Model-Based Testing of Non-Deterministic Systems

Alexander Onofrei, Marc Frappier, and Émilie Bernard

University of Sherbrooke, Sherbrooke, Canada
{onoa2801,marc.frappier,emilie.bernard4}@usherbrooke.ca

Abstract. Testing non-deterministic systems is challenging due to unpredictable behaviours arising from timing, concurrency, and random inputs. This paper explores the application of model-based testing (MBT) to tackle these challenges, leveraging formal methods and tools to ensure systematic test coverage. We employ ProB, a model checker for the B method, to analyse a formal model of the system under test (SUT) and generate test scenarios from the formal B model. As a proof of concept, we apply MBT to the TLS 1.3 protocol, a widely used complex cryptographic standard, and test one of its implementation using the BouncyCastle OpenSSL Java API. While the TLS handshake is primarily deterministic, it includes non-deterministic components like cipher selection and random value generation, making it an excellent candidate for evaluating MBT's effectiveness. We present the design and logic of our proof of concept, showcasing its flexibility to support various models and SUTs. This study demonstrates that combining formal methods, non-deterministic analysis, and state-based testing can effectively address the challenges of non-deterministic systems, enabling improved testing strategies and greater system reliability.

1 Introduction

Models are essential in software development, particularly as system complexity increases. They abstract intricate architectures, employing tools like graphs and finite state machines (FSMs) to predict behavior [2]. Ideally, a model fully represents the implementation under test (IUT), yet non-deterministic FSMs remain constrained [14]. While effective for systematic testing, real-world systems—especially distributed architectures and protocols like TLS—introduce external dependencies that complicate verification [10].

This paper presents a proof of concept: testing non-deterministic systems using B, a model-based specification method [1]. We construct an abstract B machine to model key behaviors of the system under test and develop a test generator using ProB [15] to verify the correctness of the SUT. To validate this approach, we apply it to the TLS 1.3 session protocol [21], a highly complex standard. Using a simplified model of TLS, we test the BouncyCastle OpenSSL Java API implementation [24]. The challenge in MBT of non-deterministic systems is that the test scenario is constructed online during the execution of a test, since

the next event to generate depends on the output that was non-deterministically chosen by the SUT. For example, in TLS, the response of the client depends on the response chosen by the server.

Testing, model checking, and formal proofs each offer distinct advantages and limitations. Formal proofs provide mathematical certainty but struggle with scalability and practical implementation. Model checking systematically explores all possible states but is constrained by state-space explosion [8]. Testing, while less exhaustive, remains the most practical method for verifying real-world implementations. Model-based testing bridges these approaches by automating test generation from formal models, reducing human error and increasing coverage compared to manually derived test cases. This makes MBT particularly valuable for testing non-deterministic systems.

2 Related Work

Research on testing nondeterministic systems remains limited, with few studies examining model-based approaches in this context. Key challenges include test model coverage under uncertainty, scenario generation via model checkers, and efficient testing strategies. [11] provides foundational insights, highlighting methods for handling concurrency, randomness, and the selection of test conditions. We tackle these issues using ProB, which offers greater expressiveness than FSMs, because of their lack of state variables—allowing the extraction of specific paths that satisfy a given predicate.

In MBT, nondeterministic models arise from abstraction or inherent software behavior. A single test case may follow multiple paths due to internal system decisions [12, 13]. Techniques such as probabilistic model checking and transition annotations mitigate this problem, but ensuring sufficient test coverage remains a challenge [20, 4]. By deriving tests directly from the model within the model checker, we ensure comprehensive scenario handling. To improve testing efficiency, innovative techniques have been proposed. For example, integrating model checkers with mutation analysis enables the generation of systematic tests by injecting faults into the system models [5]. We address this by leveraging the ProB API to generate abstract test cases based on specific test criteria and analyze logical operations within the model. Other approaches, such as [17], optimize state graph-based test generation to balance coverage and test set size, critical for multi-threaded and distributed systems. Providers, such as Entrust [9], offer SSL/TLS tools to help organizations assess and enhance digital security by evaluating configurations on both client and server sides, including verifying compliance with encryption algorithms.

Combinatorial testing has been used for TLS [23, 16],and several errors were found. We hope that a ProB MBT approach will enable us to find interesting combination of parameters using predicate analysis and logic solving that are not covered by traditional combinatorial testing, as shown in [22]

3 Foundations

3.1 B and ProB

B is a formal method for system specification, design and implementation, providing precise modeling through abstract machines and refinement [1]. It ensures correctness and reliability, making it essential for safety-critical systems. In this project, B is used to formally model the SUT. ProB, a model checker and animator, enables dynamic analysis, model execution, and real-time verification [15]. The ProB Java API [3] is integral to our proof of concept, facilitating interaction with the B model. It is used to generate targeted test cases for specific traces while also identifying states that satisfy given predicates.

3.2 TLS Protocol

TLS (Transport Layer Security) is a widely adopted cryptographic protocol designed to ensure secure communication over computer networks. It guarantees data integrity, confidentiality, and authenticity between a client and a server. TLS is a cornerstone of secure online interactions, used extensively in web browsers, email systems, and various online services due to its ability to prevent eavesdropping, tampering, and forgery.

The TLS handshake, which is the first phase of communication between a client and server, plays a crucial role in establishing a secure session. This process involves authentication, key exchange, and negotiation of session parameters. While largely deterministic, the handshake includes non-deterministic choice of elements such as cipher selection and initial random values. These steps can be summarized with the sequence diagram provided in Fig. 1. The Client Hello and Server Hello messages, along with the encrypted extensions, contain the essential parameters required to establish a basic TLS communication. These parameters include the supported versions, cipher suites, signature algorithms, supported groups, and shared keys, among others [21].

These nondeterministic factors necessitate careful testing to confirm the protocol's reliability and security. Thus, testing TLS for its non-deterministic components is essential to ensure it can correctly handle unexpected behaviors.

4 Methodology and Design

Our MBT approach for non-deterministic systems relies on black-box testing. This means we evaluate whether a system adheres to a predefined model based on its specification and rules, without considering its internal implementation. To achieve this, we integrate our Java implementation with the ProB model checker via the ProB Java API. This integration allows us to control, execute, and retrieve information from the model checker directly within our application, streamlining test selection and execution. The SUT in our study is the BouncyCastle OpenSSL Java Library, which will be evaluated based on model-derived tests.

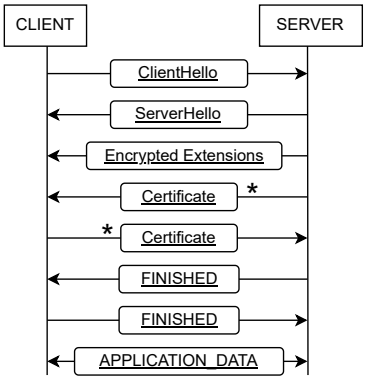

Fig. 1. TLS 1.3 Sequence Diagram

This methodology enables testing of both the client and server sides of a TLS implementation. When testing the server, ProB generates a client message, which is sent to the server. The server's response is then compared against the expected output defined by the specification. A match indicates a successful test, while any deviation results in a failed test. To test the client, the process is reversed, with the server-side behavior being modeled and validated against expected responses.

The sequence diagram provided in Fig. 2 illustrates the interactions between our application components during server-side testing. Notably, the same diagram applies to client-side testing, with the only modification being the entity that initiates the first message. This implementation is designed for flexibility, facilitating adaptation to any SUT. To achieve this, we structured each component in a generalized framework, enabling adaptation based on the specific requirements of the SUT. Since our specification abstracts from implementation details, we developed a test instantiator to convert abstract tests—generated by the test scenario generator—into concrete, executable tests. The scenario generator performs operations such as state predicate satisfaction and random trace generation. Each test consists of a sequence of events and corresponding operations with assigned parameters. An Information Capture module collects the SUT's outputs, feeding them back to the Test Instantiator, which converts them into abstract results for comparison against the specification's expected results. If the result is one of them, the test is passed and the next operation is executed. Otherwise, the test case is marked as a failure.

As a proof of concept, we test the implementation of the ClientHello and ServerHello messages in TLS 1.3. Each message is represented by a send operation for the issuer of the message, and a receive operation for the recipient, which will enable us to model man-in-the-middle attacks. The model initially generates the ClientHello message, using the SendClientHello operation, in an abstract format, which is then translated into a concrete TLS ClientHello mes-

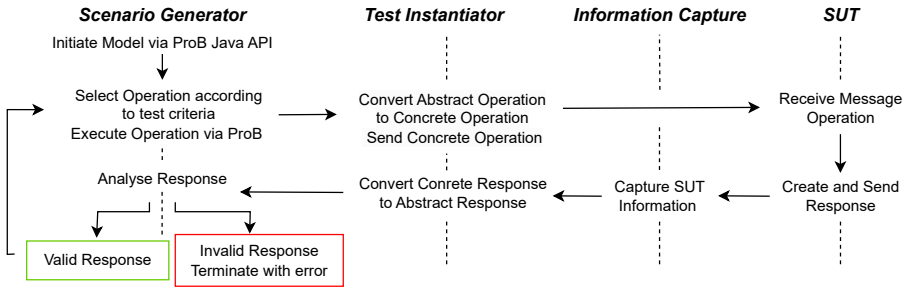

Fig. 2. Architecture of our MBT testing approach

sage. The ServerHello response from the SUT is converted back into an abstract representation and compared against the expected abstract message generated by the model.

For our proof of concept, we selected a typical sequence of operations within the model: SendClientHello, ReceiveClientHello, SendServerHello, ReceiveServerHello. The Send operations include parameters corresponding to those in a concrete TLS message, as specified in [21]. The specification enforces basic parameter validation to ensure that values are within acceptable bounds. The RFC allows for interpretation by using terms such as "SHOULD", "SHOULD NOT" and "MAY" [6], which introduce flexibility in compliance. As a result, different implementations of the TLS protocol can be derived from the same RFC, leading to variations in behavior. This might also be a source of bugs and non-interoperability, since one party's implementation may take a "SHOULD" as a practical "MUST", and the other party's implementation taking "SHOULD" as something really optional as stated in [6] ("SHOULD ... *mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.*").

5 Results

After testing and comparing BouncyCastle's server response to specific Client Hello messages generated by our model checker, we successfully generated 16 distinct ClientHello messages with varying parameters to evaluate the corresponding ServerHello responses. The generated messages included 5 cipher suites, 8 signature algorithms, 2 supported groups, 1 supported versions TLS 1.3, and 2 compression methods. The complete results are available in our GitHub [19] repository. In all 16 test cases, the server responses were consistent with the model's predictions, confirming that our specification accurately represents the SUT and that the SUT exhibits no flaws within the tested parameters.

Given the complexity of the task, we focused solely on modifying the Client Hello parameters, limiting our scope to analyzing the server's response. Nevertheless, this study demonstrates the feasibility and effectiveness of our test generation approach. Furthermore, it highlights the potential for expanding test coverage to more intricate aspects of TLS, such as certificate validation and key exchange mechanisms.

6 Discussion

Our approach opens-up new possibilities for systematically testing complex system like TLS. This becomes particularly important since TLS will have to support in the near future new quantum resistant cryptographic primitives, called post-quantum cryptography [18]. A transition period, where both classic and quantum resistant interactions must be supported by clients and servers, will induce new possibilities for attacks, bugs and interoperability issues. Model checkers can systematically explore possible input combinations to produce expected outputs, uncovering test cases that might otherwise be overlooked. However, our approach comes with certain limitations. MBT is easier to achieve if the SUT has a modular design that allows for an easy re-use of methods that send and receive messages between the SUT and the tester. For instance, BouncyCastle provides a method to send a ClientHello to the server. However, this method cannot be reused easily, because it is highly dependent on the state of the protocol, and it has several dependencies with other methods that must be executed before, but that we do not want to execute, since we use ProB to model and analyse the protocol state and drive the test generation. Sending and receiving TLS messages in the proper format is not an easy task. We had to recode these methods, with very low-level handling of the messages as bit streams, and little code could be re-used from the BouncyCastle implementation. We capture messages from the server on the communication port and manually decode them, thus we must ignore TCP messages and other irrelevant information for just testing the TLS part of the communication. We also considered to reuse the widely used OpenSSL implementation of TLS, but it was not easier, because it is written in very old-school C and harder to understand. It makes heavy use of function pointers and macros, instead of using modern object-oriented programming concepts. Unfortunately, writing the code to send and receive messages in TLS was the most difficult and time-consuming part of our work. In the next version of our implementation, we will explore the TLS-Attacker framework [7], a fuzzy-testing tool, to try to streamline this step. Additionally, refining translation methods between abstract and concrete messages will be crucial for improving automation. By incorporating test criteria, we aim to selectively generate test cases that target specific needs, ensuring comprehensive test coverage and deeper insights into system behavior. Our approach differs from the testing offered by industrial providers, as we specifically test each step of the TLS handshake implementation rather than only analyzing its overall configuration and final communication result [16].

References

[1] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Jan. 2005. ISBN: 978-0-521-02175-3.

[2] Boris Beizer. *Software testing techniques (2nd ed.)* USA: Van Nostrand Reinhold Co., 1990. ISBN: 0442206720.

[3] Jens Bendisposto et al. "ProB2-UI: A Java-Based User Interface for ProB". In: *Formal Methods for Industrial Critical Systems: 26th International Conference, FMICS 2021, Paris, France, August 24–26, 2021, Proceedings.* Paris, France: Springer-Verlag, 2021, 193–201. ISBN: 978-3-030-85247-4. DOI: 10.1007/978-3-030-85248-1_12. URL: https://doi.org/10.1007/978-3-030-85248-1_12.

[4] Nathalie Bertrand et al. "Off-line test selection with test purposes for non-deterministic timed automata". In: *Logical Methods in Computer Science* Volume 8, Issue 4, 8 (2012). ISSN: 1860-5974. DOI: 10.2168/LMCS-8(4:8)2012. URL: https://lmcs.episciences.org/1037.

[5] Sergiy Boroday, Alexandre Petrenko, and Roland Groz. "Can a Model Checker Generate Tests for Non-Deterministic Systems?" In: *Electronic Notes in Theoretical Computer Science* 190.2 (2007). Proceedings of the Third Workshop on Model Based Testing (MBT 2007), pp. 3–19. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2007.08.002. URL: https://www.sciencedirect.com/science/article/pii/S1571066107005373.

[6] Scott O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* RFC 2119. Mar. 1997. DOI: 10.17487/RFC2119. URL: https://www.rfc-editor.org/info/rfc2119.

[7] Fabian Bäumer et al. "TLS-Attacker: A Dynamic Framework for Analyzing TLS Implementations". In: *Proceedings of Cybersecurity Artifacts Competition and Impact Award (ACSAC '24).* 2024.

[8] Edmund Clarke et al. "Model Checking and the State Explosion Problem". In: Jan. 2012, pp. 1–30. ISBN: 978-3-642-35745-9. DOI: 10.1007/978-3-642-35746-6_1.

[9] Entrust Corporation. *Entrust SSL/TLS Tools.* Accessed: 2025-02-24. 2025. URL: https://www.entrust.com/knowledgebase/ssl/ssl-tls-tools.

[10] Donald Firesmith. "Testing in a Non-Deterministic World". In: *SEI Conference.* 2017.

[11] D. Graham. *Foundations of Software Testing: ISTQB Certification.* Course Technology Cengage Learning, 2008. ISBN: 9781283285186. URL: https://books.google.ca/books?id=h6h2AQAACAAJ.

[12] Natalia Kushik, Nina Yevtushenko, and Jorge López. "Probabilistic Approach for Minimizing Checking Sequences for Non-deterministic FSMs". In: *Testing Software and Systems.* Ed. by Silvia Bonfanti, Angelo Gargantini, and Paolo Salvaneschi. Cham: Springer Nature Switzerland, 2023, pp. 237–243. ISBN: 978-3-031-43240-8.

[13] Natalia Kushik, Nina Yevtushenko, and Jorge López. "Testing Against Non-deterministic FSMs: A Probabilistic Approach for Test Suite Mini-

mization". In: *Testing Software and Systems*. Ed. by David Clark, Hector Menendez, and Ana Rosa Cavalli. Cham: Springer International Publishing, 2022, pp. 55–61. ISBN: 978-3-031-04673-5.

[14] D. Lee and M. Yannakakis. "Principles and methods of testing finite state machines-a survey". In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956.

[15] Michael Leuschel and Michael Butler. "ProB: an automated analysis toolset for the B method". In: *Int. J. Softw. Tools Technol. Transf.* 10.2 (Feb. 2008), 185–203. ISSN: 1433-2779.

[16] Marcel Maehren et al. "TLS-Anvil: Adapting Combinatorial Testing for TLS Libraries". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 215–232. ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/maehren.

[17] Lev Nachmanson et al. "Optimal strategies for testing nondeterministic systems". In: *SIGSOFT Softw. Eng. Notes* 29.4 (July 2004), 55–64. ISSN: 0163-5948. DOI: 10.1145/1013886.1007520. URL: https://doi.org/10.1145/1013886.1007520.

[18] Christian Näther et al. "Migrating Software Systems Toward Post-Quantum Cryptography-A Systematic Literature Review". In: *IEEE Access* 12 (2024), 132107–132126. ISSN: 2169-3536. DOI: 10.1109/access.2024.3450306. URL: http://dx.doi.org/10.1109/ACCESS.2024.3450306.

[19] Alexander Onofrei. *MBT TLS using ProB*. https://github.com/ohnoitsalex/TLSModeling.git.

[20] I. S. W. B. Prasetya and Rick Klomp. "Test Model Coverage Analysis Under Uncertainty". In: *Software Engineering and Formal Methods*. Springer International Publishing, 2019, 222–239. ISBN: 9783030304461. DOI: 10.1007/978-3-030-30446-1_12. URL: http://dx.doi.org/10.1007/978-3-030-30446-1_12.

[21] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[22] Aymerick Savary. "Détection de vulnérabilités appliquée à la vérification de code intermédiaire de Java Card". Theses. Université de Limoges ; Université de Sherbrooke (Québec, Canada), June 2016. URL: https://theses.hal.science/tel-01369017.

[23] Dimitris E. Simos et al. "Testing TLS Using Combinatorial Methods and Execution Framework". In: *Testing Software and Systems*. Ed. by Nina Yevtushenko, Ana Rosa Cavalli, and Hüsnü Yenigün. Cham: Springer International Publishing, 2017, pp. 162–177. ISBN: 978-3-319-67549-7.

[24] The Legion of the Bouncy Castle, Inc. *Bouncy Castle Java Library*. https://www.bouncycastle.org/download/bouncy-castle-java/. Accessed: 2025-02-25. 2025.