**Title and Subtitle**

Formal Methods for Dependable Distributed Software

**Author(s)**

B. M. McMillin

**Performing Organization Name(s) and Address(es)**

The University of Missouri-Rolla
Rolla, MO 65401

**Perfomrng Organization Report Number**

AFOSR-TR-96

**ABSTRACT (Maximum 200 words)**

This work has developed a powerful concept in evaluating formal specifications concurrently with distributed program execution for the purposes of error detection, fault tolerance, and security. This concept is realized in the CCSP evaluation system for axiomatic proofs, for interval temporal formulae, and for a security calculus. We have validated this concept through nontrivial examples of distributed programs including a dynamic group membership protocol, a distributed database scheduler, of a responsive system modeling railroad trains on intersecting tracks, and of a secure warehouse management system. Temporal subsumption has been developed as a way of making these programs efficient. Moreover, the spinoff technologies from this work, in of themselves have become useful. CCSP can also be used as a debugging tool for distributed programs. Temporal Subsumption functions as a quick and powerful proof checker for Hoare triples. Both of these achievements may help to bring more use of formal methods into the mainstream.

**Supplementary Notes**

unlimited

**Distribution/Availability Statement**

unlimited

**Subject Terms**

Temporal Logic, Subsumption, Responsive Systems, Security, Distributed Systems

**Number of Pages**

13

**Price Code**

UL

**Funding Numbers**

FA9550-92-J-0546
Final Report
Formal Methods for Dependable Distributed Software
F49620-92-J-0546
Bruce McMillin
(ff@cs.umr.edu, Phone: (573)-341-6435, FAX: (573)-341-4501)
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65409

Contents

1 Overview 1

2 Technical Details 2
2.1 The Axiomatic Approach to Program Verification 2
2.2 Temporal Proof System 3
2.3 CCSP System 4
2.4 Assessment 4
2.5 Assertion Generation 4
2.6 Temporal Subsumption 5
2.7 Security 5
2.8 Transitions 6

3 Awards 6

4 Invited Talks 6

5 Students Supervised 7

6 Contribution 7

7 Other Publications Resulting From the Contract 9

1 Overview

Motivation Our original goal in this work was to find ways to execute program specifications along with the actual program’s execution for purposes of run-time assurance - namely for error detection within the scope of fault tolerance. If the execution of the program does not satisfy the specification at run time, then an error has occurred. Since error detection is conceptually the most difficult problem in fault tolerance, this quantification of error detection has proved quite powerful - a
system need not rely on hardware or software confidence to avoid or detect errors; the specification provides the absolute truth of correctness.

Actually doing this is difficult even in the sequential environment as one must ask the question “What is an appropriate level of specification and how does it correspond with the resulting program code?” In the distributed parallel environment with which we are concerned, the challenge becomes greater due to the absence of a globally consistent state in which to evaluate the specification.

Methodology The notion of “the program satisfies the specification” is a powerful abstraction as it immediately draws the researcher into the area of formal logic to express the specification. This, coupled with an existing set of axioms and inference rules for a particular (programming) language provides the appropriate level of representation for run-time error checking. Essentially, the same tools used in program verification are immediately applicable to run-time assurance, namely execution of the proof outline in either a predicate or temporal framework.

Our work provides the run-time semantics to carry out such executions, possibly in the presence of failed hardware and/or software or security intrusions. Nor are we limited to formalized verification systems; our methods work quite well with informally specified assertions. We have developed a set of tools (described below) to carry out these evaluations.

2 Technical Details

2.1 The Axiomatic Approach to Program Verification

The axiomatic approach to program verification is based on making assertions about program variables before, during and after program execution. These assertions characterize properties of program variables and relationships between them at various stages of program execution.

Overall Proof Approach. Distributed programs are composed of a set of communicating sequential processes. In many programs, it is desirable to save part of the communication sequence between processes. This is done with use of “dummy” or auxiliary variables that relate program variables of one process to program variables of another. In general, to prove properties about the program, first properties of each component process are derived in isolation. These properties are combined to obtain the properties of the whole program using “global” auxiliary variables; if the proofs do not interfere, then this composition is valid. We use Hoare’s CSP as a model.

Operational Evaluation of Axiomatic Assertions Taking an application’s proof outline from the verification environment to the distributed operational environment
is not a straightforward task. Since assertions may involve global annotations to the program state, we need some way of communicating this state, efficiently. Observing that no state change can influence a CSP process until some communication occurs (since process states are local), we can simply defer update of global state information until an algorithmic communication occurs. Thus, each communication in CSP is augmented with two functions which prepare copies of a processes' global auxiliary variables for communication and unions these variables into a process' local state, respectively. Since we only need to send the most recent copy (and only a newer copy) variables are time stamped with a Lamport clock. Then, the latest copy of each global auxiliary variable is merged with the local processes' state. These communicated auxiliary variables, in turn, along with the sequential processes' state, are what the assertions are evaluated against.

2.2 Temporal Proof System

General Liveness Properties Using the ISTL* proof system, we constructed operational evaluation semantics of temporal specifications in the distributed environment. This enables evaluation of eventuality assertions [13]. A temporal assertion \( \phi \) is satisfied iff for every state sequence \( \sigma \) of a program, \( \sigma \) satisfies \( \phi \).

Interval Assertions To achieve responsiveness, we created a new logic, ITL [12], similar to other interval logics which enable reasoning within bounded intervals, but also is amenable to embedding operational evaluation semantics of to create a temporal run-time assurance environment for liveness properties. In particular, we consider liveness assertions of the form \( (\phi \rightarrow EF\psi) \) which asserts that execution will progress from a state satisfying assertion \( \phi \) to a state satisfying assertion \( \psi \) where \( \phi \) and \( \psi \) are interval assertions.

Operational Evaluation As in the axiomatic system, above, the state must be communicated and time-stamped in some fashion. Since we reason about certain events in a temporal system, it is enough to collect and merge event sequences or histories during run time.

This requires two steps. First, every processor collects and orders events occurring within itself and within other processors, to form its event history. This history contains a processor's local events and its externally observable events. When a communication occurs, these histories are merged into equivalent histories based on causality. Note that this relies on neither monitors nor global clocks to compute event histories. Moreover, it does not place much additional computational or message-passing burden on the operation of the system.

Second, every processor examines its event history against assertions. Since an event history is a collection of events occurring in a system, it represents a processor's observation of all the processors during execution. This history can be utilized to do
evaluation of assertions at run-time. It is a simple matter, then, to break down the
temporal assertions into predicate calculus expressions on these collected histories.
We developed the temporal logic based translation system described above and
tested its ability to express specifications for the train set responsive system mentioned
under CCSP [12, 14].

2.3 CCSP System

We have built the above concepts into a translation system called CCSP (C-CSP)
which runs on Unix workstations which operationally evaluates axiomatic and tem-
poral assertions embedded within a CSP-like language [4]. This code has been used
within both the classroom and the research group. A user's guide exists, as well [2].

CCSP for the axiomatic proof system has been used in a course at the University
of Missouri in distributed computing (C.Sc. 485). The results of this have been
reported in [1]. CCSP has also been used to validate the proofs of nontrivial examples
of distributed programs including a dynamic group membership protocol [10] and a
distributed database scheduler [5], and a distributed sort [3]. The temporal version
exists as a test version and has been used to validate the proofs of a responsive system
modeling railroad trains on intersecting tracks [12].

The full CCSP source is available from http://www.cs.umr.edu/ecl.html

2.4 Assessment

One question that arises in building a fault-tolerant system is "How fault-tolerant is
it?" In [6], we characterize degrees of fault tolerance of an error detecting algorithm
based on the graphical interconnection topology of the communicating processes in a
distributed system.

2.5 Assertion Generation

One of the complaints about our approach is that it requires a formal understanding
of the program. In [7], we built a system to "fill in" missing assertions in a program's
proof outline to be used in run-time checking. Taking this one step further, we
explored the possibility of reverse engineering a program to determine its inherent
safety and liveness properties by extracting the predicate transformations made by
each program statement on the data domain as a symbolic expression. We then used
Linear Algebraic techniques to determine invariants of the program and then took
the invariants and performed an eigenvalue analysis to determine liveness constraints.
Rather than generating assertions, though, we represented the information visually
and built a tool to visualize program behavior [11].
2.6 Temporal Subsumption

To reduce the complexity of evaluating every assertion that arises from a proof outline, we noticed that, in fault tolerance, only those assertions which are postconditions of communication statements or make statements about auxiliary variables which might have been changed in a communication (indirectly) need to be checked. All other assertions are implied by correct execution of the local, sequential, program code. Since a processor cannot reliably reason about its own faulty or fault-free behavior, all of these other assertions are implied by their preconditions and are redundant and can be subsumed.

Temporal Subsumption, a new technique arising out of our research, is an assertion-based technique. The temporal nature comes from relating logical implications across sequences of predicate transformations described by an axiomatic proof system. Logical assertions that occur early in a program imply assertions later in the program only through predicate transformations which occur in a particular sequence, or order. Thus, unlike classical subsumption, temporal subsumption is defined with respect to the model of predicate transformations.

We have built a temporal subsumption tool using a flow graph generator, the CLP(R) proof checker, and "C programs for analysis [7].

2.7 Security

It is our supposition that formal security policies can also be executed as safety and liveness properties are. This provides an integrated methodology for ensuring that formally-specified properties of safety, liveness, fault tolerance, and security hold, at run time, in a distributed computing system in the presence of faulty hardware/software components and/or active intrusion. The underlying thread is that all violations of specifications are really errors and can be treated using an integrated methodology [8].

Our work provides the run-time semantics to carry out such execution of specifications, possibly in the presence of failed hardware and/or software and/or intrusions. Thus, the approach taken here adopts a formalized specification language together with a mechanized support tool to allow detection of certain types of errors and security breaches as described in [9].

In this application of the concept, we solved several problems.

To control the flow of information, for a particular security policy, requires that semantics and refinement be developed to generating portions of CCSP data communication layers that preserve privacy. This makes CCSP application-dependent for security uses. Additionally, in studying security we had to clip the size of the history information and developed and implemented techniques for doing so.

We also determined the algebraic requirements for a security policy language, which is amenable to encoding as first order expressions and is expressive enough to capture useful security policies.
The system was implemented as part of CCSP and tested on a model problem using an existing security calculus.

2.8 Transitions

The railroad track model problem and temporal run-time evaluation system work has yielded a contract with Harmon Electronics of Grain Valley, MO to build a railroad switching yard layout GUI tool using formal logic equations to express fault-tolerant switching routes.

The understanding of asynchronous systems has yielded joint work with Software Systems Specialists, Inc. of St. Louis, MO, and the ARMY under an STTR in performing real-time animation of manufacturing processes.

3 Awards

1994 Phillips Petroleum Foundation Faculty Excellence Award, University of Missouri-Rolla ($2500)

1993 IEEE Computer Society Certificate of Appreciation for contributions to the 1993 ICDCS

1992 Phillips Petroleum Foundation Faculty Excellence Award, University of Missouri-Rolla ($2500)

4 Invited Talks


1994 “Parallel Computing for Engineering Problems,” UMR Department of Civil Engineering.


1992 “Error-Detecting Concurrent Software through Changeling” University of Illinois-Chicago.
5 Students Supervised

The following student theses were completed during the time period of the award. Complete copies may be obtained by sending e-mail to csdept@cs.umr.edu enclosing a return mailing address in the text of the message and the title and name of student of the requested document.

<table>
<thead>
<tr>
<th>Name</th>
<th>Support</th>
<th>Title</th>
<th>Degree</th>
<th>Grad. D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pei-Yu Li</td>
<td></td>
<td>Fault-Tolerant Distributed Deadlock Detection</td>
<td>Ph.D.</td>
<td>1994</td>
</tr>
<tr>
<td>Jun-Lin Liu</td>
<td></td>
<td>Recoverable Ring Embeddings in Hypercubes</td>
<td>Ph.D.</td>
<td>1993</td>
</tr>
<tr>
<td>Su-Mei Tsai</td>
<td>AFOSR</td>
<td>Fault-Tolerant Distributed Real-Time Systems</td>
<td>Ph.D.</td>
<td>1994</td>
</tr>
<tr>
<td>Alan Su</td>
<td>NSF</td>
<td>A Deterministic Membership Algorithm in Asynchronous Distributed Systems</td>
<td>M.S.</td>
<td>1994</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Support</th>
<th>Area</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred Budd</td>
<td>UMR</td>
<td>Distributed Computer Security</td>
<td>M.S.</td>
</tr>
<tr>
<td>Jui-Lin Lu</td>
<td>UMR</td>
<td>Computational Mathematics (co-supervised)</td>
<td>Ph.D.</td>
</tr>
<tr>
<td>Larry Reeves</td>
<td>UMR</td>
<td>Parallel Implicit Methods</td>
<td>Ph.D.</td>
</tr>
<tr>
<td>Cristina Serban</td>
<td>AFOSR/UMR</td>
<td>Distributed Computer Security</td>
<td>Ph.D.</td>
</tr>
<tr>
<td>Aggie Sun</td>
<td>NSF/AFOSR</td>
<td>Declarative Approach to Generalizing the Understanding of Program Behavior Through Program Visualization</td>
<td>Ph.D.</td>
</tr>
</tbody>
</table>
6 Contribution

We feel we have developed a powerful concept in evaluating formal specifications concurrently with distributed program execution. Moreover, the spinoff technologies from this work, in of themselves have become useful. CCSP can also be used as a debugging tool for distributed programs. Temporal Subsumption functions as a quick and powerful proof checker for Hoare triples. Both of these achievements may help to bring more use of formal methods into the mainstream.

References


7 Other Publications Resulting From the Contract


Copies of the above technical reports may be obtained from http://www.cs.umr.edu/techreport