On the Distribution of Property Violations in Formal Models: An Initial Study

Jimin Gao, Mats Heimdahl
Dept. of Computer Science and Engineering
University of Minnesota

David Owen
Prologic Inc.

Tim Menzies
Dept. of Computer Science
Portland State University

Abstract

Model-checking techniques are successfully used in the verification of both hardware and software systems of industrial relevance. Unfortunately, the capability of current techniques is still limited and the effort required for verification can be prohibitive (if verification is possible at all). As a complement, fast, but incomplete, search tools may provide practical benefits not attainable with full verification tools, for example, reduced need for manual abstraction and fast detection of property violations during model development.

In this report we investigate the performance of a simple random search technique. We conducted an experiment on a production-sized formal model of the mode-logic of a flight guidance system. Our results indicate that random search quickly finds the vast majority of property violations in our case-example. In addition, the times to detect various property violations follow an acutely right-skewed distribution and are highly biased toward the easy side. We hypothesize that the observations reported here are related to the phase transition phenomenon seen in Boolean satisfiability and other NP-complete problems. If so, these observations could be revealing some of the fundamental aspects of software (model) faults and have implications on how software engineering activities, such as analysis, testing, and reliability modeling, should be performed.

1 Introduction

Automatic verification by model-checking has been effective in many domains including computer hardware design, networking, telecommunications protocols, automated control systems, and more [14, 7, 6]. Many real-world software models, however, are large enough that the time required for full verification can be unacceptable (if full verification is possible at all). Incomplete, but faster, analysis techniques, capable of finding property violations but not formally proving their absence (commonly referred to as refutation techniques) are useful for early feedback during development. With the wider industrial interests and adoption of *model-based development* these light-weight techniques are becoming increasingly important since quick feedback after model changes is essential to practicing engineers.

In this paper, we investigate the performance of one such incomplete technique—random search—on a close-to-production model of the model-logic of a flight guidance system. We found that random search can find the vast majority of the property violations in models with realistic seeded faults, generally much faster than the symbolic model checker NuSMV [6]. Further experiments revealed a skewed long-tailed distribution of the times to detect property violations, a distribution showing a large number of “big” (obvious) violations and a very small number of “small” (subtle) ones. We speculate that this distribution is closely linked to the phase transition effect observed in complex systems in thermodynamics, economics, neuroscience, and computation [5, 13, 17], and, therefore, this distribution of failures could be a general phenomenon of complex computational systems. If so, our results may have a broad impact on the method selection for verification and refutation, testing, debugging, and reliability estimation of software systems.

Random search is simple; we execute the software model with random inputs and monitor its internal state and outputs for property violations. We choose to explore random search for two reasons. First, in software testing there are indications that random testing can be an effective technique compared with partition testing [8, 12, 32]. Similarly, we expect random search to be an effective incomplete method in property refutation of formal models; a method that can be effectively used as a complement to more costly methods such as model-checking and theorem proving. Second, random search is essentially a Monte-Carlo simulation performed on the model input domain and can thus be used as a measurement tool to offer us statistical insights into the verification problem, e.g., the “size” of the fault (how much of the input domain leads to a property violation and a witness of a fault), the generality of a property (how many of the faulty behaviors the property covers), and the structure of the model (to what degree the model reveals its faults as deviant behaviors).

Our motivations for pursuing this work are intrinsically connected since the performance and effectiveness of random search are direct results of the collective characteristics of the model, the particular property, and the fault that causes a violation of the property. High performance and fault-finding ability of random search techniques can be expected only if the vast majority of the property violations are "big"; in other words, the probability of randomly finding input vectors that witness these property violations should be reasonably high. Many NP-complete problems, for example, Boolean satisfiability and graph coloring, are known have this characteristic. These problems exhibit what is commonly known as phase transition [5, 13]; a majority of the instances are easy to solve (either by providing a solution or to prove there is no solution), while a small set of the problem instances take disproportionately larger efforts. Consequently, many heuristic algorithms work amazingly well on typical instances of these problems. In this paper, we argue that this phenomenon may also apply to our problem domain of formal verification.

The rest of this report provides the necessary background information and our detailed experimental results. Section 2 describes the phase transition and its possible connections with formal verification. Section 3 details the experimental setup and results running the random search tool LURCH on fault-seeded flight guidance system models. Section 4 further analyzes the implications of or results. The final three sections discuss related work, internal and external threats to our results and present our conclusion.

2 Phase Transition and Stochastic Search

It has long been observed that for many NP-complete problems, the typical cases are easy to solve [5, 13]. For example, various studies [11, 9] have shown that the average-case complexity for the Boolean satisfiability (SAT) of randomly generated *variable-clause-length* conjunctive normal form (CNF) formulas is polynomial for almost all parameter settings. The hard instances are very rare.

For random *fixed-clause-length* formulas, however, the hard cases are located within a narrow parameter region. Mitchell *et al.* [25] studied the satisfiability of 3CNF formulas (CNF formulas with clause length 3) randomly generated with two parameters M and N. Each formula contained a total of M clauses and each clause was generated by randomly selecting three from N variables and randomly negating them. The ratio between M and N was found to be predictive of the fraction of the formulas that are unsatisfiable. When the ratio is low (a lot of variables from which to select and few clauses), the percentage of unsatisfiable formulas is close to zero, and when the ratio is high (a lot of clauses and few variables), the percentage of unsatisfiable formulas is close to 100. Between these two phases there is a sharp transition at the ratio value 4.3. That is, if the ratio M/N is less than 4.3 the formulas are under-constrained so that the vast majority have multiple solutions. If the ratio M/N is greater than 4.3, the formulas are over-constrained so that vast majority have no solution. In general, an under-constrained formula is easy since any reasonable algorithm could stumble on one of its many solutions. An over-constrained formula is also easy since using a backtracking algorithm most of the search paths will be cut off early. The really hard cases fall within the narrow region around the phase transition.

The phase transition can be observed in many other NP-complete problems as well, *e.g.*, graph K-coloring, Hamilton circuits and number partitioning.

In dealing with the hard instances of satisfiability problems around the phase transition, stochastic search strategies have been found to substantially outperform backtracking searches [29, 10]. In general, these algorithms first pick a random solution, and then modify the solution stepwise greedily trying to reach a valid solution. The random restarts effectively prevent the search procedure being trapped in local minima. Nevertheless, these algorithms are usually incomplete search strategies unable to show the unsatisfiability of a formula. In our experiments, we also used a random search strategy with restarts. It should be noted, however, we did not attempt to demonstrate the superiority of any stochastic search strategies. Instead, we used a simplistic pure random search, which we believe is a more objective measurement of problem instance difficulties by eliminating the idiosyncracies caused by different heuristic choices.

Although drastically different approaches have been used, temporal property verification over finite state systems can be mapped to the Boolean satisfiability problem. Bounded model checking, for example, translates the transitional behavior of the system within the initial k steps into a propositional formula, which is then input into a SAT solver [3]. More recently, a method of unbounded model checking based entirely on SAT has been proposed and experimental results on hardware verifications have shown it is substantially more efficient than a BDD (Binary Decision Diagram) based approach [21]. Since the verification problem in which we are interested is an instance of the satisfiability problem, it is likely that phase transition also applies to verification of formal software models. Should the phase transition effect be present the vast majority of property violations should be easy to reveal with a random search technique. If this phenomenon can be confirmed it has implications for the choice of verification and testing techniques; applying random techniques to catch all easy problems may be highly cost effective.

3 Flight Guidance System Experiment

In our experiment we set out to investigate two issues. First, we wanted to know what percentage of faults in a model will a random search procedure reveal (given "reasonable" time)? Second, we aimed to get an indication of how the difficulty (as measured in search time) of revealing property violations was distributed.

In an effort to get results indicative of models appearing "in the wild", we conducted a large experiment based on a model of the mode logic for a production-sized flight guidance system developed at Rockwell Collins Inc. The mode logic is captured in RSML^{-e} [19], a fully formal synchronous specification language, and automatically translated to NuSMV [6] and LURCH [27] through NIMBUS [30], the development environment for RSML^{-e}.

3.1 The Flight Guidance System

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state, generating pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS can be broken into two parts: the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time; and the flight control laws, which accept information about the aircrafts current and desired state and compute the pitch and roll guidance commands. In this case study we use only the mode logic.

3.2 LURCH

LURCH [27, 22] is a temporal random-search engine for models of finite-state concurrent systems. Like a model checker, LURCH aims to detect faults and for each fault detected produces a trace file showing a path from initial conditions to a state where the fault is revealed. LURCH is an incomplete *refutation* tool while model checkers are generally used as *verification* tools.

LURCH's basic search procedure is both partial and random. The search is partial in that there is no guarantee that the entire state space will be explored. LURCH's algorithm is random in that both the order of exploration and the choice of what portion of the state space to explore is nondeterministic. In its main search procedure, user-defined parameters *max_paths* and *max_depth* determine how long the search will run. *max_paths* is the number of iterations, each of which generates a path from the initial state through the state space. Path length is limited by *max_depth*, although shorter paths may be generated if, for example, a state is reached from which no more transitions are possible. During state exploration, a *check* function acts a synchronous observer and checks the current state to see if it represents a fault.

3.3 Experimental Setup

For our experiments we used the largest FGS model developed at Rockwell Collins Inc.—a model close to their production systems. The RSML^{-e} FGS model consists of 2,564 lines of RSML^{-e} code defining 142 state variables. When translated to NuSMV, we get 2,902 lines of code, requiring 849 BDD variables for encoding. This RSML^{-e} model has been extensively validated through testing and we have previously verified close to 300 required properties using NuSMV [24].

To get targets for our experiment, we created a collection of faulty specifications and selected a subset of the 300 properties that would reveal the faults. In an attempt to create realistic faulty specifications, we first reviewed the revision history of the FGS model to understand what types of faults were removed during the original verification process. We then implemented a random fault seeder to inject representative faults to create a suite of faulty specifications. The faults we seeded fell into the following four categories:

Variable Replacement: A variable reference was replaced with a reference to another variable of the same type.

Condition Insertion: A condition that was previously considered a "don't care" (*) in one of the RSML^{-e} condition tables was changed to T (the condition is required to be true).

Condition Removal: A condition that was previously required to be true (T) or false (F) in a table was changed to "don't care" (*).

Condition Negation: A condition that was previously required to be true (T) in a table was changed to false (F), or vice versa.

We used our fault seeder to generate 100 faulty specifications (25 for each fault class). As an example, Fig. 1 shows a missing condition fault contained in macro `When_LGA_Activated`, the fault was created by changing the table from requiring the Boolean variable `Is_This_Side_Active` to be true to a "don't care."

After the fault seeding we reran the complete verification suite of 300 properties on the 100 faulty FGS models using NuSMV. We performed this step for two reasons. First, we needed to determine which fault seeded models led to violations of at least one of the properties in the verification suite. Second, we needed to provide a baseline to use as a comparison with the fault finding ability of the random

```
MACRO When_LGA_Activated() :
 TABLE
  Select_LGA()                  : T;
  PREV_STEP(..LGA) = Selected   : F;
  Is_This_Side_Active           : *; /* Was T */
 END TABLE
END MACRO
```

Figure 1. An example fault seeded into the FGS model.

search; with this baseline we knew exactly which properties were violated in which fault seeded model. This extraordinarily time consuming exercise [28] revealed that 45 of our 100 faulty models contained faults that could be revealed by our suite of properties. In addition, we found that 60 of the original 300 properties were violated in at least one of the faulty specifications. Therefore, for our experiment we selected the 45 specifications with faults we could reveal and the 60 properties that we knew were violated by at least one faulty specifications. The properties, unfortunately, are currently considered proprietary Rockwell Collins Inc. information and we can only paraphrase their informal definitions in this report. Nevertheless, the informal examples below should give the reader some understanding of the type of properties we used in this experiment.

Property 1: If the flight director cues are off, the flight director cues shall not be turned on when the Transfer Switch is pressed, (provided that no lateral or vertical mode is selected and ⟨*additional conditions*⟩).

Property 2: If mode annunciation are off, auto pilot engagement shall cause `ROLL` mode to be selected (provided ⟨*additional conditions*⟩).

With a collection of 45 faulty models and 60 properties we were in a position to perform our experiment evaluating the effectiveness of random search as a property refutation tool as well as gaining insight into the distribution of property violations in the models.

3.4 Results

In our initial experiments we ran LURCH for 30 minutes on the faulty models and 60 properties. (The 30 minutes was an arbitrary cutoff-time selected as a “reasonable time” largely based on the extent of our patience.) To run these experiments, the faulty models were automatically translated into the LURCH input language and the properties were translated into Büchi automaton machines before they were composed with the models. To eliminate potential bias caused by accidental success in the random searches (or accidentally very poor search results), we conducted 5 test runs for each fault-property combination. Note here that since each specification (containing a single fault) may cause violations of several properties and a property may be violated in several specifications, we counted each violation of a fault-property pair separately leading to 155 property violations. The results (Fig. 2) indicated that random search can find most of the property violations before the cut-off of 30 minutes was reached. 68% of the violations were detected by every test run and 85% were detected by at least one of the five test runs, and LURCH generally found these violations much faster than NuSMV. The detailed performance results can be found in [28].

	Violations found	**Percentage of total**
Total property violations	155	
LURCH found in all 5 runs	106	68.4%
Found in at least four runs	115	74.2%
Found in at least three runs	123	79.4%
Found in at least two runs	128	82.6%
Found in at least one run	131	84.5%

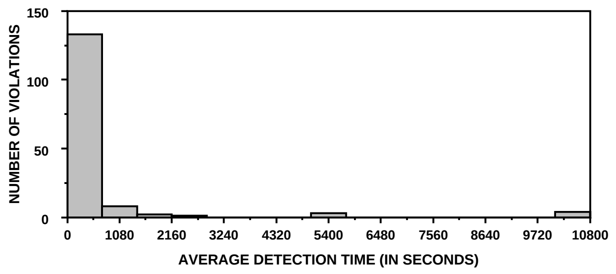
Figure 2. Summary of LURCH ’s fault finding capability.

Using a relatively crude random search technique, the results obtained with LURCH were quite surprising. To gain a better insight into the nature of the property violations, in our next experiment we attempted to quantify the inherent difficulty of revealing each property violation. LURCH is well suited to this task because its randomness eliminates any algorithmic biases.

We ran LURCH on the same faulty models and with the same properties, but this time with an extended cut-off time of 3 hours (10800 seconds) to give LURCH a chance to detect a larger percentage of the property violations and give us more data to determine the difficulty of revealing a specific property violation.

During the 3 hour run, we recorded the number of times we encountered each property violation. We then used this result to divide the total time (10800 seconds) to obtain the average detection time for each property violation. Again, we counted each violation of a fault-property pair separately. Out of the total of 155 violations found by NuSMV in all faulty specifications, 149 (96%) were detected by LURCH’s 3 hour run.

Figure 3 shows distribution histograms of the property violations in terms of their average detection time. The upper panel of Fig. 3 contains data for all detected property violations. It displays a clearly right-skewed and long-tailed distribution pattern. Although the average detection time ranged from 0.5 to 10800 seconds, 90% of all violations fell within the left 13% range (average detection time less than 1440 seconds), and 85% fell within the left 7% range (average detection time less than 720 seconds). The lower panel



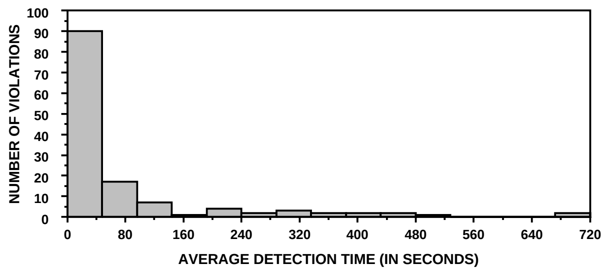


Figure 3. Distribution of the average detection time for property violations.

Probability Distributions	All Data	Truncated Data
Weibull	0.780	0.909
Lognormal	0.914	0.976
Pareto	0.903	0.980
Gamma	0.780	0.909
Birnbaum-Saunders	0.908	0.993

Figure 4. Computed max PPCC for right-skewed distributions.

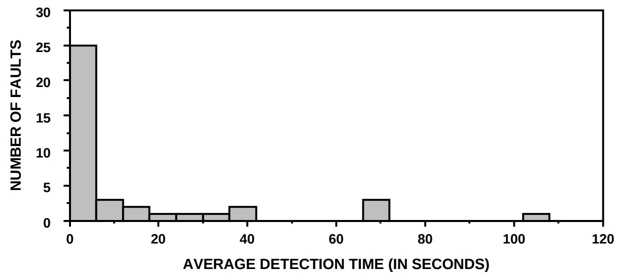


Figure 5. Distribution of the average time to detect individual faults.

of Fig. 3 shows a finer scaled histogram, displaying data contained in the first bar of the upper panel. Given such a distribution of the difficulty of finding a property violation (as measured in time for a random search to encounter the violation) the effectiveness of random search as a refutation tool is no longer surprising; the vast majority of the property violations reveal themselves very quickly.

To find a good fit for this distribution, we computed its maximum probability plot correlation coefficient (PPCC) for some well-known right-skewed probability distributions with single shape parameters using the Dataplot software [1] (Fig. 4). We also computed our results excluding the data points with average detection time greater than 2700 seconds (right column) because these property violations only saw 1 to 3 hits during the 3 hour run and their average detection time may be inaccurate due to the insufficient sampling. Out of the examined probability distributions, the Birnbaum-Saunders, Pareto and lognormal distributions seem to provide best fits for the truncated data, with max PPCC's of 0.993, 0.980, and 0.976, respectively. Because the data were truncated at the right end, yet the distinction of these distributions requires analysis of the asymptotic behavior of the tail, we leave this task to future studies.

Figure 5 illustrates the average detection time for individual faults. (To make Fig. 5 readable we have excluded four data points in the long tail of the distribution; one data point 982 seconds, two at 1800 seconds, and one at 5400 seconds.) Out of the 45 faulty specifications that caused violations (as indicated by NuSMV), 43 (96%) had faults revealed by at least one property during the 3 hour LURCH run. For each one of these specifications we calculated the average time it took LURCH to encounter any property violation. Note here that many properties might reveal the same fault in a specification. This fault detection time reflects the effort needed to find a fault when all properties are used (not individual violations corresponding to single property-fault pairs as in Fig. 3). This distribution is of interest since in actual development the same fault can cause incorrect behaviors caught by several properties; any one of these property violations is sufficient for the developer to discover and remove the fault. Again, the distribution is heavily biased towards easily detected faults.

The distributions discussed in this section support our hypothesis that random search is an effective fault removal tool far a large portion of the faults in the FGS model. From our understanding of the phase transition and previous reports on the effectiveness of random search (-like) techniques [31, 23], we believe there is sufficient data to argue that the result on the FGS model is not an isolated case. The skewed distribution could be indicative of some fundamental aspects of faults in a large class of software models/programs.

4 Discussion

At first, it may seem preposterous we want to use a technique that can only find "easy" faults for a system worthy of formal modeling and analysis. Nevertheless, our interests in such techniques was motivated by practical concerns. In our collaborations with industry we are starting to face models that are simply too large and have too many properties to be repeatedly checked with standard model-checkers (if they can be checked at all). As an extreme example, in our experiments [28] one of the faulty models took about 40 hours to model-check. Even for those non-pathological cases, the median time taken by model checking is still quite long. The duration of the change-check cycle is obviously too long to make model-checking practical for the immediate in-development feedback the engineers would like to see. Clearly, light-weight incomplete fault-finding methods can be very useful in practice if they are sufficiently effective.

Although we collected performance data form the random search implemented in LURCH and the model-checker NuSMV—and found that LURCH generally found property violations much faster than NuSMV—our experiment was never intended as a performance comparison. An incomplete refutation tool cannot be fairly compared with a verification tool since they are deigned with completely different goals in mind. Therefore, our results in this study should not be interpreted as an argument that random search would in any way be a replacement for model-checking or theorem proving. Instead, we believe random search will be a highly useful complement to model checking, especially when the model is large (and thus not practical to model check) or during early stages of model development when there are many model problems to remove. Random search would help us quickly and easily detect the obvious property violations—violations that account for the vast majority of all violations—while leaving the subtle faults for a more costly full verification effort. In sum, the most beneficial time for using random search is therefore in the beginning of model development or just after some substantial changes to a model have been made; situations when we are likely to have introduced a substantial number of new faults. Given the distribution of the difficulty of revealing faults in a model discussed in this paper, after applying a random search technique for a relatively short time (in our case a few minutes) the easy (but numerous) property violations have been detected and only very hard ones remain. At this point random (and most likely heuristic) search techniques rapidly loose their usefulness and are highly unlikely to reliably reveal any significant number of additional violations; this is the time to switch to more powerful refutation and verification techniques.

Although random search is probably the simplest of all incomplete techniques, it is implemented to monitor for all property violations simultaneously as the state space is explored. This may provide an advantage as compared to some other methods. For example, in bounded model checking the properties are checked one by one and the time required is in the worst case exponential in the length of the property under investigation. Given the goals of our study (determining the percentage of property violations revealed through random search and the distribution of the difficulty of revealing the property violations) we did not compare the effectiveness of random search with other incomplete techniques, for example, bounded model checking. Such a comparison will be a subject of future study.

Naturally, our observations have implications outside verification of formal models. Random testing is in most respects similar to the search technique we investigated in this report. Random testing with automated oracle support, for example, run-time monitors translated from properties or a monitor generated from an executable specification, can be used as the first pass to detect most easy software faults, while more costly testing techniques, for example, boundary value testing or testing to achieve MC/DC coverage, can be reserved for later stages in the testing process.

Besides analysis and testing, the distribution we observed may have important bearings on other areas of software engineering. For example, in software reliability, the two-parameter exponential model is the basic form of software reliability growth and is based on the assumption that all software faults are equally likely to be encountered and lead to a failure [15]; an assumption that is clearly erroneous given our observations in this report. Littlewood [18] has developed a generalized model that does not assume any specific fault distribution. Understandably, this model has one more parameter and—as a result—is difficult to use in practice. The two-parameter logarithmic model proposed by Musa *et. al.* [26] does assume decreasing effectiveness per fault (which amounts to non-uniform sizes of all faults) with time during random testing, though the assumed distribution is quite arbitrary. Some studies have suggested that the logarithmic model is superior in predictive validity compared with the exponential model [20]. We believe with a better understanding of program fault distribution, potentially using strictly controlled experimental methods like ours we may be able to gain significant insights into the validity of the myriads of software reliability growth models.

5 Related Work

Many studies have investigated the phase transition and stochastic searches in the SAT community [5, 25, 29, 10]. The study of the phase transition in complex systems is also increasingly gaining attention [17, 4]. Nevertheless, these studies were conducted on randomly generated systems (usually with very simple rules) and formulas, and evidence for real-world applicability is lacking.

Random walk has been studied as a state space exploration technique. In [31], West gave probably the first evidence of its effectiveness in finding faults in realistic protocol specifications. Mihail and Papadimitriou [23] identified a family of protocols, the systems of symmetric dyadic flip-flops, and theoretically argued that random walk can be used to test these protocols efficiently. Various heuristic enhancements on random walk, such as those by Kuehlmann *et al.* in [16] and Yuan *et al.* in [33] have been proposed. Nevertheless, there seem to be considerable confusions over "random walk" and "random simulation", as these two terms are often used interchangeably in the literature. Our random search is essentially a "random simulation" executing the model on *inputs* of equal probabilities, not a "random walk", which describes the approach where the probabilities of *transitions* out of the same state are equal.

A significant difference between the study in this report and many other studies is that we do not believe it is sufficient to study random simulation/walk (and other partial state space exploration techniques) in isolation. Other studies have typically explored how quickly and how much of the state space is explored by a search technique; the dimensions of properties and property violation are missing. A property violation may be revealed in a large part of the state space and we are interested in how quickly a technique will expose this corrupt portion of the state space. Thus, techniques that may seem to only cover a very small percentage of the state space may still be highly effective if the portions of the state space that lead to property violations are large and easy to reach. To our knowledge, this report is the first study applying random simulation to a large realistic software system with a large realistic property suite, and the distribution patterns for property violations presented in this report are completely new.

6 Threats to Validity

Although the results from our experiment provide some indication of the effectiveness of random search in practice and initial insight into the distribution of failures in formal-models, they only represent a small step towards understanding the interaction of properties, models, and faults in the domain of automated verification. In our study there are three threats to external validity that prevent us from generalizing our observations. First, and most seriously, we are using only one instance of a formal model. The characteristics of the FGS model—modelled only using Boolean and enumerated variables—most certainly affects our results and makes it unwise to generalize the results to systems that, for example, contain numeric variables and constraints.

Second, we are using seeded faults in our experiment. Although we took great care in selecting fault classes that represented actual faults we observed during the development of the FGS model, and some studies have shown that seeded faults do reflect realistic fault distributions [2], fault seeding always leads to a threat to external validity and may not reflect the dynamic, ever-changing fault profiles of real software development processes. However, we do believe fault seeding provides controlled conditions that better facilitate understandings on the problem itself.

Finally, we only considered a single fault per model. Using a single fault per specification makes it easier to control the experiment. Nevertheless, we cannot account for the more complex fault patterns that may occur in practice.

The question as to whether the observed effectiveness of random search is related to the specific design decision made in the LURCH tool and the specific search parameters used in the experiment pose a threat to the internal validity of the results. For example, LURCH can be set to a specific search depth; would deeper or more shallow searches have produced different results? Nevertheless, we do not believe the implementation of LURCH and the specific search parameters have any discernable impact on our results, but this is something that should be investigated in future studies.

Although there are several threats to both external and internal validity in our experiment, we believe these results are representative of a large class of models in the control systems domain and constitute a significant early step towards understanding the complex interrelationships between properties, models, and faults in automated verification.

7 Conclusion

In summary, through empirical studies on the flight guidance system we have found that random search is an effective incomplete method for property refutation of formal models; a technique that potentially can be used in combination with more powerful analysis techniques such as model checking and theorem proving. The effectiveness of random search is due to a sharply skewed distribution of property violations, a vast majority of which are relatively easy to find. We hypothesize this distribution is a manifestation of the phase transition effect observed in many NP-complete problems. We believe more investigations and better understandings of this distribution, *e.g.* its mathematical characterization and dependence on model/program structure, can lead to more effective use of the various methods for software verification, testing and reliability engineering.

References

[1] *NIST/SEMTECH e-Handbook of Statistical Methods*. NIST. Available at `http://www.itl.nist.gov/div898/handbook/`.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, New York, NY, USA, 2005. ACM Press.

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317–320. ACM Press, 1999.

[4] E. Bilotta, A. Lafusa, and P. Pantano. Research article: searching for complex CA rules with GAs. *Complex.*, 8(3):56–67, 2003.

[5] P. Cheeseman, B. Kanesfy, and W. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence IJCAI*, 1991.

[6] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4), 2000.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[8] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10:438–444, July 1984.

[9] J. Franco and M. Paull. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Math.*, 5:77–87, 1983.

[10] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Aritificial Intelligence*, pages 28–33, 1993.

[11] A. Goldberg. *On the complexity of the satisfiability problem*. Courant Computer Science Report No. 16. New York University, New York, 1979.

[12] R. Hamlet and R. Taylor. Partition testing does not inspire confidence. In *Proc. 2nd Workshop on Software Testing, Verification and Analysis*, pages 206–215, July 1988.

[13] B. Hayes. On the threshold. *American Scientist*, 91(1), 2003.

[14] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), 1997.

[15] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[16] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 574–579, Piscataway, NJ, USA, 1999. IEEE Press.

[17] C. G. Langton. Computation at the edge of chaos: phase transitions and emergent computation. *Emergent computation*, pages 12–37, 1991.

[18] B. Littlewood. Stochastic reliability growth: A model with applications to computer software faults and hardware design faults. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 139–152, New York, NY, USA, 1981. ACM Press.

[19] M. P. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, 2000.

[20] Y. K. Malaiya, N. Karunanithi, and P. Verma. Predictability of software reliability models. *IEEE Transactions on Reliability*, 41:539–546, Dec. 1992.

[21] K. L. Mcmillan. Interpolation and SAT-based model checking. In *Proceedings of Computer Aided Verfication*, pages 1–13, 2003.

[22] T. Menzies, D. Owen, and B. Cukic. Saturation effects in formal verification. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, 2002.

[23] M. Mihail and C. H. Papadimitriou. On the random walk method for protocol testing. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 132–141, London, UK, 1994. Springer-Verlag.

[24] S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedigns of the International Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, September 2003. Springer.

[25] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, July 1992.

[26] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 230–238, Piscataway, NJ, USA, 1984. IEEE Press.

[27] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *SEKE '03*, 2003.

[28] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop*, pages 75–81, December 2003.

[29] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Aritificial Intelligence*, pages 440–446, July 1992.

[30] J. M. Thompson and M. P. E. Heimdahl. An integrated development environment for prototyping safety critical systems. In *IEEE International Workshop on Rapid System Prototyping*, pages 172–177, 1999.

[31] C. H. West. Protocol validation in complex systems. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 303–312, New York, NY, USA, 1989. ACM Press.

[32] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Trans. Softw. Eng.*, 17(7):703–711, 1991.

[33] J. Yuan, J. Shen, J. A. Abraham, and A. Aziz. On combining formal and informal verification. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 376–387, London, UK, 1997. Springer-Verlag.