Validating Code-Level Behavior of Dynamic Adaptive Systems in the Face of Uncertainty

Erik M. Fredericks, Andres J. Ramirez, and Betty H. C. Cheng

Michigan State University, East Lansing, Michigan 48824-1226, USA
{freder99, ramir105, chengb}@cse.msu.edu

Abstract. A dynamically adaptive system (DAS) self-reconfigures at run time in order to handle adverse combinations of system and environmental conditions. Techniques are needed to make DASs more resilient to system and environmental uncertainty. Furthermore, automated support to validate that a DAS provides acceptable behavior even through reconfigurations are essential to address assurance concerns. This paper introduces Fenrir, an evolutionary computation-based approach to address these challenges. By explicitly searching for diverse and interesting operational contexts and examining the resulting execution traces generated by a DAS as it reconfigures in response to adverse conditions, Fenrir can discover undesirable behaviors triggered by unexpected environmental conditions at design time, which can be used to revise the system appropriately. We illustrate Fenrir by applying it to a dynamically adaptive remote data mirroring network that must efficiently diffuse data even in the face of adverse network conditions.

Keywords: search-based software engineering, novelty search, genetic algorithm, software assurance

1 Introduction

A dynamically adaptive system (DAS) can self-reconfigure at run time by triggering adaptive logic to switch between configurations in order to continually satisfy its requirements even as its operating context changes. For example, a hand-held device may need to dynamically upload an error-correction communication protocol if the network is lossy or noisy. Despite this self-reconfiguration capability, a DAS may encounter operational contexts of which it was not explicitly designed to handle. If the DAS encounters such an operational context, then it is possible that it will no longer satisfy its requirements as well as exhibit other possibly undesirable behaviors at run time. This paper presents Fenrir, a design-time approach for automatically exploring how a broad range of combinations of system and environmental conditions impact the behavior of a DAS and its ability to satisfy its requirements.

In general, it is difficult to identify *all* possible operating contexts that a DAS may encounter during execution [7, 6, 18, 28]. While design-time techniques have been developed for testing a DAS [4, 20–22, 24, 25, 31], these are typically

restricted to evaluating requirements satisfaction within specific operational contexts and do not always consider code-level behaviors. Researchers have also applied search-based heuristics, including evolutionary algorithms, to efficiently generate conditions that can cause failures in a system under test to provide code coverage [2, 15, 16]. Automated techniques are needed to make a DAS more resilient to different operational contexts as well as validate that it provides acceptable behavior even through reconfigurations.

This paper introduces FENRIR,[1] an evolutionary computation-based approach that explores how varying operational contexts affect a DAS at the code level at run time. In particular, FENRIR searches for combinations of system and environmental parameters that exercise a DAS's self-reconfiguration capabilities, possibly in unanticipated ways. Tracing the execution path of a DAS can provide insights into its behavior, including the conditions that triggered an adaptation, the adaptation path itself, and the functional behavior exhibited by the DAS after the adaptation. At design time, an adaptation engineer can analyze the resulting execution traces to identify possible bug fixes within the DAS code, as well as optimizations to improve the run-time self-adaptation capabilities of the DAS.

FENRIR leverages novelty search [17], an evolutionary computation technique, to explicitly search for diverse DAS execution traces. Specifically, FENRIR uses novelty search to guide the generation of diverse DAS operational contexts comprising combinations of system and environmental conditions that produce previously unexamined DAS execution traces. Since we do not know in advance which combinations of environmental conditions will adversely affect system behavior, we cannot define an explicit fitness function for generating the operational contexts. Instead, we opt for *diverse* operational contexts with the intent of considering a representative set of "all" operational contexts. FENRIR then executes the DAS under these differing operational contexts in order to evaluate their effects upon the DAS's execution trace. As part of its search for novel execution traces, FENRIR analyzes and compares the traces to determine which operational contexts generate the most diverse behaviors within the DAS.

We demonstrate FENRIR by applying it to an industry-provided problem, management of a remote data mirroring (RDM) network [12, 13]. An RDM network must replicate and distribute data to all mirrors within the network as links fail and messages are dropped or delayed. Experimental results demonstrate that FENRIR provides a significantly greater coverage of execution paths than can be found with randomized testing. The remainder of this paper is as follows. Section 2 provides background information on RDMs, evolutionary algorithms, and execution tracing. Section 3 describes the implementation of FENRIR with an RDM network as a motivating example. Following, Section 4 presents our experimental results, and then Section 5 discusses related work. Lastly, Section 6 summarizes our findings and presents future directions.

[1] In Norse mythology, Fenrir is the son of Loki, god of mischief

2 Background

In addition to overviewing the key enabling technologies used in this work, this section also overviews the remote data mirroring application.

2.1 Remote Data Mirroring

Remote data mirroring (RDM) [12, 13] is a data protection technique that can maintain data availability and prevent loss by storing data copies, or replicates, in physically remote locations. An RDM is configurable in terms of its network topology as well as the method and timing of data distribution among data mirrors. Network topology may be configured as a minimum spanning tree or redundant topology. Two key data distribution methods are used. *Synchronous* distribution automatically distributes each modification to all other nodes, and *asynchronous* distribution batches modifications in order to combine edits made to the data. Asynchronous propagation provides better network performance, however it also has weaker data protection as batched data could be lost when a data mirror fails. In our case, an RDM network is modeled as a DAS to dynamically manage reconfiguration of network topology and data distribution.

2.2 Genetic Algorithms

A genetic algorithm (GA) [11] is a stochastic search-based technique grounded in Darwinian evolution that leverages natural selection to efficiently find solutions for complex problems. GAs represent a solution as a population, or collection, of genomes that encode candidate solutions. A fitness function evaluates the quality of each individual genome within the population in order to guide the search process towards an optimal solution. New genomes are produced through crossover and mutation operators. In particular, crossover exchanges portions of existing genomes and mutation randomly modifies a genome. The best performing individuals are retained at the end of each iteration, or generation, via the selection operator. These operations are repeated until a viable solution is found or the maximum number of generations is reached.

Novelty search [17] is another type of genetic algorithm that explicitly searches for unique solutions in order to avoid becoming caught in local optima. Novelty search replaces the fitness function in a traditional genetic algorithm with a novelty function that uses a distance metric, such as Euclidean distance [3], to determine the distance between a candidate solution and its k-nearest solutions. Furthermore, a solution may be added to a novelty archive in order to track areas of the space of all possible solutions that have been already thoroughly explored.

2.3 Execution Tracing

Following the execution path of a software system can provide insights into system behavior at run time, as it may behave differently than intended due

to uncertainty within its operating context. Tracing system execution has been applied to various aspects of software analysis, from understanding the behavior of distributed systems [19] to identifying interactions between the software and hardware of superscalar processors [23]. An execution trace can be generated by introducing logging statements. Many different approaches to software logging exist [34], however for this paper we consider a subset of *branch coverage* [10] as our metric for tracing execution paths. Branch coverage follows all possible paths that a program may take during execution, such as method invocations, `if-else`, or `try-catch` blocks.

3 Approach

FENRIR is a design-time assurance technique for exploring possible execution paths that a DAS may take in response to changing environmental and system contexts. Conceptually, we consider a DAS (see Figure 1) to comprise a collection of target (non-adaptive) configurations, TC_i, connected by adaptive logic, A_{ij}, that moves execution from one target configuration (TC_i) to another (TC_j) [35]. Therefore, the functional logic of the system (i.e., requirements) is implemented by the target configurations, where each target configuration may differ in how it implements the requirements (e.g., different performance requirements), and how it may handle specific environmental conditions.

FENRIR starts with instrumented code for both the adaptive logic and functional logic for a DAS. Then the set of operating contexts that can trigger DAS self-reconfigurations are generated using novelty search, in order to provide a diverse and representative set of environmental and system contexts that the DAS may encounter during execution. These operating contexts are based on different combinations of identified sources of uncertainty that may affect the DAS at run time, where the combinations may be unintuitive, but feasible, and therefore not anticipated by a human developer. Next, the instrumented DAS is executed for different operational contexts, where each operational context will have a corresponding execution trace that reflects the execution path of a DAS as it self-reconfigures and executes its target configurations. This trace provides information necessary to fully realize the complexity inherent within the DAS as it executes and performs self-reconfigurations.

Figure 2 provides an overview of a DAS that has been instrumented with logging statements. The instrumented DAS is split into two parts: Configuration and Adaptation Manager. The Configuration refers to the collection of target configurations connected by the adaptive logic as shown in Figure 1, and the Adaptation Manager comprises a monitoring, decision logic, and reconfiguration engine to manage the self-adaptation capabilities for the DAS. Together, these two parts make up the system that can reconfigure itself at run time in order to handle uncertainties within its operating context.

Logging statements are then inserted into both the Configuration and Adaptation Manager to provide an engineer with information regarding the target configuration state, conditions that trigger adaptations, and steps taken dur-

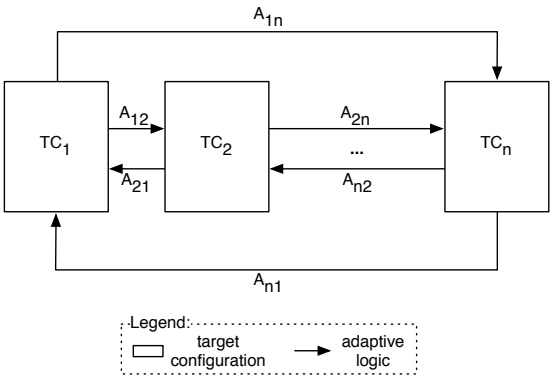


Fig. 1. DAS comprising a collection of target configurations TC_i connected by adaptation logic A_{ij}.

ing self-reconfiguration at each time step throughout execution. Furthermore, the instrumented DAS requires an operational context to specify the sources of environmental and system uncertainty, as previously identified by the domain engineer. The DAS generates a trace that represents the execution path for a given set of environmental conditions.

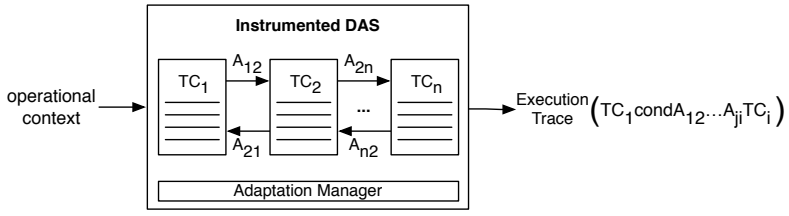


Fig. 2. Overview of an instrumented DAS.

The remainder of this section describes how FENRIR generates novel execution traces. First, we present an overview of FENRIR and state its assumptions, required inputs, and expected outputs. We then discuss each step in the FENRIR process.

3.1 Assumptions, Inputs, and Outputs

FENRIR requires instrumented executable code for a DAS to exercise the adaptive logic and functional logic triggered by different operational contexts. In particular, the instrumented code within the DAS should provide a measure of *branch coverage* in order to properly report the various execution paths that a DAS may traverse. Furthermore, the logging statements should monitor possible exceptions or error conditions that may arise.

FENRIR produces a collection of operational contexts, each with a corresponding execution trace generated by the DAS. The operational contexts specify sources of system and environmental uncertainty, their likelihood of occurrence, and their impact or severity to the system. Each operational context may trigger adaptations within the DAS, thereby creating a vast set of possible execution paths. Execution traces contain information specific to each explored path, providing insights into the overall performance of the DAS throughout execution. In particular, information regarding the invoking module, line number, a description of intended behavior, and a flag indicating if an exception has occurred is provided for further analysis.

3.2 Fenrir Process

The data flow diagram (DFD) in Figure 3 provides a process overview for using FENRIR. Each step is described next in detail.

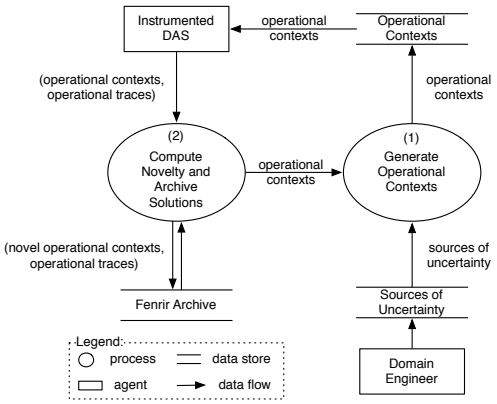


Fig. 3. DFD diagram of Fenrir process.

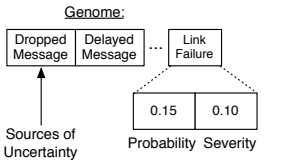


Fig. 4. Genome representation.

(1) Generate Operational Contexts. FENRIR uses novelty search [17], an evolutionary computation-based technique, to generate operational contexts that specify the sources of environmental and system uncertainty. Operational contexts are represented as genomes within a population. Each genome comprises a vector of genes of length n, where n defines the number of environmental and system sources of uncertainty. Each gene defines the likelihood and impact of occurrence for each source. Figure 4 illustrates a sample genome used by FENRIR to configure sources of uncertainty. For instance, the displayed genome has a parameter for a network link failure that has a 15% chance of occurrence, and, at most, 10% of all network links within the RDM network can fail at any given time step. Each generated operational context is then applied to an instrumented DAS, resulting in an execution trace. Both the operational context and execution

trace are provided as inputs to the novelty value calculation, as is described in the following subsection.

Novelty search is similar in approach to genetic algorithms [11], however it differs in the search objective. Novelty search aims to create a set of diverse solutions that are representative of the solution space, whereas a genetic algorithm searches instead for an optimal solution. The novelty search process is constrained by a set of parameters that govern how new solutions are created. These include *population size*, *crossover* and *mutation rates*, a *termination criterion*, and a *novelty threshold value*. Population size determines the number of genomes created per generation, and a starting population is randomly generated that specifies different sources of uncertainty based on the system and environmental conditions. The crossover and mutation rates define the number of new genomes that may be created through recombination and random modifications, respectively. The termination criterion defines the number of generations that the novelty search algorithm will run before termination, and the novelty threshold provides a baseline value for inclusion of a solution within the novelty archive. New genomes are created in each subsequent generation via the crossover and mutation operators. Crossover creates new genomes by swapping genes between two candidate genomes, and mutation produces a new genome by randomly mutating a gene within the original candidate.

(2) Compute Novelty and Archive Solutions. FENRIR calculates a novelty value for each solution within a population by first constructing a weighted call graph (WCG) [1] from each corresponding execution trace then calculating the difference against every other solution within the novelty archive in a pair-wise fashion. The WCG is an extension to a program call graph [27] and is represented as a directed graph, with nodes populated by unique identifiers corresponding to each logging statement, directed edges symbolizing execution order, and weights representing execution frequency. Figures 5(a) and 5(b) present an example of a WCG with corresponding example code, respectively, where each node represents a statement from the execution trace, and each edge label represents the execution frequency (i.e., weight).

The novelty value is computed by calculating the differences in nodes and edges between two WCGs, as shown in Equation 1, and then applying a Manhattan distance metric [3] to measure the distance between each WCG, as shown in Equation 2. Any novelty value that exceeds the novelty archive threshold, or is within the top 10% of all novelty values, is then added to the novelty archive at the end of each generation.

$$dist(\mu_i, \mu_j) = len(\{v \in g_i\} \oplus \{v \in g_j\}) + len(\{e \in g_i\} \oplus \{e \in g_j\})) \quad (1)$$

$$p(\overline{\boldsymbol{\mu}}, k) = \frac{1}{k} \sum_{i=1}^{k} dist(\mu_i, \mu_j) \quad (2)$$

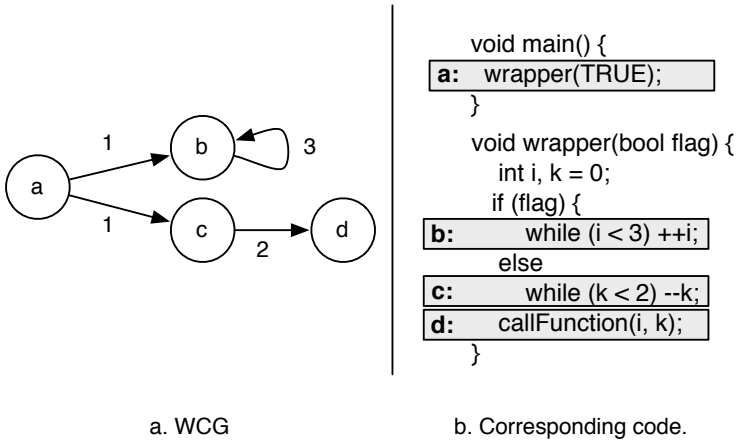


Fig. 5. WCG Example.

Upon completion, FENRIR returns a set of operational contexts, each with their corresponding execution trace stored in the novelty archive. Together, these outputs provide insight into the behavior of the DAS throughout execution, including triggers to self-reconfigurations, parameters for each target configuration, raised exceptions, and unwanted or unnecessary adaptations. Unnecessary adaptations refer to adaptations that may occur as the DAS transitions back and forth between target configurations before finally settling on a new target configuration to handle the current operating context. Unacceptable behaviors may then be corrected through bug fixes, augmentation of target configurations, or by introducing satisfaction methods such as RELAX [6, 33] that tolerate flexibility in DAS requirements.

4 Experimental Results

This section describes our experimental setup and discusses the experimental results found from applying FENRIR to an RDM application.

4.1 Experimental Setup

For this work, we implemented an RDM network as a completely connected graph. Each node represents an RDM and each edge represents a network link. Our network was configured to comprise 25 RDMs and 300 network links that may be activated and used to transfer data between RDMs. Logging statements comprise a unique identifier, module information such as function name, line number, and a custom description, and are inserted into the RDM source code to properly generate an execution trace. The RDM was executed for 150 time steps, with 20 data items randomly inserted into varying RDMs that were then responsible for distribution of those data items to all other RDMs. Furthermore,

the novelty search algorithm was configured to run for 15 generations with a population size of 20 individual solutions per generation. The crossover, mutation, and novelty threshold rates were set to 25%, 50%, and 10%, respectively.

Environmental uncertainties, such as dropped messages or unpredictable network link failures, can be applied to the RDM network throughout a given execution. The RDM network may then self-adapt in response to these adverse conditions in order to properly continue its execution. A self-adaptation results in a target system configuration and series of reconfiguration steps that enables a safe transition of the RDM network from the current configuration to target configuration. This adaptation may include updates to the underlying network topology, such as changing to a minimum spanning tree, or updating network propagation parameters, such as moving from synchronous to asynchronous propagation.

In order to validate our approach, we compared and evaluated the resulting execution traces produced by FENRIR with the novelty metric previously introduced in Equations 1 and 2. To demonstrate the effectiveness of novelty search, we compared FENRIR execution traces with those generated for random operational contexts. We compared FENRIR results to random testing since we could not define an explicit fitness function because we do not know *a priori* which operational contexts adversely impact the system. As such, FENRIR provides a means for us to consider a representative set of all possible operational contexts. For statistical purposes, we conducted 50 trials of each experiment and, where applicable, plotted or reported the mean values with corresponding error bars or deviations.

4.2 DAS Execution in an Uncertain Environment

For this experiment, we define the null hypothesis H_0 to state that there is no difference between execution traces generated by configurations produced by novelty and those created by random search. We further define the alternate hypothesis H_1 to state that there is a difference between execution traces generated from novelty search (FENRIR) and random search.

Figure 6 presents two box plots with the novelty distances obtained by the novelty archive generated by FENRIR and a randomized search algorithm. As this plot demonstrates, FENRIR generated execution traces that achieved statistically significant higher novelty values than those generated by a randomized search algorithm ($p < 0.001$, Welch Two Sample t-test). This plot also demonstrates that FENRIR discovered execution traces with negative kurtosis, thereby suggesting that the distribution of operational contexts were skewed towards larger novelty values. These results enable us to reject our null hypothesis, H_0. Furthermore, these results enable us to accept our alternate hypothesis, H_1, as novelty search discovered a significantly larger number of unique DAS execution paths when compared to the randomized search algorithm. Figure 6 also demonstrates that the solutions generated by FENRIR provide a better representation of the solution space with fewer operational contexts, as the FENRIR box plot contains novelty values from 30 solutions, and the randomized search box plot contains novelty values from 300 solutions. As such, using FENRIR enables a DAS developer to

assess behavior assurance of a DAS in uncertain environments more efficiently, both in terms of computation time and information to be analyzed.

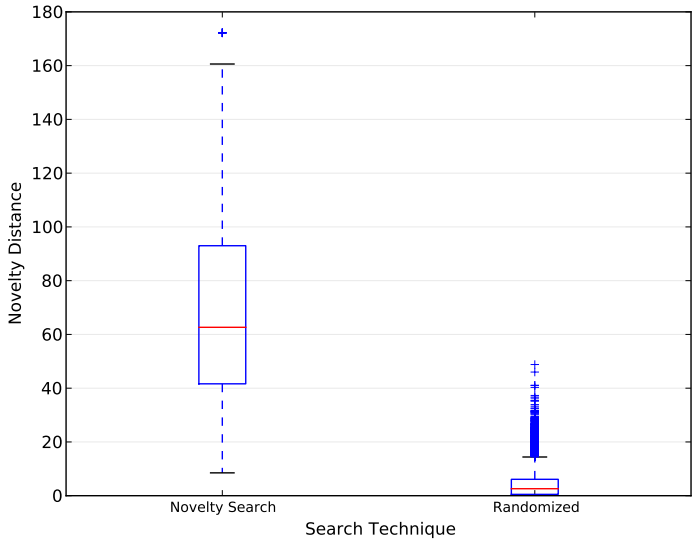
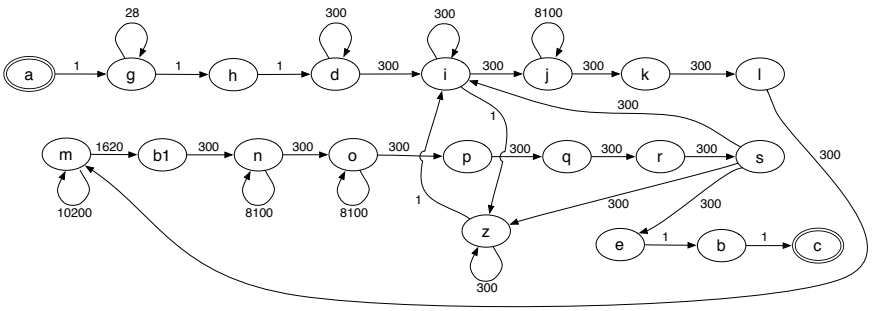


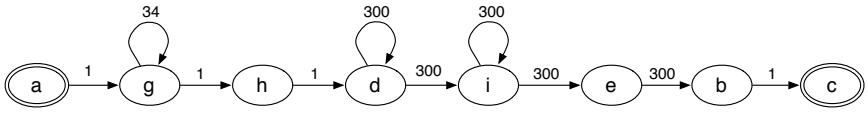
Fig. 6. Novelty distance comparison between novelty and random search.

Figure 7 presents two separate RDM network execution paths that were generated by novelty search, with each represented as a WCG. Each node represents a unique logging point and each directed edge represents a sequential execution from one node to the next. The weight on each edge indicates the frequency that the statement was executed. For instance, in Figure 7(a), the weight on the edge between Nodes (g) and (h) shows that Node (g) was executed 28 times and then Node (h) was executed once. Further analysis of the execution trace indicates that the RDM network consisted of 28 data mirrors. Visual inspection of Figures 7(a) and 7(b) indicates that FENRIR is able to find execution paths that vary greatly in both structure and in frequency of executed instructions. The large variance in structure helps us to better understand the complexity of the DAS behavior in response to different operational contexts. Furthermore, the diversity of execution traces can be used to focus our efforts in revising the functional and/or adaptive logic in order to reduce complexity, optimize configurations, or repair erroneous behavior and code.

Threats to Validity. This research was a proof of concept study to determine the feasibility of using execution trace data for a DAS to determine what evolutionary computation-generated system and system and environmental conditions warrant dynamic adaptation. We applied our technique to a problem that



(a) Execution Path 1



(b) Execution Path 2

Fig. 7. Unique execution paths.

was provided by industrial collaborators. Threats to validity include whether this technique will achieve similar results with other DAS implementations and other problem domains. Furthermore, as an optimization to maintain trace files that are manageable in size, we focused on coverage points rather than providing full branching code coverage. As such, exploration of additional code coverage may be necessary to provide extended information on the generated execution paths.

5 Related Work

This section presents related work on approaches for providing code coverage, evolving randomized unit test data, automated methods for testing distributed systems, and automatically exploring how uncertainty affects requirements.

Code Coverage. Assurance of a system can be augmented by providing a measure of code coverage testing. Chen *et al.* [5] proposed code coverage as an approach for enhancing the reliability measurements of software systems during testing. A software system may successfully pass all test cases in a testing suite and yet can still have latent errors. Augmenting traditional testing with code coverage analysis can improve testing reliability. Furthermore, instrumenting software to provide code coverage can be a non-trivial task, incurring extra time and overhead. Tikir and Hollingsworth [32] have introduced an approach that can dynamically insert and remove logging calls in a codebase. Moreover, optimizations to traditional logging techniques were introduced in order to reduce both the number of instrumentation points and program slowdown. Finally, automated code coverage, as well as automated model coverage, can be provided via gray-box testing [14]. Gray-box testing can be provided through a combination of white-box parameterized unit testing and black-box model-based testing.

In this approach, oracle-verified test sequences are combined with a suite of parameter values to maximize code coverage, and provides insights into system behaviors at both the model and code levels. Each of these approaches is concerned with providing an overall measure of code coverage, where FENRIR targets code segments that provide differing paths of execution in the DAS, including branching and self-reconfiguration paths, thereby providing a finer-grained measure of execution.

Evolved Randomized Unit Testing. A diverse set of system tests can be created automatically with evolutionary computation. Nighthawk [2] uses a genetic algorithm to explore the space of parameters that control the generation of randomized unit tests with respect to fitness values provided by coverage and method invocation metrics. EvoSuite [9] uses evolutionary algorithms to create whole test suites that focus on a single coverage criterion (i.e., introducing artificial defects into a program). In contrast to each of these approaches, FENRIR instead provides feedback through execution traces to demonstrate the vast amount of possible states that a DAS may encounter at run time, as opposed to defining test suites for validation.

Automated Testing of Distributed Systems. Distributed systems comprise asynchronous processes that can also send and receive data asynchronously, and as a result can contain a large number of possible execution paths. Sen and Agha [30] have proposed the use of *concolic* execution, or simultaneous concrete and symbolic execution, in order to determine a partial order of events incurred in an execution path. Concolic execution was proven to efficiently and exhaustively explore unique execution paths in a distributed system. Furthermore, automated fault injection can explore and evaluate fault tolerance to ensure that a distributed system continually satisfies its requirements specification [8] and ensures system dependability [29]. Conversely, FENRIR explores how a system can handle faults by studying its reaction to varying operational contexts, rather than by direct injection of faults into the system.

Automatically Exploring Uncertainty in Requirements. Ramirez *et al.* [26] introduced Loki, an approach for creating novel system and environmental conditions that can affect DAS behavior, and in doing so uncover unexpected or latent errors within a DAS's *requirements specification*. FENRIR extends Loki by exploring uncertainty at the code level in an effort to distinguish how a DAS will react in uncertain situations and attempt to uncover errors made in the *implementation* of a DAS.

6 Conclusion

In this paper we presented FENRIR, an approach that applies novelty search at design time to automatically generate operational contexts that can affect a DAS during execution at the code level. Specifically, FENRIR introduces logging statements to trace a DAS's execution path and then uses the distance between execution paths to measure the level of novelty between operational contexts. By creating a set of configurations that more extensively exercise a DAS, it is pos-

sible to identify undesirable behaviors or inconsistencies between requirements and system implementation. We demonstrated the use of Fenrir on an RDM network that was responsible for replicating data across a network. This network was subjected to uncertainty in the form of random link failures and dropped or delayed messages. Experimental results from this case study established that Fenrir was able to successfully generate more unique execution paths with a smaller set of execution traces than purely random search. Future directions for Fenrir will extend the technique and apply it to additional applications. We are investigating different distance metrics for novelty search as well as other evolutionary strategies to generate unique execution traces.

Acknowledgements

We gratefully acknowledge conceptual and implementation contributions from Jared M. Moore.

This work has been supported in part by NSF grants CCF-0854931, CCF-0750787, CCF-0820220, DBI-0939454, Army Research Office grant W911NF-08-1-0495, and Ford Motor Company. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Army, Ford, or other research sponsors.

References

1. Shin-Young Ahn, Sungwon Kang, Jongmoon Baik, and Ho-Jin Choi. A weighted call graph approach for finding relevant components in source code. In *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09. 10th ACIS International Conference on*, pages 539–544, 2009.
2. James H. Andrews, Tim Menzies, and Felix C.H. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, January 2011.
3. Paul E. Black. *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, May 2006.
4. J. Camara and R. de Lemos. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *Software Engineering for Adaptive and Self-Managing Systems.*, pages 53–62, June 2012.
5. M-H Chen, Michael R Lyu, and W Eric Wong. Effect of code coverage on software reliability measurement. *IEEE Transactions on Reliability*, 50(2):165–170, 2001.
6. Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Lecture Notes in Computer Science, pages 468–483, Denver, Colorado, USA, October 2009. Springer-Verlag.
7. Betty HC Cheng, Rogerio De Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, et al. *Software engineering for self-adaptive systems: A research roadmap*. Springer, 2009.

8. Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 404–414. IEEE, 1996.
9. Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 416–419, Szeged, Hungary, 2011. ACM.
10. R. Gupta, A.P. Mathur, and M.L. Soffa. Generating test data for branch coverage. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 219–227, 2000.
11. John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.
12. Minwen Ji, Alistair Veitch, and John Wilkes. Seneca: Remote mirroring done write. In *USENIX 2003 Annual Technical Conference*, pages 253–268, Berkeley, CA, USA, June 2003. USENIX Association.
13. Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.
14. Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann, and Victor Braberman. Achieving both model and code coverage with automated gray-box testing. In *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing*, pages 1–11. ACM, 2007.
15. M. Lajolo, L. Lavagno, and M. Rebaudengo. Automatic test bench generation for simulation-based validation. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 136–140, San Diego, California, United States, 2000. ACM.
16. Yves Ledru, Alexandre Petrenko, and Sergiy Boroday. Using string distances for test case prioritisation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE'09, pages 510–514, Auckland, New Zealand, November 2009. IEEE Computer Society.
17. Joel Lehman and Kenneth O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *Proceedings of the Eleventh International Conference on Artificial Life (ALIFE XI)*, Cambridge, MA, USA, 2008. MIT Press.
18. P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56 – 64, July 2004.
19. J Moc and David A Carr. Understanding distributed systems via execution trace data. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, pages 60–67. IEEE, 2001.
20. Cu D. Nguyen, Anna Perini, Paolo Tonella, and Fondazione Bruno Kessler. Tonella p., automated continuous testing of multiagent systems. In *Fifth European Workshop on Multi-Agent Systems (EUMAS)*, 2007.
21. Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 521–528, Budapest, Hungary, May 2009. International Foundation for Autonomous Agents and Multiagent Systems.
22. Duy Cu Nguyen, Anna Perini, and Paolo Tonella. A goal-oriented software testing methodology. In *Proceedings of the 8th International Conference on Agent-oriented software engineering VIII*, pages 58–72, Berlin, Heidelberg, 2008. Springer-Verlag.

23. Alex Ramírez, Josep-L Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero. Software trace cache. In *Proceedings of the 13th International Conference on Supercomputing*, pages 119–126. ACM, 1999.
24. Andres J. Ramirez and Betty H.C. Cheng. Verifying and analyzing adaptive logic through uml state models. In *Proceedings of the 2008 IEEE International Conference on Software Testing, Verification, and Validation*, pages 529–532, Lillehammer, Norway, April 2008.
25. Andres J. Ramirez, Erik M. Fredericks, Adam C. Jensen, and Betty H. C. Cheng. Automatically relaxing a goal model to cope with uncertainty. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 198–212. Springer Berlin Heidelberg, 2012.
26. Andres J. Ramirez, Adam C. Jensen, Betty H.C. Cheng, and David B. Knoester. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *Proceedings of the 2011 International Conference on Automatic Software Engineering*, ASE'11, Lawrence, Kansas, USA, November 2011.
27. B.G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, SE-5(3):216–226, 1979.
28. Pete Sawyer, Nelly Bencomo, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re self-adaptive systems. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 95–103, Sydney, Australia, September 2010.
29. Z Segall, D Vrsalovic, D Siewiorek, D Yaskin, J Kownacki, J Barton, R Dancey, A Robinson, and T Lin. Fiat-fault injection based automated testing environment. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 102–107. IEEE, 1988.
30. Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.
31. D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium*, pages 91 –100, 2000.
32. Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.
33. Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In *Proceedings of the 17th International Requirements Engineering Conference (RE '09)*, pages 79–88, Atlanta, Georgia, USA, September 2009.
34. Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.
35. Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software engineering*, ICSE '06, pages 371–380, Shanghai, China, 2006. ACM.