EXHAUSTIVE TESTING AS A VERIFICATION TECHNIQUE†

John C. Knight, Kevin G. Wika and Shannon Wrege

(knight | wika | shannon)@virginia.EDU

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Contact Author:

John C. Knight
Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903

(804)982-2216
knight@Virginia.EDU

†. Supported in part by the National Science Foundation under grant number CCR-9213427 and in part by NASA under grant number NAG1-1123-FDP.

ABSTRACT

For a safety-critical system, i.e., a system whose consequences of failure are very high, it is not possible to rely upon testing to provide the necessary verification. The difficulties arise mainly from the shear number of tests that are required to permit statistically meaningful conclusions to be drawn about the system. Other difficulties with testing include failure to observe erroneous output when it occurs and incorrectly defining the operational profile from which to select inputs.

Goodenough and Gerhart observed that exhaustive testing of a software system amounts to a proof of the software. This is an appealing thought for safety-critical systems because establishing proofs of system properties by more traditional techniques is difficult at best and often depends on questionable assumptions such as assuming correct translation by a compiler. Unfortunately, however, if testing that yields a statistical conclusion is infeasible it would seem that exhaustive testing would be also.

In general, exhaustive testing is infeasible. However, this infeasibility is a direct result of the goal of testing for overall functional correctness. In considering the general issue of how safety-critical systems might be tested, we have concluded that a different view of testing is required. The view we advocate is that testing should be used to show significant *properties* of safety-critical software systems rather than overall correctness. This approach to testing is analogous to the use of formal verification to demonstrate properties rather that correctness.

When testing is used as a technique to establish a property, the property of interest determines in large part the number of tests required. This is the case no matter whether the goal is to establish properties in a statistical sense or in the sense of a proof using exhaustive testing. By careful definition of the property and by the application of a technique called *specification limitation*, we have been able to prove a number of significant properties of a large software system by exhaustive testing.

In this paper we formalize the notion of specification limitation and show how it can be applied in practice. We present the details of a complex software system and associated properties that were established by applying the technique.

INTRODUCTION

Computing systems in which the consequences of failure can be very high are termed *safety-critical*. Many such systems exist in application domains such as aerospace, defense, transportation, power-generation, and medicine, and public exposure to these safety-critical systems is increasing rapidly. Since the correct operation of these systems depends on software, the possibility of serious damage resulting from a software defect is considerable and growing.

Because of the extreme consequences of failure, the probability of failure of safety-critical systems has to be very small. In fact, in some cases there are mandated limits on the probability of failure that is acceptable. The FAA, for example, requires that failure of flight-critical systems for commercial air transports be “highly improbably”, a term that is interpreted as a probability of failure of less than 10^{-9} per hour. Systems requiring such levels of dependability are sometimes referred to as *ultra dependable*.

Researchers have shown that demonstration of limits such as these by testing for non-trivial systems is infeasible mainly because of the shear number of tests that would be required. Although any safety-critical system will be tested before deployment, the result of this testing is usually informal. The developers’ “confidence” in the system is raised by the testing but this is not a quality that can be quantified.

With this limitation on testing, researchers have appealed to mathematics to provide an alternative to testing that can provide a more rigorous approach to assuring software dependability. Formal methods in general and formal verification in particular provide an approach to the analysis of software that has the potential for yielding very high levels of assurance in various software properties.

Typically, however, formal methods are limited in the view that they take of a software system. Formal methods are usually applied to high-level abstract notations rather than the machine instructions that constitute the actual software. Similarly, the modern application of formal methods is usually far short of what is sometimes called a “proof of correctness”. Rather than trying to show the equivalence of two representations, a formal specifi-

cation and an implementation for example, formal methods are more easily applied to the demonstration of certain properties of a software system. A proof might be undertaken, for example, to show that a software system cannot exceed the capacity of a critical data structure no matter what sequence of events it processes. This proof ensures that the software possesses an extremely useful property although, of course, it might still fail for some reason unrelated to the data structure.

The infeasibility of testing as a verification technique is based on the assumption that testing is being used to demonstrate overall functional correctness. In this paper, we argue that testing becomes a feasible technology in the verification of safety-critical systems if it is used to demonstrate limited but significant properties of the software rather than functional correctness. This is in direct analogy with the current successful approach being pursued with formal verification.

The approach we advocate is to test exhaustively rather than to test a sample of the input space. This permits the establishment of a proof of the property of interest rather than merely a statistical conclusion. In order to make exhaustive testing practical, we argue that safety-critical systems should be designed with the goal of making exhaustive testing of important properties feasible. We suggest various techniques that permit this together with various other techniques that deal with other issues in testing.

To explore the potential value of the approaches we propose, we discuss an application to which the techniques have been applied. We show how a variety of useful properties have been established by testing for this application.

TESTING SAFETY-CRITICAL APPLICATIONS

Naturally, safety-critical systems are tested before being deployed. In practice, it is common for a large proportion of the total effort expended during the development of such systems to be expended on testing. In fact, in some cases 90% of the development effort is used for testing. Despite this expenditure of resources, the conclusions that can be drawn from the testing process are usually informal.

In attempting to draw rigorous conclusions, researchers have raised a number of issues about testing safety-critical applications. Specific issues that are of concern are: the difficulty of quantifying reliability from test data; the difficulty of determining the correctness of the output produced by a program; and the difficulty of determining the operational profile. We elaborate briefly on each of these in turn and then discuss how the issues might be resolved.

Quantifying Reliability

The quantification of reliability by testing is common in the hardware domain where it is also referred to as *life testing*. In order to apply life testing to software, test inputs are selected from the assumed operational profile and the software is executed as if it were operating in production circumstances. In essence this is a form of functional testing. However, the goal is not to determine traditional functional correctness but to quantify its reliability, i.e., to determine how reliable the software will be in practice.

The difficulties associated with life testing of safety-critical software were first discussed in detail by Butler and Finelli [3]. The major issue with quantification based on testing in this way is the large number of test cases required. It is essential that the quantification be within a statistical framework in which a bound on the reliability goal of the software is stated as a null hypothesis and an experiment defined, i.e., the testing, can establish that the null hypothesis is true with a specific confidence. The large number of tests derives from the very low bound that is desired on the reliability and the very high confidence that is required in that bound. In practice, showing, for example, that software has a probability of failure of less than 10^{-9} per hour with a confidence of say 99.9% requires so many tests to be run that this bound with this confidence just cannot be demonstrated.

Correctness Checking

The difficulties associated with quantifying dependability by testing are just part of the problem. The analysis given by Butler and Finelli depends upon a critical assumption - that when the software fails, its failure is detected. In practice, this assumption almost certainly does not hold.

Large software systems are very complicated functions. They are, for the most part, not continuous functions and often are functions of time. Frequently, they are not even one-to-one functions. Knowing *precisely* what the output of a large software system should be for any given input is impossible. If it were possible, then, by definition, software reliability would be achieved easily since the checking system could replace the software.

This observation just makes the problem of testing safety-critical systems worse. Ammann et al have shown that the number of tests required to show a certain level of reliability (in the sense of Butler and Finelli) either rises rapidly as the number of missed failures rises or becomes infinite depending on the confidence required and the underlying distributions [1].

Operational Profile

The operational profile is the probability distribution associated with the various input values that the software will see. If all input values are equally likely, this distribution will be uniform. If some input values are more likely than others, then the distribution will be correspondingly non-uniform.

The difficulty is knowing what the actual probability distribution will be. All that can be done in practice is to make a well-reasoned guess. The guess can be based on observation of the expected operational environment, simulation, analysis, and so on, but it will always be a guess. And since it is a guess it could be arbitrarily far from the actual probability distribution that the software sees when it enters service.

This possible discrepancy is unimportant if the software operates correctly. No matter how many input values are presented from some region of the input space, and it is this number that the probability distribution predicts, the software will deal with all of them properly if it operates correctly. Unfortunately, this desirable state of affairs will not be revealed until the software is in operation. In order to make an *accurate* reliability estimate of the system based on life testing before deployment, the tests used must be generated from an accurate operational profile. If the profile is inaccurate, then so is the reliability bound.

EXHAUSTIVE TESTING

The discussion in the previous section suggests that no useful formal conclusions can be drawn from testing a safety-critical application. The first limitation is the huge number of test cases required. Even if large numbers of tests could somehow be run, failures might not be observed. And even if all failures were observed, the assumed operational profile might be wrong.

Knowing that a safety-critical software system will operate correctly is extremely important, however. For this reason and because testing is at best resource intensive, a great deal of research has been and continues to be conducted into the application of mathematics to software verification. The goal in that case, of course, is to permit conclusions to be drawn about the software without having to execute it.

Despite the merit of this approach to proof, when applied to large software systems the practical circumstances present usually limit the conclusions that are possible. For example, if a proof is established that the output of a sort program produces a valid ordering of its input, the proof usually makes assumptions such as: (1) that the compiler correctly translates the program, (2) that the linker and loader operate correctly, (3) that the operating system executes the program correctly, and (4) that no arithmetic exceptions are raised during execution. In addition, establishing proofs of significant properties of significant programs is not a routine activity for most developers at this point. Such proofs are rare rather than a standard part of development although this situation is improving quickly.

Goodenough and Gerhart pointed out that exhaustive testing of a program also constitutes a proof [4]. If a program has been executed with every input value that it could ever see and its output found to be acceptable for each of these input values, then clearly nothing more need be done. This is an appealing thought for safety-critical systems given that rigorous demonstration of dependability is so important and that traditional formal verification is quite difficult to apply. Unfortunately, however, if testing that yields a statistical conclusion is infeasible for systems requiring ultra dependability, it would seem that exhaustive testing would be infeasible also since it requires even more tests to be run.

In general, of course, exhaustive testing is infeasible. However, we claim that it can be applied to a wide variety of real programs to yield valuable results if:

- testing is used to demonstrate useful but possibly narrow properties rather than overall functional correctness, and
- the size of the set of values that a software system might read for any given input is made as small as possible consistent with the demands of the application.

We refer to the first item above as *property testing* and the second as *specification limitation*, and we describe them in more detail in the following subsections.

We do not discuss further the difficulty of determining the operational profile that a system will experience since that difficulty is eliminated trivially by the use of exhaustive testing. The operational profile is an important input to a life-testing process because the goal is to establish a statistical bound on dependability. Since exhaustive testing amounts to a proof, the operational profile is not an issue. In view of the significant impact that the operational profile has on the statistical models of life testing, the importance of its elimination should not be underestimated.

Property Testing

The large number of tests implied by the analysis of Butler and Finelli is a direct result of the goal of testing for overall functional correctness. In considering the general issue of how safety-critical systems might be tested, we have concluded that a different view of testing is required. The view we advocate is that testing should be used to show significant *properties* of safety-critical software systems rather than overall correctness. This approach to testing is analogous to the use of formal verification to demonstrate properties rather that correctness.

When testing is used as a technique to establish a property, the property of interest determines in large part the number of tests required. This is the case no matter whether the goal is to establish properties in a statistical sense or in the sense of a proof using exhaustive testing. By careful definition of the desired property, the number of tests required can be made tractable. In the limit, the simplest possible property of any system is that it operates correctly for a single set of input values. Clearly this property can be

shown by exhaustive testing since only one test is required. More generally, we hypothesize that significant properties can be defined which can be shown by exhaustive testing. We have defined and proved a number of extremely valuable properties for the example system that we discuss below.

Although it might be argued that restricting verification by exhaustive testing to specific properties rather than overall functional correctness is of limited value, we note that it is directly analogous to the statement and proof of putative theorems about a formal specification. Putative theorems are used to establish expected properties of a specification rather than some kind of overall correctness. Their power lies in the fact that they establish a result or results which the specifier expected and allow him or her to focus on other elements of the specification. In a similar way, proof by exhaustive testing that a program possess expected properties ensures that failure will not result from lack of the associated properties and permits the verifier to focus on other elements of the verification.

Specification Limitation

The input space for a program is the set of all possible input values that a program might encounter. It is the enormous size of this space for typical applications that is behind Butler and Finelli's conclusions on the infeasibility of testing for ultra dependability.

What defines the size of this space and why is it so large? Each input to a software system at the point where the software first processes it is a discrete quantity. The possible values that an input can take, the *value set* of the input, is therefore a finite set but the cardinality of the set might be quite large. For systems with several inputs, the input space is actually the Cartesian product of several separate value sets, and the size grows very rapidly as the number of inputs increases.

In theory, the value sets of the individual inputs for a particular software system are defined by the specification of the system. In practice, the value sets tend to take on default values that are determined by hardware constraints. An integer input derived from a sensor, for example, is assumed usually to have a value set equal to the hardware representation produced by an analog-to-digital converter. We claim that this approach to the definition of value sets leads to a large, undesirable, and unnecessary expansion of the

input space. By forcing the definition of the value set to be large enough to accommodate the needs of the application *but no larger*, it is possible to reduce the size of the input space dramatically without affecting the operation of the application. We refer to this approach as *specification limitation*.

As an example of specification limitation, consider a simple control system with a single sensor measuring a temperature. For room temperature applications, it is unlikely that more than eight bits would be needed for temperature representation. However, if a sixteen-bit signal is presented, this tends to be considered the definition of the value set. By careful examination of the application, it might be possible to reduce the required precision to perhaps six bits. If this is possible and inputs are deliberately truncated to six bits, the value set is reduced from a sixteen-bit integer to a six-bit integer. This is a reduction of more than three decimal orders of magnitude in the size of the value set.

To be more precise, specification limitation involves the determination of precisely what the minimal value set is for an input. In some cases, it might be possible to itemize the specific values that are possible. More generally, however, we have found that restricting the size of the value set tends to depend on the following two techniques:

- *range limitation*: by which we mean deliberately restricting the range that an input value can take, perhaps composing a value set from a set of non-intersecting subranges, and
- *granularity enhancement*: by which we mean deliberately forcing elements of the value set to be regularly spaced across the range with as large a spacing as possible between elements. The six-bit temperature value discussed above is an example.

If range limitation is applied and the valid range checked at the point of input, then only values from within the range need be tested. If granularity enhancement is applied and the granularity used by the hardware is truncated to the enhanced granularity at the point of input, then only truncated values need be tested.

Specification limitation can be applied to almost all inputs of a software system, and the result can be a significant reduction in the system's input space. The effect that this has on the feasibility of exhaustive testing is, of course, system dependent. We give examples in the application to which we have applied the technique showing at least one example in which specification limitation has made exhaustive testing feasible.

CORRECTNESS CHECKING

Correctness checking, i.e., determining whether the output produced by a program is indeed the output that is desired, is a very difficult problem. From the perspective of verification by testing, it is crucial that it be performed reliably and that it be totally automated. As noted above, undetected failures affect statistical conclusions very dramatically and they certainly invalidate a proof by exhaustive testing.

There is no perfect approach to correctness checking for any type of program and certainly not for complex safety-critical software. Each application must be examined separately and all available approaches considered. Rather than discussing the problem in general, we limit our attention here to a technique, *reversal checking*, that in our experience is powerful yet rarely used.

A reversal check is a calculation that takes the output of some computation and regenerates the associated input (see Fig. 1). If the regenerated input matches the actual input up to the limits imposed by numerical error then there are only two possibilities: either the output is correct or the forward and reverse calculations contain faults that are the *inverses* of each other. Even if the latter were the case, it is unlikely that they would be the *exact*

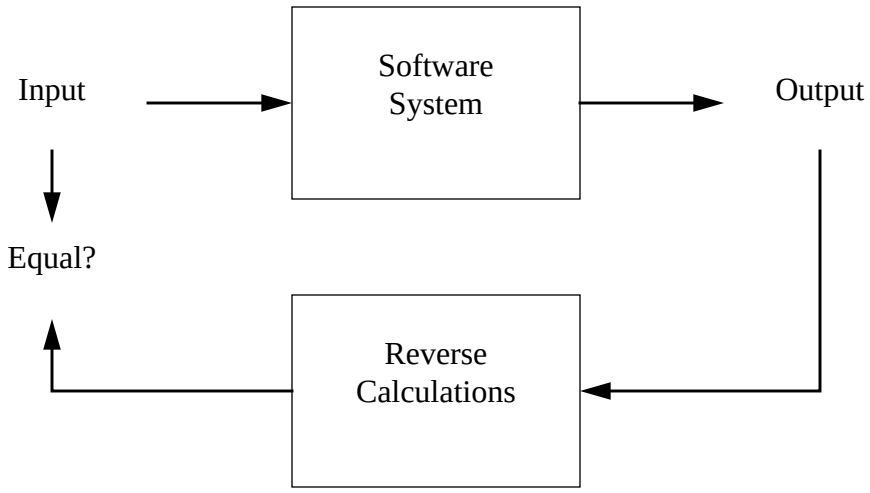

Fig. 1. Application of a reversal check.

inverses of each other. By exact inverse, we mean that the faults do not permit a failure to be detected on *any* test case whatsoever. In other words, for a fault to go undetected, the forward computation would have to fail in such a way that when its output is used as the input for the reverse computation, that computation fails in such a way that its output is indistinguishable for the original input. And this must occur on every test case for which the system's outputs are in fact wrong. This seems unlikely and so provided sufficient test cases are executed, the probability of detecting the faults can be raised to an acceptable level [2].

Reversal checking is not a cure-all for the problem of correctness checking. For example, nothing of any value can be said about the input operands to an add operation if all that is known is its output. But in many cases, relatively simple inverses do exist, and, when they do, they provide an excellent basis for high-quality, automatic error detection. We describe the extensive use of reversal checks in the application to which this work has been applied.

EXAMPLE APPLICATION

As part of a process of evaluation, all of the techniques discussed in this paper have been applied to a prototype software system that we have developed for a safety-critical application. Since we have looked at only a single application, the evaluation that we describe is merely a feasibility demonstration.

The application, *The Magnetic Stereotaxis System* (MSS), is an investigational device for performing human neurosurgery being developed in a joint effort between the Department of Physics at the University of Virginia and the Department of Neurosurgery at the University of Iowa [7].

The system operates by manipulating a small permanent magnet (known as a "seed") within the brain using an externally applied magnetic field. By varying the magnitude and gradient of the external magnetic field, the seed can be moved along a non-linear path and positioned at a site requiring therapy, e.g., a tumor. The magnetic field required for movement through brain tissue is extremely high, and, in the MSS, the required field is gener-

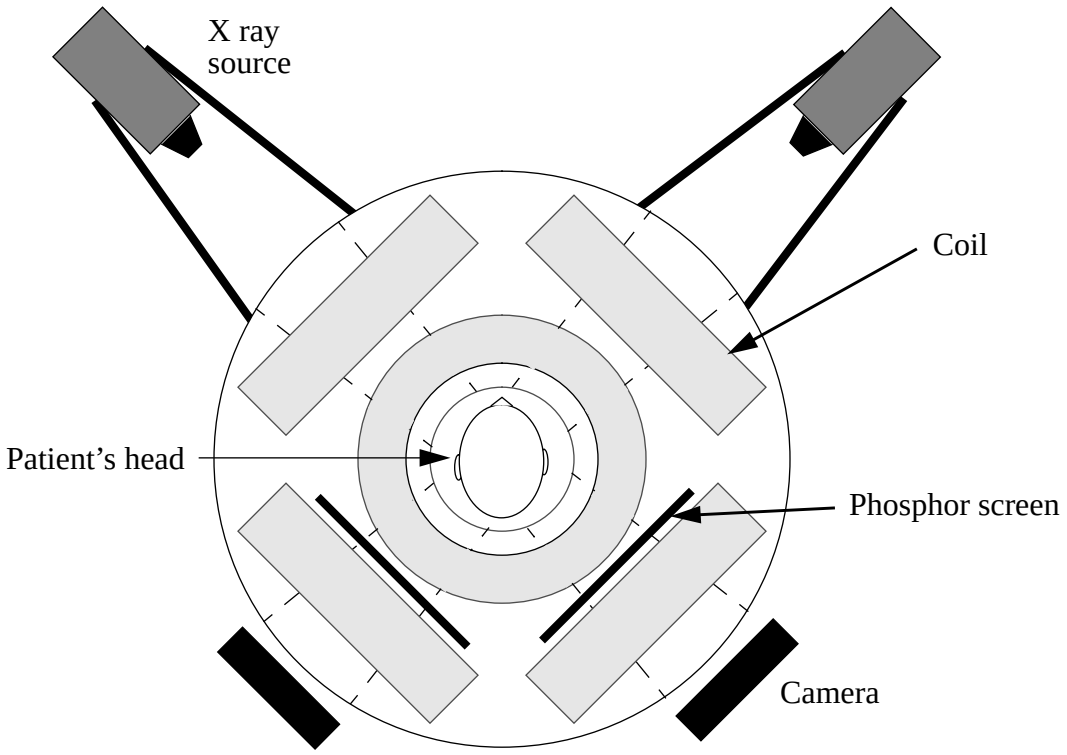

Fig. 2. View of MSS from above patient's head.

ated by a set of six superconducting magnets that are located in a housing that surrounds the patient's head. Fig. 2 shows how the system is organized.

A key element of the device is the imaging subsystem. It uses two X-ray cameras positioned at right angles to detect in real time the locations of the seed and of X-ray opaque markers affixed to the patient's skull. The X-ray images are not displayed. Instead, they are processed by the imaging subsystem so as to locate the objects of interest in a canonical frame of reference, and this information is used to display graphic representations of the seed and skull markers on pre-operative magnetic resonance (MR) images. The MR images are the primary source of information used by the operator for making control decisions.

The application requires several high-resolution graphic displays and these depend upon extensive computation for data generation. This demand for computation and display has forced the use of a distributed architecture for our prototype. This in turn has forced the use of a substantial amount of "off-the-shelf" software to perform routine operating

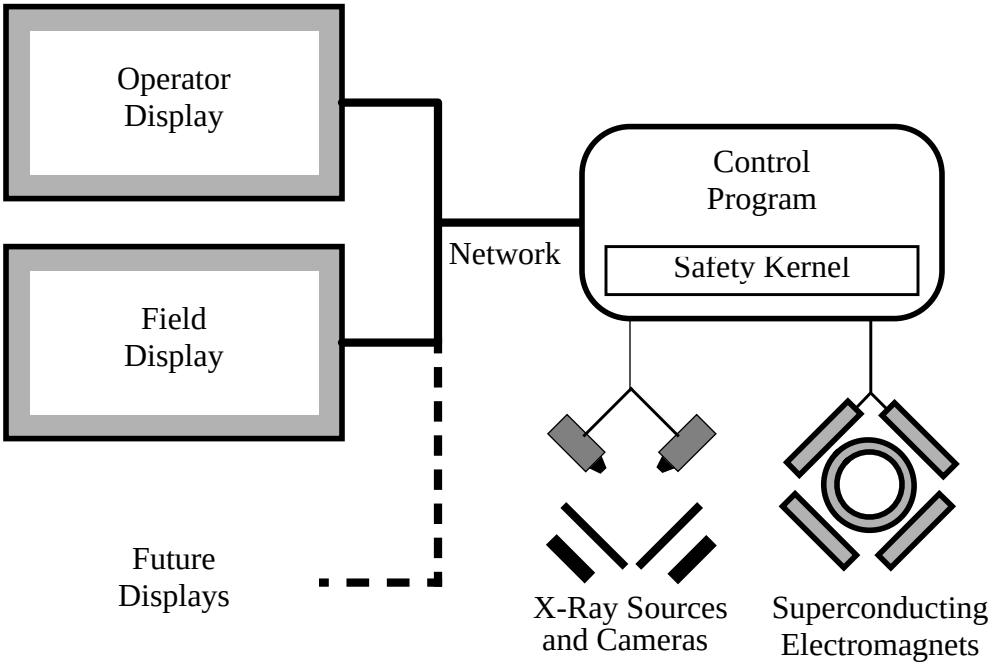

Fig. 3. Distributed MSS software architecture

system and network functions. The overall software architecture of the prototype system, shown in Fig. 3, consists of a control program that interfaces with the various peripherals and two display programs. Communication between these programs is over a local-area network. Each program executes on a separate computer running Unix and the network links are Unix socket connections. The graphic user interfaces are implemented using X Windows.

An unusual feature of our prototype system architecture is the use of a *safety kernel* [6]. Its purpose is to ensure that certain required safety policies are enforced irrespective of the actions of the remainder of the application software.

TEST SYSTEM STRUCTURE

A test harness has been developed that permits testing of the complete prototype software system or major subsystems. The test harness operates as a separate program on a different computer from the rest of the system. It communicates with the software under test via two socket connections. One of these connections is used to transfer synthetic

images to the replacement X-ray device driver. The second is used to transmit to the operator display operator “commands” that the test harness determines to be part of a test case.

Synthesized X-ray images that are produced as follows. The desired object positions in the canonical coordinate system are generated initially by the test harness as part of a test case. The projections of the objects onto the two camera sensing surfaces are then computed and used to place correctly located shadows onto the two digital images. These projections take into account a multitude of deviations from perfect positioning of the real equipment. The X-ray sources, for example, are not located precisely on a line perpendicular to the center of the camera. Once the projections have been generated, realistic distortions are applied to the images, and finally the images are mixed with real X-ray backgrounds to produce extremely high-fidelity synthetic images.

The Control Program’s interface with the synthetic image generator is identical to the real X-ray system down to the level of the device driver. The synthetic images arrive over a socket connection whereas the real X-ray images would arrive via a custom communications system.

To handle operator commands, a relatively small addition has been made to the Operator Display program. This addition accepts directions from the Test Harness that are in the form of high-level action requests such as “push the calibrate button”. The modification to the Operator Display, referred to as the *Pseudo User*, transforms the high-level directions either into X events that it injects into the event queue, or, if necessary, it calls the associated call-back functions.

VERIFICATION EXPERIMENTS

In this section we describe the application of property testing and specification limitation to our MSS prototype implementation. This section is worded in the future tense in part. This is because in most cases we have performed only subsets of the exhaustive testing that is mentioned, and used the resulting information and other analyses to predict both the feasibility of exhaustive testing and the expected resource requirements. To actually carry out a complete exhaustive test would be pointless other than to confirm the analyses

since our prototype changes frequently and any change would invalidate the verification.

Imaging Subsystem

The MSS imaging subsystem is complex, and knowing that it locates objects correctly and within required accuracy bounds is useful. The property that we have defined for proof by testing is:

> *The imaging subsystem determines the positions of objects within its field of view to within a predefined tolerance of the correct location assuming that its functionality is not affected by the background in the images upon which it operates.*

In other words, we seek to show that the object location algorithm and its implementation works correctly given shadows of objects at any points on the input images. We assume that the verification is unaffected by the variations in background that are inevitable in practice.

This property can be established by exhaustive testing. The approach we intend to undertake is to place an object in the field of view and move it systematically throughout the entire operating region, i.e., raster scan the object through the operating region. Although we speak of objects and motion here, recall that the images are synthesized, and so the raster scanning referred to is in fact performed by a set of nested loops.

Since the images are digital, there is a minimum distance that an object has to move in order for the move to be detectable. If we move the target object at the resolution limit, we are guaranteed to achieve exhaustive testing of the imaging subsystem's ability to determine the position of an object in the field of view. The estimated time required to perform these tests is only on the order of 1,500 hours.

Although this is a useful property, it is by no means a complete test of the imaging subsystem, and it is important to keep this in mind. This set of test cases does not test, for example, the imaging system's ability to detect objects in the presence of noise, its ability to distinguish between multiple objects, and so on. However, exhaustive testing of the form described above proves that one significant aspect of image processing is being performed correctly.

Coil Current Calculation

In a similar fashion to the imaging system testing, we can apply exhaustive testing to the coil current calculation to show a narrow but useful property. The current calculation is based on a desired seed movement and present seed location. The property that we have defined for proof by testing is:

> *The coil-current subsystem determines the coil currents correctly for any desired seed movement from any present seed location.*

This property can only be proved if specification limitation is applied. Although generally real-valued quantities, we apply granularity enhancement to the seed location and to the requested direction and magnitude of seed movement deliberately for the express purpose of permitting exhaustive testing. Details of the specification limitation that is used are as follows. During testing, the seed is positioned at small intervals (e.g., 1.0 mm) on a three dimensional grid. During use of the system, the seed position is always rounded so that it matches these tested locations. In addition, for each possible seed location, a finite number of movement directions and a similarly limited number of distances have been established. Thus, the coil currents for all possible combinations of seed location and requested seed movement can be computed and the total number of tests is bounded.

In our prototype software for the MSS, we have established a grid size of 1.0 mm with the working volume being approximated by a cube that is 100 mm on a side. The total number of points in this volume is 10^6. The symmetry of the coil arrangement permits the actual number of points tested to be reduced by a factor of 16. The seed movement direction interval is set at 5.0 degrees over the range of the angular spherical coordinates. The movement distance is rounded to the nearest 1.0 mm over the range 1.0 to 10.0 mm. These intervals have been selected on the basis that they permit sufficient operator control of seed movement with a worst case resolution of approximately 0.5 mm. These intervals result in approximately 700 directions for each point and 6×10^4 points to be tested. For this test, the distance of seed movement affects only the magnitude and duration of the current, so it is acceptable to test just the maximum distance (i.e., 10.0 mm) for each direction. As a result, the total number of cases to be tested is roughly 4×10^7. Testing 10 cases per second, the estimated time required to perform these tests is on the order of 1000 hrs. The result is that all possible movements that can be requested during a surgical procedure

can be evaluated during the exhaustive testing. In addition to ensuring that the magnetic force is consistent with the requested movement, testing will also incorporate other analysis, e.g., checking coil current values to ensure that the seed is not moving into a region where a rapidly increasing magnetic field would interfere with safe seed movement.

The rounding of the seed position and establishment of discrete movement requests are certainly not functional requirements of the system, but are design decisions made to enable exhaustive testing to assure a valuable safety property. The adjustments in the seed position are on the order of the resolution of the imaging system and are significantly smaller than the spacing of points used to compute the force on the seed. The discretization of the force direction and magnitude is established at a level that permits control of the seed at a resolution that is on the order of the resolution of the vision system. As a result, these minor restrictions will have no adverse impact on the functional or safety properties of the system while permitting the assurance of a significant safety property.

Safety Kernel Verification

Since the safety-kernel architecture is designed to ensure the enforcement of certain safety policies, its correct operation is crucial to safe operation of the system. Exhaustive testing has been used as part of the verification of the kernel. Specifically, it has been used to verify the two properties shown below:

> *For any safety kernel mode, a command for a device will not be executed if the command is not permitted in the given mode.*
>
> *For any safety kernel mode, a command will be executed if a transition to a valid mode has been specified.*

These properties certainly do not imply safety kernel correctness, but they contribute to the overall verification and are properties that would be difficult to establish with other techniques. For example, it is likely that random, system testing would not test these properties exhaustively and that formal verification between the policy specification and the executable safety kernel would be very complicated.

Using a specification-based test system both of these properties have been tested exhaustively. For the safety kernel component tested, there were 24 separate modes and in excess of 20 commands for each mode. Testing required a few minutes for completion.

The entire test process is automated, permitting ready application to different and more complex instances of the safety kernel.

ERROR DETECTION

Somewhat surprisingly, the entire MSS computation sequence that effects coil control can be covered by reversal checks. It is not possible to discuss this coverage here in detail and so we describe the reversal checks used in two major subsystems as examples.

Object Location

From the location of objects in the two input X-ray images and using previously-obtained calibration data, the imaging system determines the location of the objects in the canonical three-dimensional coordinate system. Recall, however, that in the test system, the images are synthesized. Synthesis is done from object-location data that is generated as part of the test case, and this object-location data is with reference to the canonical coordinate system. Thus, the imaging system is analyzing images in an effort to determine the very data that was originally part of the test case. The correctness check is, therefore, merely to compare the output of the imaging system with the initial test case data. The sequence of steps is shown in detail in Fig. 4.

Coil Current Calculation

The computation of the required currents through the superconducting coils is a surprisingly difficult task. The difficulty arises because the input to the computation is a required force. The output is a six-element vector, i.e., the six coil currents required. There is an infinite number of current combinations that could provide a particular requested force. There is no known way to compute the currents in any optimal sense and so various complex approximations are used.

Fortunately, the force produced by a set of six currents passing through coils in a known geometric configuration, i.e. the reverse computation, is easily and exactly computable. Thus, in this case, the reversal check is merely to compute the force that would be produced by the set of six currents determined by the system and compare the value with

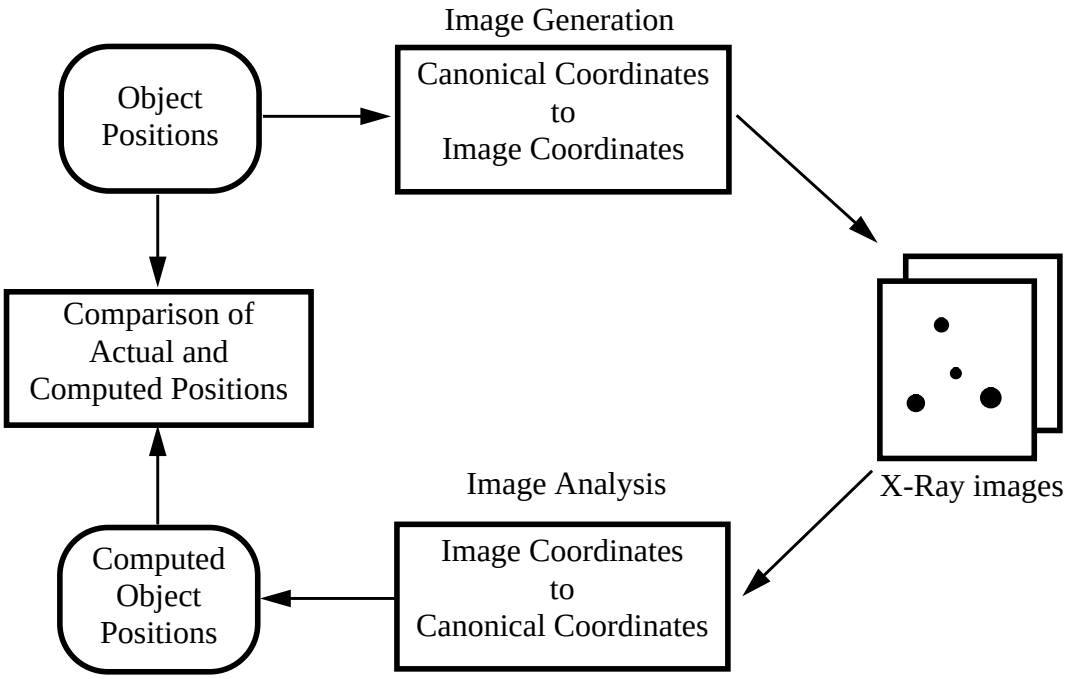

Fig. 4. Reversal check testing of imaging system.

the desired force.

We find the application of reversal checks to the entire system to be an appealing approach to error detection. Determining rigorously whether the systematic use of reversal checks as an error detection mechanism is indeed an approach with quantifiable benefits is the subject of ongoing research. We note the additional benefit that many of the reversal checks could be employed as execution-time assertions. In particular, the reversal check on the current calculations can provide run-time assurance that the difference between the actual and requested force on the seed are within a specified tolerance.

CONCLUSIONS

Assurance of dependability in complex safety-critical systems is difficult. No one verification technique is sufficient, and in particular, testing has been shown convincingly to be insufficient for demonstrating functional correctness in nontrivial applications. However, in spite of its limitations, testing can play an important role in demonstrating impor-

tant properties of safety-critical systems.

We have shown that exhaustive testing can be a practical approach to proving that a system possesses significant properties. We have introduced the notions of property testing and specification limitation as techniques that can vastly improve the practicality of exhaustive testing.

Test cases identified for the MSS application will enable the rigorous demonstration of important properties of the imaging system and current calculation algorithms. The test cases utilize reversal checks for error detection and rely on exhaustive sets of inputs. Although the complete set of safety properties of the MSS (or any other system) will need to be established by a range of verification techniques, we assert that testing can play an important role in demonstrating a subset of safety properties — properties that might otherwise be very difficult to establish with other verification techniques.

Finally, we note that the proof of a property by exhaustive testing is a proof of the property for the actually machine representation of a program precisely as it will operate in practice. The is a valuable characteristic of the technique.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant number CCR-9213427, and in part by NASA under grant number NAG1-1123-FDP.

REFERENCES

1. Ammann, P. E., S. S. Brilliant, and J. C. Knight, "The Effect of Imperfect Error Detection on Reliability Assessment via Life Testing," *IEEE Transactions on Software Engineering*, Vol. 20-2, February 1994.

2. Brilliant, S. S., J. C. Knight, and P. E. Ammann, "On the Performance of Software Testing Using Multiple Versions," in *Proceedings of Fault-Tolerant Computing: The Twentieth International Symposium*, Newcastle upon Tyne, England, 1990.

3. Butler, R. W. and G. B. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, Vol. 19-1, pp. 3 - 12, January 1993.

4. Goodenough, J. B. and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, SE-1, June 1975

5. Weyuker, E. J., "On Testing Non-Testable Programs," *Computer Journal* Vol. 25-4, November 1982.

6. Wika, K.G., "Safety Kernel Enforcement of Software Safety Policies," Doctoral Dissertation, University of Virginia, May 1995.

7. Wika, K. G., "A User Interface and Control Algorithm for the Video Tumor Fighter," Masters Thesis, University of Virginia, May 1991.

8. Wika, K. G. and J. C. Knight, "A Safety Kernel Architecture," Department of Computer Science, University of Virginia, Technical Report No. CS-94-04, February 1994.