ANALYSIS AND PREDICTION OF SET-TOP-BOX RELIABILITY IN MULTI-APPLICATION ENVIRONMENTS USING ARTIFICIAL INTELLIGENCE TECHNIQUES

Louis P. Slothouber
BIAP Systems, Inc.

Abstract

We present an Artificial Intelligence based method for improving the reliability of software applications, especially in digital cable TV set-top-box and other embedded environments. Initially a small finite state model of the software system and all relevant applications is constructed to define all user input events and application states of interest. A small set of expert system rules is then defined that analyzes state transitions in testing data. When these rules are applied to actual testing data a quantitative measure of suspicion is assigned to all event transitions in the original finite state model. Analysis of this annotated model can then uncover the source of otherwise intermittent inter-application failures.

INTRODUCTION

No amount of software testing can guarantee the quality of an application outside of the environment in which it is tested [4]. However, it is often prohibitively expensive or difficult to exactly replicate the software and hardware environment during testing that will be present once an application is deployed. This is especially true in some embedded environments, like those found inside most digital cable TV set-top-boxes (STB).

Software applications designed to operate in such constrained computational environments must provide a highly reliable, quality, interactive user-experience while simultaneously coexisting with other applications —usually from multiple vendors— and sharing the limited computational resources available. In such environments, applications that operate flawlessly by themselves may wreak havoc in a system where limited memory, CPU cycles or network bandwidth must be shared amongst several applications. Such software incompatibilities can occur intermittently and be difficult to trace. Often the actual source of a fault is elusive and may not be obviously related to the point of failure.

The results of such inter-application failures can be disastrous in a highly distributed service-oriented industry, like the digital cable TV industry, where one software failure might be replicated throughout all devices in a system, rendering the service (i.e., TV) inoperable for millions of customers. Despite the very real possibility of adverse interactions between applications, it is often logistically impossible or prohibitively expensive to test inter-application interactions prior to installation in an actual deployment environment. In the case of digital cable TV applications, software vendors rarely have access to each other's products, nor can they easily afford the cost of the hardware and software infrastructure required by another vendor's product. Laboratory environments provided by a system operator do help this situation, but individual systems are often quite unique in terms of hardware and software versions, the mix of deployed applications, and other subtle but important differences.

Inter-application testing in an environment of many distributed, embedded devices —like that found in a digital cable TV system— involves a normal installation of the application on a live system or lab. The actual testing process is rarely fully automated because —at least in the TV environment— application results are often visual, and human interpretation is required to detect defects. Further, the services provided by such applications (e.g., TV programming) often have relatively long durations, so testing cycles may be prolonged. Given these limitations and the large variety of possible adverse interactions between applications, no practical testing plan can be expected to uncover all problems [4].

We present a different approach. Rather than attempting to design and execute large, comprehensive testing plans our method gathers test data from a series of random user events, and employs a combination of Artificial Intelligence techniques to automatically analyze the data and deduce the likely sources of faults and defects.

The method we present has been employed successfully to identify the sources of inter-application failures and to predict system reliability during a recent software trial with a leading digital cable TV operator. The examples presented below are abstracted from that real world case study.

METHOD

The method we present is designed to deduce the sources of software failures from the sequences of user actions that are likely to induce those failures. It is composed of five distinct steps, as follows:

STEP 1 - Define a finite state model that abstracts the relevant software system, applications, and user events.

STEP 2 - Define a set of IF-THEN rules with associated certainty factors that identify sequences of events and states that tend to lead to failure.

STEP 3 - Gather test data from a number of randomized testing trials.

STEP 4 - Apply the rules from step 2 to the testing data from step 3 to annotate the model from step 1 with certainty factors.

STEP 5 - Analyze the annotated model produced in step 4 to deduce likely sources of failure and problem user event sequences.

Each of these steps will be discussed in detail below.

STEP 1 - The Finite State Model

The first necessary element in this process is the definition of a simple finite state model[1] that abstracts the relevant system and application states along with the user input events that cause transitions from one state to another. Initially, this model may be quite simple, with only a handful of states representing the applications involved. Later, once the method deduces specific states as the sources of failure, these problem states may be expanded into more complex sub-models to further refine the source of the failures. Table 1 below depicts a finite state model that will be used later in Example 1.

In this table each row depicts a state with the state name in the first column followed by the state transitions for the five user key-press events named at the head of the columns. Events that have no effect in the various applications have no transitions listed in the corresponding rows of the model. Pictorially, this same finite state model can be depicted by the directed graph in Figure 1. The “Power Off” state is the start state of the model, but no states are identified as final states. The only final state is the “REBOOT!” or failure state,

[1] Also known as a Finite State Automaton (FSA), Finite State Machine (FSM), or Deterministic FSA. [2]

which is accessible from all other states, and is therefore not shown.

Given a finite state model of a system, test results for that system may be represented by a sequence of (state, event, state) triples that

Model States	User Events				
	Pwr Key	**A Key**	**B Key**	**Exit Key**	**Guide Key**
Power Off	Watch TV				
Watch TV	Power Off	App1			IPG App
IPG App	Power Off	App2	App3	Watch TV	Watch TV
App1	Power Off			Watch TV	IPG App
App2	Power Off			Watch TV	IPG App
App3	Power Off			Watch TV	IPG App
REBOOT!	Power Off				

Table 1. Simple finite state model for Example 1

define the transitions from state to state caused by a sequence of user input events. For example:

```
(Power Off, Pwr Key,   Watch TV)
(Watch TV,  A Key,     App1    )
(App1,      Guide Key, IPG App )
(IPG App,   A Key,     App2    )
(App2,      Exit Key,  REBOOT! )
```

This depicts a sample transition sequence that culminates in failure. All sequences are assumed to start in the initial state (e.g., "Power Off") and end in the final, failure state (e.g., "REBOOT!").

STEP 2 - IF-THEN Rules & Certainty Factors

Next, a set of simple IF-THEN rules is defined that, when applied to a sequence of finite state transitions, will assign a numerical value to all transitions that estimates the likelihood of that transition is plays a part in an inter-application failure. For example, using the finite state model of Table 1, one sample rule might be:

```
IF (transition doesn't appear in a sequence)
THEN likelihood is -0.8.
```

Notice that the state transition:

`(PowerOff, Pwr Key, Watch TV)`

appears in the testing sequence above. Thus, the sample rule above would not apply to that transition for that sequence. However, the transition:

`(App3, Exit Key, Watch TV)`

is not found in the test data sequence, and therefore it is highly unlikely that the transition is involved in the failure. This reasoning is quantified and represented by the likelihood value of -0.8 in the sample rule.

A collection of IF-THEN rules defines a simple form of Artificial Intelligence ***expert system***, a programming paradigm that is particularly adept at encoding and applying imprecise expert knowledge about a very narrow topic [1,3]. In this case, the rules assess the likelihood that the transitions found in testing sequences play a part in software failures. Most real world expert systems contain hundreds or thousands of rules, and require specialized software language support. However, the expert systems defined in this paper are quite simple, containing only a handful of simple rules, and are easily implemented by conventional programming languages.

Similarly, the "likelihood" numeric values in rules are actually ***certainty factors (CF),*** a mechanism for quantifying uncertainty [1,3,5]. Unlike probabilities, which are often difficult to apply to real world situations, cer-

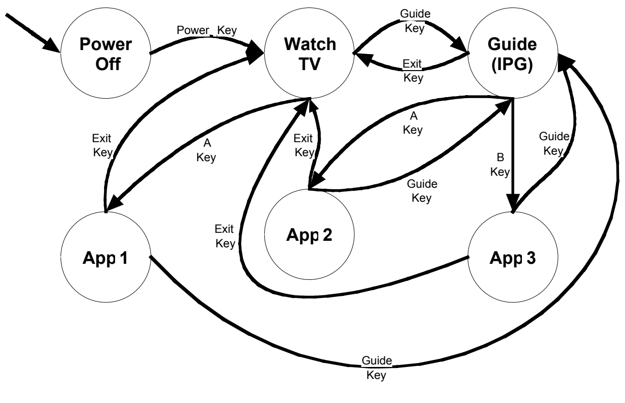

Figure 1. Graph of finite state model of Example 1

tainty factors are quite easy to implement. In addition, certainty factors can also quantify a lack of knowledge, which probability values cannot. Certainty factors are real numbers ranging between -1.0, meaning total disbelief in a conclusion, to +1.0, which denotes total belief. Any value in between denotes some measure of uncertainty. For example, a certainty factor of +0.5 means the corresponding conclusion is probably true, while a certainty factor of –0.9 means it is almost certainly not true. A certainty factor of 0.0 denotes no knowledge either way (i.e., unknown) and is used to initialize all CFs.

To combine two certainty factors CF_1 and CF_2, that are derived for the same transition by two different rules, we use the following formula [1,5]:

$$\begin{aligned} CF_1 \oplus CF_2 &= CF_1 + CF_2 * (1 - CF_2) && \text{if both are positive} \\ &= CF_1 + CF_2 * (1 + CF_2) && \text{if both negative} \\ &= \frac{CF_1 + CF_2}{1 - \min(|CF_1|,|CF_2|)} && \text{otherwise.} \end{aligned}$$

One important reason for choosing certainty factors to represent potentially inconsistent test data is that certainty factors are both *commutative* and *asymptotic* [1,3,5]. The former means that it does not matter in what order we combine certainty factors, the same result is obtained. This implies that it does not matter what in order our rules are applied, which greatly simplifies implementation. The asymptotic property implies that as new evidence is found to support (or discredit) a conclusion, we increase (decrease) the CF value incrementally. Fore example, if two distinct rules both generate a strong belief in a given transition (e.g., 0.7 and 0.8) we will tend to believe somewhat more strongly (e.g., 0.7+0.8*[1–0.7] = 0.94). The asymptotic property also keeps certainty factors nicely between -1.0 and +1.0.

Some of the rules that have been most useful to date appear to be generally applicable to almost any type of failure. Two of the most important are:

RULE 1:
IF (s_1,e,s_2) does not appear in a sequence
THEN $CF(s_1,e,s_2) = CF(s_1,e,s_2) \oplus -0.9$

RULE 2:
IF a test sequence enters a state s_i N times
THEN $\forall j \mid (s_j,e,s_i)$ is in the sequence
$CF(s_1,e,s_2) = CF(s_1,e,s_2) \oplus (+0.2 / N)$

Because we are only interested in transition sequences that end in failure, and we assume that the number of inter-application problems is small, it is logical to assume that a transition that appears in the test data may be related to that failure. Unfortunately, this deduction doesn't help much, because many transitions that are not related to the failure may also be in the test data sequences. However, by reversing that same logic, a transition that *does not* appear in the test data is most likely not related to the failure. This is codified in RULE 1, where the "most likely not related" translates into a CF of -0.9.

RULE 2 is based on the supposition that a state that appears many times in a test data sequence that ends in failure has more opportunities to cause problems. A small CF value (i.e., +0.2) is thus distributed amongst the transitions in the test data that exit from the suspect state.

Other rules are more effective for certain types of failures. Multiple, different sets of rules may be applied to the same test data sequences to deduce different types of errors. Consider RULE 3, which is effective when one application is suspected of starving another of a computational resource (e.g., a memory leak).

RULE 3:
IF transition *t* appears in the test data
THEN CF(*t*) = CF(t) ⊕ (+0.5 * (1 - *P*(*t*))

In this rule, *P*(*t*) denotes the probability estimate any random transition will be *t.* While a true probability value would be difficult to calculate, an acceptable estimate can be derived easily as follows:

Let $t = (s_1, e, s_2)$. Let T_{fsm} be the number of non-empty transitions in the finite state model, and T_{s1} be the number of transitions that lead to state s_1. Finally, let T_{s2} be the number of transitions that leave state s_2. We then define *P*(*t*) as follows:

$$P(t) = \frac{T_{s1}}{T_{fsm} * T_{s2}}$$

Finally, consider RULE 4, which helps to deduce the state in which an inter-application failure originates, even though the actual failure may occur many transitions later.

RULE 4:
IF transition t is one of the last N transitions in the test sequence
THEN CF(*t*) = CF(t) ⊕ +0.5

This rule is predicated on the assumption that the detrimental situation precipitated by the first adversely interacting state is severe enough to produce a failure soon after. For example, if one digital cable TV application corrupts the video heap in a set-top-box, a crash will often occur the next time something changes on the screen.

STEP 3 - Gather Test Data

The next step is to gather test data in the form of state-event-state transition triples as defined in the finite state model. The actual testing may be performed at any time. Old testing results generated for other purposes may also be converted to the necessary transition triples, as long as the test data still represents the current system and applications. It is understood that the initial model may be quite abstract and simplistic, and that many of the actual states and events exhibited by the system and applications are not represented by the model. But the test data must be modified to fit the model, removing extraneous transitions if necessary.

Ideally, no strict testing plan will be used to drive the sequence of user events that are input to the tested system. Rather, a random sequence of user input events is preferred. There are several reasons for this counterintuitive preference. First, a non-random test plan embodies an implicit bias toward one or more a priori results. While this is a good thing if the bias is in the right direction, test results would be useless if the bias is in the wrong direction; important state transitions might never be seen. Second, because the rules apply to a domain of uncertain data, and are statistical in nature, we suspect that many of the most effective rules work best with random test sequences. Finally, in some of the supplementary probability analyses performed on the test results, the mathematics require randomized test sequences.

Implicitly, all test sequences will begin in the start state. For all practical purposes only test sequences that end in a failure state are of interest. Because most failures are intermittent in nature, little useful information about failures can be deduced from a sequence that does not fail.

STEP 4 - Apply Rules to Test Data

This step is a straightforward application of the rules from step 2 to the testing sequences gathered in step 3. All certainty factors are initialized to 0.0 before applying any rules[2]. As rules are applied the certainty factors associated with the various transitions of the finite state model (e.g., Table 1) are modified accordingly. Once all rule processing is complete, two certainty factors should be computed for each state (e.g., *s*). The first combines the certainty factors for all transitions leaving state *s* for another different state. Similarly, the second certainty factor is a combination of the certainty factors of all transitions that are entering state *s* from other states. Ignore transitions from state *s* back to itself.

STEP 5 - Analyze the Results

This is a very interesting step, and at the time of this writing, we continue to find new and interesting results in the data produced by this process. However, several observations are generally applicable to all annotated finite state models:

(1) Positive CFs (e.g., $\geq +0.2$) suggest that a transition is somehow associated with a failure.

(2) Negative CFs (e.g., ≤ -0.2) suggest that a transition is not associated with a failure.

(3) Many transitions with positive CFs merely provide a path connecting the original source state of the failure to the state in which it fails. These states do not appear to contribute to the failure. Along such transition paths the transition CFs tend to increase. However, the corresponding entry and exit CFs of the states along this path tend to be nearly equal.

(4) States that actually fail tend to have a high entry CF and a low exit CF, because the likelihood of failure decreases after passing through the state.

(5) *Most importantly*, states that are likely to be an original source of failure, but seldom fail themselves, tend to have a much lower entry CF than exit CF, because the likelihood of failure *increases* after passing through the state. This type of result is often very hard to find using conventional testing techniques.

EXAMPLES

The examples in this section present results derived via the method presented above. The test data used has been generated by probabilistic simulation software to simplify the problem for purposes of example, while retaining the salient features of real world test data.

Example 1: Memory Leak

The test data generator was constructed to simulate the system defined by the finite state model of Table 1 and Figure 1. Five applications of varying characteristics, including two system applications and three third-party applications were simulated. Each application had different memory usage patterns and requirements. Each exit from state "App2" to another state generated a small simulated memory leak of random size. The simulation was sensitive to both memory exhaustion and fragmentation, and would enter the "REBOOT!" state whenever insufficient memory was available for an application to function.

Test data files containing 500, 1000, and 2000 legal transitions were generated by the simulation, each file containing a variable number of test sequences that end in failure states.

[2] If substantial prior evidence exists to implicate one or more transitions, the initial certainty factors may be initialized accordingly. Regardless of the evidence small initial certainty factors are recommended.

RULES 1, 2, and 3 were applied to these test files to annotate the states and transitions of the finite state model. Optimum results were obtained by the files containing 1000 transitions. Files with fewer transitions resulted in less clear distinctions between high and low CFs. Files with more than 1000 transitions tended to wash out the CFs, so that all values were approximately +1.0 or -1.0, thus obliterating valuable information about relative certainties.

The resulting annotated finite state model appears below in Table 2. Entry and exit CFs for the states are shown in Table 3.

CF(s, e)	Pwr Key	A Key	B Key	Exit Key	Guide Key
Power Off	0.000	N/A	N/A	N/A	N/A
Watch TV	0.000	0.766	N/A	N/A	0.649
IPG App	0.000	0.734	0.175	-0.604	-0.932
App1	0.000	-0.998	-0.982	-0.899	-0.980
App2	**0.000**	**N/A**	**N/A**	**0.984**	**0.977**
App3	0.000	N/A	N/A	-0.934	-0.964
REBOOT!	0.000	N/A	N/A	N/A	N/A

Table 2. Transition CFs for Memory Leak

CF(s)	CF(s) ENTRY	CF(s) EXIT
Power Off	0.000	0.000
Watch TV	-0.999	0.917
IPG App	-0.999	-0.878
App1	-0.998	-0.998
App2	**0.734**	**0.998**
App3	0.175	-0.996
REBOOT!	0.000	0.000

Table 3. State entry and exit CFs

Notice that the high CFs associated with transitions out of state "App2" (i.e., +0.98) indicate that this state is almost certainly related to the failure. The "Watch TV" and "IPG App" states also have several substantial CFs associated with transitions. These transitions tend to be on the path from state "App2" to the actual failure transition. Because failures happen in a variety of states and transitions, CF values are distributed amongst the transitions on the paths from "App2" to the failing states. Additional evidence pointing at state "App2" as the source of the failure are the entry and exit CF values for that state. Notice that the entry CF is significantly less than the exit CF. From this evidence we conclude that the memory leak originates in "App2".

Additional simulations were generated that assigned the memory leak to random applications to remove any experimental bias. The results were similar, clearly pointing to the offending state in each case.

<u>Example 2: Adverse Interaction</u>

Another test data generator was constructed to simulate the system defined by the finite state model of Table 4, below. In this example, we expand the finite state model from Table 1 to add additional sub-states and transitions within the previous "App1" state. The simulation then generated random failures with a 25% chance whenever state "App1.2" was entered sometime after exiting from state "App3". In other words, the system simulates an inter-application failure with the source of the failure in "App3", but the actual failure occurring eventually in "App1.2".

Test data files containing 1000 valid transitions were generated by the simulation. Each file contained many actual test sequences ending in a failure state.

Model States	User Events				
	Pwr Key	A Key	B Key	Exit Key	Guide Ke
Power Off	Watch TV				
Watch TV	Power Off	App1.1			IPG App
IPG App	Power Off	App2	App3	Watch TV	Watch TV
App1.1	Power Off	App1.2	App1.4	Watch TV	IPG App
App1.2	Power Off	App1.3	App1.1	Watch TV	IPG App
App1.3	Power Off	App1.4	App1.2	Watch TV	IPG App
App1.4	Power Off	App1.1	App1.3	Watch TV	IPG App
App2	Power Off			Watch TV	IPG App
App3	Power Off			Watch TV	IPG App
REBOOT!	Power Off				

Table 4. Simple finite state model for Example 2

RULES 1, 2, and 4 as defined above were applied to this test files to annotate the states and transitions of the finite state model.

The resulting annotated finite state model appears below in Table 5. Entry and exit CFs for the states are shown in Table 6.

CF(s, e)	Pwr Key	A Key	B Key	Exit Key	Guide Key
Power Off	0.000	N/A	N/A	N/A	N/A
Watch TV	0.967	0.151	N/A	N/A	0.998
IPG App	0.000	-0.220	0.179	-0.220	0.999
App1.1	0.000	0.999	0.924	-0.166	0.040
App1.2	0.000	0.998	0.147	-0.083	-0.083
App1.3	0.000	-0.089	0.724	-0.111	-0.543
App1.4	0.000	0.398	0.998	-0.468	-0.169
App2	0.000	N/A	N/A	-0.104	0.106
App3	**0.000**	**N/A**	**N/A**	**0.529**	**0.266**
REBOOT!	0.000	N/A	N/A	N/A	N/A

Table 5. Transition CFs for Memory Leak

CF(s)	CF(s) ENTRY	CF(s) EXIT
Power Off	0.000	0.000
Watch TV	0.999	0.998
IPG App	0.998	0.999
App1.1	0.563	0.999
App1.2	0.999	0.998
App1.3	0.998	0.254
App1.4	0.916	0.998
App2	-0.220	0.001
App3	**0.179**	**0.654**
REBOOT!	0.000	0.000

Table 6. State entry and exit CFs

Again, notice the higher CF values in transitions for state "App3" the disparity between entry and exit CFs for this state. This evidence again correctly suggests that state "App3" is the original source of the inter-application failure.

CONCLUSION

Digital cable TV systems, and other similarly large, distributed computing systems present unique difficulties for application vendors and system operators. Time and resources for inter-application testing is often severely limited, even though the hardware and software resource constraints within these computing environments make them susceptible to inter-application interactions and failures [4]. We present a new method that applies two simple techniques from the field of Artificial Intelligence to the problem. Rather than generating large complex test plans and lengthy testing programs, the sometimes inconsistent results from a relatively small quantity of randomly generated tests produces sufficient information for a small A.I. expert system to deduce various points of failure. In particular, we have demonstrated how the often asymptomatic sources of inter-application failures can be deduced.

This method has been applied to a real world case in the digital cable TV industry, and has successfully discovered a previously unknown memory leak in another vendor's application, and also identified an operating system anomaly that can cause exhaustion of video memory and a subsequent system crash.

CONTACT INFORMATION

Dr. Louis Slothouber
Chief Scientist,
BIAP Systems, Inc.,
www.biap.com
lpslot@biap.com

References

1. Durkin, J., *Expert Systems: Design and Development,* Macmillan Publishing Company, New York, NY, 1994.

2. Hopcroft, J.E., and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, 1979.

3. Rich, E., and K. Knight, *Artificial Intelligence --2nd ed.,* pp. 231-239, McGraw-Hill, Inc., 1991.

4. Schulmeyer, G.G., and McManus, J.I., *Handbook of Software Quality Assurance --2nd ed.,* Van Nostrand Reinhold Publishing, 1992.

5. Shortliffe, E.H., and B.G. Buchannan, A Model of Inexact Reasoning in *Medicine, Mathematical Biosciences,* vol. 23, pp. 351-379, 1975.