in part by NSF Grant#CNS-0428738, and Intel Corporation.

Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, and Todd C. Mowry in CSD at Carnegie Mellon University, 5000 Forbes Ave, 15213 {mpa,seth,petel,tcm}@cs.cmu.edu

Padmanabhan Pillai is at Intel Research Pittsburgh padmanabhan.s.pillai@intel.com

```
RULE1: Dist(S,D):- At(S,P),
                   P_d = destination(),
                   D = |P - P_d|,
                   D > robot radius.

RULE2: Farther(S,T):- Neighbor(S,T),
                      Dist(S,D_S),
                      Dist(T,D_T),
                      D_S ≥ D_T.

RULE3: MoveAround(S,T,U):- Farther(S,T),
                           Farther(S,U),
                           U ≠ T.
```

Fig. 1. Walk program.

the problem of independently executing robots where inter-robot communication is limited to immediate (in contact) neighbors. Meld allows the code for an ensemble of robots to be written from a global perspective, focusing on information and logic for decisions, as opposed to managing a large collection of independent robot programs and dealing with low-level communication and data management tasks. This greatly simplifies the thought process needed for programming. Initial experience shows, just as with P2, that this leads to a considerable reduction in code size and complexity.

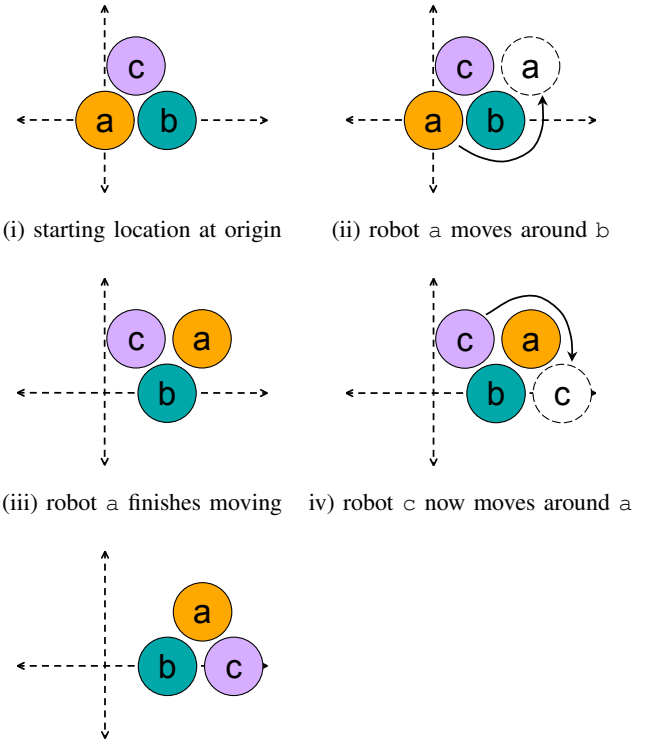
In this paper we have already described some of the difficulties that arise in programming modular robotics. Next we introduce the logic programming language we wish to use for modular robotics and discuss how it addresses these problems. Finally, we evaluate the new challenges that arise from the logic programming approach and discuss an approach for tackling them. An initial implementation of Meld has been completed and used to demonstrate its effectiveness in dprsim[2], a Claytronics[4] simulator. Early results indicate that Meld programs are 20x shorter, while as efficient as the same programs written in C++.

II. THE MELD LANGUAGE

In this section we give a presentation of the Meld language. We explain how logical programming works, and provide a simple example that introduces some of the salient features of the language.

A. The Meld Logic

Logical programming languages use a collection of facts and a set of production rules for combining existing facts to produce new ones. Each rule specifies a set of conditions (expressions relating facts and pieces of facts), and a new fact that can be proven (i.e., generated safely) if these conditions are satisfied. As a program is executed, the facts are combined to satisfy the rules and produce new facts which are in turn used to satisfy additional rules. This process, called *forward chaining*, continues until all provable facts have been proven. A logic program, therefore, consists of the rules for combining facts while the execution environment is the set of base facts that are known to be true before execution begins. It is important to understand that the base



(i) starting location at origin (ii) robot a moves around b

(iii) robot a finishes moving iv) robot c now moves around a

(v) robot c finishes moving

Fig. 2. Illustration of robots collectively "walking".

(a) Base facts:

```
Neighbor(a,b)    Neighbor(b,a)
Neighbor(b,c)    Neighbor(c,b)
Neighbor(a,c)    Neighbor(c,a)
At(a,(0,0))      At(b,(0,2))
At(c,(1,√3))
```

(b) Facts added after applying RULE1:

```
Dist(b,2)    Dist(c,~2.5)
Dist(a,4)
```

(c) Facts added after applying RULE2:

```
Farther(a,b)    Farther(a,c)
Farther(c,b)
```

(d) Facts added after applying RULE3:

```
MoveAround(a,c,b)    MoveAround(a,b,c)
```

Fig. 3. Example of how the facts are updated for the Walk program.

facts represent the assumptions being made about the system and the information that is available on that system.

We use a simple example to show how logical programming works in Meld. In our example, we consider a set of cylindrical robots that move across a flat plain by rolling against one another. We will assume that each robot is able to identify its neighbors and knows its location in a globally consistent coordinate system. We use two kinds of base facts, one to represent each type of information available in the system: a `Neighbor(S,T)` fact which specifies that robot `S` is adjacent and connected to robot `T`; and an `At(S,P)`

fact which specifies that robot S is at the point P=(X,Y).[1]

Consider the example program, shown in Figure 1, which takes an ensemble of three robots and "walks" them to a specified location. To support the specified location, we will assume there is a function destination() which returns the target location. We will also assume that the three robots each start adjacent to each other; we will try to maintain this invariant. Assuming the starting positions are a triangle at the origin, as shown in Figure 2, will give us the base facts shown in Figure 3(a). Suppose also that destination() always returns (0,4).

As we look at our collection of rules, shown in Figure 1, and our base facts, shown in Figure 3(a), we can see that we cannot satisfy RULE2 or RULE3 as we have neither Dist(S,D) nor Farther(S,T) facts available. We are, however, able to apply RULE1 as it only requires a single At(S,P) fact, of which we have three. This gives us a new Dist(S,P) fact for each catom, as depicted in Figure 3(b). Notice that after we have applied this rule, RULE3 is still inapplicable, but we now have enough facts to apply RULE2 giving us a collection of Farther(S,T) facts as shown in Figure 3(c). Finally, RULE3 can be applied to produce two instances of MoveAround(S,T,U) as shown in Figure 3(d). At this point we can verify by inspection that no additional facts are derivable using any of our rules. Without additional facts, rules, or actual motion, our program has run to completion.

We note that this entire program and the processing has been described in a global perspective, and not in the view of individual robots. Meld provides the programmer this global abstraction, and will compile down to code which accomplishes this processing in a distributed fashion across the participating robots, as described later in Section III.

B. Facts With Side-effects

As we have seen, our program quickly determined that robot A should move around one of the other robots, and then immediately halted. Rather than halt we would like the program to cause robot A to move and then continue moving other robots until the ensemble reaches the destination. We accomplish this by allowing some facts to cause a side-effect to occur. Thus, for our example, we will interpret MoveAround(S,T,U) as a motion primitive which will cause S to roll around the outside of T until it touches U. If S and U are already in contact, it will roll S around T to the other side of U as shown in Figure 2.

It is worth noting that there is nothing particularly special about this MoveAround(S,T,U) primitive for motion. Any other motion primitive would work equally well from the perspective of the language; which motion primitives make sense is dependent upon the particular system of modular robots. If the low-level hardware drivers only provide functionality to step clockwise and counter-clockwise, then we would use MoveAround(S,T,U) facts to derive instances of StepCW(T) and StepCCW(T) using appropriate rules. It may even be desirable to have a variety of motion primitives available as well as other side-effects relating to sensors, changing color, or any other piece of physical state.

[1]One can consider a weaker set of assumptions where robots cannot determine their global locations, but only the relative locations of their neighbors. This could easily be represented using base facts of the form neighbor(S,T,D) which indicate that S and T are neighbors and that T is in the D direction from S, where D is specified in some local coordinate system known only to S. For a sufficiently dense ensemble, this can be used to generate a global coordinate system and the At(S,P) facts [9]. In general, the base facts should reflect the available sensor information and assumed knowledge of the world.

```
AGGREGATE: Dist(A, min<n>).

RULE1A: Dist(A,0):- At(A, P),
                    P = destination().

RULE1B: Dist(A,n+1):- Neighbor(A,B),
                      Dist(B,n).
```

Fig. 4. Modified rules for walk program using gradient distances and min aggregates.

The key issue with side-effects is that they can result in a change in the base facts. Recall that our base facts At(S,P) and Neighbor(S,T) were dependent upon the physical layout of the ensemble. If one robot moves, it will effect changes in these base facts. Some new base facts will become true as robots gain new neighbors and some old base facts will become false as robots lose neighbors. The results of these changes are automatically handled by the language. Any fact that is true given the new base facts will eventually be proven and any fact which is supported by a proof including a fact that has changed will eventually cease to exist via a process called *Deletion*, explained in Section III-B. It is important for the programmer to be aware that while untrue facts will vanish, they will not always do so instantaneously and thus it is necessary to be aware of when facts may potentially be stale.

When MoveAround(S,T,U) is proven (e.g., in the walking program example) it causes the motion to occur; a new set of base facts is created, and old facts inconsistent with these facts are deleted. The program can continue to run, and proves additional MoveAround(S,T,U) facts until the target location is reached.

C. Aggregates

In addition to the regular rules which prove all possible instances of a fact, rules may also be used to produce an aggregate over all provable instances. There are two types of aggregate rules. One type of aggregate picks out a particular fact from all provable facts, such as minimizing or maximizing over one of the fields in the resulting fact. The other type of aggregate computes a result based on all provable facts, such as a summation over one field. The ability to take minima and maxima and compute sums can be useful for many things such as calculating a shortest path. In addition, aggregates could be done for any arbitrary method of determining the "best" value for a field as long as it defines a total ordering and some method for munging a collection of facts to produce just one.

As an example, suppose RULE1 of the Walk program in Figure 1 was rewritten to be based on gradients from the destination as in Figure 4. Observe, then, that using RULE1B, a robot will obtain a distance from each of its neighbors, and potentially have many different distances available to it. Instead of this behavior, we wish to pick a single distance and the one we want is the shortest known. Thus, by using the min aggregate, only the smallest known distance will be considered to be valid at each robot. When a robot receives a new *Dist* that is greater than the current one, it will ignore it and not prove anything new with it. When, however, it gets a *Dist* that shows the robot to be closer than it previously supposed, it will drop the old value, all the facts proven from the old value will disappear and be replaced by facts proven from the new best distance.

III. IMPLEMENTING MELD ON MODULAR ROBOTS

In this section, we consider some of the implementation issues that arise when applying the Meld language to modular robots, particularly with regard to distributed execution and complexities in deleting facts. We will also consider how Meld helps solve some critical problems faced when programming modular robots, and also investigate some new concerns that arise in our new language.

A. Distributed Execution

Although we have discussed Meld programs from a purely global perspective, in practice, the actual execution of the programs will be distributed across a modular robotic ensemble. In a distributed environment, it will be necessary to choose a robot where each fact will be located. Furthermore, a method of sharing facts is need to permit rules to use facts that reside on multiple different robots.

To distribute the computation across the ensemble, we adopt the idea of localization from P2 [5], which requires explicit specification of the location of a fact. As a matter of convention we take the first element of any fact to be the robot where it is known. Furthermore, if a computation involves facts known at different locations, then we require a way for the robots that own the facts to communicate. We therefore stipulate if a rule includes facts known at both S and T then the fact `neighbor`(S,T) must also be included. If we again consider the example shown in Figure 1 we notice that each `Dist` is known at the catom being talked about, `MoveAround` is known at the catom that needs to move, and `Farther` is known at the father away catom, all of these facts are where we would expect them. Also notice that RULE1 and RULE3 are local rules, while RULE2 contains the `Neighbor`(S,T) fact as required.

Given our convention, it is relatively straightforward to implement the sharing of facts across the distributed environment. If a rule requires some facts known to A and some facts known to B, then A can send B a message whenever it gets a complete set of its half of the facts. B can then combine this message from A with its share of the facts to complete the rule. To reduce the number of messages sent, we always take B to be the node where the resulting fact should be known and A to be the other involved node, if any. Applying this to RULE2, we see that each node must send it's `Dist` to its neighbors so that they can determine if they are farthest from the destination. For a more detailed description of how this works, see [5].

RULE1: `Foo(A):- Bar(A).`

RULE2: `Foo(A):- Neighbor(A,B), Foo(B).`

Fig. 5. Rules resulting in a cyclic proof.

B. Deletion

In a modular robotic setting with actuation, the base facts representing physical state will change frequently, so we must ensure that deletions of facts occur efficiently and correctly. When a fact is deleted, all facts that were derived from it must be deleted as well. This can be accomplished by applying the rules to determine which other facts were derived and need to be deleted, just as when deriving new facts. Each of these facts is then deleted. A problem, however, arises if we delete a fact that has an alternative proof available that does not rely on any deleted facts and is, therefore, still valid. This problem can be resolved through the use of reference counts to keep track of how many derivations exist for a given fact. Then, when a fact is deleted, we decrement its reference count and only if the count goes to zero do we perform the regular deletion operation [5].

Even with this approach, a problem arises when we have a cyclic proof, i.e., one where a fact may be, indirectly, derived from itself. To illustrate this, we consider the example program in Figure 5 running on two adjacent robots. Assuming we start out with the `Bar`(a) fact (Figure 6(a)), running the rules gives us the facts shown in Figure 6(b). In this example, when `Bar`(a) is deleted, we see that `Foo`(a) must be deleted by looking at RULE1. When we delete `Foo`(a) we simply decrement its reference count, see that it is non-zero, and presume that an alternate proof exists. This leaves us with the facts shown in Figure 6(c). Notice that we believe `Foo`(a) and `Foo`(b) are both true, but if we were to wipe out all of the derived facts and rerun the program, we would not be able to prove either. Thus, reference counting alone does not fix all of the problems with deletion.

As this problem arises directly from cyclic proofs, we would like a way to break the cyclic dependencies. This can be implemented in the compiler with a relatively simple transformation of any program into a cycle-free version. We first add to each fact an extra term representing the depth of the derivation used to prove the fact. In each rule, the depth is set to one plus the maximum depth of the facts used as shown in Figure 7. This serves to distinguish two copies of the same fact when one was proven from the other. Note that if A was used in the derivation of B then Depth(A) $<$ Depth(B). This is adequate to remove any cyclic proofs where A is used to prove A as Depth(A) $\not<$ Depth(A). The use of the min aggregate on the rewritten facts ensures that

(a) Initial facts:

```
Neighbor(a,b) (x1)   Neighbor(b,a) (x1)
Bar(a) (x1)
```

(b) Facts after application of rules:

```
Neighbor(a,b) (x1)   Neighbor(b,a) (x1)
Bar(a) (x1)          Foo(a) (x2)
Foo(b) (x1)
```

(c) Facts after deletion of Bar(a) using naïve method:

```
Neighbor(a,b) (x1)   Neighbor(b,a) (x1)
Foo(a) (x1)          Foo(b) (x1)
```

(d) Facts after deletion of Bar(a) using advanced method:

```
Neighbor(a,b) (x1)   Neighbor(b,a) (x1)
```

Fig. 6. Example of facts for the cyclic proof program. The reference count of number of derivations is shown in parentheses after each fact.

```
AGGREGATE: Foo(A, min<n>).

RULE1: Foo(A,n):- Bar(A,m), n = m+1.

RULE2: Foo(A,n):- Neighbor(A,B,m1),
                  Foo(B,m2), n = 1+max(m1,m2).
```

Fig. 7. Rewritten rules for cyclic proof program.

only the version of a fact with the shallowest derivation tree can be used in further proofs, preventing cyclic programs from looping forever.

However, the use of aggregates can lead to additional problems if we are not careful. When an aggregate fact is deleted, it is necessary to instate the next best value for the aggregate in its place, which may trigger the proof of additional new facts. If deletion is immediately followed by the update of the aggregate facts, in the above code, the system can continue to prove new, deeper derivations of `Foo(a)` and `Foo(b)` while deleting the shallower ones. This will result in the code running forever, as deletions chase behind, but never catch up to the newly proven facts. This problem can be simply resolved by waiting for deletion to recursively complete before updating values for any affected aggregates.

C. Addressing the Issues in Programming Modular Robots

The Meld language can help resolve some of the critical obstacles to effective programming of modular robotic ensembles. In particular, it alleviates problems relating to:

1) Perspective: Most programming languages force code to be written in the local perspective of individual robots. As we noted before, it is often easier to describe what we would like a modular robotics system to do while looking at a subset of the robots rather than at one individual in the system. Meld allows us to write rules across multiple nodes, making it easy to write programs from a group or global perspective. Meld automatically compiles down to code that is distributed across the individual robots.

2) Fluctuating Topology: Another critical problem is that motion within a robotic ensemble results in an ever changing ensemble topology. As discussed in the previous section, Meld neatly takes care of these changes by deleting any facts that depend on the lost neighbor relations and building up new facts based on the new ones. This means that the program can ignore the changes in the ensemble topology as they are handled automatically. Each rule the programmer creates depends only on the current neighboring nodes and need not care about which nodes were neighbors in the past and which will be neighbors in the future. The programmer need only worry about the current state of the ensemble.

3) Uncertainty: While it will take additional work to determine whether this system can be extended to be inherently fault-tolerant, it does lend itself to writing fault-tolerant code. After a robot stops moving, it will return to figuring out what it ought to be doing independently of how it arrived at it's current location. Thus, whether or not the robot reached it's intended location, it will determine it's best next step. Any location that may have been expecting the robot will cease to expect it as the broken `neighbor` relation from the moving robot will automatically delete any expectations that depended on the moved robot. Thus, as long as the program was written to allow robots to compute an appropriate course of action from any location within the ensemble the algorithms will simply continue.

D. Potential Concerns

As with any change of representation, Meld will make some things easier to represent, while others become more difficult. In particular, classical logic programming is poor at representing state. As long as all of the interesting state is a direct consequence of the base facts and as long as it changes when the base facts change, the state is easy to keep track of as it follows "for free" from the changes in the base facts. While the presentation here takes only the `neighbor` and `at` relations as base facts, it is a simple matter to extend these with other useful facts including things like sensor data. Thus it is easy to allow state to change dynamically in conjunction with more things than just the neighbors and locations. Other state that changes independently from the base facts is much more difficult to represent.

This begs the question: how important is state that changes independently from the ensemble topology, sensor data, or other base facts and, if truly needed, is there a way we can extend the system to make this dynamic state easier to deal with? Contrary to our initial thoughts, our experiences using Meld seem to indicate that aggregates may eliminate the need to represent intermediate dynamic state. If, however, other problem domains end up requiring significant dynamic state then we propose moving to a linear version of the logic for these programs.

The idea behind a linear logic [3] is that each fact can only be used once for one rule and it is then deleted from the table. This makes deleting old state trivial as there is now a method, apart from motion, of deleting arbitrary old facts. This allows one to maintain state by deleting the old piece of state whenever a newer piece is generated.

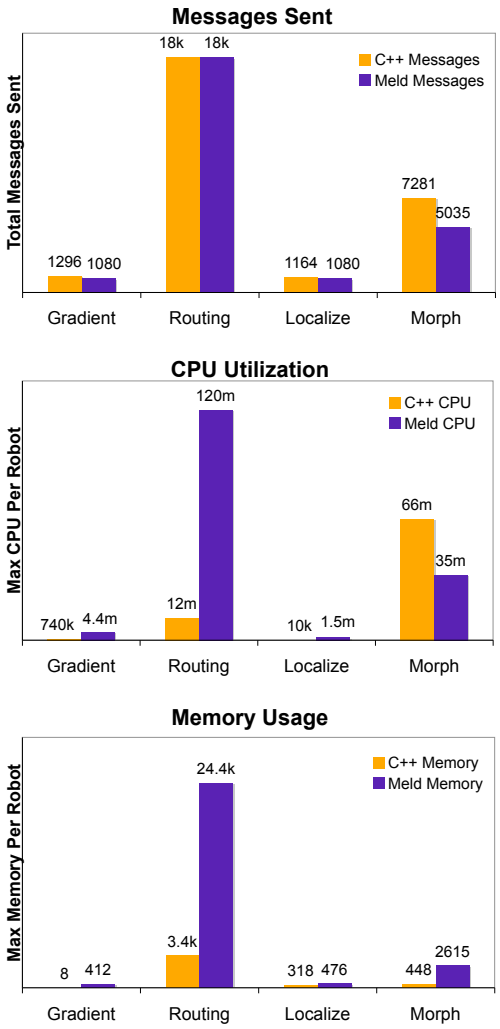


Fig. 8. Performance of Meld applications compared with C++ equivalents. All were run on dprsim [2].

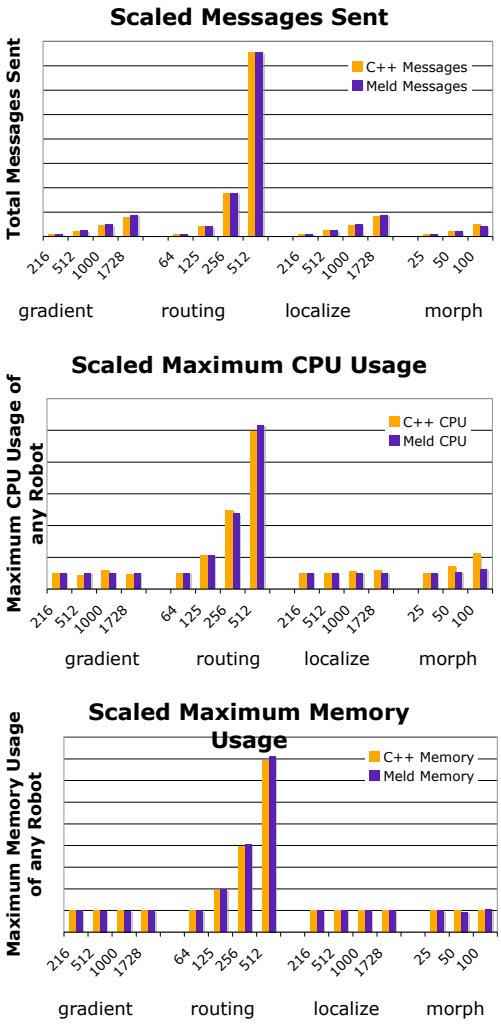


Fig. 9. Scaling properties of Meld applications compared with C++ equivalents over varying ensemble sizes. For each application, the bars are normalized to 1.0 for the smallest ensemble for each language.

IV. EVALUATION

A. *Efficiency and Scalability*

We evaluated the performance of Meld in terms of network traffic, memory usage, and CPU utilization by comparing applications written in both Meld and C++. The programs used are all of interest to modular robots, and were run on dprsim [2], the modular robotic simulator from the Claytronics project. Gradient generates a gradient field across the ensemble, which is the basis of one approach to programming modular robots. Routing implements a standard distance vector network routing algorithm and is useful for communication between arbitrary robots. Localize computes a global coordinate system from the local observations of individual robots [9]. Morph changes the physical topology of the ensemble from one arbitrary shape to another.[2]

For each application, the total number of messages sent between robots, as well as the maximum memory and CPU used on any one robot, were measured (see Figure 8). Meld and C++ applications generated comparable network traffic, but Meld used an order of magnitude more memory and CPU. For large ensembles, however, a more important metric is how well performance scales as the ensemble size increases. The measurements for varying sized ensembles, shown in Figure 9, demonstrate that Meld applications scale just as well as C++ ones.

Further investigation showed that the extra memory used by Meld was due to the need of Meld to hold onto every fact ever proven, whereas a C++ programmer can better predict which information will be useful again in the future and which will not. It is not clear that the memory comparison was really fair, as Meld is prepared to deal with fact deletion as a result of network topology changes, while the C++ programs are unable to do so. The added cost in memory of adding this support to the C++ programs is presently unknown. In addition, the Meld compiler is immature and not well optimized, so it may be possible to make up a large part of the performance difference through compiler optimization. In particular, significant CPU time is currently

[2]All of these programs along with videos of the Morph program are available at `www.cs.cmu.edu/~claytronics/iros07-meld.html`.

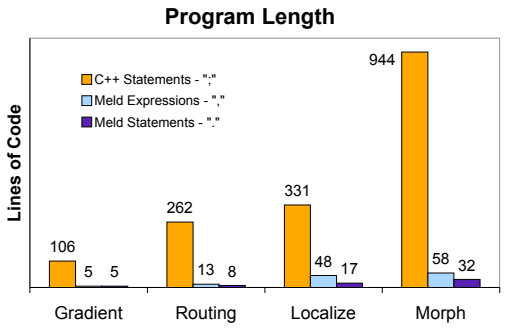


Fig. 10. Lines of code in C++ vs Meld

expended performing database lookups using a naïve linear table-space, so there is potential for much improvement with simple optimizations.

B. Length of Programs

As one purpose of Meld is to simplify the process of writing programs for modular robots, it is necessary to somehow quantify the results. To do this, we have measured the number of lines of code in each of the applications described above. For C++, the metric used for counting lines of code was the number of lines containing semicolons in the program source. For Meld there is no established metric, so we measured both commas and periods. These measurements are shown in Figure 10. These metrics show that Meld programs are frequently twenty to thirty times more concise than the equivalent C++ applications. This difference is a compelling demonstration of the conciseness possible when writing programs with Meld.

V. RELATED WORK

Existing approaches to programming modular robots are almost as varied as the robots themselves. Most of the work to date has focused on making it easier to program *individual* robots, whereas this work focuses on how to program the *ensemble* as a whole.

Meld is directly inspired by P2 [5], which uses a declarative logic programming style to program overlay networks. Meld extends P2 in several ways to make it suitable for programming modular robots: e.g., we added new primitives, we changed the compiler and runtime system to support movement, and we implemented an efficient form of deletion.

Declarative programming styles have also been used effectively in the context of sensor networks [6], [8], [10]. While these languages use a declarative approach to programming ensembles, they focus on data aggregation and analysis rather than causing movement and state changes. In addition to the fact that units in a sensor network generally do not move, another key difference in this domain that is of prime importance is which nodes a given node can communicate with, not where the nodes are physically located.

Previous work that is more closely related to robots and the ensemble perspective includes OSL [7] and Proto [1]. These languages are declarative, and allow programmers to specify ensemble behavior at a fairly high level of abstraction (which is then compiled to local rules that run on each unit). In this, these have been inspirational to our work. A key difference, however, is that the genesis of these languages is stationary nodes that communicate wirelessly: hence it is the communication topology—e.g., who you can communicate with in less than x hops—that matters rather than the actual geometry of your neighbors.

VI. CONCLUSIONS

While programming modular robotics is presently a difficult task, it can be greatly simplified by making use of the appropriate tools. Meld provides a big step forward in our development of these tools as it demonstrates that logic programming can greatly simplify the programming of modular robots by making programs very concise while maintaining reasonable efficiency.

VII. ACKNOWLEDGEMENTS

We would like to thank the Claytronics group at Carnegie Mellon University and the DPR group at Intel Corporation for their assistance. In particular, we thank Casey Helfrich for his support on dprsim, and Jason Campbell, Michael De Rosa, and Siddharta Srinivasa for their ideas and valuable feedback. We would also like to thank Jake Donham, Bryant Lee, and Garth Gibson for their assistance with the initial version of Meld.

REFERENCES

[1] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

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

[3] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

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

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

[6] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[7] 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.

[8] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proc. of the Int'l conf. on Information Processing in Sensor Networks (IPSN'07)*, April 2007.

[9] P. Pillai, J. Campbell, G. Kedia, S. Moudgal, and K. Sheth. A 3d fax machine based on claytronics. In *IEEE/RSK Int'l Conf. on Intelligent Robots and Systems*, pages 4728–35, Oct. 2006.

[10] K. Whitehouse, C. Sharp, E. Brewer, and D. 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.