Distributed Watchpoints: Debugging Large Modular Robot Systems

Michael De Rosa, Seth Goldstein, Peter Lee, Jason Campbell and Padmanabhan Pillai

The International Journal of Robotics Research 2008; 27; 315
DOI: 10.1177/0278364907084986

Distributed Watchpoints: Debugging Large Modular Robot Systems

Michael De Rosa
Seth Goldstein
Peter Lee
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891, USA
{mderosa, seth, petel}@cs.cmu.edu

Jason Campbell
Padmanabhan Pillai
Intel Research Pittsburgh
4720 Forbes Avenue, Suite 410
Pittsburgh, PA 15213, USA
{jason.campbell, padmanabhan.s.pillai}@intel.com

Abstract

Distributed systems frequently exhibit properties of interest which span multiple entities. These properties cannot easily be recognized from any single entity, but can be readily detected by combining the knowledge of multiple entities. Testing for distributed properties is especially important in debugging or verifying software for modular robots. We have developed a technique we call distributed watchpoint triggers which can efficiently recognize distributed conditions. Our watchpoint description language can handle a variety of temporal, spatial and logical properties spanning multiple robots. In this paper we present the specification language, describe the distributed online mechanism for detecting distributed conditions in a running system and evaluate the performance of our implementation.

KEY WORDS—cellular and modular robotics, distributed robot systems, programming environment

1. Introduction

1.1. Modular Robotics

Modular robotics has attracted considerable research attention over the past decade by pursuing the idea that many simple, often homogeneous modules can form an ensemble whose functionality meets or exceeds that of a traditional monolithic robot. The flexibility of such a composite system could be high and the fabrication of individual modules would benefit from the economies of scale inherent in fabricating larger numbers of similar units (Chen and Burdick 1995; Pamecha et al. 1996). There are (generally speaking) two main classes of modular robots: lattice type and chain type. Lattice-type robots are arrayed in some regular repeating structure and move only between the grid points in the lattice (Kotay et al. 1998). Chain-type robots operate without a constraining lattice and typically form one or more chains (either open or closed) of modules (Støy 2004; De Rosa et al. 2006).

In addition to the expected work on hardware design and control for such robots, modular robotics has proven to be a fertile research area for inquiry into a variety of distributed programming tasks. In the modular robotics community, notable work has been published on such topics as distributed sensor fusion, distributed localization (Reshko 2004), leader election and distributed motion planning (Støy 2004; De Rosa et al. 2006).

Modular robotic ensembles of the order of tens of modules have been constructed for research purposes (Yim et al. 1997; Murata et al. 2002; Jorgensen et al. 2003), but ensembles of larger sizes are much less common. This has not stopped several projects (Nagpal 2001; Støy 2004; Goldstein et al. 2005; Fitch and Butler 2006) from exploring algorithms for thousands to millions of robots, but such experiments must currently be carried out in simulation.

Debugging tools for robots are typically integrated into a simulation (Collett et al. 2005) or development (Tansley 2007) frameworks and provide support for debugging motion plans and other algorithm details for single or small groups of ro-

The International Journal of Robotics Research
Vol. 27, No. 3–4, March/April 2008, pp. 315–329
DOI: 10.1177/0278364907084986

bots (Gerkey et al. 2003). Stand-alone debugging tools for robotics are relatively rare (Lamine and Kabanza 2002) and, to the best of our knowledge, there are currently no generally accepted techniques for debugging the distributed behavior of large numbers of robots.

1.2. Distributed Debugging

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

Tools to assist programmers in debugging distributed algorithms are few in number and generally inadequate. Most are forced to fall back on standard debugging methods, such as debuggers (e.g. GDB; see FSF (2007)) or logging through console input/output (I/O) such as printf(). Both debuggers and logging must be used very cautiously, or their file/console I/O can impose unintended serialization (altering the timing behavior of the robot ensemble) and possibly masking some bug manifestations. Debuggers are useful for detecting errors local to an individual thread or process, but are not effective for errors resulting from the interactions or states of multiple threads of execution that span multiple modules. Console or file logging may be used to detect some of these errors, but this requires collecting all potentially relevant state information at each robot. The information must then be centrally collected, correlated and post-processed in order to extract the details of the error condition. This requires significant effort, skill and even luck on the part of the programmer.

One would like to have a tool that can allow programmers to easily specify and detect *distributed conditions* in a multi-robot ensemble. Such conditions constitute logical relations between state variables that are distributed both temporally and spatially across the ensemble. In general, distributed conditions cannot be detected by observing the state of any single robot or even the whole system at any single time. For example, in debugging a distributed motion planning algorithm, we may wish to detect whether two adjacent modules each initiate motion within four iterations of their respective main loops. A tool which can detect such conditions can provide insights into the logical and temporal behavior of the systems and help to pinpoint defects in distributed algorithms.

Unfortunately, detecting arbitrary distributed conditions which may range freely inside an ensemble of modular robots can be quite difficult, given the large number of module subsets that must be examined. As an example, a condition that can occur in any three-module subset within an ensemble of 1,000 modules would require examining almost 10^9 possible subsets of size three. To avoid this problem, we limit ourselves to the subclass of conditions which can be detected within *fixed-size, connected subensembles* of the entire ensemble. That is to say, there must be at least one network connection (and usually physical adjacency) between each element of the subensemble and some other element of the same subensemble. Taking the above example, given a condition involving three modules, in a topology where each module has at most four neighbors, one would have to examine at most 4 million different subensembles. This is a reduction of nearly three orders of magnitude when compared with the unconstrained case above.

We have developed a *distributed watchpoint* mechanism that permits programmers to easily specify spatiotemporal conditions across an ensemble of robots and trigger debugging actions when they are detected. We have also developed a corresponding fully distributed, online detection algorithm that can be used to efficiently detect such distributed spatiotemporal conditions in both simulated robotics systems and physical hardware.

The remainder of this paper is primarily devoted to the introduction of our mechanism for specifying distributed conditions and our design and implementation of a distributed algorithm to detect these conditions. We also present an evaluation and analysis of the distributed watchpoint system, followed by a discussion of its potential applicability to other disciplines.

2. Related Work

In considering the design of a distributed debugging system for modular robots, there are three relevant areas of research to consider. The first of these is distributed and parallel debugging, including the Chandy–Lamport global snapshot algorithm (Chandy and Lamport 1985) and subsequent related work on global predicate evaluation (Fromentin et al. 1994; Chase and Garg 1998; Hurfin et al. 1998). Snapshotting and global predicate evaluation are valuable tools, but they are geared towards the problem of finding a single instance of a particular global configuration, where the conditions that manifest in modular robots can be numerous and localized to connected subensembles. In addition, global snapshots require the aggregation of all relevant data at a central point, resulting in a large communications overhead (Yang and Marsland 1992). A related technique is deterministic replay (Bacon and Goldstein 1991; Xu et al. 2003), which provides the capability to reproduce the distributed actions of a multi-agent system by capturing the order and timing of messages. Unfortunately, deterministic replay algorithms typically require a shared bus or memory and assume that all important causal channels can be recorded and replayed (which is not the case for modular robots). Other important parallel debugging tools include static code analysis tools such as race detectors (Savage et al. 1997; Carr et al. 2001), which can detect many (but not all) data races. While race detectors are important tools, they are not general debugging aids.

Another relevant research area involves the development of logic-based verification and proof tools. Specifically, linear temporal logic (Pnueli 1977), an instance of a modal temporal logic, is capable of representing and reasoning on infinite state sequences, such as those that might be generated by FSM-style robot programs. This capability of linear temporal logic was exploited by Lamine and Kabanza (2002), who developed a linear temporal logic-based model verification tool for single mobile robots.

Finally, declarative overlay network systems, such as P2 (Loo et al. 2005), provide a general purpose tool for the computation of distributed flow functions, which could include debugging primitives. In fact, P2 includes debugging support which leverages the system's ability to compute arbitrary distributed functions (Singh et al. 2006). However, the focus of the P2 project is not on robotics, and as such it does not explicitly deal with the dynamic point-to-point topologies found in modular robotics. In addition, the use of P2 for debugging implies the adoption of the P2 programming paradigm, which may not be appropriate for all applications.

3. Describing Errors

The first step in detecting distributed error conditions is being able to represent them concisely. To that end, we have created a simple watchpoint description language, based on a fragment of linear temporal logic with the addition of predicates for state variable comparison and topological restriction (Figure 1). Linear temporal logic serves as the inspiration for this simple descriptive syntax and provides meaningful semantics for several of the operators.

3.1. Abstract Machine Model

We consider a simplified machine model for each modular robot: each module is represented by a number of named numeric state variables and an array of neighbors. We assume that each module iterates through three atomic phases: computation, state variable assignment and communication. Computation may take an arbitrary amount of time and each module can communicate only with its neighbors via a reliable first in first out (FIFO) channel. Furthermore, we assume that each module has a copy of the watchpoint and that each module has the state variables needed by the watchpoint expression. We explicitly do not require that all modules have the same code image, merely that they have compatible state variables. Finally, we assume no synchronous execution or shared clock between modules. This simplified model does not entail loss of generality, as we can express run-loop, finite-automata or event-driven programs using it.

⟨watchpoint⟩ ⟶ ⟨module decl.⟩ ⟨bool⟩
⟨module decl.⟩ ⟶ `modules(`*string*$^{+}$`)`
⟨bool⟩ ⟶ `not` ⟨bool⟩
| ⟨bool⟩ `and` ⟨bool⟩
| ⟨bool⟩ `or` ⟨bool⟩
| `neighbor(`*string string*`)`
| `(` ⟨compare⟩ `)`
| `(` ⟨bool⟩ `)`
⟨module⟩ ⟶ *string*
| `last.` ⟨module⟩
| `next.` ⟨module⟩
⟨compare⟩ ⟶ ⟨numeric⟩ ⟨c-op⟩ ⟨numeric⟩
⟨numeric⟩ ⟶ ⟨state var⟩
| *integer*
| (⟨numeric⟩ ⟨m-op⟩ ⟨numeric⟩)
⟨state var⟩ ⟶ ⟨module⟩ `.` *string*
⟨c-op⟩ ⟶ `<` | `>` | `==` | `!=` | `>=` | `<=`
⟨m-op⟩ ⟶ `+` | `-` | `*` | `/`

Fig. 1. Extended BNF grammar for the watchpoint description language.

3.2. Watchpoint Expressions

As was mentioned in Section 1.2, for practical representation of distributed error conditions, we make a key assumption: *the error must be able to be represented by a fixed-size, connected subensemble of robots in specific states*. Allowing disconnected subensembles would imply an exponential search through all appropriately sized subsets of the total ensemble and distributing information between the members of these subsets would require significant multi-hop messaging.

Watchpoint descriptions begin with a named list of module variables. The length of this list defines the size of the matching subensemble (the connected subgroup whose state satisfies the watchpoint expression) and is implicitly quantified over all connected subgroups of this size in the ensemble. The language includes the standard boolean and grouping primitives, basic mathematical operators, topological restrictions and state variable comparisons. Topological restrictions take the form of the predicate `neighbor(a b)`, and indicate that the two specified modules are neighbors (i.e. in physical or network connectivity). When used in combination and in conjunction with the boolean negation operator, arbitrary module topologies can be expressed. State variable comparisons allow for the comparison of named state variables in one module against constants, other local variables or remote variables on other robots. In addition, state variable comparisons may include ar-

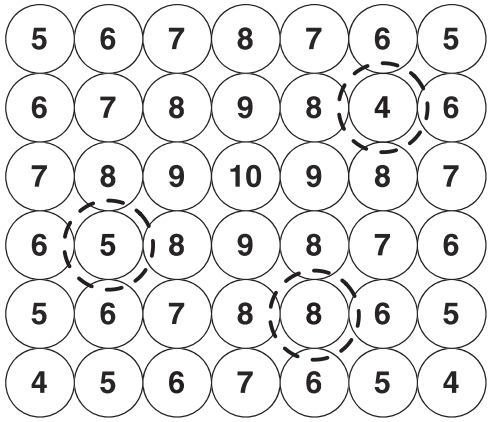

Fig. 2. A gradient field with several non-smooth values (highlighted).

bitrarily nested uses of the `last` and `next` temporal modal operators, which provide access to the past and future states of the robot's state variables. In the case of the `next` operator, this implies that the watchpoint triggers in the "future" and that the state of the robots would need to be rolled back one or more timesteps when the watchpoint triggers.

These simple primitives give us the ability to represent very complex distributed conditions. We can reason along three different axes of a configuration: numeric state variables, topological configuration and temporal progression. Topological restrictions allow us to model (in an abstract fashion) the configuration space of the robots, so that error states related to the physical positioning of neighboring modules may be represented. Temporal modal operators can be used to represent sequences of states, a useful capability for debugging distributed finite state automata.

4. Example Watchpoints

To clarify the use of the watchpoint description language, we present three example watchpoints that demonstrate the detection of errors in common algorithms found in modular robotics: gradient fields, leader election and token-passing.

4.1. Gradient Fields

In many distributed algorithms for modular robotics, gradient fields are used to disseminate information to an entire ensemble of robots and to provide (in the case of monotonic gradients) a shortest-path function available at all points. Gradient fields have been used for motion planning (Støy 2004) and for localization/role assignment (Nagpal 2001).

In the example configuration of Figure 2, we have a two-dimensional array of robots in a rectilinear lattice. Gradient values are propagated from the center to the edge, decreasing by one at each hop. The gradient as a whole should maintain a property of smoothness. However, there are several nodes (marked with dashed circles) where the gradient value has been miscalculated by the underlying algorithm. These errors cannot be detected merely through inspection of a single robot's gradient value, as the property of smoothness exists only in the context of two (or more) gradient values. To represent this error state, we can use the watchpoint shown in Figure 3(a). This watchpoint detects when two neighboring robots have a gradient value that differs by more than one.

An interesting observation about this watchpoint is that we do not need two clauses or an absolute value operator to detect the case where the difference between the values is less than -1. This is due to the fact that this watchpoint is implicitly evaluated for all size-two connected subgroups in the ensemble, and thus will detect the case when a difference is less than -1 by examining the separate subensemble where the assignments of the variables `a` and `b` are reverse, giving a difference of more than 1. This saves effort in one way, but turns out to be a false economy, as it will require the examination of twice as many subensembles. This suggests an as-yet undiscovered optimization may be possible for expressions with similar logical symmetry.

4.2. Leader Election

Figure 4 shows a hexagonally packed array of robots which are attempting to select leaders using some (unspecified) leader election protocol. Each leader must have a path distance of at least two hops to any other leader. It is obvious from inspection of Figure 4 that the algorithm has yielded incorrect results, as there are two leaders within two hops of each other. While this is readily discernible from an omniscient perspective, any single robot will not be able to detect this error condition without communicating with its neighbors. To represent this error state, we use the watchpoint shown in Figure 3(b). Deconstructing the watchpoint expression, we have a connected subensemble of three distinct modules, where two claim to be leaders. As there can be at most two hops between any two modules, the presence of two modules in such a group which are both leaders constitutes a violation of the path distance property from above. It should be noted that this approach will grow significantly more expensive as the hop-distance requirement grows larger (as the size of the watchpoint will increase with the number of hops between leaders).

4.3. Token Passing

As a slightly more complex example, let us consider the problem of token passing in a closed ring network (Figure 5). We would like to enforce the condition that, if robot `x` has the token, then exactly one of its neighbors must have had the token

```
(a) modules(a b); (a.gradient - b.gradient > 1)

(b) modules(a b c); (a.isLeader = 1) and (c.isLeader = 1)

(c) modules(a x b); neighbor(a x) and neighbor(x b) and (x.tok = 1)
      and (((last.a.tok = 1) and (last.b.tok = 1)) or
      ((last.a.tok = 0) and (last.b.tok = 0)))
```

Fig. 3. Example Watchpoints.

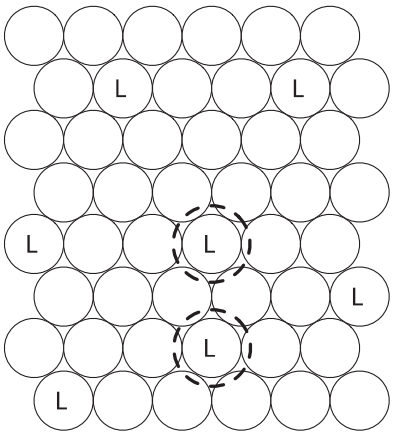

Fig. 4. Incorrect two-hop leader election. Conflicting leaders (with a path distance of less than two hops) are circled.

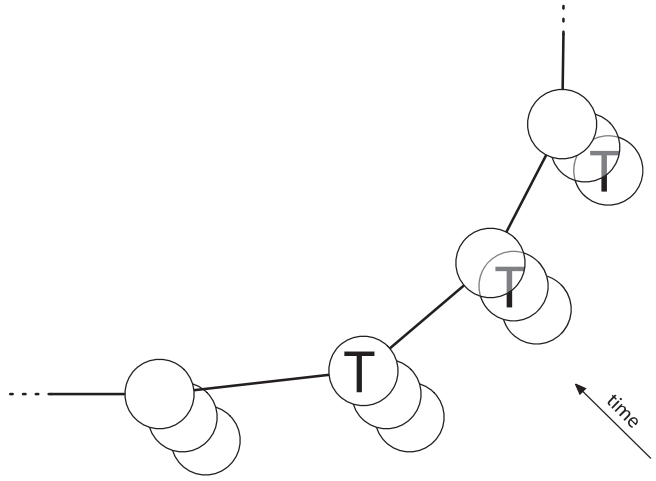

Fig. 5. Part of a token-passing ring. Previous states shown stacked behind current states.

in the last timestep. We can express the violation of this condition with the watchpoint shown in Figure 3(c). Here we have three modules, with module x currently holding the token. An error occurs if both or neither of neighbors of x previously had the token. Note that we need to use topological restriction in this watchpoint, as the requirement that x, a, b form a connected subensemble is insufficient to ensure that both a and b are neighbors of x.

4.4. Summary

The above examples illustrate some of the unique and important characteristics of our watchpoint description system. From the first example, we see that watchpoints are *implicitly quantified over all connected subensembles of appropriate size*, meaning that we do not have to explicitly represent the translations, rotations or skews of possible matching subensembles. The second and third examples together show the conditions where *topological restrictions* are and are not needed. The third example also shows how reasoning over distributed sequences of states allows for the *detection of time-sensitive conditions*.

5. Detecting Errors

The main component in our distributed watchpoint implementation is the PatternMatcher object (Figure 6). A PatternMatcher consists of three subunits:

- an ordered list of n named slots that hold robot id numbers;
- an $n \times n$ bit adjacency matrix;
- an expression tree that both represents the watchpoint expression and stores any accumulated state variables.

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

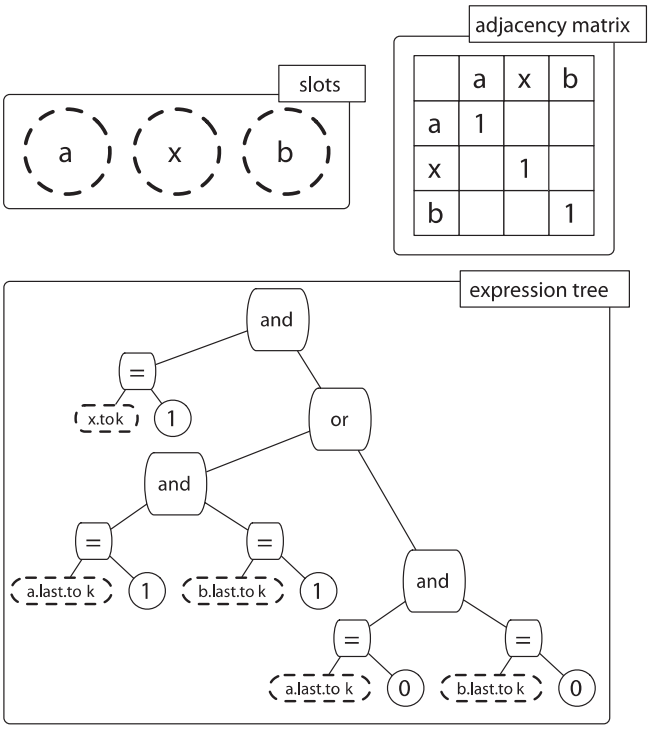

Fig. 6. PatternMatcher object. The expression tree corresponds to the watchpoint expression of Figure 3(c). Variables are shown with dotted outlines.

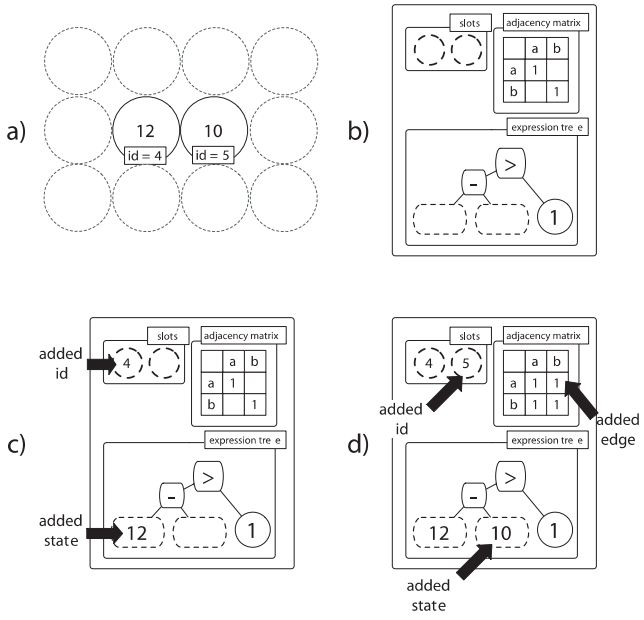

Fig. 7. The population process, with arrows indicating where data is added. (a) Module configuration, showing gradient value and module id. (b) Initial (empty) PatternMatcher for the expression in Figure 3(a). (c) PatternMatcher after being populated with module id 4. (d) PatternMatcher after being populated with module id 5.

5.1. Populating PatternMatcher Variables

In order to use PatternMatchers to determine whether watchpoint expressions have been satisfied, we *populate* the components of the PatternMatchers with information from different modules. This process has three stages that occur at each timestep: (i) filling the next available slot; (ii) filling the expression tree; and (iii) updating the adjacency matrix. When filling slots, we proceed in order: find the first slot that has not been assigned a value, then copy the local module's unique id number to that slot variable. The name of the slot variable is then used in the second phase, where we traverse the expression tree and instantiate any references to the current slot's state variables with the values from the current module. Finally, we query the module for its current list of neighbors and update the adjacency matrix with any connections between the current module and any modules in previously filled slots. These steps are repeated for every module that the PatternMatcher acquires state information from.

As an example, we illustrate the action of the populating process on the watchpoint expression from Figure 3(a). We examine a single PatternMatcher as it is populated by two modules in succession. The two modules have unique ids 4 and 5, and gradient values of 12 and 10, respectively (Figure 7(a)). The PatternMatcher is first populated by module 4. This places the value 4 in slot `a` and the value 12 in the variable `a.gradient`, as indicated by the arrows in Figure 7(c). The PatternMatcher is then populated by module 5. This fills the slot `b` and the variable `b.gradient`. In addition, as `a` is a neighbor of `b`, the relevant entries in the adjacency matrix are updated. The expression tree is now satisfied and the PatternMatcher has matched the condition described by the watchpoint expression.

5.2. Centralized Algorithm

Our initial implementation relied on a single centralized procedure to update all PatternMatchers across an entire (simulated) ensemble. A a set of vectors maintained each robot's state variables. At each timestep, the current values of all state variables used by active watchpoints were appended to the vectors, providing state history for the variables. The simulator also maintained a single set of PatternMatchers (S), which were updated and processed every timestep as described in Algorithm 1. This centralized algorithm is useful for simulated robot ensembles, but debugging actual hardware (or even multi-threaded simulations) required the development of a distributed technique for detecting watchpoint conditions.

Algorithm 1 Centralized watchpoint update.

```
S = ∅
for all modules m do
  create new PatternMatcher p from the watchpoint text
  populate first open slot of p with m
  S = S ∪ p
end for
while S ≠ ∅ do
  T = ∅
  for all PatternMatchers p ∈ S do
    if p matches then
      execute trigger action for p
    else if p is indeterminate then
      for all neighbors n of modules in p's slots do
        p1 = clone(p)
        populate first open slot of p1 with n
        T = T ∪ p1
      end for
    else
      #failure: PatternMatcher is ignored
    end if
    S = S − p
  end for
  S = T
end while
```

Algorithm 2 Distributed algorithm: message handler.

```
message m received by module n
if m.isMultihop then
  if m.dest = n then
    n.R = n.R ∪ m.p
  else
    r = nextRoutingStep(n,m.dest,m.p)
    m.source = n
    send m to r
  end if
else
  if !m.possibleDup or !isDuplicate(m, n) then
    n.S = n.S ∪ m.p
  end if
end if

isDuplicate(message m,module n)
  i_s = index of m.s in m.p's slots
  i_max =max index of n's neighbors in m.p's slots
  return i_s ≥ i_max

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

5.3. The Need For Multi-hop Messaging in a Distributed Implementation

To detect the presence of distributed conditions without the presence of a central control entity, we use PatternMatchers as mobile state aggregators. PatternMatchers are transmitted from module to module via single-hop messaging, accumulating state until they fail or trigger. Unfortunately, using single-hop messaging, the subensemble topologies that can be detected are quite limited. Consider the two subensembles in Figure 8. Figure 8(a) illustrates a subensemble whose modules lie in a linear path, but are not ordered sequentially. With only single-hop messaging, there is no way to propagate the slots (A B C D) in order. Figure 8(b) shows a subensemble which has an ordered path, if one is allowed to "backtrack" to previously visited modules. In order to implement this backtracking (and thus detect this class of subensembles) we introduce multi-hop messaging and rerouting.

Multi-hop messaging is used to move a PatternMatcher back through previously visited modules, so that it can continue to propagate from a different point in the subensemble. PatternMatchers that return to a module that they have already visited in this way are said to have been *rerouted* to the module. Multi-hop messaging and rerouting allow us to detect subensembles of the kind seen in Figure 8(b), which we term *non-linear* ensembles, but not to detect those seen in Figure 8(a), which are *unordered*. To detect unordered subensembles we would have to reorder the slots of the PatternMatcher to correspond to the ordering in the subensemble. To detect all such unordered configurations, we would have to search all permutations of the slots, a potentially expensive proposition (as a watchpoint of size four would have 24 such permutations).

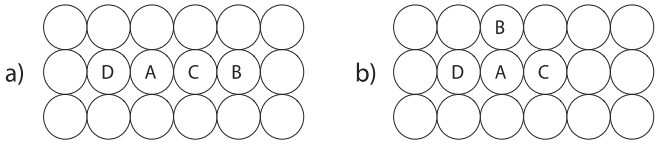

Fig. 8. (a) Unordered and (b) non-linear subensembles for the set of modules (A B C D).

5.4. Distributed Algorithm

The distributed implementation of our watchpoint system involves two components on each module: an update function and a message handler for incoming PatternMatchers. The update function is used at each timestep to create, populate and spread PatternMatchers. Each module maintains three data structures:

Algorithm 3 Distributed algorithm: PatternMatcher update.

```
on each module n
create new PatternMatcher p' from the watchpoint text
S = S ∪ p'
for all PatternMatchers p1 ∈ S do
  populate first open slot of p1 with n
  if p1 matches then
    execute trigger action for p1
  else if p1 is indeterminate then
    message m1 = ⟨p1,false,*,*,*⟩
    send m1 to each of n's neighbors
    for all modules d in p1's slots s.t. d ≠ n do
      message m2 = ⟨p1,true,*,n, d⟩
      module r =nextRoutingStep(n,d,p1)
      send m2 to r
    end for
  else
    #failure: PatternMatcher is ignored
  end if
end for
for all PatternMatchers p2 ∈ R do
  message m = ⟨p2,false,true,*,*⟩
  send m to each of n's neighbors
end for
S = ∅, R = ∅
```

1. a set S of active PatternMatchers;
2. a set R of rerouted PatternMatchers;
3. a local history of watched state variables (used by the *populate* routine).

The PatternMatcher update function of the algorithm is executed at each timestep and is composed of four phases. In the first phase, a new (empty) PatternMatcher is created and added to the set S. Then for each PatternMatcher in S, its first open slot (and associated expression tree variables) is populated with information from the local module. If a PatternMatcher's expression tree is satisfied, the trigger action for the watchpoint is executed. If still indeterminate, the PatternMatcher is then spread via two types of messages: local messages sent to each of the module's neighbors and multi-hop messages sent to all of the other modules already identified as being in the PatternMatcher's slots. Every rerouted PatternMatcher in R is then sent via a local message to all of the module's neighbors. Finally, the sets S and R are cleared.

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

1. the PatternMatcher `p`;
2. a boolean flag `isMultihop`;
3. a boolean flag `possibleDup`;
4. the destination module `dest`; and
5. the source module `source`.

We represent a complete message as $\langle p, isMultihop, possibleDup, source, dest \rangle$. "$*$" is used to denote field values that we do not care about. Message handling is based on whether a given message is multi-hop or local. Local messages are handled by adding their PatternMatcher p to the local set S of active PatternMatchers.

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

5.5. Example

To illustrate how the distributed algorithm functions, we use the watchpoint expression in Figure 9, applied to the five-module ensemble in Figure 10(a). We note that the two subensembles which satisfy this watchpoint are [3 4 2] and [4 3 2]. For clarity of explanation, we assume that all actions at each step occur simultaneously and that message propagation incurs no delay.

The algorithm begins by converting the text of the watchpoint expression into a PatternMatcher (Figure 10(b)). As the contents of the expression tree and adjacency matrix have been described elsewhere, we abbreviate the PatternMatcher as just the sequence of numbers stored in its slots (Figure 10(c)). Thus, an empty PatternMatcher will be abbreviated [$-$ $-$ $-$], while one that had been populated by module 3 and then by module 4 would be [3 4 $-$].

Once an empty PatternMatcher has been created at a module, its first slot is filled by that module (step (a)). PatternMatchers which have failed at this point are then discarded (step (b)). Conversely, those which succeed activate their assigned trigger actions. The remaining PatternMatchers are then spread to the single-hop neighbors. In this example, the PatternMatcher [3 $-$ $-$] at module 3 is spread to modules 2 and 4 (step (c)). All PatternMatchers are then filled by the module that they occupy (step (d)) and failed PatternMatchers are again discarded (step (e)). As the PatternMatchers have two slots filled, they are now spread both by single-hop messaging (to modules 2 and 5) and by rerouting (to modules 3 and 4). Rerouted PatternMatchers are marked with a star and are not filled by the module they arrive at (step (f)). Finally, the rerouted PatternMatchers are spread to the neighbors of the modules they occupy (step (g)), where they are filled again.

```
modules(a b c); (a.var = 0) and (b.var = 0) and (c.var = 2)
```

Fig. 9. Watchpoint for the distributed algorithm example.

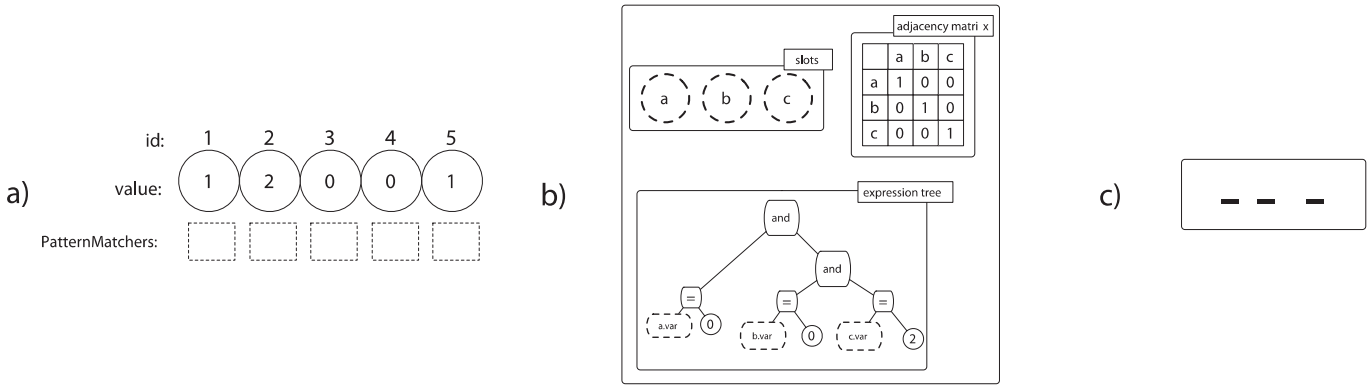

Fig. 10. a) Module configuration. (b) Empty PatternMatcher. (c) Abbreviated PatternMatcher.

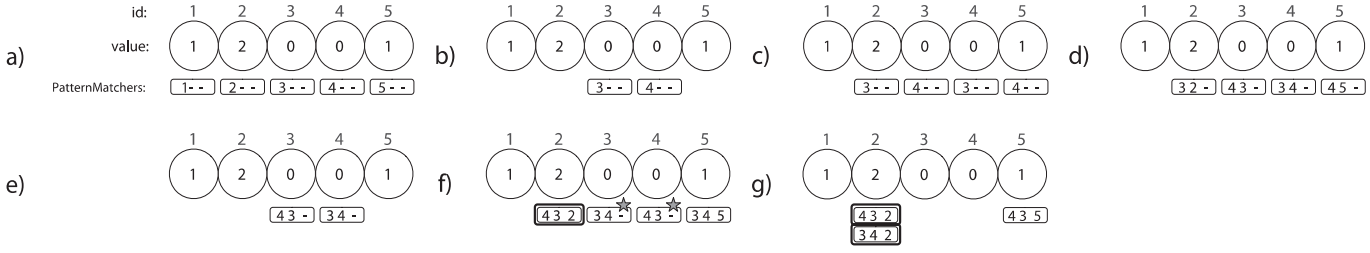

Fig. 11. Distributed algorithm execution. Rerouted PatternMatchers are indicated with a star, while those that have triggered are marked with a double border.

6. Operational Concerns

6.1. Machine Model

As mentioned in Section 3.1, the algorithms we have described assume a specific machine model: namely one that loops through three atomic phases: computation, communication and state variable assignment. This abstract model is easily adapted to accommodate more general programming styles. We address the applicability of our machine model to three common styles: state machine, event-handling and run-loop.

The state-machine programming style (popular in embedded devices) treats the robot as a finite-state automata, which transitions to different states upon the reception of input signals. In this case, we can treat each state transition as a period of computation, followed by variable assignment (the new automata state and any associated variables). The generation and reception of input signals are then the communications phase.

Event-handling systems are similar in nature to state machines, as they respond to external interrupts with event-handling routines. These systems lack the requirement of remaining within a transition system containing a finite number of states and typically handle many more interrupts than a state machine handles transitions. In this case, we can potentially treat each interrupt service request as a discrete timestep, with the interrupt serving as the communications phase. As this may impose an excessive burden on the communications infrastructure (it would imply sending out PatternMatchers after every interrupt is handled) we can group a number of closely spaced interrupt requests together to form a timestep. In this case, the interrupt handlers may access the same state variables, so one can either discard the intermediate values assigned within a timestep or create additional intermediate state variables that can be used to track them.

Run-loop programming is the simplest of all programming styles to adapt for our purposes. We can treat each pass through the main loop as a timestep and record only the variable values

that are present at the end of each loop. If a program sends or receives messages multiple times within the run loop, we can arbitrarily designate one of those occurrences to be the "end" of the loop, and thus mark the point where variable values are recorded and PatternMatchers are propagated. Alternatively, we can make every instance of inter-module communication correspond to the end of a timestep, at which point a single pass through the run loop may contain multiple timesteps.

6.2. Snapshot Correctness

In any distributed algorithm which requires state from multiple asynchronously executing programs, there arises the question of consistency. Without some prior guarantees on the length of a timestep or the presence of a shared clock, there is generally no way to obtain a "snapshot" of the state of multiple modules that corresponds to one instant of wallclock time. What can be obtained is a *consistent* snapshot or a snapshot that represents a set of states that could exist without any messages passed between the modules. We examine the issue of snapshot coherence in two phases: in linear subensembles (which do not require multi-hop messaging) and then in non-linear subensembles.

In linear subensembles, the movement of a PatternMatcher from module to module serves as both the marker and state aggregator for a bounded-size Chandy–Lamport snapshot (Chandy and Lamport 1985). This allows us to capture a consistent snapshot of the modules' state for any state variables whose values persist for at least one timestep.

In non-linear subensembles, the problem becomes more complicated. As the multi-hop messaging phase of the algorithm introduces additional delays in state capture, it is no longer possible to ensure a consistent snapshot with the PatternMatchers alone. There are two possible solutions to this problem. If the timesteps of all modules are known to be of equal length, then one can simply apply a backward temporal shift of 1 to all unfilled variables whenever a PatternMatcher moves as part of multi-hop routing. This ensures that the data the PatternMatcher collects when it reaches a new module has "aged" an amount equal to the number of multi-hop routing steps, which will be equal to the number of timesteps that have elapsed (by the assumption above and the abstract machine model).

If there is no guarantee as to the length of timesteps on different modules, the following approach may be used. When a PatternMatcher with m slots is created, it is given a timestamp by the local module. That timestamp is broadcast to all neighbors within m hops, which record the relevant state values and associate them with the timestamp. When a PatternMatcher arrives at a module, it requests the state variables associated with its timestamp. This is equivalent to separating the beacon and state aggregation phases of the Chandy–Lamport snapshot algorithm. The beacon is the timestamp broadcast and the aggregation is performed by the rerouted PatternMatcher.

6.3. Hardware Overhead

The resources required to implement this technique on real rather than simulated modular robots are modest: RAM needs per module would typically be tens of kilobytes or less (including storage for pattern matchers plus local state memory). Each module should ideally have a unique identifier, but lacking such a facility, a randomized identifier with low likelihood of local repeats would be sufficient (as module ids are needed only to identify modules in the context of a small local group of modules).

The full implementation of our distributed implementation is less than 1,500 lines of C++, so code size is modest as well. In many cases the required communications could be piggybacked onto other pre-existing messages between modules and since exchanges are limited to nearest neighbors many designs would be able to take advantage of nearest-neighbor wired or infrared links for such data. The most constrained resource would probably be CPU for systems already operating close to their computational or communications limits. In those cases, the additional load of transmitting and processing PatternMatchers may result in the system breaking potential real-time constraints. However, it is increasingly feasible to provision all but the smallest robot modules with powerful processors.

7. Optimizations

To reduce the storage, processing and communications demands of our watchpoint system, we implement three optimizations: temporal span detection, early termination of candidate pattern matchers and aggressive neighbor culling.

7.1. Automatic Temporal Span Detection

For each state variable, we must determine the minimum amount of history that must be maintained by each robot. We call this quantity the temporal span of the variable. In addition, we must determine the minimum amount of total state (all state variables plus neighbor information) that must be maintained to allow for watchpoints that trigger in the future. This is the temporal extent of the system. To calculate the temporal span of a variable, we inspect each use of that variable in the watchpoint expression. For each use of the variable, we calculate the temporal extent by assigning a value of $+1$ to each `next` and a value of -1 to each `last`. The sum is then the temporal extent for that particular use of the variable. The temporal span for the variable is the maximal difference between any two temporal extents. This is the amount of history which must be maintained for that variable. Similarly, the maximum positive extent over all variables specifies the size of the total state vector that must be maintained.

7.2. Early Termination

To reduce the number of active PatternMatchers and thus the bandwidth and processing cost of the algorithm, we aggressively cull PatternMatchers that have no chance of succeeding. Before propagating a PatternMatcher to other modules, we first check whether the expression tree can ever match. Just as with short-circuit evaluation of a boolean expression in programming languages such as C, even if the PatternMatcher is not completely filled, subclauses of the expression tree may have already made the whole unsatisfiable. If a match can already be ruled out, the PatternMatcher is deleted and no messaging and computation is required to check any descendant PatternMatchers that would have propagated from it.

7.3. Neighbor Culling

Finally, we reduce the set of neighbors to which a given PatternMatcher can spread by examining the topological constraints of the expression tree. If the constraint `neighbor(a b)` exists in the watchpoint, `b` is the next open slot, and `a` is already filled, then the PatternMatcher can only spread to neighbors of `a`. In the case of multiple topological restrictions, we generate a set of possible neighbors by traversing the tree from the bottom up, treating `and` as set intersection and `or` as set union. Without this optimization, we could conceivably spread a PatternMatcher to a large number of other modules whose connectivity relationships would prevent the watchpoint expression from being satisfied.

8. Evaluation and Analysis

We evaluated the algorithms using a massively multi-threaded multi-robot simulator. To more accurately measure the differential overhead of watchpoint support, we disabled the physics and graphic rendering portions of the simulator. All tests were performed on 100 to 1,000 robots arranged in a cubic lattice packing in stacked planes of 10 × 10 modules. Simulations were conducted for 100 virtual timesteps, where each timestep allowed for arbitrary computation, including message transmission/reception. As computation at each module was of very similar duration, we used the temporal shift technique described in Section 6.2 to ensure snapshot correctness. Message travel time was one timestep. Test configurations were monitored for total execution time, number of active PatternMatchers (segmented by number of slots populated) and the number of successful matches.

8.1. Evaluation Criteria

When evaluating the performance of the algorithms, we noted four significant variables that controlled system performance:

1. the size of the ensemble;
2. the behavior of the program being debugged;
3. the watchpoint detection algorithm (centralized or distributed);
4. the topological restrictions present in the watchpoint expression.

To express the behavior of the program being debugged, in terms of its effect on the PatternMatchers, we used the watchpoint expressions shown in Figure 12: x_1 through x_4 are four independent uniformly distributed integer random variables generated by the host program. Each variable x_n ranges over the integral values from zero to $\max_{x_n} - 1$. We can thus represent the behavior of the host program with the tuple $[\max_{x_1}{:}\max_{x_2}{:}\max_{x_3}{:}\max_{x_4}]$. For example, the tuple [2:2:2:2] will cause half of all PatternMatchers to be discarded after each slot is filled. In contrast, the tuple [4:1:1:1] will cause three-quarters of all PatternMatchers to be discarded after filling the first slot, but then all remaining PatternMatchers will survive until they are fully filled, at which point they will match.

To vary the topological restrictions found in the watchpoint, we compared the results of monitoring the expressions in Figure 12: (a) contains no neighbor restrictions, meaning that it can match any ordered, non-linear subensemble that passes the other criteria in the watchpoint; (b) is the opposite extreme, requiring that all modules in the subensemble lie in an ordered linear path.

8.2. Host Program Dependence

With these test cases we can now examine how variation in host program behavior impacts the number of active PatternMatchers (and thus the execution time). We begin with the "worst case" tuple, [1:1:1:1], where all generated PatternMatchers will always survive, leading to an exponential explosion of PatternMatchers as seen at the top of Figure 13 (note the log scale). After 100 timesteps on 100 robots, over 250,000 successful matches have accumulated in the linear case, while over 1.2 million have been found in the non-linear case (upper line of Figure 13). We can easily halve the number of active PatternMatchers by using the tuple [2:1:1:1], which halves the number of PatternMatchers that survive having the first slot filled. Halving the number of active PatternMatchers at each step (as in [2:2:2:2]) results in the expected decrease by an additional factor of eight (middle line of Figure 13). Comparing [1:1:1:100] and [100:1:1:1] is quite instructive. Both eventually generate an almost identical number of successful matches, but over wildly different trajectories, as can be seen in the area under the curve in Figure 13. The tuple [1:1:1:100], which culls almost all of its PatternMatchers after the last slot has been filled, is much less efficient than [100:1:1:1]. This

(a)
```
modules(a b c d);(a.x1 = 0) and (b.x2 = 0) and (c.x3 = 0) and (d.x4 = 0)
```

(b)
```
modules(a b c d);neighbor(a b) and neighbor(b c) and neighbor(c d) and (a.x1 = 0)
    and (b.x2 = 0) and (c.x3 = 0) and (d.x4 = 0)
```

Fig. 12. Watchpoint expressions for evaluating host program dependance. `x1` through `x4` are per-module independent random variables. (a) Non-linear watchpoint. (b) Linear watchpoint.

Table 1. Successful matches versus execution time.

	Linear			Non-linear		
Program tuple	Number of matches	Centralized time(s)	Distributed time(s)	Number of matches	Centralized time(s)	Distributed time(s)
[1:1:1:1]	265,600	28.1	31.0	1,278,400	72.3	175.2
[2:1:1:1]	133,855	15.2	15.9	647,817	37.0	84.7
[2:2:2:2]	17,231	4.5	7.2	81,447	12.0	24.8
[1:2:4:8]	3,981	10.4	10.1	20,544	15.0	31.1
[8:4:2:1]	4,174	3.0	3.0	20,648	3.3	5.0
[1:1:1:100]	2,547	36.1	31.7	13,388	78.3	181.7
[100:1:1:1]	3,047	2.5	2.5	10,838	2.5	3.2

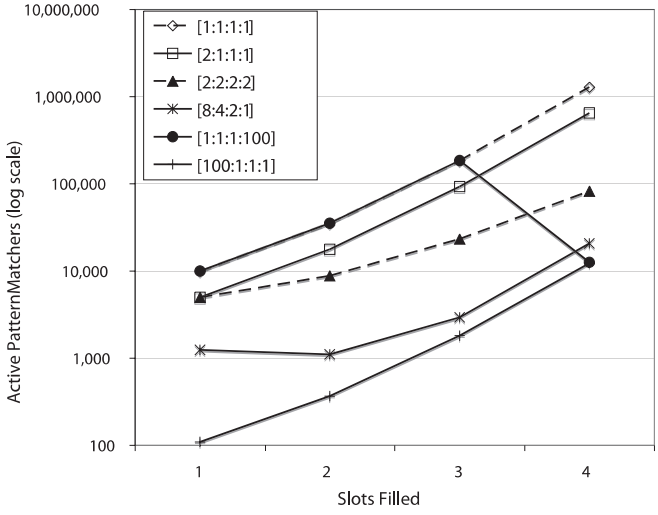

Fig. 13. Active PatternMatchers versus slots filled.

can be seen at the bottom of Table 1, especially in the non-linear case, where [1:1:1:100] takes over 60 times as long as [100:1:1:1]. Continuing to examine Table 1, we note a general linear increase in the amount of time taken by the distributed algorithm as the number of successful matches increases.

8.3. Expected Behavior of PatternMatchers

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

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

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

For a fully linear watchpoint, the expression becomes

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

We note that both expressions are exponential in m (with base k) and linear in n. That is to say, the number of active

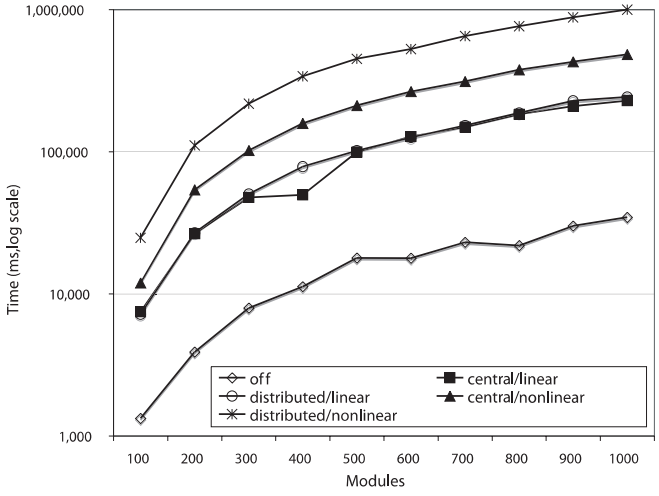

Fig. 14. Execution time versus ensemble size.

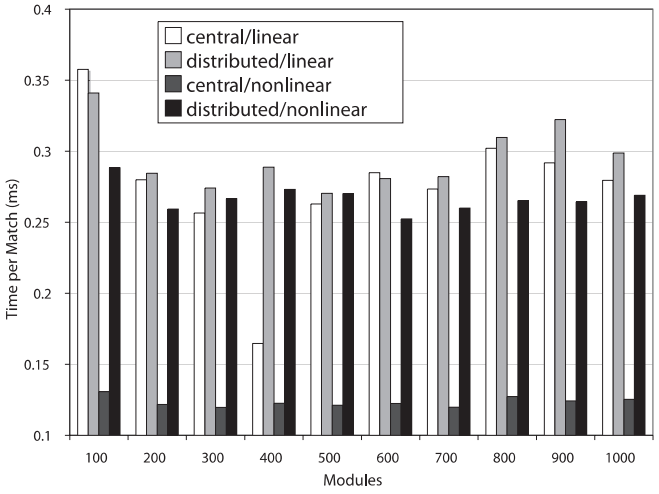

Fig. 15. Time per match versus ensemble size.

PatternMatchers is exponential in the number of slots (as each additional slot can be filled with at least k neighbors), but only linear in the number of modules (as each module begins the search process in parallel). Comparing this with the results above, we have an ensemble of size 100, over 100 timesteps, with program tuple [1:1:1:1] and degree 3 (an approximation), giving an expected number of matches of 270,000, which is quite close to the actual value of 265,600.

8.4. Watchpoint System Overhead

We also analyzed the overall execution time of the algorithms, and their scaling behavior as the size of the robot ensemble grows (Figure 14). In these tests, we used the same watchpoint expressions as above, with the host program generating random variables according to the [2:2:2:2] scheme. Each robot also ran a data aggregation and landmark routing program, to simulate a medium-intensity workload on the system. Tests were run on ensembles of various sizes, using the centralized and distributed implementations, as well as without any watchpoints enabled (for comparison).

The overhead for linear expressions is quite reasonable, with the centralized algorithm having an average penalty of 540% and the distributed algorithm an average of 582%. The overhead for both algorithms is well within the range of other debugging tools such as GDB (FSF 2007) and Valgrind (Nethercote and Seward 2003). Unfortunately, the overhead for non-linear expressions is much higher, being as high as 2,708% in the worst case. While this may seem unacceptably high, we must consider the number of successful matches being found via the algorithm. With 1,000 modules, the non-linear watchpoint finds 3.59 million matches over 100 timesteps. If we plot the time expended per match (Figure 15), we see that the time required as a function of the number of matches is actually quite low: less than 1 ms per match. If we assume that the system will be used to detect relatively infrequent events, then this level of performance is quite adequate.

To test the hypothesis that the large number of matches plays a dominant role in the system's overhead, we compared a set of experiments on 1,000 modules using the [100:1:1:1] program tuple. Results were quite encouraging, with an overhead of only 41% (a reduction by a factor of 10) for linear subensembles detected via the centralized algorithm and 194% (a reduction by a factor of almost 14) for non-linear subensembles detected with the distributed algorithm.

8.5. Topology Dependence

Finally, we explored the effect that the topology of the ensemble has on the runtime and communications cost of the system. As we varied the number of modules from 100 (a single 10×10 plane) to 1,000 (a $10 \times 10 \times 10$ cube), we noted that the average degree of the modules increased from 3.6 to 5.4, asymptotically approaching 5.6. This happens as the fraction of "interior" modules to "surface" modules increases as we move to a taller and taller prism of modules. This increases the number of possible neighbors at each step in the search process and thus the number of different subensembles that must be searched.

This increase in degree causes a corresponding increase in the number of messages required to propagate a PatternMatcher to all neighbors. We can see this clearly in Figure 16. Note that the largest increases in per-module communications cost occurs as we move from 100 to 200 modules (moving from a maximum degree of 4 to a maximum of 5) and 200 to 300 modules (moving from 5 to 6). Similarly, the computation

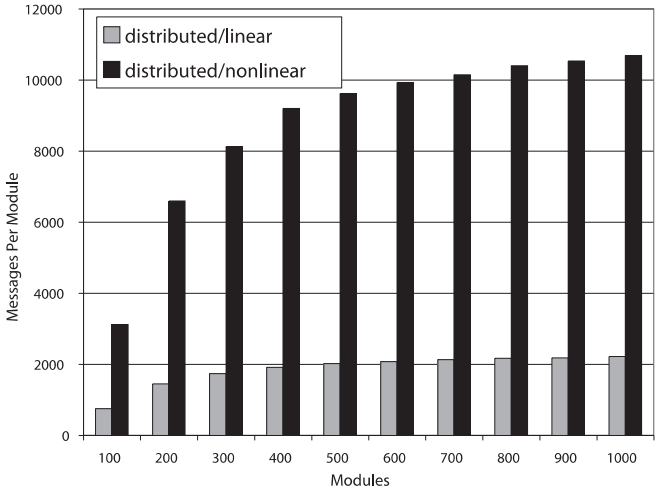

Fig. 16. Messages per module versus ensemble size.

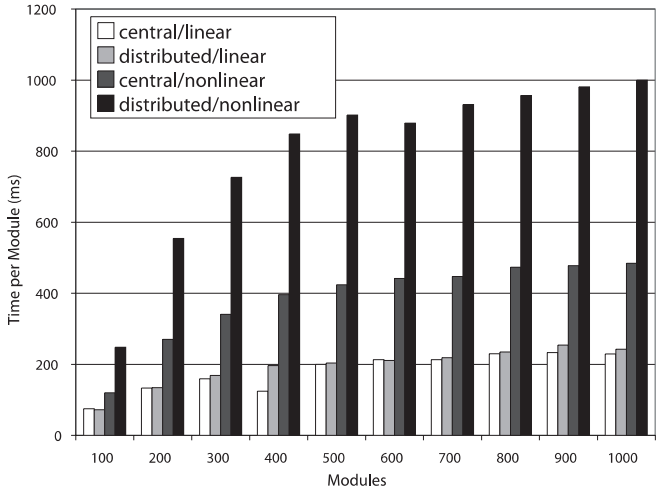

Fig. 17. Time per module versus ensemble size.

time per node increases asymptotically, as the average degree increases (Figure 17).

9. Conclusions

We have demonstrated two contributions: (1) the ability to express a large class of distributed error conditions; and (2) a set of algorithms to detect these conditions in modular robotic ensembles. Our watchpoint description language allows for the concise expression of multi-robot conditions that include state variables, temporal offsets and restrictions on module topology. These conditions are quantified over all connected subensembles of robots by our detection algorithms. The two presented algorithms have execution overheads low enough to make them practical in most cases. Finally, we explored the system's dependencies on host program behavior and illustrated the importance of attaching criteria that eliminate large percentages of the PatternMatchers to the earlier slots of the watchpoint.

Acknowledgments

This research was partially sponsored by the National Science Foundation (NSF) under grant no. CNS-0428738. Additional funding provided by Intel Research.

References

Bacon, D. F. and Goldstein, S. C. (1991). Hardware-assisted replay of multiprocessor programs. *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*. New York, NY, ACM Press, pp. 194–206.

Carr, S., Mayo, J. and Shene, C.-K. (2001). Race conditions: a case study. *The Journal of Computing in Small Colleges*, **17**(1): 88–102.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states in distributed systems. *ACM Transactions on Computer Systems*, **3**(1): 63–75.

Chase, C. M. and Garg, V. K. (1998). Detection of global predicates: techniques and their limitations. *Distributed Computing*, **11**(4): 191–201.

Chen, I. and Burdick, J. W. (1995). Determining task optimal modular robot assembly configurations. *Proceedings of the 1995 IEEE International Conference on Robotics and Automation (ICRA '05)*.

Collett, T. H., MacDonald, B. A. and Gerkey, B. P. (2005). Player 2.0: toward a practical robot programming framework. *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*.

De Rosa, M. et al. (2006). Scalable shape sculpting via hole motion: motion planning in lattice constrained modular robots. *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '06)*.

Fitch, R. and Butler, Z. (2006). Million module march: scalable locomotion planning for large self-reconfiguring robots. *Robotics: Science and Systems, Workshop on Self-Reconfigurable Robots*.

Fromentin, E. et al. (1994). On the fly testing of regular patterns in distributed computations. *Proceedings of the International Conference on Parallel Processing*, pp. 73–76.

FSF (2007). *GDB: The GNU Project Debugger*. Free Software Foundation.

Gerkey, B., Vaughan, R. T. and Howard, A. (2003). The player/stage project: tools for multi-robot and distributed sensor systems. *Proceedings of the 11th International Conference on Advanced Robotics (ICAR 2003)*.

Goldstein, S., Campbell, J. and Mowry, T. (2005). Programmable matter. *IEEE Computer*, **38**(6): 99–101.

Hurfin, M. et al. (1998). Efficient distributed detection of conjunctions of local predicates. *Software Engineering*, **24**(8):664–677.

Jorgensen, M. W., Ostergaard, E. H. and Lund, H. (2003). Modular atron: modules for a self-reconfigurable robot. *Proceedings of IEEE/RSJ International Conference on Robots and Systems (IROS)*.

Kotay, K. et al. (1998). The self-reconfiguring robotic molecule. *Proceedings of the IEEE International Conference on Robotics and Automation*.

Lamine, K. B. and Kabanza, L. (2002). Reasoning about robot actions: a model checking approach. *Advances in Plan-Based Control of Robotic Agents*, pp. 123–139.

Loo, B. et al. (2005). Implementing declarative overlays. *Proceedings of ACM Symposium on Operating System Principles (SOSP)*.

Murata, S. et al. (2002). M-tran: self-reconfigurable modular robotic system. *IEEE/ASME Transactions on Mechatronics*, **7**(4): 431–441.

Nagpal, R. (2001). Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics. *Ph.D. Thesis*, Cambridge, MA, MIT, Department of Electrical Engineering and Computer Science.

Nethercote, N. and Seward, J. (2003). Valgrind: a program supervision framework. *Electronic Notes in Theoretical Computer Science*, **89**(2).

Pamecha, A. et al. (1996). Design and implementation of metamorphic robots. *Proceedings of the 1996 ASME Design Engineering Technical Conference and Computers in Engineering Conference*.

Pnueli, A. (1977). The temporal logic of programs. *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pp. 46–67.

Reshko, G. (2004). Localization techniques for synthetic reality. *Master's Thesis*, Pittsburgh, PA, Carnegie Mellon University.

Savage, S. et al. (1997). Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, **15**(4): 391–411.

Singh, A. et al. (2006). Using queries for distributed monitoring and forensics. *Proceedings of EuroSys 2006*, pp. 389–402.

Støy, K. (2004). Controlling self-reconfiguration using cellular automata and gradients. *Proceedings of the 8th International Conference on Intelligent Autonomous Systems (IAS-8)*, pp. 693–702.

Støy, K., Shen, W.-M. and Will, P. (2002). Using role based control to produce locomotion in chain-type self-reconfigurable robots. *IEEE Transactions on Mechatronics*, **7**(4): 410–417.

Tansley, S. (2007). Trends in embedded systems—a Microsoft perspective. *Proceedings fo the IEEE International Conference on Microelectronic Systems Education (MSE '07)*.

Xu, M., Bodik, R. and Hill, M. D. (2003). A "flight data recorder" for enabling full-system multiprocessor deterministic replay. *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, New York, NY, ACM Press, pp. 122–135.

Yang, Z. and Marsland, T. A. (1992). Global snapshots for distributed debugging. *Proceedings of the International Conference on Computing and Information*, pp. 436–440.

Yim, M., Homans, S. and Roufas, K. (2001). Climbing with snake-like robots. *Proceedings of IFAC Workshop on Mobile Robot Technology*.

Yim, M. et al. (1997). Rhombic dodecahedron shape for self-assembling robots. *Technical Report P9710777*, Xerox PARC SPL.