Beyond The Golden Run: Evaluating the Use of Reference Run Models in Fault Injection Analysis

Matthew Leeke Arshad Jhumka *

Abstract

Fault injection (FI) has been shown to be an effective approach to assessing the dependability of software systems. To determine the impact of faults injected during FI, a given oracle is needed. This oracle can take a variety of forms, however prominent oracles include (i) specifications, (ii) error detection mechanisms and (iii) golden runs. Focusing on golden runs, in this paper we show that there are classes of software which a golden run based approach can not be used to analyse. Specifically we demonstrate that a golden run based approach can not be used when analysing systems which employ a main control loop with an irregular period. Further, we show how a simple model, which has been refined using FI, can be employed as an oracle in the analysis of such a system.

1 Introduction

1.1 Motivation

As computer systems become pervasive the functionality of systems will increasingly be defined by software, thus making software dependability a critical issue for systems developers. Fault injection (FI) is one approach that can be used to assess the dependability of a given software system. In general, FI involves analysing the response of a given system to the artificial insertion of faults or errors, usually with a view to assessing the coverage and latency of error detection and correction mechanisms [13]. FI techniques can be grouped into three, not necessarily distinct, categories; simulation based fault injection, physical fault injection and software implemented fault injection (SWIFI)[1] [6] [16] [18] [19]. To assess the impact of an injected fault upon a software system being analysed, an oracle is needed. There are various types of oracle that can be employed, but perhaps the most prominent are (i) specification-based [9], (ii) error detection mechanisms [20], and (iii) golden runs [8].

However, it is often the case that either software systems are not equipped with error detection mechanisms or the mechanisms themselves may not be correct [14]. Further, it is also the case that the availability of a complete formal specification cannot be guaranteed. In such cases a golden run based approach

*High Performance Systems Group, Department of Computer Science, The University of Warwick, Coventry, {matt, arshad}@dcs.warwick.ac.uk

can be used to capture the correct functioning of a software system. In this approach the system is run under normal conditions for a set of test cases. For each test case, the system is executed and the run is termed as the *golden run* for that test case. This golden run then acts as a reproducible reference run of the system for that particular test case, capturing information about the state of the system during execution. Then, for the same set of test cases, the system is executed under various fault injections to assess it's dependability. For each test case the corresponding golden run and fault-injected run are compared, where the golden run is used as an oracle. The impact of faults can then be determined based upon any deviation from the golden run.

Despite the conceptual appeal of the described approach, there are many situations in which it is very difficult, if not impossible, to perform an analysis based on a golden run. For example, it is stated that in order to analyse software using a golden run, the software itself should (i) be deterministic in all single executions, (ii) compile independently of the fault injection suite used and (iii) not incorporate loops which make it impossible to determine when a single execution has completed [23]. In general (ii) and (iii) will be fulfilled by the majority of software that a developer would realistically want to analyse using the golden run based approach. However, there are classes of software, including concurrent systems, that will exhibit non-deterministic behaviour. Thus, requirement (i) could impact upon the applicability of fault injection to such systems. To address the described limitation this work proposes a variant of the golden based approach that allows a specific class of non-deterministic software to be analysed. Specifically, we show that a class of sequential, non-deterministic software, such as those with an irregular main control loop or an unstable initialisation due to environmental fluctuations, can be analysed using a variant of golden run based FI.

1.2 Contributions

In this paper we show that a sequential software system which exhibits non-deterministic behaviour, and for which no clear specification is available and for which the error detection based approach is not applicable, can be amenable to FI analysis. To achieve this we first construct a simple *reference run model* derived from data collected over a significant number of normal executions. We then refine this reference run model by performing fault injection experiments using the Propagation Analysis Environment (PROPANE) tool [10]. In evaluating our approach we focus upon any observed improvements in the proportion of failed executions which the reference run model correctly classifies as erroneous.

In this paper we make the following specific contributions:

- We demonstrate the limitations of employing a golden run based approach when performing fault injection analysis upon sequential software systems which exhibit non-deterministic behaviour.

- We develop approaches whereby analysis based upon SWIFI techniques can be applied to sequential software systems which exhibit some degree of non-deterministic behaviour.

- We evaluate our approaches to constructing reference run models that can be used to detect and classify errors resulting in failure.

The remainder of this paper is organised as follows: Section 2 reviews related work, discussing the use of oracles and their association with fault injection. Section 3 outlines the models assumed in this paper, including both the fault and system model. Section 4 details the nature of the selected target system. Section 5 describes the experimental setup used in our analysis. Section 6 presents our reference run model construction and results. Section 7 addresses the limitations of the described approach. Finally, Section 8 concludes the paper with a summary and discussion of future work.

2 Related Work

To perform FI analysis it is essential that some oracle be used to determine whether a particular execution is erroneous. Without such an oracle, the impact of any fault injected into an execution could not be assessed and thus little meaningful information could be derived from the associated experiments. Many current approaches to fault injection analysis rely upon the use of a single reproducible golden run to capture information regarding internal state and system outputs during the execution of a test case. This captured information may then be used as a basis for comparison with subsequent executions of the test case, effectively serving as a template for what should happen during each subsequent execution [3] [6]. However, as rigid use of a golden run assumes that the state of the system is precisely the same during all valid executions of the test case, this approach can be inadequate when validating the dependability of non-deterministic software systems. Arguably a proportion of these systems could be analysed by decomposing or isolating the component of the system under test, however these suggestions would overlook the significance of the component in the wider systems context.

To overcome the naivety of using a single set of previously observed values, many current fault injection techniques and tools allow a tolerance value to be associated with specified variables or execution points, thus permitting some degree of deviation from the value previously encountered [4] [7]. However, as this type of approach still involves the use of a single golden run, it is difficult to suggest that an observed value combined with an associated tolerance will both account for the full extent of the permissible deviation and discriminate between erroneous and valid executions, primarily because the observed value itself is subject to deviations.

Specification-based testing is an approach to testing which uses a system specification to derive a testing oracle. This testing oracle can then be used to classify failures or assure correctness in particular test cases [11] [15] [17]. Despite the appeal of this formal approach to classification, it relies upon a complete specification being available for the system under test. As it is somewhat unrealistic to expect such as specification to be available for all software systems, the applicability of specification-based testing can be limited.

In addition to golden run and specification based approaches there are a variety of dynamic mechanisms which have been employed to determine whether a particular execution is erroneous. For example, code duplication has been shown to be an effective method for discriminating between successful and erroneous executions [2] [22]. A similar effect can also be achieved through the repeated execution of code segments [12]. However, as with code duplication, it can be

difficult to determine precisely what to replicate and how to keep overheads down to an acceptable level. Further to these approaches, it is also possible to determine whether a particular test case execution is erroneous by employing runtime checks in the form of executable assertions [20]. However, as the checks to be performed may be difficult to derive, inexpressible or unknown, this approach can only be applied to target systems which are thoroughly understood or have a clearly defined operational specification.

In contrast to existing approaches, a reference run model based upon multiple executions can offer the wide applicability of dynamic mechanisms, whilst facilitating retrospective analysis similar to that afforded by golden run based approaches. Moreover, whilst the reference run models evaluated in this paper are inappropriate for dynamic analysis, the information contained within the models could be used to inform the design of such mechanisms [21].

3 Models

3.1 System Model

To conduct the required FI experiments, access to application source code is required for both system instrumentation and data collection. With this in mind we adopt a grey-box view of the target system. Moreover, we view the target systems as being composed of a series of interconnected but conceptually distinct subsystems, each of which is composed of a collection of grey-box modules. These modules are in turn considered to be composed of a collection of functions. In the case of subsystems the adoption of this grey-box system view means that, whilst the inputs and outputs of subsystems can be observed for analysis purposes, no assumptions about the interconnections between subsystems are made. For the modules which comprise subsystems the grey-box view mean that, whilst the required access to application source code is afforded, the precise functionality of each module need not be known. A visualisation of the described system model is provided in Figure 1.

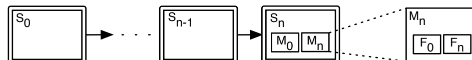

Figure 1: System model visualisation

3.2 Fault Model

The fault model adopted in this paper is a transient fault model, i.e. a fault occurs and may never appear again. Such a fault model implies that single bit-flips faults are used to target variables within the selected system modules, allowing transient failures to be mimicked. We do not discriminate among variables and assume that all variables stored in local memory are equally susceptible to corruption.

4 Target System

The FlightGear Flight Simulator project is a collaborative open-source project which seeks to build a extensible yet highly sophisticated flight simulator to serve the needs of the academic and hobbyists communities [5]. The flight simulator itself is written in C/C++ and is configured using collections of XML files. The software is deliberately designed such that it can be built and run on multiple platforms, including variants of Linux, Windows, and Macintosh OS X. All source code and resources related to the project are made available under the GNU General Public License.

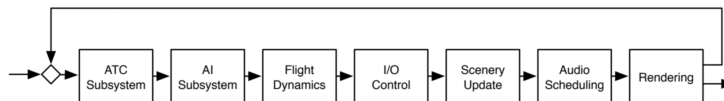

Figure 2: Structure of the FlightGear Flight Simulator main control loop

As is common in many software simulations, the FlightGear architecture depends upon a single main control loop, which is responsible for periodically sending update requests to subsystems within the simulation. The details of the subsystems which are updated during each iteration of the main control loop are given in Figure 2. As FlightGear is fundamentally a single-threaded sequential application, program control is passed to each of the individual subsystems in turn, with the main control loop being held up whilst each update request is processed. As a consequence of this architecture the performance of each individual subsystem impacts heavily upon the overall performance of the simulator. Further, as a request for an update of the graphical display is sent to the renderer subsystem at every iteration of the main control loop, the frequency at which the loop iterates is directly linked to the framerate. It is this direct link, combined with the fact that framerate is a function of scene complexity, which results in the main control loop having an irregular period and thus exhibiting non-deterministic behaviour.

5 Experimental Setup

To ensure that the target system could be evaluated in a consistent manner, steps were taken to ensure that separate executions of the system could be justifiably compared. Firstly, to ensure that a distinct start and end point could be observed for each separate execution, it was determined that the main control loop would be allowed to execute for a fixed number of iterations before the simulation was gracefully terminated. Moreover, as the single test case used in the FI experiments involved the simulator performing a simple aircraft takeoff, the number of iterations was fixed at 1400. This execution duration was a conservative, empirically determined value which allowed sufficient time for a complete takeoff. To ensure that the process of termination did not impact upon the system executions, the same termination sequence that is employed by the target system to initiate a standard simulation shut down was used.

As some means of providing the target system with valid input vectors at each iteration of the main control loop was required, a simple input module was

Module ID	Subsystem	Module Name	Function	Variables
A	Flight Dynamics	FGEngine	FGEngine	3
B	Flight Dynamics	FGEngine	ConsumeFuel	4
C	Flight Dynamics	FGPiston	Calculate	5
D	Flight Dynamics	FGPropeller	Calculate	4

Table 1: Instrumentation details for FI experiments used in model refinement

developed. This module was implemented using the environment simulation feature afforded by the PROPANE tool, thus providing a degree of separation from the target system. Further, to ensure that the irregular period of the main control loop did not severely impact upon or nullify the relevance of the input vector provided at each iteration, the XML configuration files associated with the single test case were modified such that, whilst the simulator was able to perform the takeoff procedure, precisely the same input vector could be provided at every iteration of the main control loop.

In line with the described system model, the overall system outputs were recorded at each iteration of the main control loop. Specifically, the logging capabilities of the PROPANE tool were used to record the current altitude and speed of the aircraft immediately following each iteration of the main control loop. For the remainder of this paper the current altitude and speed outputs are referred to as observed system outputs.

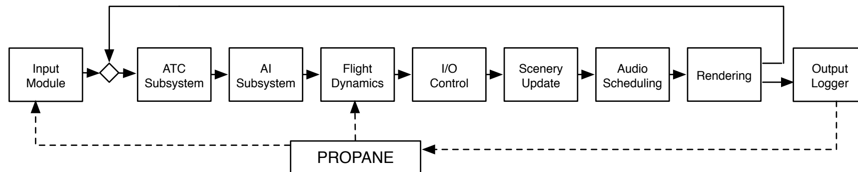

Figure 3: Structure of the experimental setup

The instrumentation details for the FI experiments used in the refinement of the reference run models are shown in Table 1. All experiments were focused upon modules within the Flight Dynamics subsystem, primarily because this subsystem is the largest and has the most significant impact upon the overall output of the system, thus it would be desirable for errors resulting in failure to be more readily detected in this subsystem. The variables instrumented were randomly selected with no prior knowledge of their function or meaning. To assess the capacity of each reference run model to classify executions consistently, regardless of variations in observed system outputs, 7 executions of each distinct experiment (i.e. each randomly chosen bit flip fault) were performed.

By introducing significant deviations in internal state it is possible that the task of discriminating between erroneous and valid executions could be made easier, as it is possible that the observed outputs may vary significantly. With this in mind the extent of the bit-flip faults injected was limited to the first ten least significant bits of instrumented variables, each of which had a internal representation greater than this threshold. Henceforth each subsystem-module-function triple shown in Table 1 will be referred to by the assigned Module ID.

6 Model Construction & Results

6.1 Golden Run

Prior to the development of the reference run model, an attempt was made to perform FI analysis upon the target system using a golden run based approach. A single execution of the target system was performed, with the observed system outputs being recorded at each iteration of the main control loop. However, during initial fault injection experiments it became evident that all executions were being classified as erroneous. To verify that this was indeed a problem with the oracle employed and not a result of the faults injected, an attempt was made to recreate the golden run and the information collected from it. This recreation proved unsuccessful, with each iteration yielding inconsistent values for all observed system outputs. To verify what had been observed, a further 150 executions were performed and compared. As no two of these executions yielded consistent values, it was determined that employing a golden run based FI approach in the analysis of the target system was inappropriate. Further, as no formal specification or sufficiently detailed functional documentation existed for the target system, an alternative approach to deriving an oracle for the classification of executions was motivated.

6.2 Range Model

In this section we propose a range model which classifies executions as being either erroneous or valid based upon the values of observed system outputs. Following the construction of an initial model based upon values observed during multiple executions of the target system, the model is validated and refined using FI experiments. Ideally these experiments would have demonstrated that the range model can correctly classify all failed executions as erroneous, whilst avoiding the misclassification of valid executions. As this was the case for only one of the modules under test, modifications which address the limitations of the range model are necessitated.

The motivation for the use of tolerances in golden run comparisons is the need to acknowledge deviations in the expected runtime values of particular variables or system outputs. The range model admits the potential for deviation by assuming that each observed output has an associated range of valid values for each iteration of the main control loop and that any occurrence outside this range is indicative of an erroneous execution. As it is not possible to guarantee that a true maximum or minimum has been encountered for particular iteration of the main control loop, the model must be somewhat permissive of deviations which cause observed outputs to go outside the bounds established through observation. We acknowledged that our sample executions could not be assumed to contain information that precisely represented the distribution of acceptable values by extending the bounds of the range by a percentage of the bound itself.

Table 2 shows the result of fault injections experiments classified using the range model. The range deviation percentage was set to 10% above and below the previously observed minimum and maximum respectively. The results presented were obtained under the fault model and experimental setup described previously. The Erroneous column of the table indicates how many of the executions were classified as erroneous. To reiterate, an execution if classified as

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	210	182	0.867	7	0.033	1.000
B	280	182	0.650	56	0.200	0.875
C	350	252	0.720	70	0.200	0.700
D	280	238	0.850	63	0.225	0.667
	1120	854	0.763	196	0.175	0.750

Table 2: Classifications for Range Model (±10% Range Deviation)

erroneous if any observed output value strays outside the valid range for at least one iteration of the main control loop. The Error Rate expresses the number of erroneous executions as a proportion of the total number of executions. The Failures column gives the number of executions which could not be completed due a failure, with the Failure Rate expressing that number as a proportion of the total number of executions performed. Finally, Detection Rate refers to the proportion of failed executions that were classified as erroneous before failing.

The most noticeable result presented is the average failure detection rate of 0.750. This is a relatively low detection rate that could prevent this parameterisation of the model from being used as an oracle in any form of dependability analysis. Given that each distinct experiment was performed on 7 separate occasions, it is unsurprising that the number of runs classified as erroneous for each module is a multiple of 7. However, it is important to recognise that a repeated execution may not necessarily be given the same classification as the executions that it attempts to recreate. This is due to the variations in observed system outputs, which can determine whether a particular fault manifests as a error or whether the overall range is violated by a particular execution. It is also interesting to note that, whilst the error rate for module A is the highest of the four modules under test, the associated fatality rate is remarkably low, a fact that is made noteworthy because it is the only class constructor under test.

To ensure that range model is able to correctly discriminate between valid and erroneous executions, 150 executions with no fault injections were classified. The model correctly classified each of these executions as being non-erroneous.

Having stressed the importance of detecting errors which lead to failures and discussed how the range deviation percentage determines the range of values which are considered acceptable, we now show how adjusting the bounds of the range impacts upon the capacity of the model to correctly detect errors resulting in failure. Specifically, we show how narrowing the valid range can increase the failure detection rate without unduly impacting upon the error rate.

The results in Table 3 demonstrate an improvement in the detection of errors which result in failures. Of the 14 executions newly classified as erroneous, each one ultimately resulted in a failure. Further to this, these 14 executions are associated with only 2 experiments, with each of the 7 separate executions of each experiment being correctly classified as erroneous. When interpreting these results it is important to note that the selected range percentage deviation was not chosen blindly. The figure was reached by systematically narrowing the extent of the valid range and carefully observing the response of the model. During this process it became apparent that executions resulting in failures

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	210	182	0.867	7	0.033	1.000
B	280	182	0.650	56	0.200	0.875
C	350	259	0.740	70	0.200	0.800
D	280	245	0.875	63	0.225	0.778
	1120	868	0.775	196	0.175	0.821

Table 3: Classifications for Range Model (±7.8% Range Deviation)

were first to be excluded as the range narrowed, indicating that repetitions of the same fault injected execution produce similar values in at least one of the observed system outputs. Using this approach it was found that a range deviation of 7.8% was the last point at which a failure was encountered before the range narrowing process began to reclassify executions which did not result in failure. The highly focused improvement presented not only gives credence to the assumption that there is a valid range of values associated with each of the observed variables, it may also suggest that certain observed system output values are associated with failures.

To this point the results derived from FI experiments have been used to improve the failure detection rate of the range model. To demonstrate that this improvement has not also impacted upon the capacity of the model to discriminate failures, 150 executions with no fault injections were classified using the refined range model. The model was able to correctly classify each of these executions as being valid, thus suggesting that the improved results were not due to the eagerness of the model to classify executions as erroneous.

A comparison of the results presented so far would indicate that the range deviation percentage does impact upon the capacity of the model to detect errors which result in failures. Interestingly, as the range deviation percentage can be controlled, there is potential for it to be tailored to fit the requirements of any target system that is amenable to the approach described. For example, when evaluating safety-critical applications using the range model as a oracle, a lesser range deviation percentage could be adopted to ensure that the values are both confined and well understood. In contrast, performance applications which seek to reduce the amount of time spent undertaking recover actions may be more permissive of false negatives.

6.3 Clustered Model

The clustered reference run model is a further refinement of the range model. The clustered model seeks to ensure that errors associated with failures within the valid range of observed output values are correctly detected. The greatest weakness of the range model is the underlying assumption that an error-free execution could potentially produce values across the entire range, thus ignoring groupings or arbitrary values that may occur within the range. The clustered model addresses this limitation by identifying values within the range that are grouped sufficiently closely to suggest that they may combine to form a valid sub-range. The range deviation percentage used in the range model is retained.

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	210	182	0.867	7	0.033	1.000
B	280	187	0.668	56	0.200	0.964
C	350	259	0.740	70	0.200	0.800
D	280	238	0.850	63	0.225	0.667
	1120	866	0.773	196	0.175	0.811

Table 4: Classifications for Clustered Model (±10% Range, ±10% Sub-Range)

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	210	182	0.867	7	0.033	1.000
B	280	188	0.671	56	0.200	0.982
C	350	271	0.774	70	0.200	0.971
D	280	245	0.875	63	0.225	0.778
	1120	886	0.791	196	0.175	0.913

Table 5: Classifications for Clustered Model (±7.8% Range, ±7.8% Sub-Range)

In addition, a new sub-range deviation percentage is used to specify how closely observed output values must be to be considered part of the same sub-range.

Tables 4 and 5 show the results of fault injections experiments classified using the clustered model with both deviation parameters and set to 10% and 7.8% respectively. All experiments were conducted under the fault model and experimental setup described previously.

The results shown in Table 4 and 5 demonstrate the superiority of the clustered model, with both parameterisation yielding better failure detection rates than their range model counterparts. Further, this improved detection rate does not lead to an unnecessary increase in the overall error detection rate, as only those executions which are newly classified as resulting in failures contributed to the increase. This efficiency in failure detection can also be seen across different parameterisations of the clustered model, as once again only newly recognised failures contribute to the increased error rate.

The matched range and sub-range deviation percentages shown were selected to allow a broad, yet imperfect, comparison with the results presented for the range model. In order to further refine the clustered model it is possible to use unmatched values for the range and sub-range deviation parameters. As the analysis performed in the refinement of range model yielded a justifiable extent for the overall range, the range deviation percentage was not modified. The sub-range deviation was systematically decreased and set to the value associated with last point at which a failure was encountered before executions which did not end in failure were excluded.

The results presented in Table 6 represent both the final refinement of the clustered reference run model and a significant improvement over the refined range model. Not only is the failure detection rate superior to that achieved by any parameterisation of the range model, the clustered model also correctly

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	210	182	0.867	7	0.033	1.000
B	280	189	0.675	56	0.200	1.000
C	350	273	0.780	70	0.200	1.000
D	280	245	0.875	63	0.225	0.778
	1120	889	0.794	196	0.175	0.929

Table 6: Classifications for Clustered Model (±7.8% Range, ±4.5% Sub-Range)

Module ID	Runs	Erroneous	Error Rate	Failures	Failure Rate	Detection Rate
A	150	130	0.867	5	0.033	1.000
B	200	130	0.650	45	0.225	1.000
C	250	195	0.780	50	0.200	1.000
D	200	170	0.850	45	0.225	0.778
	800	625	0.781	145	0.181	0.931

Table 7: Validation of Clustered Model (±7.8% Range, ±4.5% Sub-Range)

classified all failures as erroneous in three of the four modules under test. A subsequent inspection of the data collected during experimentation revealed that those failures which the refined model incorrectly classified were associated with errors that caused the target system to terminate before completing the current iteration of main control loop.

To demonstrate that the clustered model could discriminate between valid and erroneous executions, 150 non-fault injected execution were classified using the clustered model with range and sub-range deviation set to 7.8% and 4.5% respectively. The model correctly classified each of these executions as valid.

As the target system exhibits a degree of non-deterministic behaviour, it is possible that the improvements presented may only be relevant to the set of experiments conducted in the refinement of the model. To validate the clustered model, a further set of fault injection experiments were conducted under the fault model described previously. A single change to the experimental setup was made, which saw the number of executions performed reduced from 7 to 5 for each distinct experiment.

The results in Table 7 suggest that the refined clustered model can effectively detect those errors which result in failures under the adopted fault model. The error rates obtained are consistent with those presented previously, thus indicating that the clustered model is consistent in its classification of errors. The fact that the model can be seen to have maintained a near-identical failure detection rate suggests that the model is indeed capable of consistently detecting errors which lead to failures. Indeed, the small discrepancy between the two failure rates can be explained by the difference in the number of repeated experiments, as both models were in agreement with respect to each distinct experiment. A subsequent inspection of the information recorded during the FI experiments revealed that failure misclassifications were associated with errors which did not

allow the current iteration of the main control loop to end.

Having constructed, refined and validated a reference run model which is capable of detecting a significant proportion of errors leading to failures in the target system, it is reasonable to conclude that reference run models demonstrate potential for use as oracles in FI analysis. Further, the information implicitly stored within the reference run model can be used in the design of error detection mechanisms. For example, the identified valid ranged associated with the refined model could be used to inform the implementation of dynamic mechanisms such as runtime checks.

7 Limitations

The framerate associated with the selected target system governs the period of the main control loop and thus the rate at which observed system outputs are recorded. This means that the extent of the valid range and hence the correctness of the model can be impacted upon if a relatively stable framerate is not maintained in each execution of the target system. Whilst this limitation is not well represented in the results presented, this potential source of variation must be considered when recreating the experimental setup described. More significantly, it should be remembered that the outlined approach can not be employed in the analysis of all software systems. The assumptions that underlie the model and the exploitation of signals which model continuous real-world phenomena are relatively specific to the type of software under test. However, despite this acknowledgement it should be noted that the intention of this paper was not propose such a general purpose approach, rather it was to demonstrate that it is possible to use reference run models to perform FI analysis upon sequential software which exhibits non-deterministic behaviour.

8 Summary & Future Work

The desire for determinism in all single executions of a target system can be considered to be a limiting characteristic of golden run based FI analysis. In this paper we have demonstrated that it is possible to analyse a sequential system which exhibits a degree of non-deterministic behaviour. Specifically, we have shown that reference run models, including the range and clustered models presented, could be employed as an oracle for the classification of executions associated with such a system. Further, we have gone on to propose that the information derived during the construction and refinement of reference run models could be use to inform the design and placement of runtime error detection mechanisms.

The majority of the work presented has focused upon the detection of errors which result in failures. If reference run models are to be applied more generally then it is also important that error-oriented analyses can be performed. Thus, one way to advance the work presented would be to verify that the error rates derived from reference run models can be consistent with those derived from a single golden run based approach, thereby demonstrating that model based approaches could be applied when estimating coverage values and other error-oriented metrics.

References

[1] J H Barton, E W Czeck, Z Z Segall, and D P Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, April 1990.

[2] A Benso, S Chiusano, P Prinetto, and L Tagliaferri. A c/c++ source-to-source compiler for dependable applications. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 71–78. IEEE Computer Society, June 2000.

[3] R Chandra, R M Lefever, M Cukier, and W H Sanders. Loki: A state-driven fault injector for distributed systems. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 237–242. IEEE Computer Society, June 2000.

[4] J-C Fabre, M Rodriguez, J Arlat, and J-M Sizun. Building dependable cots microkernel-based systems using mafalda. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependalbe Computing*, pages 85–92. IEEE Computer Society, August 2000.

[5] FlightGearFlightSimulator. http://www.flightgear.org/.

[6] K K Goswami, R K Iyer, and L Young. Depend: A simulation-based environment for system level dependability analysis. *IEEE Transactions on Computers*, 46(1):60–74, January 1997.

[7] M Hiller. *A Software Profiling Methodology for Design and Assessment of Dependable Software*. Doctoral thesis, Department of Computer Engineering, School of Computer Science and Engineering, Chalmers University of Technology, Goteborg, Sweden, June 2002.

[8] M Hiller, A Jhumka, and N Suri. An approach for analysing the propagation of data errors in software. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 161–172, July 2001.

[9] M Hiller, A Jhumka, and N Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 135–144, June 2002.

[10] M Hiller, A Jhumka, and N Suri. Propane: An environment for examining the propagation of errors in software. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 81–85. ACM, July 2002.

[11] L J Jagadeesan, A Porter, C Puchol, J C Ramming, and L G Votta. Specification-based testing of reactive software: Tools and experiments experience report. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 525–535. IEEE Computer Society, May 1997.

[12] B W Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley, 1989.

[13] G A Kanawati, N A Kanawati, and J A Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, February 1995.

[14] N G Leveson, S S Cha, J C Knight, and T J Shimeall. The use of self checks and voting in software error detection: An empirical study. *IEEE Transactions on Software Engineering*, 16(4):432–443, April 1990.

[15] D J Richardson, S L Aha, and T O O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, July 1992.

[16] V Sieh, O Tschäche, and F Ballbach. Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions. In *Proceedings of the 27th International Symposium on Fault Tolerant Computing*, pages 32–36. IEEE Computer Society, June 1997.

[17] P Stocks and D Carrington. A framework for specification based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, November 1996.

[18] D T Stott, B Floering, Z Kalbarczyk, and R K Iyer. Nftape: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the 4th International Symposium on Computer Performance and Dependability*, pages 91–100. IEEE Computer Society, June 2000.

[19] T K Tsai and R K Iyer. Measuring fault tolerance with the ftape fault injectiontool. *Lecture Notes in Computer Science*, 977/1995:26–40, April 1995.

[20] J Vinter, J Aidemark, P Folkesson, and J Karlsson. Reducing critical failures for control algorithms using executable assertions and best effort recovery. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 347–356. IEEE Computer Society, July 2001.

[21] J Vinter, O Hannius, T Norlander, P Folkesson, and J Karlsson. Experimental dependability evaluation of a fail-bounded jet engine control system for unmanned aerial vehicles. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 666–671. IEEE Computer Society, June 2005.

[22] J Vinter, A Johansson, P Folkesson, and J Karlsson. On the design of robust integrators for fail-bounded control systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 415–424. IEEE Computer Society, June 2003.

[23] J M Voas and G McGraw. *Software Fault Injection.* John Wiley and Sons, 1998.