BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2026
http://excel.fit.vutbr.cz

Eliot: A Tool for Generating Code from Declarative Fault Models

Samuel Štefánik*

Abstract
Defining fault injection scenarios in network communication often requires manual implementation, which is error-prone and limits flexibility. This work presents Eliot, a tool that enables declarative specification of fault models in YAML or JSON and their transformation into efficient C++ code using a DAG-based representation. The generated implementation removes runtime interpretation overhead, ensures deterministic behavior, and achieves up to $2.6\times$ higher throughput compared to the initial, non-optimized version of the generated code. The approach simplifies the design of complex fault scenarios while demonstrating that code generation combined with targeted optimizations can significantly improve performance.

*xstefas00@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Fault injection is a common technique for testing the robustness of networked systems under non-ideal conditions, such as packet loss, delays, or reordering [1]. In practice, tools such as NetLoiter [2] provide two main approaches to defining fault behavior. A higher-level interface enables flexible configuration but introduces performance overhead, while a lower-level implementation in C++ achieves high throughput at the cost of requiring manual implementation of the fault logic.

This creates a trade-off between flexibility and performance. Defining more complex scenarios, including stateful or probabilistic behavior, becomes difficult when implemented manually, while interpreted approaches are not suitable for high-performance packet processing. As a result, there is a need for a solution that allows high-level specification of fault models without sacrificing execution efficiency.

This work presents Eliot, a tool for automated generation of source code from declarative fault models. Fault behavior is described using a YAML-based specification, which is internally transformed into a directed acyclic graph representation. Based on this representation, the tool generates C++ code that can be compiled and used within an external fault injection framework.

The proposed approach removes the need for manual low-level implementation while preserving the performance characteristics of compiled code.

2. System overview

This section outlines the proposed approach, including the transformation pipeline, model structure, and runtime behavior.

2.1 Processing pipeline

The overall workflow of the proposed approach is illustrated in Figure 1. The process starts with a declarative specification of a fault model written in YAML or JSON. This specification describes conditions, actions, and state transitions that define how network traffic should be affected.

The input model is first parsed and validated to ensure structural correctness. It is then transformed into an internal representation in the form of a directed acyclic graph (DAG). Each node in the graph represents either a condition, an action, or a state transition.

Based on this representation, source code is generated in C++. The generated code follows the structure of the DAG and implements the described behavior without the need for runtime interpretation. The resulting implementation is compiled and used together with an external fault injector, which provides access to network traffic.

An example of the transformation from YAML to the DAG representation is shown in Figure 2, together with a simplified pseudocode of the generated output.

2.2 Model functionality

The declarative model allows describing fault behavior using a combination of conditions, actions, value generators, and state nodes, as illustrated in the functionality overview on the poster.

Conditions define when a rule is applied. They can operate on packet attributes or external context and support both stateless and stateful evaluation. In addition, conditions may incorporate time-based or probabilistic logic, enabling more realistic fault scenarios.

Actions specify how packets are modified once a condition is satisfied. Supported operations include dropping, delaying, reordering, or modifying packets. More advanced behavior, such as replication or throttling, can also be expressed within the model.

Value generators provide dynamic parameters for conditions and actions. These generators allow the use of random distributions or time and sequence-based values, making it possible to model non-deterministic or evolving behavior over time.

State nodes enable the definition of internal state within the model. The behavior of the system can change depending on the current state, and transitions between states are triggered by actions. This allows expressing complex scenarios that depend on previous events or accumulated conditions.

2.3 Processing and scheduling

The runtime behavior of the generated implementation is illustrated in Figure 3. Packet processing is driven by a loop that reacts to incoming traffic and scheduled events. For each packet, the fault model is evaluated, and the corresponding actions are executed based on the current state and defined conditions.

Some actions, such as delaying packets, require scheduling their future processing. This is handled by an internal scheduling mechanism, which ensures that packets are released at the correct time. The processing loop, therefore, alternates between handling newly intercepted packets and processing scheduled ones.

The concept of packet scheduling is shown in Figure 4. Packets can be deferred and later reinserted into processing according to their assigned delay. To efficiently manage scheduled events, a hierarchical timer wheel structure [3] is used, as illustrated in Figure 5. This structure enables efficient handling of a large number of timed events with minimal overhead.

By combining immediate processing with scheduled execution, the system is able to model complex temporal behavior while maintaining high performance.

3. Performance evaluation

The performance evaluation of the generated code is shown in Figure 6, Figure 7 and Table 1. The results compare different versions of the generated code and highlight the impact of the applied optimizations.

The evaluation focuses on the effect of optimization techniques applied to the generated code. The results show that these optimizations lead to a significant increase in throughput. In particular, the optimized version achieves up to $2.6\times$ higher performance compared to the initial non-optimized version.

These results demonstrate that the proposed approach improves usability and allows further performance tuning through code generation and optimization.

4. Conclusions

This work introduced Eliot, a tool for generating efficient implementations of network fault models from a declarative specification. The approach combines a high-level YAML-based language with a transformation pipeline that produces optimized C++ code.

The main benefit of the approach is the removal of the trade-off between flexibility and performance present in existing solutions. Users can define complex, stateful, and probabilistic fault scenarios without the need for manual low-level implementation while still achieving high runtime efficiency.

The results show that the generated code can be further optimized, leading to significant performance improvements compared to the initial implementation. This confirms that code generation is a suitable approach for bridging the gap between usability and performance in fault injection systems.

Future work may focus on extending the expressiveness of the specification language and improving integration with existing fault injection frameworks.

Acknowledgements

I would like to thank my supervisor Ing. Michal Rozsíval for his guidance, valuable insights, and support throughout the development of this work.

References

[1] Michal Rozsíval, Aleš Smrčka, and Tomáš Vojnar. Automated testing of reliability of networked systems. In *2024 IEEE 17th International Scientific Conference on Informatics (Informatics)*, pages 302–307, Poprad, 11 2024. IEEE.

[2] Michal Rozsíval and Aleš Smrčka. Netloiter: A tool for automated testing of network applications using fault-injection. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 207–210. IEEE, 2023.

[3] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking*, 5(6):824–834, 1997.