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

Integrated Debugging of Large Modular Robot Ensembles

Benjamin D. Rister, Jason Campbell, Padmanabhan Pillai, and Todd C. Mowry

Abstract— **Creatively misquoting Thomas Hobbes, the process of software debugging is *nasty, brutish, and all too long*. This holds all the more true in robotics, which frequently involves concurrency, extensive nondeterminisism, event-driven components, complex state machines, and difficult platform limitations. Inspired by the challenges we have encountered while attempting to debug software on simulated ensembles of tens of thousands of modular robots, we have developed a new debugging tool particularly suited to the characteristics of highly parallel, event- and state-driven robotics software. Our state capture and introspection system also provides data that may be used in higher-level debugging tools as well. We report on the design of this promising debugging system, and on our experiences with it so far.**

I. Introduction

As ever-more-powerful embedded computers and PCs have become ubiquitous, robotics software designers have exploited this additional computational horsepower to perform increasingly complex tasks, including vision processing, online learning, and sophisticated planning operations. Many modern robot designs rely on multiple software modules executing as concurrent threads, either time-sliced or spread across multiple cooperating CPUs. While this increased software complexity can help enable more sophisticated functionality, it comes at the cost of a greater likelihood of implementation or algorithmic errors as well as bugs due to concurrency, nondeterminism, and subtle interactions between the multiple threads of execution. Debugging such multithreaded robotics software is challenging even when it is run on a simulated platform, and yet more difficult on the actual embedded hardware itself.

Software development for modular and metamorphic robotic systems compounds these issues, as it involves systems of many robots, each with one or more threads of execution. Programs for even modest modular robot ensembles may involve hundreds of concurrently executing threads. The Claytronics project [9] envisions using thousands to millions of tiny modular robots (*catoms,* short for "Claytronics atoms") that dynamically reconfigure to render and animate 3D scenes and structures. Our current simulations routinely involve over 50,000 robot modules (each with at least one independent stream of execution), and in the near future we expect to simulate over a half-million modules. Debugging software and algorithms at this scale of concurrency is a daunting task.

B. Rister is with the School of Computer Science, Carnegie Mellon University, bdr@cs.cmu.edu

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

T. Mowry is with both Carnegie Mellon University and Intel Research Pittsburgh, tcm@cs.cmu.edu

Based on our frustrations with debugging large modular robot simulations, we have developed a novel execution analysis tool that can capture extensive runtime information including state transitions, intermodule messaging, and variable values. With this information, the tool permits interactive visualization and analysis of a program's operation, both during and after execution. While originally designed to debug modular robot code, many aspects of this tool appear relevant to other robot software and even to general purpose programming.

We distinguish between several classes of errors experienced in modular robotic systems: algorithmic errors, implementation errors, and physical errors. Algorithmic errors are encountered when an algorithm does not always (or ever) correctly perform its task due to logical flaws. For instance, a distributed algorithm might never converge to a consensus. Implementation errors come from improper coding of an algorithm, not the algorithm itself; one example would be failing to properly lock a shared data structure. Finally, physical errors are caused by the underlying hardware, such as a component failure or actuator inaccuracies.

Our tool is designed to help when debugging algorithmic errors and many implementation errors. Physical errors and some low-level implementation errors lie outside the scope of this work.

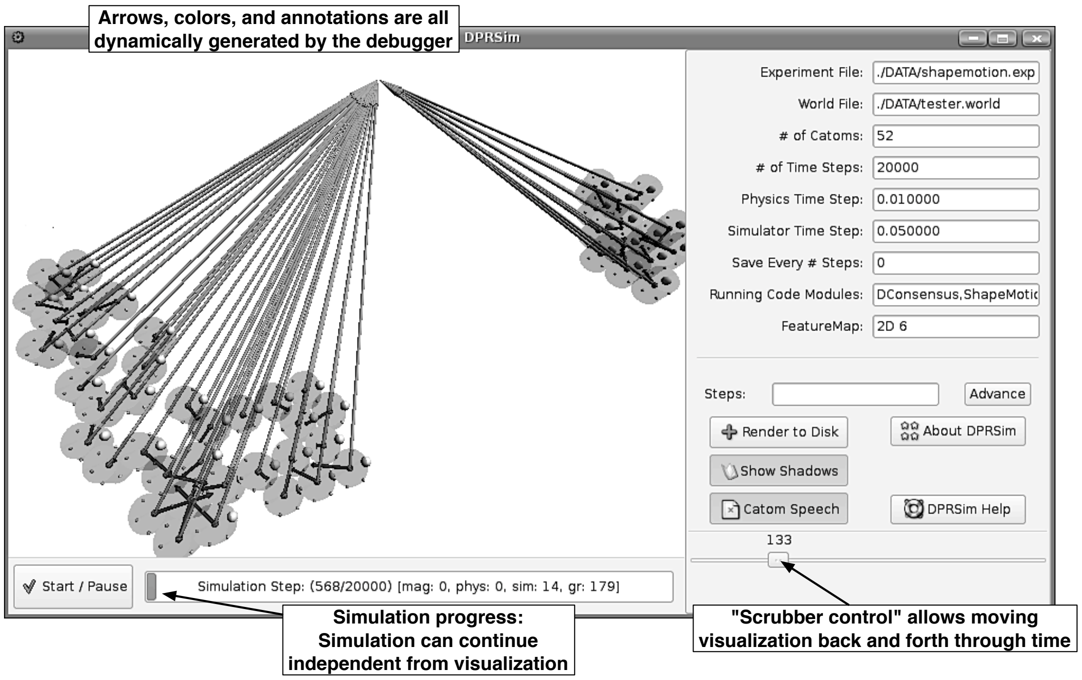
II. Related Work

A. Interactive Debuggers

There is a long history of work on interactive debugging software (e.g., gdb [1] and its many counterparts), and they have been indispensable tools to programmers for many decades. From the perspective of debugging modular robotic systems, however, the problem with these tools is that they are designed to look at a particular instant in time (*time insensitivity*) on a particular thread (*thread insensitivity*).

Thread insensitivity: When examining a multithreaded program using a traditional interactive debugger, the interface typically works on only a single thread at a time. Although you can switch between threads, it may be unclear which thread contains the information that you need. Furthermore, in order to understand the relationships between the states of different threads, you must switch back and forth to get information about those relationships. It is also well known that breakpoints are awkward to implement and interpret in multithreaded systems [3], [6] since they are defined from the perspective of a single thread.

Time insensitivity: Traditional debuggers are designed to show a snapshot of the program state at a particular instant in time. Unfortunately, by the time that a symptom of problem


Arrows, colors, and annotations are all dynamically generated by the debugger
DPRSim
Simulation progress: Simulation can continue independent from visualization
"Scrubber control" allows moving visualization back and forth through time
Advance
Render to Disk
About DPRSim
Show Shadows
Catom Speech
DPRSim Help
Start / Pause

Experiment File: ./DATA/shapemotion.exp
World File: ./DATA/tester.world
# of Catoms: 52
# of Time Steps: 20000
Physics Time Step: 0.010000
Simulator Time Step: 0.050000
Save Every # Steps: 0
Running Code Modules: DConsensus,ShapeMotic
FeatureMap: 2D 6
Steps:
133
Simulation Step: (568/20000) [mag: 0, phys: 0, sim: 14, gr: 179]

Fig. 1. An example debugging visualization of a Claytronics researcher's experiment using planar catoms. Long green arrows are displayed to indicate each catom's eventual goal, and blue arrows indicate its current estimation of its subgroup's center of mass. A "scrubber" control, like that found in media players, allows the programmer to browse through the historical state of the program independent from the simulation's progression.

becomes obvious, the root cause of the problem has often occurred sometime in past (possibly on a different thread). To uncover the root cause of a problem, programmers must typically resort to an iterative process of re-running the program, setting earlier and earlier breakpoints in an effort to "sneak up" on the bug from behind (often guided by leaps of intuition). Unfortunately, parallel systems are frequently non-deterministic, making an iterative approach problematic at best. Even if the program can be replayed deterministically (e.g., through simulation), this process can be extremely time-consuming.

In contrast with previous thread-insensitive, time-insensitive debuggers, the tool that we present in this paper allows us to easily analyze program behavior *across threads* and *across time*.

B. Tools Designed for Specific Classes of Bugs

Tools such as Purify [10] and race-condition detectors [8] are very good at solving the specific problem for which they are built, but offer little leverage against generic bugs. One potential way to surmount this restriction is by using a debugging-oriented programming framework such as Valgrind [7] or Pin [5]. Unfortunately, while these frameworks can ease the task of writing new debugging tools, this process is still too labor-intensive for day-to-day debugging.

III. Example Scenarios

While designing our debugging tool, we gathered information and stories from programmers about their experiences in debugging modular robot algorithms. We present of these several experiences to concretely frame classes of needed improvement in this area.

A. Messaging in a Hierarchy

One researcher is developing a hierarchical computation framework to manage scale and dynamism in large modular robotic ensembles. After the introduction of a new communications protocol into the system, messages were never arriving at their destinations. Debugging this issue was particularly difficult because a message could pass through thousands of modules on the way to its eventual destination, and there was no indication of where along these paths the messages were being lost. It was furthermore unclear whether the problem lay in the construction of the routing tables or in the implementation of the routing itself.

This developer's only foothold in the problem was that the simulator could be made deterministic—a luxury not present in many systems (particularly physical ones). Once a problem case had been identified, he was able to compile-in `printf` statements to follow the progress of a particular message, work out on paper the causality relationships between certain values, and iterate back through the causality chain (recompiling-in new `printf`s each step along the way). He eventually determined that the routing tables could contain incorrect entries. These entries were first spotted *visually* in a mass of numbers because they were entirely different than their counterparts in other modules.

B. Spanning Tree Construction

Another researcher we studied is developing a toolkit of fundamental algorithms for coordinating actions in local neighborhoods, such as spanning tree construction and distributed locking. He altered the simulator to color code the 3D rendering of modules based on the progress of an internal state machine in order to try to discover the reason for anomalous behavior. The researcher quickly discovered that a module was in an unexpected state.

In order to understand how the module came to be in its unexpected state, the researcher needed to know which messages were arriving during a short time span at a small set of nodes in the much larger ensemble. Unfortunately, due to technical reasons, this simulation could *not* be run deterministically. The point of failure was different each run and could only be identified after running the algorithm for varying, usually substantial amounts of time.

This bug remained unsolved until the use of an early prototype of our system. The capture phase of the implementation was complete by that point, and we were able to extract the information needed by the researcher from our database manually.

IV. Debugging Tool Structure

As our debugger is specifically for use in diagnosing errors in Claytronics modular robotic algorithms, there are several constraints and assumptions in place that influenced our design and development:

- *Focus Initially on Simulation:* The ideas behind the work can also be applied to physical implementations of our modular robotic ensembles; however, the manufacturing of our hardware at the scales we desire remains several years away. Thus, our focus is on the simulated programming environment in which we currently perform our research. Nonetheless, when describing our simulator-based tool, we will point out some important design points pertinent to future work on a hardware-based implementation. Conversely, as the simulator itself is a research system, we allow ourselves to exploit our control over the runtime environment to improve the programmer's experience.
- *Embrace a "Legacy" C++ Codebase:* While other languages provide much richer support than C++ for introspection, the combination of our existing C++ codebase and our use of third-party code and libraries (including an entire physics engine) meant that changing language platforms was not an option.
- *Reward Programmer Interaction:* Automating analyses is an important part of our debugger; however, we feel that the programmer has the best understanding of the program. Debugging is an inherently interactive process, so we adopt the goal of providing good debugging results for no programmer effort, with better results being available for a proportionate amount of programmer effort.
- *Address Current Needs First:* As a research system, the engineering effort required to implement the debugger must be considered more carefully than in other environments where techniques may be previously established, systems may be in less continual refinement, and/or engineering teams may be larger. We emphasize practicality and to produce tools that will actually be used. For instance, as most important state is stored in class instance variables in our system, we focus on these locations as our primary instrumentation points, while leaving other sites such as function local variables unaddressed.

TABLE I
Code instrumentation features available to the programmer.

```
MONITOREDVAR(int,varName)
MONITOREDCLASS(className, parentClass)
BLOCK_SIGNPOST("signpost name")
SIGNPOST("signpost name")
```

Our tool is logically separated into three phases of data management: A) capture, B) aggregation/storage, and C) processing/presentation. The capture phase requires a mechanism for obtaining information about events and the state of the program, and is the source of all information about the program. Once acquired, this data must be stored somewhere in a usable form—the task of the aggregation and storage phase. Processing and presentation operations then alternate as information is presented to the user, who queries the debugger, resulting in additional processing and information to present.

A. Capture

Capture of program state and execution details requires making a tradeoff between generality, programmer ease, and performance. Ideally, our tool would be able to capture all relevant information all of the time, without any direct action by the programmer, and at low overhead, but these desires often conflict in implementation requirements.

Scalar value monitoring: In our solution, scalar class instance variables may be included in the captured data by changing a declaration such as `int varName;` to `MONITOREDVAR(int,varName);`. This macro substitutes a templated wrapper class `monitored_value` around the scalar value that intercepts all operations performed on the variable, allowing us to capture not only every value ever adopted by the variable, but also the origin and uses of those values. When operating inside the space of monitored variables, full causality data can also be maintained through our instrumentation of the operations on the values—we can form a link from the inputs of the operation, if they are monitored, to the result.

It's important to note that *no other code must be changed* in order to capture the entire history of a monitored instance variable. This makes our system extremely amenable to lazy instrumentation during the development of the program. The programmer may simply tag variables as they are needed, rather than having to guess in advance which info may be buggy or pertinent to other bugs (and potentially hurt performance by capturing excess data), or worse, to have to change coding habits to accommodate the debugger.

Object class monitoring: Classes are similarly

Before instrumentation:

```
class exampleClass : public exampleParent {
  . . .
  int exampleInstanceVar;
  void doStuff() {
     for(int i=0; i<exampleInstanceVar; i++) {
        doSomethingElse();
        doAnotherThing();
     }
  }
  . . .
}
```

After instrumentation:

```
MONITOREDCLASS (exampleClass, exampleParent)
  . . .
  MONITOREDVAR(int,exampleInstanceVar);
  void doStuff() {
     for(int i=0; i<exampleInstanceVar; i++) {
        BLOCK_SIGNPOST("loop signpost");
        doSomethingElse();
        SIGNPOST("between something and another");
        doAnotherThing();
     }
  }
  . . .
}
```

Fig. 2. Example class code before and after instrumentation. After annotating the instance variable declarations, no further modifications to the code are needed to track changes to the variable's value. (In particular, *no annotations are required when the variable is used.*) Note that the signpost markers are *entirely optional* and in this case have been added because the programmer nominally wanted better localization of events within this section of code.

handled by a simple substitution of the macro `MONITOREDCLASS(className, parentClass)` for the usual `class className : public parentClass`, which both turns the class into a subclass of `monitored_class` and inserts an instrumented instance variable into the class itself. Because this instance variable is constructed with each instance of the class itself, and destructed when the instance is destructed, it allows us to execute code at precisely those times without needing to modify the code or prototype of the monitored class's constructor, and without forcing the programmer to manually insert instrumentation in those locations. Again, no other code must be changed besides that localized modification to gain the full benefits of the monitored class.

The `monitored_values`, `monitored_classes`, and instrumented instance variables inside monitored classes all register themselves into an index over the memory space, identifying each class's location and extent. "Belongs-to" relationships can be calculated with this data, allowing us to reason about and display information concerning each specific instantiation of the variables and classes. We can determine which class, code module, and catom any particular variable belongs to, and, conversely, find a variable based on the same criteria. In a philosophical sense, these are the addresses to all of the "needles" in our "haystack!"

Signpost marking: To mark locations of note, "signposts" may be placed in the code both automatically by the system and manually by the programmer. A special type of signpost is automatically inserted into each function by a simple automated script during the compilation process to mark the entrance and exit from the function. This provides a stack trace for every operation performed on the monitored values. The programmer can easily add additional signposts in locations where he or she would like better resolution. e.g. as done in Figure 2.

Causality splicing: The signposts and value tracking produce threads of causality for values. For a given value, we know which other monitored values contributed to the generation of that value, as well as the traces of signposts through the code that led to their assignments. However, there are some causal relationships that this does not capture.

For instance, when one catom sends a message to another, the message is created, handled in some simulator-internal ways, and then later reappears at the destination in the local event handling code. This leaves us with two threads of causality—the one leading to the sending, and the one commencing from the receipt—but without any relationship between them. We splice together these causality threads by instrumenting the pertinent data structures in the simulator, embedding a pointer to the previous causality thread into the message structure itself.

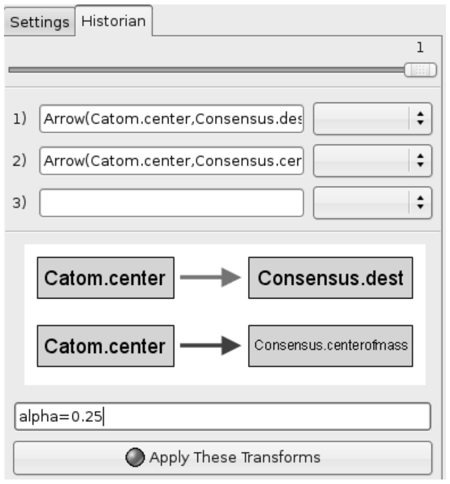
B. Aggregation and Storage

The main design decision in creating the storage part of the system was how much processing of the data should happen at capture time and how much should be deferred to when the user is performing queries. It is both impractical and unnecessary to try to perform all analyses at capture time, but some form of processing can significantly simplify the implementation of queries. Additionally, there are performance concerns involved in selecting different points along this line—see Section V for more information.

For our implementation of the debugging tool, we chose a SQL database (SQLite [4]) as a good compromise, requiring minimal a priori data structure, and providing a rich query language and optimized implementation for subsequent analyses. The database schema is relatively straightforward, and is based around three tables which track signposts, events, and values. The online state manager only needs to maintain the row ID of the most recent insertion in each table for cross-referencing purposes; all detailed information can be immediately discarded following insertion.

The storage phase is also the most interesting one in terms of moving the debugger into a real distributed, physical ensemble. In the case of physical robots, the data streams generated by each module will need to be exported across some external transport. This process will almost certainly be bandwidth-limited when confronted with the debugging information from millions of modules. We will need to minimize the size of the data stream, likely sacrificing substantial information and/or real-time qualities.

In contrast, our simulator-based desktop implementation is only concerned with the size of the data being stored inasmuch as the resulting disk accesses may impact performance. In the simulator, "performance" is a soft restriction,



1) Arrow(Catom.center,Consensus.des
2) Arrow(Catom.center,Consensus.cer
3)

Catom.center → Consensus.dest
Catom.center → Consensus.centerofmass

alpha=0.25

Fig. 3. Part of the configuration used to generate Figure 1. The programmer has specified that arrows should be drawn from each catom's center to the point contained in the "dest" and "centerofmass" variables in the "Consensus" code module. Using our debugger's interface, the programmer can interactively configure many attributes of the display based upon the captured state of the system across time.

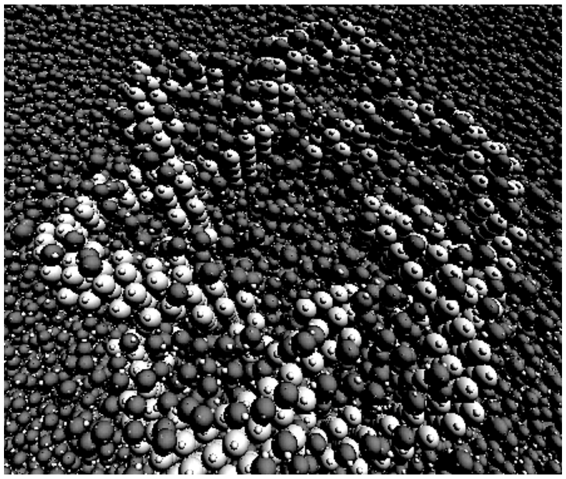


Fig. 4. An example visualization where the color of each rendered module is based upon its internal state. In this case, the lighter-colored catoms have self-identified as part of a larger structure, while the darker catoms are attempting to move away. Despite the very large number of modules visible, the programmer can easily identify which self-assigned role each catom has taken.

as we can simply wait for operations to finish, whereas a real ensemble would need to proceed regardless of whether the debugger was able to keep up.

C. Presentation

Unlike classical debuggers, our debugger is designed to answer questions, not just present facts. Substantial processing may be required to generate the representation presented to the user from the raw data which was captured during the run. There are four main aspects to our user presentation: spatial visualization, state-space visualization, causality tracing, and captured state browsing.

Spatial visualization: The human brain can quickly assimilate very large amounts of information visually. We exploit this by providing very flexible customization of the (typically 3D) rendering of the physical configuration of the system. The displayed location, color, transparency, and textual label of each catom can be adjusted dynamically based upon any monitored state. Lines and arrows can also be generated automatically and rendered based upon monitored state as well.

Additionally, because we have the entire historical state available, we are able to provide the user with the ability to smoothly review data from any point in the simulation without any need to rerun. Such "time travel" alone can save users many hours of (re)simulation time.

The most basic application of this visualization is changing the colors of the catoms in response to their internal state. Other, more sophisticated possibilities are also available through a collection of simple value transformers that can be hooked together to form arbitrarily complex expressions on the catom's state.

For instance, one can instruct the debugger to "superimpose on each catom the sum of its variables 'varName' and 'otherVar,' but only if 'anotherVar'==2." Important catoms can be highlighted, unimportant ones can be faded out, and distributed data structures can be rendered through arrows in the real space of the world.

Figure 1, near the beginning of the paper, shows an example visualization generated by our system which uses the ability to render arrows in the visualized product. The arrows are dynamically generated and colored based upon the monitored state of the catoms. The user has (at debug time) decreased the opacity of the catoms in order to see the lines more clearly. The "north" point in each cylindrical robot is marked by a small white dot. Figure 3 shows part of the configuration used to produce Figure 1.

Figure 4 is another visualization example, this time using our ability to color catoms based on their internal state. The algorithm being run during this simulation starts with a solid block of catoms, and attempts to form a shape by removing everything but the intended shape. As the different self-assigned roles result in the catoms being colored differently, the programmer can easily spot any discrepancies in the shape being formed, or errant catom behavior. The marked slider allows the user to scroll the visualization and state introspection back and forth through time.

State-space visualization: In addition to the 3D visualization in a "real world" space, we also provide 2D graphs for data values. The distribution of values across time (how a particular variable changes over time) or space (which values a variable has across all modules) can be valuable information when debugging a large modular robotic ensemble.

For instance, the first graph of Figure 5 shows the distribution of a "level" variable partway into the construction of the hierarchy mentioned in Section III-A. While the distribution is proper for a hierarchy, given that most of the nodes are leaf nodes (at level 0), the dropoff between levels 0 and 1 is proportionately much sharper than between subsequent levels. The programmer could then ask the debugger to show

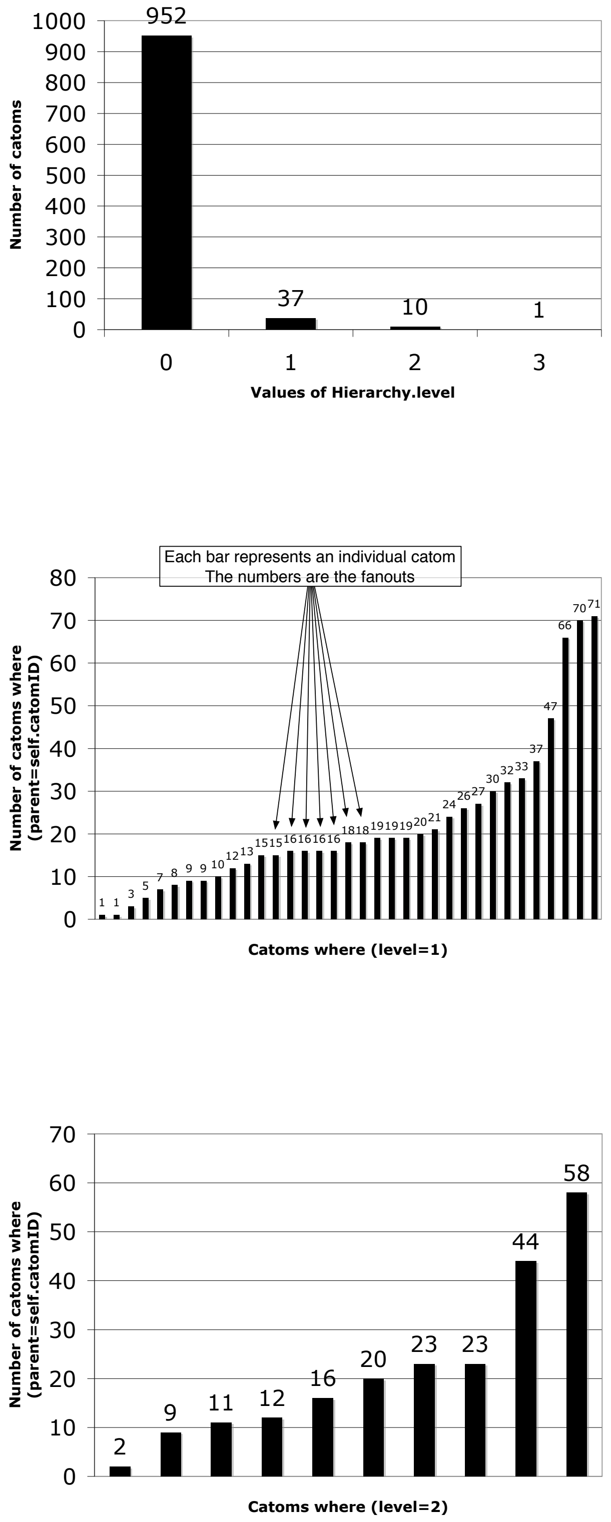


Fig. 5. An example state-space debugging visualization of variables and attributes in a hierarchical computation structure. The second and third graphs caused the initial realization that there was a fanout problem in the hierarchical system. If the system were working correctly, the graph would be approximately flat.

the fanout for the higher levels to see what the distributions are, as shown in the second and third graphs.

This type of information is simply not handily accessible in a traditional debugging system. Moreover, if this tool were available during the debugging described in Section III-A, the errant values in the routing table would have clearly stood out as outliers in such a visual representation.

Captured state browsing: As with most debuggers, the programmer can access the captured state of the system. Unlike temporally insensitive debuggers, which are snapshot-based, the programmer can scroll this state view backwards and forwards in time. This functionality is also the jumping-off point for the state-space visualization and causality tracing features of the debugger—by selecting a monitored variable, the time and space distributions may be generated, or the value history and causality chain of a particular value displayed.

Causality tracing: When a value is selected by the user, the debugger displays the trace of signposts (including function calls) leading to the operation. If the value was derived from other monitored values, say through computation or basic assignment, the contributing values are also displayed and can be used to follow the causality chain further. Thus, when a value is determined to be in an invalid state, the programmer can easily determine what other piece of invalid state caused the visible anomaly, and determine the original cause.

In the presentation and analysis phase of the debugger, the structure and sophistication of the earlier phases pays off dramatically through the ability to automate these analyses and convey processed information in an effective way to the programmer. All of the work that the debugger performs saves the programmer from having to do that work manually, whether mentally or on paper. Parsing, collating, and processing data is why computers exist, and properly exploiting this produces a large improvement in the debugging experience.

V. Performance

We measure the performance characteristics of our debugger on the most extensive code module[1] to date—the hierarchical computation framework described in Section III-A. The simulation was run on a world containing 8008 catoms, and involved two simultaneous instances of hierarchical computations using the framework. Each instance of the code module contained three monitored variables: `hostCatom`, `parent`, and `level`. These three variables represent the most frequently used variables in the modular robot code in question. We also included one "dummy" variable which was never accessed, to demonstrate that the overhead scales with the amount of use each monitored variable receives rather than the number of variables being monitored.

[1]Code written for our simulated catoms takes the form of a "code module" which encapsulates a single unit of behavior. To ease development, these can be enabled or disabled in simulations without recompiling the simulator itself, and interact with the rest of the simulator through a well-defined API.

TABLE II
OVERHEAD BY INSTRUMENTATION. 8008 CATOMS RUN AGGREGATION AND ELLIPSIS CODE MODULES FOR 25 SIMULATOR TICKS.

	Time Overhead mm:ss	Time Overhead multiple	DB size
baseline	1:30	n/a	n/a
world state	0:08	0.09x	17MB
signposts	8:48	5.87x	1.7GB
hostCatom	2:02	1.36x	204MB
parent	0:52	0.58x	61MB
level	0:51	0.58x	89MB
dummy	0:00	0x	0MB
total	14:11	9.45x	2GB

The variable `hostCatom` is part of the simulator interface and provides the gateway to the catom's internal state. It is accessed multiple times in virtually every function in every code module, and results in an average of 20 events per code module per catom per simulator tick. The `parent` and `level` variables are part of the hierarchy system and are accessed at least once in most functions, for an average of 8 events each per code module per catom per tick. According to the author of the hierarchical framework, monitoring these variables would be sufficient to debug about 90% of problems encountered to date in that system.

A. Instrumentation Overhead

Table II shows the amount of overhead incurred by the debugger, broken down into the different areas of instrumentation. The total slowdown was 9.45x, placing our technique between the overheads of traditional general purpose debuggers like gdb, and more intense special-purpose analysis tools like Purify and Valgrind.

Most of the overhead is incurred by the signposts—64%, to be precise. We track the entire history of signposts that are encountered to provide the maximal amount of information to the user about the code paths that led to the events he or she is interested in. However, of the approximately 24 million signposts we record, only about 8 million are directly referenced by recorded events. If the user is willing to receive only the information about the file and function in which each event occurred, without intermediate signpost traces between the events, the time overhead from the signposts can be reduced to approximately 2x the baseline runtime (from 5.87x), reducing the overall runtime to about 7.45x the baseline.

The dummy variable had no measurable impact on the run time, and used no database space. The overhead of monitoring variables is proportional to the number of accesses to the variables, and does not necessarily grow linearly as additional variables are monitored. Although we are only monitoring 3 variables in each code module, because these are the most accessed variables, the overhead we measured is a significant portion of the overhead that would be measured if *all* instance variables were monitored.

Breakdown by phase: With the storage phase disabled, the capture phase (with all instrumentation above enabled) incurs an overhead of about 0.51x of the baseline time. The remainder of the overhead belongs to the storage phase.

How much of the overhead of the storage phase is due to the storage of the raw data, and how much is due to the structuring and indexing provided by the database? To answer this question, we temporarily bypassed the database and dumped the raw, unstructured data directly to disk and found that the storage phase was approximately 9x faster. However, in a sense, the most important benchmark is the duration between the time between when the simulation starts and the time when the debugging information may be accessed in a reasonable way, not the time when the simulation completes. Feeding the data into a database online during the simulation also allows for debugging before the run is complete, without having to carefully manage and process a constantly-growing file of raw data.

Disk usage and data compression: We consider the disk usage of 2GB quite acceptable for this purpose, particularly as the database will generally be transient, existing only for the duration of the debugging session. If longer-term storage is desired, or if disk space is at a premium, compression is highly effective on this data. The 2GB database compresses down to 244 MB using `gzip --best`, for a compression ratio of about 89%.

High compressibility also bodes well for implementation in a real physical modular robotics ensemble, where inter-module communications bandwidth may be at a premium. As the data compresses well, we can transmit substantially more information across a limited bandwidth connection, at the expense of the CPU resources needed to perform the compression and decompression at each end.

B. Presentation

The processing performed during the storage phase pays off in a snappy, responsive interface during the presentation of information to the user. As it is generally hard to quantify how responsive an interface feels, we present some basic numbers and descriptions to convey the experience of using our debugger.

The spatial rendering of the system through processed historical data runs at a comfortable 30fps for most simulated worlds on a desktop machine equipped with modern accelerated 3D graphics. This is approximately 2x slower than when drawing directly from the current state of the simulator. The user can smoothly scroll through time using the slider in the interface, and the world follows gracefully, only exhibiting visible lag on the largest of simulated worlds.

Generation of a state visualization showing the distribution of all values across time or space typically will complete in about 5 seconds, a short wait given the benefit received from the analysis. Without the structure of the database, generation of the analysis can take several minutes or more.

The basic introspection interface, being simpler and involving minimal processing, is instantaneous from the user's perspective.

C. Resource Limits

Our simulator infrastructure is primarily limited in scaling by memory consumption—the amount of state maintained for each catom overwhelms the memory system far before the speed of simulation becomes unbearably slow.

The debugging instrumentation adds a memory overhead of about 20 bytes per monitored variable or class (depending on the name of the variable, which must be stored with it). Compared to the remainder of the state maintained about each catom, this is non-trivial, but usually not large. In the experiments mentioned above, memory consumption with our instrumentation enabled rested about 10% above the baseline memory consumption.

In a physical modular robotics ensemble, we anticipate that bandwidth will be the limiting factor. As discussed above, the data streams we create are highly amenable to compression.

VI. Summary

We have built a debugger for Claytronics modular robotic ensembles that is *temporally-aware* and explicitly attempts to assist the programmer in debugging within the highly parallel environment. Historical state from all stages of execution is available and can be easily accessed by the programmer. Visualizations in "physical space" and graphs in data space exploit the visual processing capabilities of human beings to efficiently and effectively convey very large amounts of information in a way comprehensible to the programmer. Finally, our tool has demonstrated effectiveness against the types of real-world bugs encountered during actual Claytronics development, and is moving into regular use by the researchers on the project.

Limitations: At present, our debugger does not handle pointers; as with most times they come up in program analysis, they pose formidable challenges. Nonetheless, we have found the tool applicable in a wide variety of scenarios without pointer support, and have not yet felt any strong impetus to add such support. There are also corner cases in subclassing monitored classes that can confuse the "belongs-to" analysis, but to date, this has remained a theoretical limitation—no user has yet encountered any of them in actual practice.

Similarly, while our system for causality weaving is tied into our particular simulator's implementation, the possibility of more generic causality weaving is an interesting prospect. However, as messaging is the only case where this has been required to date, we decided to directly instrument the simulator instead. This may change in future versions as researchers grow more sophisticated in their Claytronics development over time.

Future work: Improvements to our approach to signposting, perhaps using techniques akin to path profiling [2], could provide a significant boost to performance.

A future implementation of the capture system might be best done by a compiler in alliance with a runtime environment, or even through modifications to a traditional debugger. However, the implementation described above enables us to gain research experience with this type of a debugging environment with a more modest engineering overhead than a compiler-based solution would incur.

More generally, it may be possible to take our experience in building a temporally-sensitive, parallelism-facilitating debugger in the context of modular robotics and move towards the creation of a tool that would apply outside the context to which we have limited ourselves here. Every programmer has at some point wished they could simply click on a variable and ask why it has the value that it has—perhaps such a tool is not out of the question. This will require techniques to monitor other types of values, such as those belonging to global, static, and function local variables.

Finally, as the project's hardware development continues, we will likely wish to use a similar tool in the non-simulated system as well. The stronger real-time requirements of a physical system coupled with the distributed nature of the computation will require extensions and changes to the tool as described here.

VII. Acknowledgments

We thank Casey Helfrich and Michael Ryan, who are the primary developers of DPRSim and implementers of many of the user interface components of the debugging tool. We also thank the other developers of catom code, including Michael De Rosa, Ram Ravichandran, Stano Funiak, David Christensen, Preethi Bhat, and Seth Goldstein for allowing us to observe their debugging practices, and for answering the authors' questions about their methodologies.

References

[1] GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[2] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[3] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 134–141, Washington, DC, 1990. IEEE Computer Society.

[4] D. Richard Hipp. SQLite. `http://sqlite.org`.

[5] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[6] B. P. Miller and J. D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 316–325, Washington, DC, 1988. IEEE Computer Society.

[7] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 2003.

[8] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[9] S.Goldstein, J. Campbell, and T. Mowry. Programmable matter. *IEEE Computer*, 38, 6:99–101, May 2005.

[10] Rational Software. Purify: Fast detection of memory leaks and access errors.