2007 IEEE International Conference on
Robotics and Automation
Roma, Italy, 10-14 April 2007

Distributed Watchpoints: Debugging Large Multi-Robot Systems

Michael De Rosa *Student Member, IEEE*, Jason Campbell *Senior Member, IEEE*
Padmanabhan Pillai *Member, IEEE*, Seth Goldstein *Senior Member, IEEE*
Peter Lee, and Todd Mowry *Member, IEEE*

***Abstract*— Tightly-coupled multi-agent systems such as modular robots frequently exhibit properties of interest that span multiple modules. These properties cannot easily be detected from any single module, though they might readily be detected by combining the knowledge of multiple modules. Testing for distributed conditions is especially important in debugging or verifying the correctness of software for modular robots.**

We have developed a technique we call *distributed watchpoint triggers* which can efficiently recognize such distributed conditions. Our watchpoint description language can handle a variety of temporal, spatial, and logical properties spanning multiple robots. This paper presents that language, describes our fully-distributed, online mechanism for detecting distributed conditions in a running system, and evaluates the performance of our implementation. We found that the performance of the system is highly dependent on the program being debugged, scales linearly with ensemble size, and is small enough to make the system practical in all but the worst case scenarios.

I. Introduction

Designing algorithms for distributed systems is a difficult and error-prone process. Concurrency, non-deterministic timing, and combinatorial explosion of possible states all contribute to the likelihood of bugs in even the most meticulously designed software. Likewise, these factors also make detection of bugs very difficult. Several modular robotics systems, such as Claytronics [1], envision very large distributed systems consisting of millions of modules, further exacerbating this problem.

Tools to assist programmers in debugging distributed algorithms are few, and generally inadequate. Most are forced to fall back on standard debugging methods, such as GDB [2] or logging through `printf`. GDB is useful for debugging errors local to an individual thread or process, but is not effective for errors resulting from the interactions or states of multiple threads of execution that span multiple modules.

While `printf` may be used to detect some of these errors, this requires logging all potentially relevant state information at each robot, then centrally collecting, correlating, and post-processing the data to extract the details of the error condition. This requires significant effort, skill, and often luck on the part of the programmer. Additionally, both GDB and `printf` 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.

De Rosa is with the School of Computer Science, Carnegie Mellon University, mderosa@cs.cmu.edu

Goldstein and Lee are with the School of Computer Science, Carnegie Mellon University, [seth,petel]@cs.cmu.edu

Campbell and Pillai are with Intel Research Pittsburgh, jason.campbell@intel.com, padmanabhan.s.pillai@intel.com

Mowry is with both CMU and Intel Research Pittsburgh, tcm@cs.cmu.edu

A. Related Work

In considering the design of a distributed debugging system for modular robots, there are three relevant areas of existing research to consider. The first of these is work on distributed and parallel debugging, including the Chandy-Lamport global snapshot algorithm [4], and subsequent related work on global predicate evaluation [5]–[7]. 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 are numerous and localized. Additionally, global snapshots require the aggregation of all relevant data at a central point, resulting in a large communications overhead [8]. Other important parallel debugging tools include static code analysis tools such as race detectors [9], [10], 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 is the development of logic-based verification/proof tools. Specifically, linear temporal logic (LTL) [11], 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 LTL was exploited by Lamine et al [12], who developed an LTL-based model verification tool for single mobile robots.

Finally, declarative overlay network systems, such as P2 [13], 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 [14]. However, the focus of the P2 project is not on robotics, and as such it does not explicitly deal with the rapidly changing topologies inherent in modular robotics. Additionally, the use of P2 for debugging implies the adoption of the P2 programming paradigm, which may not be appropriate for all applications.

1-4244-0602-1/07/$20.00 ©2007 IEEE.

⟨watchpoint⟩ ⟶ ⟨module decl.⟩ ⟨bool⟩
⟨module decl.⟩ ⟶ `modules` (⟨string⟩$^{+}$)
⟨bool⟩ ⟶ `not` ⟨bool⟩
| ⟨bool⟩ `and` ⟨bool⟩
| ⟨bool⟩ `or` ⟨bool⟩
| `neighbor` (⟨module⟩ ⟨module⟩)
| (⟨compare⟩)
| (⟨bool⟩)
⟨module⟩ ⟶ ⟨string⟩
| ⟨module⟩ `.last`
| ⟨module⟩ `.next`
⟨compare⟩ ⟶ ⟨state var⟩ ⟨op⟩ ⟨r val⟩
⟨state var⟩ ⟶ ⟨module⟩ `.` ⟨string⟩
⟨op⟩ ⟶ < | > | = | != | >= | <=
⟨r val⟩ ⟶ ⟨state var⟩ | ⟨numeric constant⟩

Fig. 1. Extended BNF grammar for watchpoint description language

B. Distributed Watchpoints: Our Approach

The key to enabling effective distributed debugging is to allow programmers to easily specify and detect *distributed conditions* in a multi-robot ensemble. Such conditions constitute logical relations between state variables which are distributed both temporally and spatially across the ensemble. Generally, they cannot be detected by observing the state of any single robot, or even the whole system at any single point in time. For example, in debugging a distributed motion planning algorithm, we may wish to detect if two adjacent modules each initiate motion within four iterations of the algorithm main loop. A tool which can detect such conditions can provide insights into the logical and temporal behavior of the system, and help pinpoint defects in distributed algorithms.

To this end, we introduced the concept of a *distributed watchpoint* in [3]. The present paper extends that work by formalizing the watchpoint specification language (see Section II), describing a fully distributed implementation of the watchpoint mechanism (see Section III), providing some simple performance-enhancing optimizations (see Section IV), and evaluating the performance of our approach (see Section V).

II. Describing Errors

The first step in detecting distributed error conditions is to represent them effectively. To that end, we have created a simple watchpoint description language, based on a fragment of LTL [11] with the addition of predicates for state variable comparison and topological restriction (see Figure 1).

In representing distributed error conditions, we make a key assumption: the error must be able to be represented by a fixed-size, connected sub-ensemble of robots in specific states. Allowing disconnected sub-ensembles would imply an exponential search through all subsets of the total ensemble, and distributing information between the members of these subsets would require significant multi-hop messaging.

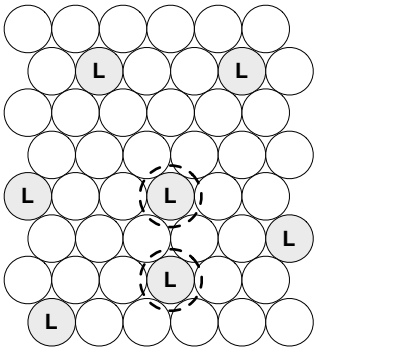

Fig. 2. Incorrect 2-hop leader election. Conflicting leaders are circled. The watchpoint text shows the error condition where two leaders exist in the same two-hop radius.

Watchpoint descriptions begin with a list of module names. This list defines the size of the matching sub-ensemble, and is implicitly quantified over all connected subgroups of this size in the ensemble. The language includes the standard boolean and grouping primitives plus topological restrictions and state variable comparisons. Topological restrictions take the form `neighbor(a b)` and indicate that the two specified modules are neighbors. 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. Additionally, state variable comparisons may include arbitrary 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 configuration: numeric state variables, topological configuration, and temporal progression. Topological restrictions allow us to model (in some 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.

We illustrate the utility of the watchpoint description language with two debugging examples: incorrect leader election and token passing. As shown in Figure 2, we have 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

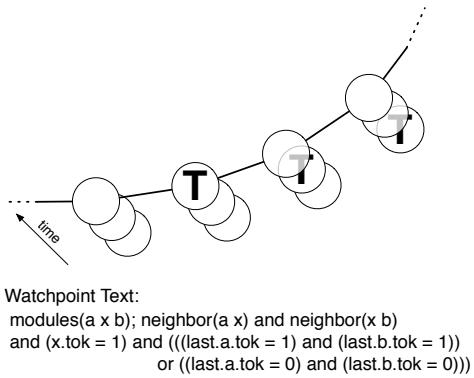

Fig. 3. Part of a token-passing ring. Previous states shown stacked behind current ones. The watchpoint text corresponds to error conditions where zero or two modules previously had the token.

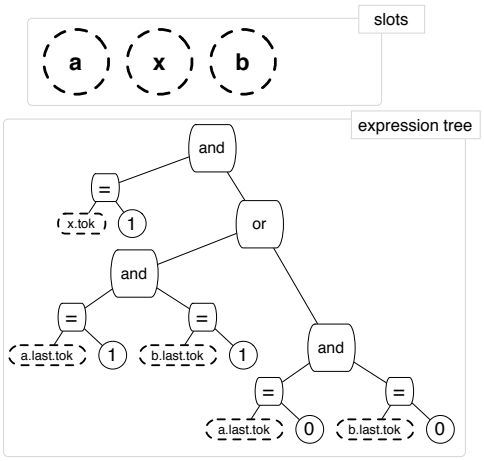

Fig. 4. PatternMatcher object for the watchpoint example in Figure 3. Empty variables shown with dotted outlines. At each step, the first empty slot is filled with the local node ID, and corresponding variables filled with local data. Copies of a partially filled PatternMatcher propagate to neighboring nodes until the expression tree can be definitively evaluated.

Figure 2 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 in Figure 2. Deconstructing the watchpoint expression, we see it specifies three modules, with a linear path from `a` to `c`, and where both `a` and `c` are leaders. A match for this watchpoint indicates a violation of the path-distance criteria given above.

As a slightly more complex example, let us consider the problem of token passing in a ring network (see Figure 3). 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 in the last timestep. We can express the violation of this condition with the watchpoint shown in Figure 3. Here the watchpoint again specifies three modules, with module `x` currently holding the token. An error occurs if both or neither of `x`'s neighbors previously had the token. Note that we do not need to use topological restriction in this watchpoint, as the requirement that `x,a,b` form a connected sub-ensemble is sufficient.

III. Detecting Errors

We consider a simplified machine model for each modular robot: each robot is represented by a number of named integer state variables, and an array of neighbors. We assume that each robot iterates through three atomic phases: computation, state variable assignment, and communication. Computation may take an arbitrary amount of time, and each robot can communicate only with its immediate neighbors. Furthermore, we assume that each robot has a copy of the watchpoint, and that each robot has the relevant local state variables needed by the watchpoint. We explicitly do not require that all robots have the same code image, merely that they have compatible state variables. This simplified model does not entail a large loss of generality, as we can express most run-loop, finite-automata, and event-driven programs within it.

The main component in our distributed watchpoint implementation is the PatternMatcher object (Figure 4). A

Algorithm 1 Centralized Watchpoint Update

$S = \emptyset$
for all modules m **do**
  create new PatternMatcher p from the watchpoint text
  fill p's first slot with m
  $S = S \cup p$
end for
while $S \neq \emptyset$ **do**
  $T = \emptyset$
  for all PatternMatchers $p \in S$ **do**
    if p matches **then**
      execute trigger action for watchpoint
    else if p is indeterminate **then**
      for all neighbors n of modules in p's slots **do**
        p_1 = clone(p)
        fill p_1's first open slot with n
        $T = T \cup p_1$
      end for
    end if
    $S = S - p$
  end for
  $S = T$
end while

PatternMatcher consists of two subunits: a set of named slots that hold robot ID numbers, and 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.

Once a PatternMatcher has matched, the error condition

Algorithm 2 Distributed Watchpoint Update

```
create new PatternMatcher p from the watchpoint text
fill p's first slot with local module m
S = S ∪ p
for all PatternMatchers p ∈ S do
  if p matches then
    execute trigger action for watchpoint
  else if p is indeterminate then
    for all neighbors n of m that are not already in p do
      p1 = clone(p)
      send p1 to n via messaging system
    end for
  end if
  S = S − p
end for
```

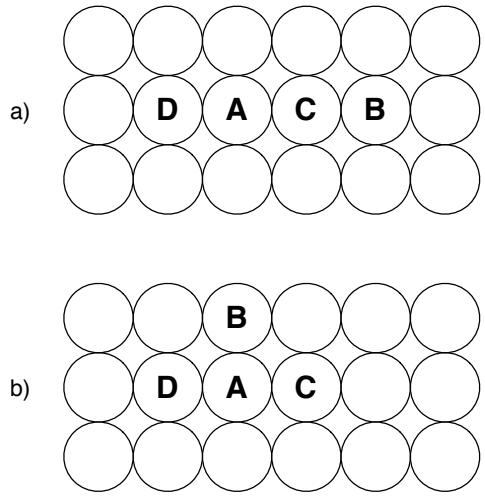

Fig. 5. Non-linear (a) and non-ordered (b) sub-ensembles for the set of modules (a b c d e).

has been detected at the final robot added to the sub-ensemble, and an arbitrary action can be executed. This can be as simple as halting the robots, or as complicated as initiating some expensive logging or recovery operation.

A. Centralized Implementation

Our initial implementation, introduced in [3], relied on a single centralized procedure to update all PatternMatchers across an entire (simulated) ensemble. The watchpoint system maintains a set of vectors for each robot's state variables. At each timestep, the current values of all state variables used by active watchpoints are appended to the vectors, providing state history for the variables. The simulator also maintains a single set of PatternMatchers (S), which are updated and processed every timestep as described in Algorithm 1.

B. Distributed Implementation

For our distributed implementation of watchpoint functionality, rather than having one central state vector and PatternMatcher array, each robot maintains its own state history and set of active PatternMatchers. Robots then independently (and asynchronously) execute two behaviors:

1) When an incoming message is received containing a PatternMatcher, the robot fills the PatternMatcher's next open slot with its information, and adds it to a local set S of active PatternMatchers.
2) Each timestep, every robot m updates its local state information and then runs Algorithm 2, to process any active PatternMatchers.

We note that this algorithm limits the topologies of triggering sub-ensembles to linear chains that match the watchpoint's variables in order. This is intentional, as it removes the need for multi-hop communication in trigger ensembles with non-linear or non-ordered topologies (Figure 5). We are currently working to remove this limitation, at the cost of increased latency.

It is interesting to note that, as we store relevant state information in the expression tree of the PatternMatcher while it migrates between robots, the above algorithm is equivalent to a Chandy-Lamport snapshot [4] of bounded radius, as the message carrying the PatternMatcher serves as both a snapshot beacon and data aggregator. The lack of synchronization between modules implies that we can obtain only *consistent* sets of states, not *simultaneous* sets of states, as simultaneity is ill-defined in asynchronous distributed systems.

IV. 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.

A. 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. Additionally, 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, we calculate the temporal extent by assigning a value of $+1$ to each `next` occurrence, 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 that 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, is it bounds how far the watchpoint must "rewind" for expressions that use the `next` operator.

B. 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

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

Fig. 6. Performance Evaluation Watchpoint

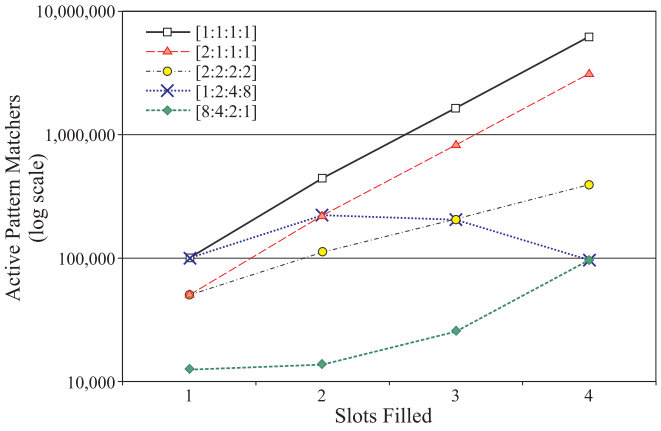

Fig. 7. Total number of Pattern Matchers generated versus number of slots filled after 100 timesteps of execution for the 1000 node ensemble described in Section V. Note that certain host program behavior triggers an exponential increase in the number of PatternMatchers.

succeeding. Whenever we generate new children, we first check whether the parent's expression tree can never match. This can happen even if the PatternMatcher is not completely filled, as subclauses of the expression tree may have become unsatisfiable. If this condition is detected, the parent is deleted, and no children are created.

C. 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. To facilitate this, we add a set of 1-hop neighbors for each slot in the PatternMatcher, allowing for fast local computation of set operations.

V. Evaluation

We evaluated the algorithms using DPRSim [15], our massively multithreaded multi-robot simulator. Evaluation was performed on dual-core machines under Linux. 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 ensembles of 100 to 1000 modules arranged in a cubic lattice packing in 10 by 10 stacked planes. Simulations were conducted for 100 virtual timesteps, where each timestep allowed for arbitrary computation, including message transmission/reception. Message travel time was 1 timestep. Test configurations were monitored for total execution time, number of active PatternMatchers (segmented by number of slots filled), and number of successful matches.

TABLE I
Successful Matches vs. Execution Time for the 1000 node ensemble described in Section V.

Program tuple	# matches	time(secs)
[1:1:1:1]	6233141	715
[2:1:1:1]	3119538	358
[2:2:2:2]	394103	128
[1:2:4:8]	96507	183
[8:4:2:1]	96390	32

A. Host Program Behavior

When evaluating the performance of the algorithms, we were immediately struck by how dependent runtimes were on the behavior of the host program being debugged. To illustrate this, we used the watchpoint shown in Figure 6. 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 0 to $max_{x_n} - 1$ but matches the test watchpoint only when the value is precisely 0. 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] represents a host program that causes half of all PatternMatchers to be discarded after each slot is filled. In contrast, the tuple [4:1:1:1] describes the case where $\frac{3}{4}$ of all PatternMatchers are discarded after filling the first slot, but then all remaining PatternMatchers survive until they are fully filled, at which point they match.

With this test case 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 7. After 100 timesteps on 1000 robots, over 6 million successful PatternMatchers have accumulated. We can easily halve the number of active PatternMatchers by using the tuple [2:1:1:1], which halves the number of PatternMatchers that survive having the first slot filled. Halving the number of active PatternMatchers at each step (as in [2:2:2:2]) results in the expected decrease by a factor of 16. Finally, comparing [1:2:4:8] and [8:4:2:1] is quite instructive. Both eventually generate an almost identical number of successful matches, but over wildly different trajectories. [1:2:4:8], which culls most of its PatternMatchers after the last slot has been filled, is much less efficient than [8:4:2:1]. This can be seen at the bottom of Table I, where one takes almost 6 times as long as the other. Continuing to examine Table I, we note a general

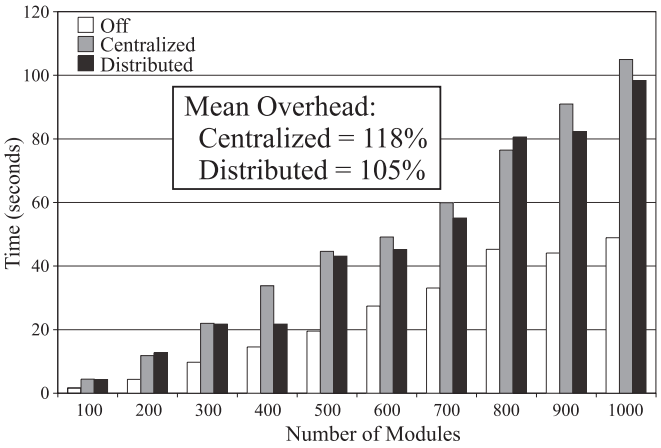

Fig. 8. A medium-intensity workload involving unrelated distributed algorithms. It runs simultaneously with the watchpoint system to provide a baseline for the overhead of both the centralize and distributed detection algorithms.

linear increase in the amount of time taken by the distributed algorithm as the number of successful matches increases.

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. And by shifting the criteria that are least frequently true to early in the watchpoint evaluation, we can dramatically cut execution time, even though the total number of successful matches remains the same.

B. Execution Overhead

We also analyzed the overall execution time of the algorithms, and their scaling behavior as the size of the robot ensemble grows (Figure 8). In these tests, we used the same watchpoint expression 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). We note that all three datasets scale linearly in time as the ensemble size increases. The centralized algorithm required a mean overhead of 118%, while the overhead of the distributed version was only 105%. The distributed implementation must do at least as much work as the centralized algorithm, plus the cost of messaging, so the speedup in the distributed case was quite counterintuitive, until we realized that the distributed implementation was naturally taking advantage of the potential for parallel execution on the dual-core test system. The overhead for both algorithms is well within the range of other debugging tools like GDB [2] and Valgrind [16].

TABLE II
WATCHPOINT OVERHEAD AS EXPRESSION SIZE VARIES FOR AN ENSEMBLE OF 1000 ROBOTS

Expression Size	Frequency Low ($max_{x_i} = 100$)	Medium ($max_{x_i} = 8$)	High ($max_{x_i} = 2$)
1	7.6%	8.6%	3.6%
2	25.2%	32.4%	54.4%
3	39.2%	33.0%	104.7%
4	42.6%	52.0%	369.8%
5	23.0%	83.4%	1432.5%

C. Varying Watchpoint Sizes

Finally, we analyzed the overhead of the watchpoint system as the size of the expression grew. Using a similar expression to that in Figure 6, we varied the number of modules in the expression from 1 to 5. We executed the watchpoints on the centralized algorithm, using a cube of 1000 modules over 100 timesteps. For each expression size, we evaluated the overhead using low- ($max_{x_i} = 100$) medium- ($max_{x_i} = 8$) and high-frequency ($max_{x_i} = 2$) program behaviors. We note that the overhead for the low-frequency behavior is dominated by the random chance that a matching sub-ensemble exists, and thus varies only a little as expression size increases. As shown in Table V-C, medium- and high-frequency behaviors show a noticeable increase in overhead as expression size increases, which corresponds to the exponential increase in the number of PatternMatchers that one would expect as the expression size increases.

D. Hardware Requirements

The resources required to implement this technique in real, rather than simulated, modular robots are modest. Memory needs per module would typically be tens of kilobytes or less (including storage for pattern matchers plus local state memory). Likewise code size could be modest: the full implementation of our distributed implementation is less than 1500 lines of C++. 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 neighbor to neighbor wired or infrared links for such data. The most constrained resource would probably be processor cycles for systems already operating close to their computational or communications limits. In those cases, the additional load of transmitting and processing pattern matchers could require the system to break potential real-time constraints. It is increasingly feasible , however to provision all but the smallest robot modules with powerful processors.

VI. CONCLUSIONS

We have demonstrated two significant contributions: the ability to express a large class of distributed error conditions and two algorithms to detect these conditions both in

simulation and in real robotic ensembles. Our watchpoint description language allows for the expression of complex distributed conditions along three different axes of configuration: numeric state variables, topological configuration, and temporal progression. We describe two algorithms, a centralized algorithm and a distributed algorithm, which evaluate the watchpoints over all connected sub-ensembles in the system.

Both of the presented algorithms have execution overheads low enough to make them practical. The main component of the overheads is directly related to the number of PatternMatcher objects that are generated and propagated through our system. Thus, we found that the overhead of the system depends heavily on the host program being monitored and the structure of the watchpoint. The sooner it can be shown that a particular PatternMatcher object cannot trigger the watchpoint, the fewer PatternMatcher objects are spawned. In the worst case, an exponential number of PatternMatchers will be spawned which can lead to a significant slowdown (on our most extreme example a slowdown of about fourteen times). Note, however, that the user can control the amount of overhead introduced by structuring the watchpoint to fail early. We have found that reasonable watchpoints introduce overhead of 100% or less. This is on par with overheads from such powerful (and heavily used) tools as Valgrind [16] and a small price to pay for the power of finding bugs which involve multiple robots in the ensemble.

ACKNOWLEDGMENTS

This research was sponsored by the National Science Foundation (NSF) under grant number CNS 0428738 (ITR: Synthetic Reality). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity. The authors wish to thank Casey Helfrich and Michael Ryan for developing the simulator, Benjamin Rister for providing a sample workload, Deepak Garg and Frank Pfenning for a tutorial on LTL, and David O'Halloran and TianKai Tu for an insightful discussion on current practices in parallel debugging.

REFERENCES

[1] S.Goldstein, J. Campbell, and T. Mowry, "Programmable matter," *IEEE Computer*, vol. 38, 6, pp. 99–101, May 2005.

[2] GDB: The GNU Project Debugger. [Online]. Available: http://www.gnu.org/software/gdb/

[3] M. DeRosa, S. Goldstein, P.Lee, J. Campbell, and P. Pillai, "Distributed watchpoints: Debugging very large ensembles of robots (extended abstract)," in *RSS'06 Workshop on Self-reconfigurable Modular Robotics*, August 2006.

[4] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63–75, February 1985.

[5] C. M. Chase and V. K. Garg, "Detection of global predicates: Techniques and their limitations," *Distributed Computing*, vol. 11, no. 4, pp. 191–201, 1998.

[6] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson, "On the fly testing of regular patterns in distributed computations," in *International Conference on Parallel Processing*, 1994, pp. 73–76.

[7] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Efficient distributed detection of conjunctions of local predicates," *Software Engineering*, vol. 24, no. 8, pp. 664–677, 1998.

[8] Z. Yang and T. A. Marsland, "Global snapshots for distributed debugging," in *International Conference on Computing and Information*, 1992, pp. 436–440.

[9] S. Carr, J. Mayo, and C.-K. Shene, "Race conditions: A case study," *The Journal of Computing in Small Colleges*, vol. 17, no. 1, pp. 88–102, October 2001.

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[11] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, 1977, pp. 46–67.

[12] K. B. Lamine and L. Kabanza, "Reasoning about robot actions: A model checking approach," *Advances in Plan-Based Control of Robotic Agents*, pp. 123–139, 2002.

[13] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica, "Implementing declarative overlays," in *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[14] A. Singh, T. Roscoe, P. Maniatis, and P. Druschel, "Using queries for distributed monitoring and forensics," in *Proceedings of EuroSys 2006*, 2006, pp. 389–402.

[15] [Online]. Available: http://www.pittsburgh.intel-research.net/dprweb/

[16] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic Notes in Theoretical Computer Science*, 2003.