Towards an Approach to Overcome Software Brittleness

Gordon C. Osbourn

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Towards an Approach to Overcome Software Britteness

Gordon C. Osbourn
Vision Science, Pattern Recognition & Multisensor Algorithms
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1423

Abstract

Development of bug-free, high-surety, complex software is quite difficult using current tools. The brittle nature of the programming constructs in popular languages such as C/C++ is one root cause. Brittle commands force the designer to rigidly specify the minutiae of tasks, e.g. using "for(index=0;index<total;index++)", rather than specifying the goals or intent of the tasks, e.g. "ensure that all relevant data elements have been examined". Specification of task minutiae makes code hard to comprehend, which in turn encourages design errors/limitations and makes future modifications quite difficult. This report describes an LDRD project to seed the development of a surety computer language, for stand-alone computing environments, to be implemented using the swarm intelligence of autonomous agents. The long term vision of this project was to develop a language with the following surety capabilities: (1) Reliability -- Autonomous agents can appropriately decide when to act and when a task is complete, provide a natural means for avoiding brittle task specifications, and can overcome many hardware glitches. (2) Safety, security -- Watchdog safety and security agents can monitor other agents to prevent unauthorized or dangerous actions. (3) An immune system -- The small chunks of agent code can have an encryption scheme to enable detection and elimination of unauthorized and corrupted agents. This report describes the progress achieved during this small 9 month project and describes lessons learned.
Background and Motivation

The development of bug-free, high-surety, complex software is quite difficult using current tools. A recent survey (Ref. 1) indicated: 5 out of 6 corporate software projects don’t produce code that performs as required; 1 out of 3 corporate software projects are canceled; all delivered systems are unable to accommodate future changes. The unavoidable presence of numerous bugs (i.e. design or implementation flaws) in commercial software indicates that better development tools are widely needed. The implications of ubiquitous bugs for high-consequence software utilization are obvious. The frequent and frustrating failures associated with software bugs also have increasingly costly consequences for consumers and businesses.

Why is robust software so difficult to design and implement? We believe that the brittle nature of the programming constructs in popular languages such as C/C++ is one root cause. Brittle commands force the designer to rigidly specify the minutiae of tasks, e.g. using “for(index=0;index<total;index++)”, rather than specifying the goals or intent of the tasks, e.g. “ensure that all relevant data elements have been examined”. These types of commands are at the heart of essentially all general-purpose software languages because: they correspond closely to the available operations in microprocessors; they are needed to allow automatic parsing of the individual software statements into a form appropriate for microprocessors. Specification of task minutiae has several negative effects on software production:

1. It makes code difficult to comprehend, so that even experienced coders consistently produce many undetected code/design flaws in large software projects. Unfortunately, finding subtle flaws is like finding needles in a haystack. Also, the act of “fixing” a flaw often introduces new flaws. This leads to the chronic presence of flaws in complex code.

2. It produces brittleness in the code operation against unanticipated conditions of operation, e.g. unexpected inputs and hardware “glitches”.

3. Future code modifications require someone to “decipher” the code minutiae created by other authors. Modifying old software is difficult to do without disabling it in unexpected ways, and typically adds new bugs to the “working” modified code.

Many computer scientists have attempted to address this difficult problem. Much progress has been made with object-oriented techniques and a variety of high-level architectures and design philosophies. However, many experts agree (see Ref. 2) that these approaches have not solved the problem. In particular, these high-level approaches still require the use of the same, brittle low-level language constructs that we would like to replace for surety software.

Approach and Goals of this Project

The long term vision that motivated this work was to develop a new path to making large, reliable, fault-tolerant software code that is less expensive to create and maintain. The
main project goal was to improve software reliability and overcome the brittleness of software constructs through the creation of a new software language. However, another goal was to create the opportunity to incorporate novel safety and security language commands. The language was to be implemented using the "swarm intelligence" of autonomous agents. The swarm-based language would address reliability (of design and operation), security and safety in an integrated fashion.

This language concept would combine the best of two worlds: The ability of human designers to see the big picture and to specify goals and constraints; The ability of agent swarms to robustly perform particular tasks without requiring task minutiae specification. In particular, the ability of agents to accomplish tasks without minutiae specification could allow the new language commands to be simple, general, easy to understand, and less prone to make mistakes with. The code developer could use these new commands to write robust, high-level programs without any knowledge of swarm concepts. This conceptual approach might provide a unique combination of software capabilities:

1. Reliability -- Autonomous agents can appropriately decide when to act and when a task is complete, and can work around many hardware glitches.
2. Safety, security -- Watchdog safety and security agents can monitor other agents to prevent unauthorized or dangerous actions.
3. An immune system -- The small chunks of agent code can have an encryption scheme to enable detection of unauthorized and corrupted agents.

Why base a new software language on autonomous agents? To motivate this, consider that autonomous agents are typically collections of small software programs that accomplish complex tasks by simple individual behaviors. Agents act by "sensing" their environments, sharing information by communicating with each other, and performing together in a collective fashion. Researchers at the Santa Fe Institute (SFI), Massachusetts Institute of Technology (MIT), Sandia National Laboratories (SNL) and elsewhere have found that the task performance of a collection of autonomous agents can be quite robust and can adapt to changing or unexpected environmental conditions. There are several reasons for these interesting properties of agent "swarms":

1. The performance of a group of agents can be insensitive to malfunctions of a few agents.
2. There are many agents attempting to accomplish the same end result, so that there are many possible paths to success for the agent group.
3. Agents can benefit from the "knowledge" and successes of other agents through communication.
4. Agents rely on the "sensed" condition of the environment they are in to initiate and terminate behaviors. This allows agents to behave appropriately in a variety of circumstances, and eliminates the need for specifying many details of the overall task that the agents collectively carry out.

SFI research has shown that agents can accomplish many relevant computational tasks -- searching, sorting, and allocation of processor resources between concurrent tasks. We expect that agents could be designed to have relevant surety properties as well. For example, undesirable or hazardous situations can be collectively detected and responded
Agents can also monitor each other for code corruption. Further, groups of agents can readily operate in fault-tolerant hardware environments with redundant processors and memory. We intended to exploit all of these properties to build robust language constructs.

Software autonomous agents are typically programmed using existing languages. Our plan was to turn this around, i.e. to build language commands from autonomous agents. The agents would consist of packets of (virtual) machine language instructions that would have primitive “sensing”, labeling and communication abilities when executed in the RAM of a stand-alone computer. “Sensing” refers to detecting the state of the computing environment, e.g. the presence and status of data and other agents in RAM. Each language command would be associated with general computational goals, e.g. “examine all relevant data elements”, and each would be carried out by a unique set of autonomous agents invoked by the executable program. These agents would bring the same robustness properties described above to each command that would be implemented in this fashion.

Another element of the plan was to control the development of software in this new language through a development GUI. All language constructs, including commands and variable names, would be selected from menus. This, in effect, would allow the software developer to directly parse the program inputs, since execution tokens could be directly associated with each menu entry. Such a software development scheme might avoid the many subtle bugs that occur due to "typos". More importantly, readable English phrases could also be associated with the menu items. An "English" version of the source code (that would be too difficult to parse by machine) would be produced as menu items are selected. This part of the language would free us from needing either a parser or a conventional compiler.

**Project Plan and Milestones**

We could not completely develop a new software language in 9 months with the small level of funding provided. Instead, the plan was to seed the development of a larger follow-on effort by carefully focusing this year's activities. The specific goals were: to implement and demonstrate some of the key core properties we expect from this approach; to build a constituency of supporters, interested observers, and partners across the laboratory. The intent was to make the potential of this new approach more evident to nonexperts by the end of the year. Through the initial progress, we intended to make future development of this language a much lower risk investment for the laboratory.

The milestones of this work were: design a few language commands so that minutiae specifications are not needed to use them; design their "English-like" names and usage so that their purpose and function is always clear; complete a plan for of the "look and feel" and the required functionality of the GUI that will ultimately enable program building from these commands; demonstrate that machine-level swarm agents can carry out the intended functions of these commands; decide on an effective demo to illustrate the novel language properties and implement it; SAND report; secure follow-on funding.
Accomplishments

A significant shortcut, based on the existing language FORTH, was recognized at the start of this project. FORTH is a simple, extendable language that was popular in the early days of personal computers. It consists of a set of words in one or more dictionaries, that are invoked to carry out computational tasks. A core of words defines the standard version of FORTH, but additional words are added to the core command set by each programmer. FORTH is still used in embedded applications and for writing the small boot programs that initialize some modern computers. It was recognized that some of the key surety features of a new language could be demonstrated by appropriately modifying FORTH.

To mimic the development of a surety language, we constructed our own FORTH core to ensure that there were no hidden and dangerous functionalities in the language. While the standard method for constructing FORTH is to implement machine language versions of the core words, we saved time by simply using C-compiled executables for each core word in the dictionary. It was recognized that the concept of surety agents could be mimicked in the context of a FORTH dictionary by preventing a user's FORTH program from having direct access to the FORTH core words. Instead, a higher level of "safe" words could be created for use by a programmer. These higher level words would contain the surety agent code that would allow safety checks as needed. For example, we used this scheme to prevent access to particular RAM addresses by any code written using the "safe" FORTH words. This example suggests the feasibility of the "watchdog" surety agent concept. We also completed a plan and layout for the program-building GUI discussed above, and made some progress in implementing a simple version. For this simple case, we decided to use FORTH words as the execution tokens associated with each menu entry.

However, there was not enough time to complete the GUI, and so we did not produce FORTH code or associated English text versions of such code from a GUI. We were also unable to obtain follow on funding for this work. This is discussed below.

Lessons learned and Conclusions

The LDRD Programmatic team responsible for Software Surety at SNL has declined to invest in this surety language project for two consecutive years (full proposal 99-0203 for FY99 and proposal idea 00-0135 for FY00). There were two primary reasons given for the FY99 Surety PAT decision to not invest in this potential surety technology. First, it was argued that the technical barriers in this effort are too large, and the project is too high risk, for a SNL research activity. The capabilities of autonomous agents were not seen as clearly sufficient to guarantee the language properties that were desired. Put candidly, SNL'S LDRD Surety PAT and it's technical reviewer did not believe that the ideas described here would work. Second, it was argued that even a successful language faces a large social barrier to widespread acceptance. The example of the lack of general acceptance of the DOD's ADA language was cited by the LDRD review committee. A
concern of our own should also be noted here. We expect that the inclusion of surety functionality into any language is likely to reduce the execution speed of programs written in such a language, compared to code written in C or C++. This may be an unavoidable price to pay for achieving safe, reliable, and predictable software that can also be modified safely.

The problem of developing and maintaining large, robust software programs is a difficult one that remains after decades of computer science research. The progress achieved in this 9 month seed project did not uncover any clear "showstopper" difficulties. However, we were unable to make sufficient progress to convince skeptics that a swarm-based surety language would address this long-standing "software crisis". A much larger scale R&D effort than was available to this project would be needed first to understand how to realize and then to evaluate the intriguing approach suggested here.
Acknowledgment

J. W. Bartholomew, Dept. 1155, carried out the implementation of a C-based FORTH core for this project. We also acknowledge the programmatic support of S. Tom Picraux in Center 1100.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.
References


### DISTRIBUTION:

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Code</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MS0149</td>
<td>Chuck Meyers, 4000 – LDRD office</td>
</tr>
<tr>
<td>1</td>
<td>MS0612</td>
<td>Review and Approval Desk, 4912 for DOE/OSTI</td>
</tr>
<tr>
<td>2</td>
<td>MS0899</td>
<td>Technical Library, 4916</td>
</tr>
<tr>
<td>5</td>
<td>MS1423</td>
<td>Gordon C. Osbourn, 1155</td>
</tr>
<tr>
<td>1</td>
<td>MS9018</td>
<td>Central Technical Files, 8940-2</td>
</tr>
</tbody>
</table>