CONF-970337--1
SAND96-2351C

Ensuring Critical Event Sequences in High Consequence Computer Based Systems as Inspired by Path Expressions

Marie-Elena C. Kidd, Sandia National Laboratories[1]

RECEIVED
OCT 03 1996
OSTI

Abstract

The goal of our work is to provide a high level of confidence that critical software driven event sequences are maintained in the face of hardware failures, malevolent attacks and harsh or unstable operating environments. This will be accomplished by providing dynamic fault management measures directly to the software developer and to their varied development environments. The methodology employed here is inspired by previous work in path expressions. This paper discusses the perceived problems, a brief overview of path expressions, the proposed methods, and a discussion of the differences between the proposed methods and traditional path expression usage and implementation.

Introduction

Currently, our work focuses on dynamic (run-time) fault detection to ensure critical software driven event sequences in single processor environments. If these methods prove valuable and practical, they will be extended to distributed environments and fault management. We are in the early phases of applying our initial methods to real world projects which are predominantly in the embedded systems area. The initial methods are manually embedded in software models and code. Later work will concentrate on adding the extensions to the software development environments through compilers, assemblers, and modeling tools. It is important to note that since high consequence software is often embedded software, the compilers are often cross-compilers from a high level programming language like C to a target processor assembly language like 8051 or 68020. Also, assembly language is, at times, the only programming language used. Thus, our methods must be general enough to work in these varied environments.

Perceived challenges and problems

A major concern when developing high consequence software is ensuring the integrity of critical event sequences. The system must be able to execute correctly, safely, and reliably even in the face of faulty hardware or software, external malevolent forces, and environmental stimuli such as lightning strikes or static. These forces could, and have been proven to, perturb the normal software flow of execution. If, for example, the program counter gets corrupted, the software flow is moved to an unintended software point. The software should not continue executing through the code from the failure point, but instead should realize that the expected flow was corrupted and assume a fail safe mode.

Figure 1 provides an example of how the sequence of events is important. This is the sequence of events involved in making a bowl of instant soup. First you heat the water. When the water boils, you mix it with the soup packet. Then, you must wait for the soup to reconstitute and to cool to a temperature that is safe for consumption. There is a minor safety problem if the cooling stage is skipped. If, for example, you got distracted at just the right moment, the result might be that you skip the cooling stage and burn your tongue and throat by drinking boiling liquid.

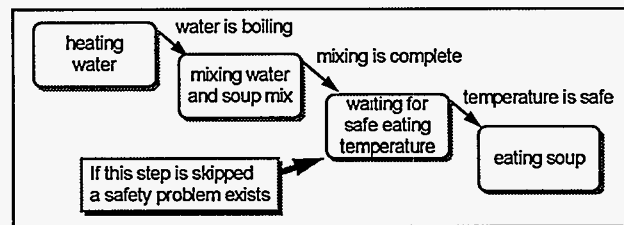

Figure 1 - Example of a Simple Event Sequence

The analogy can be extended to a high consequence

[1] The work presented in this paper is part of the High Integrity Software (HIS) Project which is supported by the Strategic Surety Backbone of the Defense Programs Sector at Sandia National Laboratories. Although our funding and initial focus stems from defense applications, our methods will be applicable to the general high integrity software developer.

This work was supported by the United States Department of Energy under contract DE-AC04-94AL85000.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

MASTER

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.

computer-based system having a problem such as being hit by lightning, zapped by static, or a hardware malfunction which leads to a critical event being skipped. In that case, rather than a burned tongue, the resulting safety problem may involve taking the lives of many innocent people.

Currently, no formalized methods exist to handle the problem of ensuring critical event sequences. Therefore, many ad-hoc and creative methods are employed which result in the injection of more software bugs, creating hard to maintain software, highly clever yet unrepeatable software, or increased complexity.

A recurring informal method has been used in the past. It consists of creating a variable that holds information describing what events have occurred at any point in the execution of a software program. Some schemes simply assign a numeric value to each critical output event and add that value to the variable at runtime. In another method, a byte (or word) is bit encoded so that each bit represents the state of an event, 0 for not done and 1 for done. Usually, the variable's value is derived in real-time during execution through clever logical or mathematical equations, but sometimes it is simply assigned to the variable. This is a creative and manual process done by the software developer and embedded in the code. The methods for matching an event with a value or figuring out which bits to attach to an event are mainly cleverness and trial & error. The author was part of one such effort.

Clearly, a need exists for more reliable, repeatable, and easily employed methods for ensuring critical software driven event sequences in harsh and unstable environments. This work takes the informal, ad-hoc "methods" into consideration and applies existing computer science theory to create a more formal and reliable method for ensuring software event sequences.

Introduction to supporting computer science theory

In order to understand path expressions and our methods, it is first necessary to understand its theoretical basis. Therefore, a brief review on finite automata and regular expressions will be addressed before discussing path expressions.

Finite Automata basics

The review information in this section is derived from [7].

A Finite Automaton (FA) is defined as a quintuple involving states and input values.

FA = $(Q, \Sigma, \delta, q_0, F)$.

- **Q** is the finite set of states.
- Σ is the finite input alphabet.
- δ is the transition function mapping **Q x** Σ to **Q** such that the signature of the transition function is $\delta: Q \times \Sigma \rightarrow Q$. Using function notation, this is $\delta(q_i,a) = q_j$. This means, when in state q_i, which is an element of **Q**, with input **a**, which is an element of Σ, the resulting state, q_j, is given by the transition function, δ. Another way to describe this is that the transition function takes each possible state and input pair and defines the resulting state.
- q_0 is the start state (also known as the initial state). And, $q_0 \in Q$, which means q_0 is an element of the set of states, **Q**.
- F is the finite set of final states. And, $F \subseteq Q$, which means the final states, **F**, are a subset of the set of states, **Q**.

Two standard representations for finite automatons are transition diagrams represented as directed graphs and transition tables. Figure 2 displays a finite automaton in the form of a transition diagram represented as a directed graph. Notice that the circles represent states and the arrows represent elements of the input alphabet. Final states are often marked with a double circle.

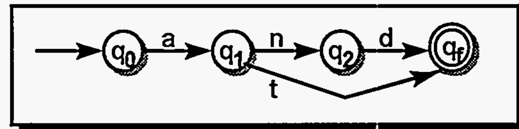

Figure 2 - Example Of A Finite Automaton

Table 1 is the transition table associated with the transition diagram in Figure 2. Notice that this example allows only "*and*" and "*at*" as acceptable input strings. This means that the "language", or set of strings, accepted by this finite automaton consists of "*and*" and "*at*" and nothing else. A string is accepted only when the finite automaton finishes in a final state.

Table 1 - Example Of A Transition Table

states in Q	inputs in Σ			
	a	n	d	t
q_0	q_1			
q_1		q_2		q_f
q_2			q_f	
q_f				

One could visualize the input to a finite automaton as an input stream, perhaps written on a tape that arrives and is read by a reading head. As the input stream is read one character at a time, the transition diagram or table executes based on the input symbols. This is pictured in the sequence in Figure 3. The "execution" of one path through the finite automaton is simulated by highlighting the active state.

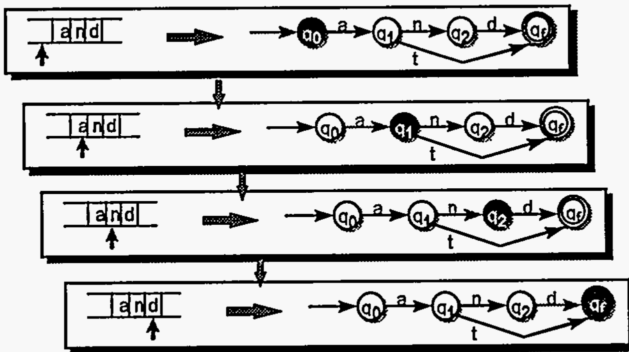

Figure 3 - An Execution Path Through A Finite Automaton

We have provided only a very basic review of finite automata. Indeed, there are more complex and advanced areas within automata theory. But, they are not necessary for our discussion.

Regular Expression basics

The review information in this section is derived from [7]. Regular expressions are simple expressions describing languages that are accepted by an associated finite automaton. For example, the previous section gave a finite automaton that accepts the set of input strings of the form ***'a' followed by 'nd' or 'a' followed by 't'***. This is a long winded way of describing a very simple expression. Regular expressions give us a simple and compact way to describe such expressions. Table 2 gives the basic syntax of regular expressions. A and B are sets of input symbols.

Table 2 - Regular Expression Syntax

Syntax	Meaning
A B	This is sequence or concatenation. It means A followed by B.
A + B	This is selection. It means A or B, but not both.
A*	This is called Kleene Star or Kleene closure. It means 0 or more occurrences of A which is repeated concatenation.
A^+	This is called positive closure. It means 1 or more occurrences of A. It is just like Kleene closure except that the minimum number of occurrences is one.
A^0	This is the empty string.

In general, capital letters represent sets of strings and lower case letters represent set elements (strings). Here are some examples using regular expressions. Regular expressions may appear in terms of sets (capital letters) or elements (lower case letters). The "=" below means "denotes the set".

Given A = {a} and B = {x, y, z}

- AB = {ax, ay, az}
- xy = {xy}
- A^* = {ε, a, aa, aaa,...}
- A^+ = {a, aa, aaa,...}
- $A + B$ = {a, x, y, z}
- $x + y$ = {x, y}

Perhaps a more meaningful example would be to let A = {b, c} and B = {all, oat, at}.

- AB = {ball, boat, bat, call, coat, cat}
- $A + B$ = {b, c, all, oat, at}

Here is an example of the sequence notation. Given that a specific person is 60 years old, the life sequence they went through was birth then infancy then childhood and then adulthood. This could be described by the following notation ***birth infancy childhood adulthood***. If we let b represent birth, i represent infancy, c represent childhood, and a

represent adulthood then we can compress the notation above to ***b i c a***.

Here is an example of the selection notation. Common house pets are dogs, cats, reptiles, and fish. Given one common house pet, that pet is either a dog, a cat, a reptile, or a fish. A notation is ***dog + cat + reptile + fish***. If we let d represent dog, c represent cat, r represent reptile, and f represent fish then we can again compress the notation above to $d + c + r + f$. Unless my understanding of animal classification is mistaken, this is true selection since a given pet can be exactly one of these types of animals with the odd cases of multiple inheritance like the duck-billed platypus aside.

Here is an example of the Kleene Star notation. Entering the world of "make believe", assume we have an infinite length freeway and an infinite number of automobiles. Each automobile has an associated driver. This freeway can hold zero automobiles, or one automobile, or two automobiles,... or an infinite number of automobiles traveling at once. Now, if we let A represent the set of all automobiles that can be on the freeway, we can represent the freeway activity as A^*.

Here is an example of the repetition of 1 or more notation. We must remain in the world of "make believe" for this example. Given a functioning and infinitely large Emergency Room in a typical hospital, there should always be at least one physician on duty. So, there will be one physician, or two physicians, ... or an infinite number of physicians on duty at a given time. If we let A denote the set of possible physicians, we can represent this example as A^+.

Again, we have only reviewed enough of regular expression theory to allow us to talk about path expressions. In compiler theory, regular expressions are expanded to cover very complex expressions and languages.

Path expression basics

A look at path expressions

Figure 4 is a look at a basic path expression represented as a finite automaton via a directed graph. This path set is interpreted as ***"a is followed by either b then d or a is followed by c followed by zero or more repetitions of g followed by e. Then, f comes last."*** This is long winded and somewhat confusing, not to mention open to different interpretations. So, we will use a regular expression to specify all acceptable paths through this directed graph. Interpreting the finite automaton in Figure 4 produces a regular expression, $a(bd + c(g^*)e) f$, which is an algebraic representation of the path. This is also called a path expression since it expresses paths through the graph. In this particular case, it represents all paths through the given graph. Path expressions give us a more concise way to express the acceptable sequences just as regular expressions did in the earlier section. This example in Figure 4 depicts one of the many graphical models and notations found in the literature.

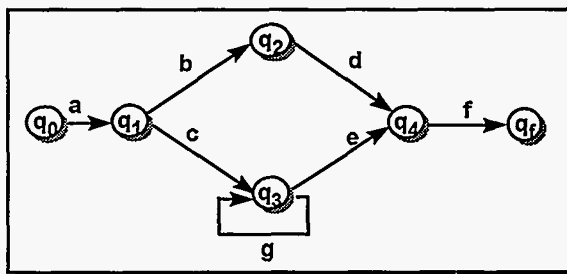

Figure 4 - Example Graphical Representation Of A Path Expression

Path expressions are basically extended regular expressions[2] that denote a specified set of paths through a graph where the graph depicts a model of flow through software code units. The uses of path expressions in the literature vary and will be discussed later in this paper. The notations found in the literature vary greatly sometimes with good reason. For simplicity and consistency, we will continue to use regular expression notation throughout this paper.

Current related path expression usage by application area

The literature on path expressions introduces many variations of path expressions. For example, regular path expressions were the first non-shuffle operator path expressions based on regular expressions and were used to describe synchronization relationships among processes sharing resources. Open path expressions were created to allow inherent unrestricted concurrency. Predicate path expressions extend regular path expressions to allow for a level of granularity beyond the process/module level and

[2] Not all of the path expression derivatives are based on regular expressions. However, for the context of this paper, we are interested in the regular expression based uses.

to add predicates to the decision process before performing an action. Generalized path expressions grew out of predicate path expressions and are mainly used in the verification and validation area. This list goes on.

However, for our purposes, the different ways in which path expressions are used is more important than the many specific versions of path expressions. Therefore, the term "path expression" in this paper refers to the general class of path expressions except when a specific version is listed. We focus on the concurrent systems and verification & validation areas because their uses are somewhat similar to our own.

Partial Chronology of Path Expression

Figure 5 shows a partial overview of the chronology of Path Expression and related methodologies. This figure is included for a historical perspective and was derived from [3, 6, 8, and 9].

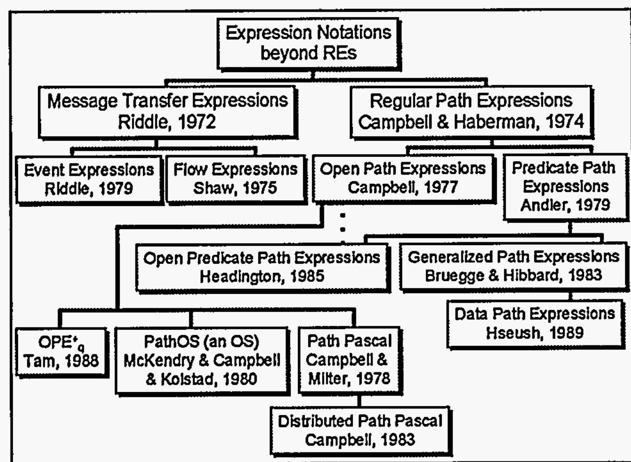

Figure 5 - Partial Chronology of Path Expressions

Concurrent systems usage

Path expressions were originally introduced by R. Campbell and A. Haberman in 1974 to describe synchronization relationships and rules. Path expressions are initially based on regular expressions. [4, 5]

Traditional usage in the concurrent area, whether used on distributed processes or not, is based on synchronizing concurrent access to shared resources like data. Resource allocation is the main objective. Furthermore, from the literature, it is clear that most traditional uses do not consider harsh environments that could throw the software execution sequence "out of whack". Figure 6 depicts the general usage scenario.

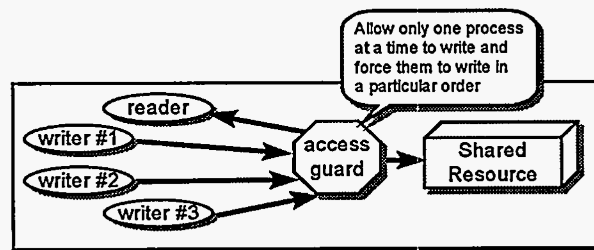

Figure 6 - Concurrent Systems Path Expression Usage Scenario

In this area, path expressions are derived during the analysis and design phases. They are then implemented, usually with semaphores or object oriented implementation constructs. Path Pascal and PPE ALGOL 68 [1] are programming languages that have been extended to include path expressions.

Verification & Validation (V&V) usage

In Verification & Validation, path expressions have been used to optimize test case coverage and create external monitors.

Path expressions are used to select software test paths. The paths are derived from control flowgraphs of the software. Flowgraphs can be used at various levels of granularity and are based on the actual execution time flow of control through the software. A procedure for the conversion of a flowgraph into a path expressions is given in the literature. Methods exist for determining the longest path, shortest path, and other specific paths through the software. [2]

Another application of path expressions in the Validation & Verification area focuses on picking actual software paths and verifying that those paths occurred during execution as expected. Some methods actually implement an external path recognizer for this purpose. These methods are employed on single processor as well as distributed systems. As one can imagine, an external recognizer could become quite complex when watching the output from many parallel processors in a system. Figure 7 shows this scenario.

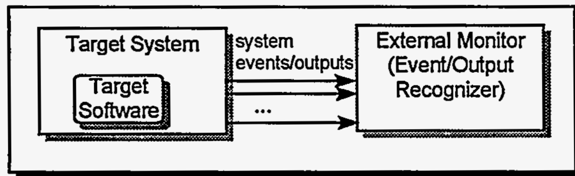

Figure 7 - Verification & Validation Path Expression Usage Scenario

Our proposed methodology which was inspired by path expressions

Basic goals

The goals are: ensure critical event sequences in unstable and harsh operating environments; ensure critical event sequences with adjustable granularity; and provide software fault management where the faults could come from the hardware, software, or the operating environment

Critical event sequence fault management method implemented by the developer

The critical event sequence fault detection method implemented by the developer consists of deriving regular expressions from a software model or the software requirements and then embedding (1) check points and (2) update points based on those regular expressions into the target code along with a (3) module (or object) that implements the underlying finite automaton. This extra software is added to the target code to verify that the correct event sequence is maintained. The granularity of the regular expression is flexible and should be determined by the software requirements. Examples of appropriate software models are data flow diagrams, state-transition diagrams, and flowgraphs. All of these models chart out a type of software flow. It is the flow that regular expressions will be used to enforce whether protecting an actual software path or a software sequence.

Basically, a module "in the background" tracks the critical event sequence by calls made from the epilogues and prologues of each critical event. This serves to localize or encapsulate the functionality in one module. That module will determine when and how to fail safe if necessary as determined by fault detection. This one function also maintains the history of execution at the critical event sequence level of granularity. It is therefore easy to access this history after execution if non-volatile memory is used (and the memory was not destroyed during execution).

During the current phase of this work, the focus is on fault detection in the single processor environment. Later phases will deal with more complicated fault management issues and distributed environments.

This method lends itself to easy additions of fault management at the critical event sequence level because the last known good event is known. It is also easy to add time bounds or loop iteration bounds to the finite automaton nodes.

Critical event sequence fault management method in the development environment

In the future, the critical event sequence fault management method may be embedded in the software development environment by placing it in extensions to compilers, assemblers, or other development tools. In this case, the software developer does not have to do anything extra because the compiler or other development tools do the work. However, the regular expression still must be created to express the critical event sequence.

The two areas of interest are generic extensions to any language and language-specific extensions. In the language-specific area, languages like Path Pascal already exist. Extensions to Ada have also been made. However, these are for specific compilers and have different intents. The problem for embedded software is that other languages are used such as C or Assembly language. In these cases, the microprocessor used will dictate a subset of compilers, cross-compilers, or assemblers. Many compiler/assembler options exist and to add to the variability, commercial compiler/assembler companies constantly change their products and at times go out of business. For these reasons, a generic set of extensions would be a better method due to the variability and dynamic nature of the market.

Using Event sequence based regular expressions vs. path based regular expressions

Our work is more concerned with critical software driven event sequences than with the actual paths chosen between the events. Figure 8 shows an expansion of the basic path based regular expression diagram into a path expression application. The nodes are now pieces of code which could be code fragments, objects, or entire modules. The inverted triangle is a check point which could be thought of

as a yield point. The large arrow is an update point which occurs after the critical output and will update the state appropriately. This method tracks the path that is taken to get to the events.

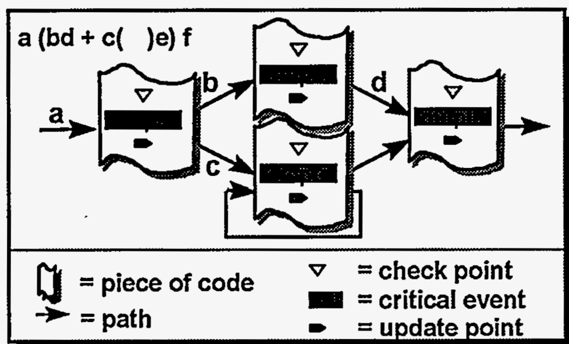

Figure 8 - Our Usage Of Path Based Regular Expressions

Another way of using path expressions is to use them as "event sequence expressions" where the event sequence is tracked rather than the path between the events. In Figure 9, the "event sequence expression" depicted is $a(b+c^{+})d$.

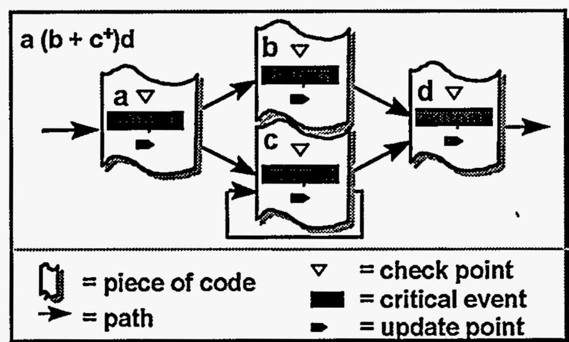

Figure 9 - Our Event Sequence Expression Scenario

Both path based regular expressions and "event sequence expressions" use regular expressions and the underlying finite automaton as a foundation. The use of one or the other should be driven by what is appropriate for the software requirements. If the path is important, use path based regular expressions. If the event sequence is important, use "event sequence expressions." These two methods are ways to derive the regular expression that will be tracked and implemented in the target code.

Identification of path expression usage in the Software Engineering life cycle

Consider some of the very basic software engineering life cycle phases: requirements, design, and implementation. During the Requirements phase, regular expressions will be derived from the analysis diagrams and safety or reliability requirements. During the Design phase, regular expressions will be embedded into the design diagrams. Finally, during the implementation phase, regular expressions will be embedded in the code as directed by the design. Figure 10 shows our usage scenario.

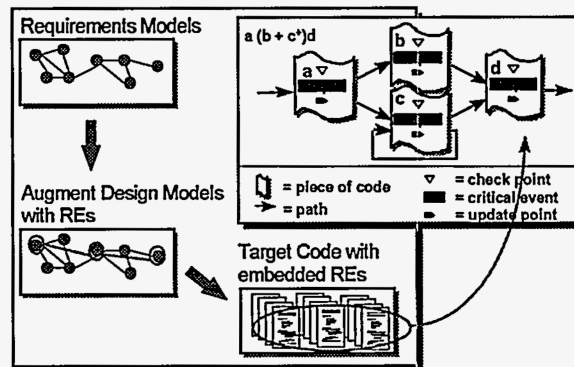

Figure 10 - Our Event Sequence Expression Usage Scenario

The level of granularity of the event sequence is flexible. It should be the level that is appropriate to the surety requirement. This can be at the module level in some areas, above the module level in other areas, and even close to the line by line level in others. The similarity is that all monitoring mechanisms based on the regular expressions are internal to the code.

Differences from traditional path expression work

To help understand the different usage scenarios used by the concurrency area and our area, the following anthropomorphic questions may help. The basic question that is asked in a traditional concurrent path expression usage is, "May I have the shared resource now?" The answer is either, "yes, continue" or, "no, wait until it is your turn." In our usage, the basic question is, "Am I supposed to be here now based on order of events?" The answer is either "yes, continue" or, "no, fail safe or correct the execution flow based on the last known good state."

Conclusions

A major concern when developing high consequence software is ensuring the integrity of critical event sequences. The system must be able to execute correctly, safely, and reliably even in the face of

faulty hardware or software, external malevolent forces, and environmental stimuli such as lightning strikes or static. If, for example, the program counter gets corrupted which results in the software flow being moved to an unintended software point, the software should not continue executing through the code from the failure point, but should instead realize that the expected flow was corrupted and assume a fail safe mode.

Currently, no formalized methods exist to handle this problem. So, many ad-hoc methods are employed. The possible results are infection of more bugs into the software, sometimes hard to maintain software, and increased complexity.

Path expressions in software have been used to protect shared resources, optimize data base queries, for test case coverage optimization, and to create external test monitors. This work will extend the use to cover critical event sequence concerns in high consequence software. This is a unique extension set according to the literature and appears to be a reasonable and logical direction.

Because our method is based on deriving a regular expression of the critical event sequence, adding check points and update points, and adding a module that implements the functionality of the underlying finite automaton and its tracking, this method is repeatable and easy to maintain because of the inherent encapsulation. Because the method is based on mathematical foundations, it is inherently more reliable than the ad-hoc "on the fly" cleverness.

Upon completion of this work, the deliverable will be dynamic fault management (both fault detection and fault correction) methods through path expression extension inspired methods for ensuring critical event sequences in high consequence software. These will be in the form of user embedded and compiler embedded methods. These methods will also work in distributed, multiprocessor environments.

References

1. Sten Andler, *Predicate Path Expressions: A High-Level Synchronization Mechanism,* Ph.D. thesis, Computer Science Department, Carnagie Mellon University, 1979.

2. Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 1990, Chapters 3 and 8.

3. Bernd Bruegge and Peter Hibbard, "Generalized Path Expressions: A High-Level Debugging Mechanism," *Journal of Systems and Software*, Vol. 3, 1983, pp. 256-276.

4. Roy H. Campbell, A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Proceedings of an International Symposium on Operating Systems*, Rocquecourt, France, April 1974, *Lecture Notes in Computer Science, Springer Verlag*, Vol. 16, pp. 89-102.

5. Roy H. Campbell, *PATH EXPRESSIONS: A technique for specifying process synchronization*, Ph.D. thesis, Computing Laboratory, The University of Newcastle Upon Tyne, Newcastle Upon Tyne, England, August 1976. Reprinted by the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, May 1977.

6. Mark R. Headington and Arthur E. Oldehoeft, "Open Predicate Path Expressions and their Implementation in Highly Parallel Computing Environments," *Proceedings of the 1985 International Conference on Parallel Processing,* IEEE Computer Society Press, Washington, DC, 1985, pp. 239-46.

7. John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass. 1979.

8. C. Samuel Hsieh, "Timing Analysis of Cyclic Concurrent Programs," *11th International Conference on Software Engineering*, ACM, 1989, pp. 312-318.

9. Alan C. Shaw, "Software Specification Languages Based on Regular Expressions," *Proceedings of a Workshop on Software Development Tools, Lecture Notes in Computer Science, Springer-Verlag,* Berlin, West Germany, 1980, pp. 148-75.

Marie-Elena C. Kidd is a computer scientist and Senior Member of the Technical Staff at Sandia National Laboratories. During her ten years at Sandia, she has worked as a software engineer on embedded, real-time software systems for such applications as robotics, nuclear weapon components, and control systems. She has also worked on lab-wide information sharing software systems and software engineering initiatives. She has a B.S. in Computing and Information Sciences from Trinity University in San Antonio, Texas and an M.S. in Computer Science from Purdue University in West Lafayette, Indiana.