Direct Qualification of Digital Components

J. Dennis Lawrence
G. Gary Preckshot
Gary L. Johnson

This paper was prepared for submittal to the
American Nuclear Society
International Topical Meeting on Safety of Operating Reactors
October 11-14, 1998
San Francisco, California

July 8, 1998
Lawrence Livermore National Laboratory

This is a preprint of a paper intended for publication in a journal or proceedings.
Since changes may be made before publication, this preprint is made available with
the understanding that it will not be cited or reproduced without the permission of the author.
Disclaimer
This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes 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 the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.
ABSTRACT

Existing methods for qualifying digital system software for use in safety-critical systems are expensive and are based on inferences that are of doubtful validity. This report on work-in-progress describes a new approach to qualifying a class of small safety systems that can meet a number of design restrictions, where the restrictions are carefully crafted to permit safety qualification to be determined by means of static analysis of the software combined with a limited amount of testing. This approach differs from attempts to qualify unrestricted programs in the general case. Work that has been accomplished towards this goal is discussed in summary terms. The technique relies on transforming a program into a form of directed graph known as a flowgraph. Existing testing theory is used, sometimes with minor modifications, to derive a set of design restrictions that permit reasoning about safety properties of the program, based on analysis and limited testing. Future work required to complete the research is outlined.

I. INTRODUCTION

The developers of safety-critical instrumentation and control (I&C) systems must qualify the design of the components used in these systems. Examples of safety critical I&C systems include nuclear power plant protection systems, plutonium building ventilation system controls, hazardous-area monitoring, medical devices, and access control. For computer-based components, this qualification must include design qualification of the embedded software, which confirms that a component can be trusted to perform its safety functions under the full range of operating conditions. Design qualification is legally required for safety-critical equipment used in many applications. For example, 10 CFR 830 requires qualification for application in DOE nuclear facilities, and 10 CFR 50 requires qualification for applications in nuclear power plants.¹

In systems using analog technology the necessary confidence is often developed through a combination of design evaluation, testing and inspecting the equipment, and operating experience review.¹ It is possible to develop confidence for analog systems this way because (1) they are built from standard modules with known properties, (2) design documents are available and described in a well-understood language (e.g., schematic diagrams), (3) the performance is constrained by well-understood physics, and (4) physics models exist to predict the performance. These attributes allow modeling based upon physics principles to predict system functional performance. Most importantly, the models allow designers to predict discontinuities in the system transfer function. A limited set of testing or operating experience can then be used to test the design and validate conformance to the models. Knowledge of the predicted performance, and knowledge of the existence and location of transfer function discontinuities, allows the performance under the full range of input conditions

¹ 10 CFR 50 Appendix B requires rigorous quality assurance for in-house designed safety elements, but other provisions in the regulations permit the use of commercial products provided there is an acceptance procedure that assures quality similar to an Appendix B product.
Digital computer-based components are being used to replace obsolete analog components in safety systems. Computer-based components offer greater functionality and flexibility compared to the existing analog systems. For many applications, computer-based components offer greater measurement stability, which allows operation closer to safety limits while maintaining the same safety margins provided by the older analog systems. Operation closer to safety limits directly translates to income for commercial plants and translates to increased capabilities and availability for defense facilities. These advantages encourage the operators of such facilities to incorporate computer-based components into their safety systems through design modifications. Also, inevitably the obsolete analog components used in safety systems are becoming unavailable. In many cases the best available replacement components include embedded computer systems.

Unfortunately, the qualification approach outlined above for analog systems does not work for computer-based systems because typically: (1) the software in these systems is not constructed using standard modules, (2) software design information is unavailable, and (3) software functions are not constrained by physical laws. Therefore, it is difficult to construct practical design models to support the use of testing and operating experience for design qualification.

Design qualification of computer-based systems is currently based upon developing confidence in equipment by gaining confidence in the developer’s design, verification, and validation activities. This approach has at least three undesirable characteristics: (1) it does not directly measure the safety of the product, but draws an inference about safety from observation of the process that developed the product, (2) it requires labor-intensive inspections of development processes and records, and (3) it often cannot be applied to existing systems because development process records are unavailable. The first characteristic is unsettling because safety judgments depend upon a difficult-to-support inference. In fact, some software experts assert that there is no relationship between the quality of the software process and the intrinsic quality of the resulting code. The second characteristic means that the qualification process is time-consuming and expensive. The inspection process for software process and records is potentially unbounded, and the value of the results is uncertain. The third characteristic means that costly inspections often fail due to the unavailability of development records.

II. APPROACH

The project described here is exploring an alternative approach that allows qualification by test for an important subset of I&C components used in safety-critical applications. It is known to be possible to exhaustively test trivial systems. The project goal is to determine if a combination of static analysis and limited testing can be used to qualify a class of simple, but practical computer-based I&C components for safety application. This goal is accomplished by identifying design constraints that enable meaningful analysis and testing. Once such design constraints are identified, digital systems can be designed to allow for testing, or existing systems may be tested for conformance to the design constraints as a first step in a qualification process. This will considerably reduce the cost and monetary risk involved in qualifying commercial components for safety-critical service. The effect will be more rapid introduction of new technology into safety service, thus reducing costs and increasing safety margins.

Programs that can be analyzed by the new technique have a number of restrictions. Some of these are common in safety applications, so do not pose an unreasonable burden on developers. Other restrictions, primarily on program design structure, are necessary if the qualification approach is to work. At the present stage of research, we are satisfied with a sufficient set of design restrictions; later research will investigate the elimination or relaxation of the more onerous restrictions. As a beginning, the following fundamental restrictions are imposed.

- All input to the program comes from sensors, and all output goes to actuators. This is specifically intended to eliminate human operator interaction with the program. Such programs are typically used in embedded control applications.
- The program operates under a cyclic executive system. This executive causes the program to operate as a single repetitively executed program; the executive starts the program periodically, the program carries out its calculations and sets any necessary actuator results, and this process repeats indefinitely.
The program lacks memory. That is, the results on one execution of the program are not used in the next execution. In particular, this eliminates trajectories of input values through time. We anticipate removing this restriction later.

The safety assertion to be satisfied by the program is known precisely, as a mathematical relation on the sensor input values and actuator output values.

The analysis process begins by creating an abstraction of the program into the form of a flowgraph, and then translating this into a path tree form. There are four classes of faults in programs; existing testing theory is used to characterize these, and to create a method of examining the program by static analysis and limited testing to determine if faults exist that may compromise safety. Restrictions are imposed on the structure of the program that make this analysis possible.

III. FLOWGRAPHS

Mathematical reasoning about programs and program testing can be done by transforming the program into an abstract structure that preserves aspects of the program that are essential to the mathematical argument, and that eliminates aspects of the program that are irrelevant to the argument. The abstraction used for this project is one that is widely used in computer science — the flowgraph.

A flowgraph is a form of directed graph, where the nodes of the graph represent (possibly null) statements in the program and edges in the graph represent possible transfers of control. There is a single distinct node, the start node, to which transfer from within the program is not possible, and from which every node in the graph can be reached. There is another distinct node, the stop node, from which no transfer to the program takes place, and which can be reached from every node in the graph. See Fenton, Whitty and Kaposi for a more complete description of flowgraphs. A simple program is shown in Figure 1, and the corresponding flowgraph is shown in Figure 2.

Paths in the flowgraph can be defined from the start node to the stop node. Note that there may be an infinite number of such paths if there are loops within the flowgraph. These paths permit the flowgraph to be transformed into a tree structure, the flowgraph path tree, with root corresponding to the start node, and with one leaf of the tree for every path in the flowgraph. The example in Figure 1 can be transformed into the tree in Figure 3. Each path then can be analyzed, using the assignment and predicate statements represented by the nodes of the tree, to give a set of path conditions that cause the path to be executed and a path function that describes the computation carried out for each path of the tree. The path conditions and path functions for the tree in Figure 3 are shown in Figure 4. Note that two paths in the tree cannot be executed since the path conditions for those paths are internally inconsistent.

The path conditions can be used to define the input domains that cause the program to execute each path. There are two sets of such domains, the actual path domains used by the program (as defined by the actual predicates in the program) and the problem domains which partition the entire input domain into sets such that all points within a set should be treated the same way by the program, as defined by the requirements and safety assertions.

IV. SOFTWARE TESTING THEORY

Faults in program that can affect safety can be classified into four types, the first two have subtypes. These are as follows:

1. A domain fault occurs when a specific input value causes the program to execute on the wrong path due to a fault in the control flow of the program. Domain faults occur within flowgraph nodes that represent predicates.
   a. An incorrect boundary fault occurs when the boundaries between path domains do not correspond with boundaries in the problem domain.
   b. A wild jump fault occurs when the program makes an unexpected transfer of control, so that the expected path is not executed. A wild jump can occur only where a priori determination of a flow graph is impossible.

2. A missing path fault occurs when a required predicate does not appear in the program.
   a. An externally caused missing path fault occurs if some predicate required to distinguish one set

---

b start nodes and stop nodes are NULL computations into which and out of which control is transferred to and from the program.

---
in the problem domain from another is missing from the program.

b. An internally caused missing path fault occurs if a program exception causes execution of the path not to complete.

3. A timing fault occurs when the execution of the program for a specific input violates a timing requirement. This can occur if a time delay is incorrectly computed, or if a domain fault results in a longer or shorter loop than expected.

4. A computational fault occurs when a fault in some assignment statement causes the wrong function to be computed by the program for some specific input value. Computational faults occur within procedure nodes of the flowgraph.

It is possible to place limitations on the program structure in such a way that the presence or absence of all of these types of faults can be detected.

For example, incorrect boundary faults can be detected using a method originally developed by White,6 provided that all predicates are linear in the input variables. This means that every predicate can be written in the form \( ax + by + \ldots + cz \ op k \), where \( x, y, \ldots, z \) are input variables; \( a, b, \ldots, c, k \) are constants; and \( \op \) is a relational operator (\(<\), \(\leq\), \(=\), \(\neq\), \(>\), \(\geq\)). In this case, no more than \(3p(n+1)\) tests are required, where \(n\) is the number of input sensors and \(p\) is maximum number of predicates on a path.

As another example, timing faults can be determined using only static analysis if the time required to traverse the flowgraph path tree to each leaf can be determined statically. Otherwise, a test will be required on each path to determine the minimum and maximum time for that path; static analysis can then be used to determine the minimum and maximum time requirements for the entire flowgraph path tree.

Arguments similar to these can be developed for the remaining types of faults. Our arguments are limited to safety considerations; correctness is not the issue here. Incorrect safe results are explicitly permitted.

The various restrictions that are used in the analysis and test methods lead to the design restrictions placed on programs. These are listed in the next section.

V. DESIGN RESTRICTIONS

The analysis described above can be carried out if a number of conditions are satisfied. These are described on two levels. First, a general statement of a condition is given. Second, a list of specific requirements are provided that, if found to be true, is sufficient to meet the general statement. We do not imply that the list of specific requirements is necessary or that the list is minimal. We state only that it is sufficient. Relaxing these requirements is left for later research. The specific requirements are stated so that it can be determined by static analysis whether or not a program meets the requirements. Some of the second-level restrictions occur multiple times, since the same second-level restriction may be necessary to meet different first-level restrictions.

1. It must be possible to create a flowgraph that completely represents the execution of the program. That is to say, the flow of control given in the flowgraph is presumed not subject to operating system interruptions and can be statically determined.

   • There shall be no interrupts, except for timer interrupts which occur at fixed intervals and have a known, maximum duration. Other kinds of interrupts make it difficult to draw a flowgraph.

   • The operating system shall not perform time slicing. This is a special form of interrupt, so is subsumed by the first requirement. It is included because operating system functions tend to be overlooked.

   • There shall be no multi-processing and no multi-tasking on a single processor.

   • There shall be no dynamically variable program structures (such as go-to statements, where the address is a variable).

2. It must be possible to draw a finite flowgraph path tree from the flowgraph.

   • Each loop within the program must have an upper bound on the number of repetitions, where this bound is independent of any program input value.

   • There shall be no unbounded recursion. Specifically, the maximum number of times a...
recursive call may occur must be statically determinable independent of any input value.

3. It must be possible to determine if domain faults exist.
   - All predicates within the flowgraph tree must be simple\(^c\) and linear in the input variables.
   - The path computed by adjacent portions of the path domain must differ in a way that can be detected by static analysis or testing.
   - Path domain sets must be finite.
   - There shall be no dynamically variable program structures (such as go-to statements, where the address is a variable).

4. It must be possible to determine if any paths have been omitted from the flowgraph path tree.
   - Functions computed for each problem domain must be known and computable.
   - The path function for each path in the flowgraph tree shall compute the same function as that required by the problem domain within which the input for that path resides.
   - There shall be no unbounded recursion.
   - The maximum size of every data structure shall be fixed and finite.
   - There shall be no dynamic memory allocation.
   - Floating point arithmetic shall be forbidden.
   - There shall be no width conversion that reduces the word size unless it can be statically shown that such conversion cannot cause a failure. That is, width conversion is permitted if it can be demonstrated that the value in the larger-sized variable will actually fit in the smaller-sized variable.
   - Variant structures without tags shall be forbidden.

\(^c\) No side effects.

5. It must be possible to determine a finite upper and lower bound on the time required to execute each path in the flowgraph path tree.
   - Minimum and maximum timing requirements for each node in the flowgraph must be computable.
   - There shall be no interrupts (except timer interrupts), multi-processing, multi-tasking or time slicing. Each of these makes timing analysis difficult to impossible. Timer interrupts that occur at fixed intervals of time and are of known duration are permitted, since their impact on path timing can be determined.
   - Each loop within the program must have a predetermined upper bound on the number of repetitions.
   - Each recursive call within the program must have a predetermined upper bound on the depth of recursion.

6. It must be possible to either detect or to prevent computational faults.
   - Floating point arithmetic shall not be used.
   - Multi-tasking shall be prohibited. This eliminates a variety of synchronization errors and misuse of shared data items that is difficult to detect.
   - All procedure and function calls must be protected by a prototype declaration supported by strict use of data typing. This helps reduce data incompatibility errors during function usage.
   - There shall be no width conversion that reduces the word size unless it can be statically shown that such conversion cannot cause a failure. That is, width conversion is permitted if it can be demonstrated that the value in the larger-sized variable will actually fit in the smaller-sized variable.
   - All variables shall be declared and typed according to usage in the problem domain. This
enables compiler type checking, which is known to be a very effective means of reducing computational errors.

- Unspecified, undefined and implementation-defined programming language features must be avoided.

VI. FUTURE WORK

Much remains to be done. The sufficiency of this set of design rules appears plausible, but has not yet been proved. The theory is being tested using a series of increasingly complex programs. Some of the limitations — particularly two of those listed in Section II — need to be removed. In particular, we wish to extend the theory to cover inputs coming from and outputs going to other computers, as well as sensors and actuators. Programs with memory need to be incorporated. Some of the design rules in Section V — particularly the prohibitions on floating point and restrictions on pointers — should be re-examined to see if they can be partially relaxed.

The primary goal for the coming year will be to complete this theoretical work, and use it to qualify some existing commercial product for a safety application.

Our ultimate goal is to deduce a set of design rules and corresponding validation methods that will enable reusable and qualifiable commercial software to be written for the safety-critical marketplace. We already know that the general problem of qualifying any program is insoluble; we think the problem may be soluble for an important subset.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

REFERENCES


Read $i, j$

if $i \leq j + 1$

then $k = i + j - 1$

else $k = 2 \times i + 1$

if $k \geq i + 1$

then $L = i + 1$

else $L = j - 1$

if $i = 5$

then $m = 2 \times L + k$

else $m = L + 2 \times k - 1$

write $m$

Figure 1. Sample Program from White and Cohen

Figure 2. Flowgraph for Sample Program
Figure 3. Flowgraph Tree for Sample Program

<table>
<thead>
<tr>
<th></th>
<th>$i \leq j + 1$</th>
<th>$k \geq i + 1$</th>
<th>$i = 5$</th>
<th>$k$</th>
<th>$L$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTT</td>
<td>$i \leq j + 1$</td>
<td>$j \geq 2$</td>
<td>$i = 5$</td>
<td>$i + j - 1$</td>
<td>$i + 1$</td>
<td>$3i + j + 1$</td>
</tr>
<tr>
<td>TTF</td>
<td>$i \leq j + 1$</td>
<td>$j \geq 2$</td>
<td>$i = 5$</td>
<td>$i + j - 1$</td>
<td>$i + 1$</td>
<td>$3i + 2j - 2$</td>
</tr>
<tr>
<td>TFT</td>
<td>$i \leq j + 1$</td>
<td>$j &lt; 2$</td>
<td>$i = 5$</td>
<td>This path domain is nonexistent since it implies $i = 5$ and $i \leq 3$.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TFF</td>
<td>$i \leq j + 1$</td>
<td>$j &lt; 2$</td>
<td>$i = 5$</td>
<td>$i + j - 1$</td>
<td>$j - 1$</td>
<td>$2i + 3j$</td>
</tr>
<tr>
<td>FTT</td>
<td>$i &gt; j + 1$</td>
<td>$i \geq 0$</td>
<td>$i = 5$</td>
<td>$2i + 1$</td>
<td>$i + 1$</td>
<td>$4i + 3$</td>
</tr>
<tr>
<td>FFT</td>
<td>$i &gt; j + 1$</td>
<td>$i \geq 0$</td>
<td>$i = 5$</td>
<td>$2i + 1$</td>
<td>$i + 1$</td>
<td>$5i + 2$</td>
</tr>
<tr>
<td>FFF</td>
<td>$i &gt; j + 1$</td>
<td>$i &lt; 0$</td>
<td>$i = 5$</td>
<td>This path domain is nonexistent since it implies $i &lt; 0$ and $i = 5$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Predicates and Path Functions for Sample Program