Ensuring Critical Event Sequences in High Integrity Software
by Applying Path Expressions

Marie-Elena C. Kidd*; Sandia National Laboratories; Albuquerque, NM

Abstract

The goal of this work is to extend the use of existing path expression theory and methodologies to ensure that critical software event sequences are maintained even in the face of 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. This paper discusses the perceived problems, a brief overview of path expressions, and our proposed extension areas. We will discuss how the traditional path expression usage and implementation differs from our intended usage and implementation.

Introduction

The path expressions work presented in this paper is part of the Systems Immunology™ Track of the High Integrity Software (HIS) Project. The High Integrity Software project is part of 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.

Initially, our work will focus on path expression extensions in single processor environments and for fault detection. If our methods prove valuable, we will extend them to distributed environments and fault correction. We are currently in the early phases of applying our initial methods to real world software projects. Another initial interest is methods that the user manually embeds in their software models and code. We will later concentrate on adding the extensions to the software development environment through compilers, assemblers, and modeling tools. It is important to point out that since high integrity 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. So, our methods must be general enough to work in these varied environments.

* This work was performed at Sandia National Laboratories, which is operated by Lockheed-Martin Corporation for the U.S. Department of Energy under contract DE-AC04-94-AL85000.
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. If, for example, the program counter gets corrupted, the software should not "music box" through the code from the failure point. Currently, no formalized methods exist to handle this problem. As a result, many ad-hoc methods are employed. The results are often injection of more bugs into the software, sometimes hard to maintain software, and increased complexity.

Within Sandia National Laboratories, a recurring informal method has been used. It consists of creating a set of variables 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. Usually, the numeric value is derived in real-time during execution, 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 and easily employed methods for ensuring critical software event sequences in harsh and unstable environments.

Figure 1 provides an example where the sequence of events is important. This is the sequence of events involved in making a plain cup of instant coffee. First you heat the water. When the water boils, you mix in the coffee. Then, you must wait for the beverage to cool to a temperature that is safe for consumption. There is a minor safety problem if the cooling stage is skipped. If you got distracted at just the right moment, the result might be that you burn your tongue.

![Diagram of event sequence](image-url)

**Figure 1 - Example Of An Event Sequence**
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, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or 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.
The analogy can be extended to a high consequence computer-based system having a problem such as being hit by lightning or the hardware malfunctioning which leads to a critical event being skipped. In that case rather than a burned tongue, the resulting safety problem may be along the lines of many innocent people dying.

Our work will take the informal, ad-hoc “methods” and apply existing computer science theory to create a more formal and reliable method for ensuring software event sequences. A method to attach timing to the events will also be explored. The mathematical and logical formula research may also be applicable to output signal integrity.

Introduction to supporting computer science theory

In order to understand path expressions, it is first necessary to understand its theoretical basis. Therefore, a brief refresher 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 Ref.5.

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.
- \(\Sigma\) is the finite input alphabet.
- \(\delta\) is the transition function mapping \(Q \times \Sigma\) 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 \(\Sigma\), the resulting state, \(q_j\), is given by the transition function, \(\delta\). 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 automata 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.
Table 1 is the transition table associated with the transition diagram. Notice 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

<table>
<thead>
<tr>
<th>states in Q</th>
<th>inputs in Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>q₀</td>
<td>q₁</td>
</tr>
<tr>
<td>q₁</td>
<td>Ø</td>
</tr>
<tr>
<td>q₂</td>
<td>Ø</td>
</tr>
<tr>
<td>qₚ</td>
<td>Ø</td>
</tr>
</tbody>
</table>

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.
We have reviewed only the very basic area of finite automata. Indeed, there are more complex and advanced areas within automata theory. But, they are not necessary for our discussion of path expressions.

Regular Expression basics
The review information in this section is derived from Ref.5. 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 to describe 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>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
<th>Also denoted as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>This is concatenation. It means A followed by B.</td>
<td>{x y</td>
</tr>
<tr>
<td>A + B</td>
<td>This is selection. It means A or B, but not both.</td>
<td>{x</td>
</tr>
<tr>
<td>A*</td>
<td>This is called Kleene Star or Kleene closure. It means 0 or more occurrences of A which is repeated concatenation.</td>
<td>A* = \bigcup_{i=0}^{\infty} A^i</td>
</tr>
<tr>
<td>A+</td>
<td>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.</td>
<td>A+ = \bigcup_{i=1}^{\infty} A^i</td>
</tr>
<tr>
<td>A0</td>
<td>This is the empty string.</td>
<td>{\varepsilon}</td>
</tr>
</tbody>
</table>
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 is meant to mean “denotes the set”.

Given \( A = \{a\} \) and \( B = \{x, y, z\} \)
- \( AB = \{ax, ay, az\} \)
- \( xy = \{xy\} \)
- \( A^* = \{\epsilon, a, aa, aaa, ...\} \)
- \( A^+ = \{a, aa, aaa, ...\} \)
- \( A + B = \{a, x, y, z\} \)
- \( x + y = \{x, y\} \)

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 an average person is 60 years old, the life sequence they went through is birth then infancy then childhood and then adulthood. This could be described by the following notation: birth ; infancy ; childhood ; adulthood. An alternate notation is 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 notations above to \( b ; i ; c ; a \) and \( b i c a \).

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. An alternative 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 notations above to \( d + c + r + f \) and \( d / c / r / f \). 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 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 at all
times. 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 $A^\mathbb{N}$.

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
The following figure is a look at a basic path expression represented as a finite automaton via a directed graph. Interpreting the finite automaton produces the algebraic representation of the path, $a(bd + c(g^*e))f$. This algebraic expression is a regular expression specifying all acceptable paths through the directed graph. This is also called a path expression since it expresses paths through the 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." Path expressions give us a more compact 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](image)

Path expressions are basically extended regular expressions 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 and some 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 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 expressions 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 ours.

**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. (Refs.3,4)

Traditional usage in the concurrent area, whether used on distributed processes or not, is based on synchronizing concurrent access to shared data. Resource allocation is the main objective. Furthermore, from the literature, it is clear that most traditional uses do not care about harsh environments that could throw the execution sequence "out of whack". Figure 5 depicts the general usage scenario.

![Figure 5 - Concurrent Systems Path Expression Usage Scenario](image)

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 (Ref.1) are programming languages that have been extended to include path expressions.
Verification & Validation (V&V) usage
The software testing realm uses path expressions to optimize test case coverage and for creating 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. He has devised ways to determine the longest path, shortest path, and other specific paths through the software. (Ref.2)

Another usage 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. Figure 6 shows this scenario.

![Figure 6 - Verification & Validation Path Expression Usage Scenario](image)

Our proposed usage of path expressions
We are focusing on three main deployment methods for path expressions to ensure critical event sequences.

Path expression methods implemented by the developer
Path expression methods implemented by the developer consist of deriving path expressions from a software model and then embedding check points and update points based on those path expressions into the target code. Extra software is added to the target code to verify that the correct event sequence is maintained. The granularity of the path 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 path expressions will be used to enforce whether we are protecting an actual software path or a software sequence.
During the initial phase of our work, the focus will be on fault detection in the single processor environment. Later phases will deal with more complicated fault management issues and distributed processes.

Path expression methods in the development environment
Path expression methods may be embedded in the software development environment by placing them in 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.

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 with quite 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. We believe a generic set of extensions would be a superior method due to the variability and dynamic nature of the market.

Hybrid of Hardware Systems Immunology™ with the above
The Digital Isolation & Incompatibility project, which is also part of the Systems Immunology™ track of the High Integrity Software project, is working on hardware solutions that are complementary to this work. They will provide hardware solutions that check path expression variables. Specific path expression values will enable hardware state machines that can check activity at the line-by-line of code level if desired. The hardware would then enable or leave disabled a specific hardware output based on the state machine.

This merger will handle situations where a software interlock or a hardware interlock alone is not enough protection to meet system surety requirements. One example of a threat requiring both methods is as follows. A system has software embedded in a microprocessor and at least one critical output. The operating environment has hazards that may corrupt the program counter in the microprocessor. Given that the microprocessor instructions vary from one to three bytes in length, if the program counter is corrupted it could “wake up” on the third byte of an instruction instead of the first byte.

Our uniqueness in the path expression area

Our basic goals
We seek to ensure critical sequence of events in unstable and harsh operating environments. Our usage of path expressions has two related, main goals. First, ensure critical event sequences with adjustable granularity. Second, provide software fault
tolerance where the faults could come from the hardware, software, or the operating environment.

It may be possible to use the path expression derivation techniques from the V&V area with a flexible level of granularity (e.g. module level, object level, near line by line level) and to capture event sequence rather than path sequences.

The implementation techniques, however, will be different from the current implementation techniques in both areas. The implementation will consist of embedding check points and update points into the target system code.

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 of path expressions, 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."

**Event sequence expressions vs. path expressions**

Our environment is more concerned with critical software event sequences than with the actual paths chosen between the events. Figure 7 shows an expansion of the basic path 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 path variable appropriately. This method tracks the path that is taken to get to the events.

![Path expression diagram](image)

**Figure 7 - Our Usage Of Path 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 8, the "event sequence expression" depicted is \( a(b+c^*)d \).
Both path expressions and “event sequence expressions” use regular expressions 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 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.

Path formulas
Mathematical and logical formulas will be used to check and update path variables. Some guidelines for formula use are needed. Consideration for items such as the following will be considered: placement of check points and update points for path variables, reduction rules and state minimization, recursion, the arithmetic bounds of the processor, and synchronization issues.

Identification of path expression usage in the Software Engineering Lifecycle
Consider the very basic software engineering lifecycle phases: requirements, design, and implementation. During the Requirements phase, path expressions will be derived from the analysis diagrams. During the Design phase, path expressions will be embedded into the design diagrams. Finally, during the implementation phase, path expressions will be embedded in the code as directed by the design. Figure 9 shows our 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 with path expressions is internal to the code.

**Conclusions**

A major concern when developing high consequence software is ensuring critical event sequence integrity. 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. If, for example, the program counter gets corrupted, the software should not “music box” through the code from the failure point.

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 database 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.
Upon completion of this work, the deliverable will be dynamic fault management methods through path expression extensions 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


Biography

Marie-Elena C. Kidd
Sandia National Laboratories
Albuquerque, NM 87185-0535 USA

Marie-Elena 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 and an M.S. in Computer Science from Purdue University.