arXiv:2505.09769v2 [cs.SE] 30 Jun 2025

Automated Statistical Testing and Certification of a Reliable Model-Coupling Server for Scientific Computing

Seth Wolfgang[1], Lan Lin[2], Fengguang Song[1]

[1]*Luddy School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN, USA*
{seawolfg, fgsong}@iu.edu
[2]*Department of Computer Science, Ball State University, Muncie, IN, USA*
llin4@bsu.edu

Abstract

Sequence-based specification and usage-driven statistical testing are designed for rigorous and cost-effective software development, offering a semi-formal approach to assessing the behavior of complex systems and interactions between various components. While this approach has been successfully applied to a number of domains ranging from medical devices to scientific instrumentation, it is particularly valuable for scientific computing applications in which comprehensive tests are needed to prevent flawed results or conclusions. As scientific discovery becomes increasingly more complex, domain scientists couple multiple scientific computing models or simulations to solve intricate multiphysics and multiscale problems. These model-coupling applications use a hardwired coupling program or a flexible web service to link and combine different models. In this paper, we focus on the quality assurance of the more elastic web service by automatically generating, executing, and evaluating 5,204 test cases via a combination of rigorous specification and testing methods. The application of statistical testing exposes problems ignored by pre-written unit tests and highlights areas in the code where failures might occur. We certify the model-coupling server controller with a derived reliability statistic, offering a quantitative measure to support a claim of its robustness.

1 Introduction

Multiscale and multiphysics problems often need to couple different models to address their complex, interactive, and mutually influential natures. One classical scientific computing problem is Earth System Models, such as E3SM [3], which rely on highly specialized couplers to facilitate the exchange of data between participant models. Although domain-specific couplers, like E3SM's CPL7, enable highly efficient data transfer, they are largely inflexible and cannot be utilized with other models outside of their ecosystem.

To help domain scientists integrate various users' models, the NSF's Cyberwater framework [2] is built to use a data-exchange service to connect distinct models that execute in distributed computing systems. As a part of the framework, the Data Exchange Service (DES) is a web service-based coupler that allows the exchange of data between scientific models through a generic service component. The Data Exchange Service promotes interoperability among models, with the exception that the models must adhere to sharing certain units of measurement.

Developers are able to build such a coupling service, however, it is a challenging task to verify the service's reliability and functionality. Considering the criticality of the correctness of scientific computing simulations, rigorous testing methods are essential to ensure data is handled correctly and to certify the reliability of the DES Controller, as coupling logic resides in this component.

This paper is organized as follows. Section 2 provides related work. Section 3 describes our testing methodology. Section 4 and Section 5 show how we apply rigorous specification and testing to the DES Controller, with results in Section 5. Conclusion and future work are given in Section 6.

2 Related Work

Finite State Machines (FSMs) are commonly used to generate test cases for event-driven software. In [17], the

authors use a sequence-based approach to test interactions of shared objects and pages in websites. Others use FSMs to discover navigation errors in web pages [5]. [1] uses combinatorial test generation to create initial test sequences from an FSM and repairs or discards invalid sequences. The authors of [16] use ant colony optimization to minimize the cost of test sequences of a Markov-chain usage model.

T-way sequences [4, 7] are used in combinatorial testing to create a reliable test environment. In [10] the authors apply specification-based testing to cruise control software and record coverage of system interactions and state transitions. Cayley graphs may be used, with respect to a metric, to generate full coverage test sequences as seen in [6].

3 Rigorous Specification and Testing Methodologies

Statistical usage-based testing, combined with *sequence-based specification*, provides a rigorous testing method to systematically examine the behavior of software in all possible real-world usage scenarios, and to assess its reliability based on the testing experience. As outlined in [11, 15], the benefits of statistical testing lie in weighted testing towards the most frequent operational uses of the software. Sequence-based specification, as a *black box* specification method, considers only the external inputs and outputs of the module being tested [9, 14, 15].

3.1 Sequence-Based Specification

First, a *system boundary* is defined to identify the inputs and outputs between the system, the module being tested, and the software's *environment*. The software's environment consists of the interfaces used to communicate with the system.

Next, a functional mapping is created to associate all possible input, or *stimulus*, sequences with their expected outputs, or *responses*, and equivalencies, if applicable, to length-lexicographically smaller sequences. This mapping is discovered through a systematic process called *sequence enumeration* and defines the test oracle for subsequent usage-based statistical testing.

To enumerate, start with the empty sequence λ with length 0. Then we extend each length n sequence by every possible stimulus to get all length $n+1$ sequences and consider them in lexicographical order.

For each new sequence, a decision is made to map to an expected response according to the requirements, and *reduce* to a prior sequence if it takes the system to a previously seen state. Otherwise, the sequence is designated as *unreduced*, serving as an unseen state. Some sequences are *illegal* per software specifications and are denoted as ω in their response and not extended further. Sequences which are reduced or illegal are not extended further. Enumeration is terminated when all enumerated sequences of a certain length are reduced or illegal.

The unreduced and legal sequences are *canonical sequences* which represent unique states within the system. Canonical sequences enable us to construct a *Mealy machine*, a finite-state machine composed of canonical sequences as nodes and arcs defined by stimuli and responses.

3.2 Statistical Testing

A *Markov-chain usage model* is necessary for statistical usage-based testing. By defining probabilities for each arc of the Mealy machine, obtained through specification, a usage model is derived. From each usage state, higher probabilities should be defined on the most common stimuli, allowing these functions to be tested more often. The purpose of the usage model is to characterize a population of all possible and *the most frequent use cases* of the system. To validate the model, standard Markov analysis is performed to determine if the model reflects the expected usage.

Test cases are then generated from the usage model by random, weighted, or coverage sampling. A test case is a sequence of stimuli following the arcs of the usage model, starting from the *source* and ending at the *sink*. At each step, one checks whether the output is expected and if the system is in the correct state. If either output or state is incorrect, the test step fails. If the state is incorrect and the next stimulus is illegal, then the test ends with a *stop failure*, otherwise the failure step is a *continue failure*. To certify the system, quantitative measures, like *Single Use Reliability*, are calculated taking into consideration the usage model, the sample of generated test cases, and test results.

4 Applying Rigorous Specification and Testing to the Data Exchange Service

The DES follows the *Model-View-Controller* pattern, wherein sessions function as the model, the Data Exchange Controller serves as the controller, and clients represent the view. We apply our testing methods to the controller. Correct functionality of the server largely depends on a session's interaction with the controller. To establish a clear system boundary for testing, we first derive a comprehensive set of high-level requirements from available documents and the initial system.

4.1 Requirements

The DES serves as a way for different scientific models to communicate with each other via sessions. Sessions are designed to be model-agnostic through user-defined parameters. Information about participant model IDs, initiator and

invitee IDs, variables, and variable size are required when starting a session.

An initiator initiates a session, and invitees join sessions. However, a session cannot be joined unless the invitee possesses the ID specified by the initiator. Variables are denoted by a user-defined integer ID serving as a key in the session's *Flag Status* table telling users if data is ready to be received. A 0 represents data is not present and a 1 tells the users data is available. Data cannot be overwritten, as the controller will reject data for variables with a flag equal to 1. Detailed requirements for handling the controller are shown in Table 1.

Tag	Requirement
1	**Session Creation:**
1a	The initiator shall send a request to the server to create a new session.
1b	On receiving the request the server shall create a new session.
1c	The server shall send a reply/acknowledgment message to the initiator.
2	**Joining Sessions:**
2a	The client shall send a request to the server to join a session.
2b	The server shall check the request to see if the session exists.
2c	The server shall check the request to see if the client is an invitee in the existing session.
3	**Sending data:**
3a	The client shall be in a session in order to send data.
3b	The client shall send a send-data request to the server.
3c	The server shall reject send-data requests for sessions that don't exist.
4	**Receiving data:**
4a	The client shall be in a session in order to receive data.
4b	The client shall send a receive-data request to the server.
4c	The server shall check (via the data's flag) if data is present.
5	**Ending Sessions:**
5a	Either client shall send an end-session request if data exchange is no longer needed from the client.
5b	The server shall reject end-session requests for sessions that do not exist.
5c	The server shall reject end-session requests from clients not in the session.
6	**Session Requirements:**
6a	The session shall be created first.
6b	A session shall be independent of each other.
6c	A new session shall have default flags set to 0.

Table 1. Excerpts from Data Exchange Controller requirements

4.2 System Boundary

The system boundary is defined around the Data Exchange Controller, as all inputs and outputs route through it and it contains most of the logic within the package. The system boundary and software's environment are shown in Figure 1. The uninvited client is included due to requirements of joining sessions. While sessions do not include the uninvited client, one can attempt to join a session.

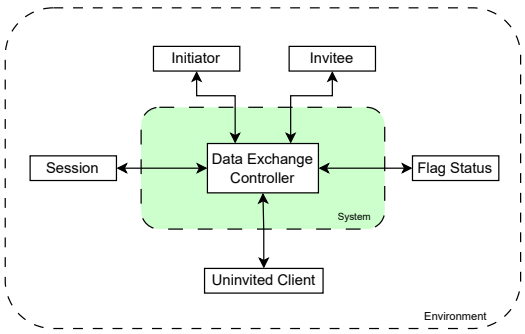

Figure 1. The system boundary for the Data Exchange Controller and its interactions with the environment

4.3 Enumeration and Mealy Machine

Stimuli and responses are identified and grouped by HTTP POST endpoints, which manipulate session states. Table 2 shows the mapping between stimulus keys, HTTP POST endpoints, and responses on error or success.

To further refine the enumeration, *predicates* are used to specify what a stimulus should do. We specify *invalid* inputs with an $_f$ and *valid* inputs with a $_t$, e.g., R_f and R_t are invalid and valid receive data requests respectively. Error responses are used to check invalid inputs, such as an uninvited client attempting to join a session. Predicate refinements may not be needed for some sequences during enumeration. For instance, any $/receive_data$ request is invalid if data is not present.

We enumerate sequences of stimuli in length-lexicographical order that represent histories of events received by the controller. An excerpt of the enumeration table is presented in Table 3 with canonical sequences highlighted in blue. The first sequence (λ) is canonical, representing the lack of inputs, or more specifically, no session. The next sequence is C_f. C_f is a create session request with bad input data, so an error response is returned and the sequence is logically equivalent to λ as no new session is created. The next sequence (C_t) is canonical, which returns new session information in the acknowledgment. The remaining length one sequences are all illegal since a session is required to perform their actions. Because of this, ω is shown in response denoting these sequences are illegal.

Enumeration continues with a length of two starting at C_tC, as C_t is the only canonical sequence of length one that is extended further. A predicate is not specified for the second C as creation of a new session is not possible within a session. The next sequence is C_tE; creating and deleting

Stimulus Key	Endpoint	Success Response	Error Response
C	/create_session	create_session, create_session_ack	create_session_err
E	/end_session	end_session_ack, clear_flag_and_data	end_session_err
J	/join_session	join_session, join_session_ack	join_session_err
R	/receive_data	retrieve_data, recv_data_ack, update_flag(0)	recv_data_err
S	/send_data	send_data_ack, store_data, update_flag(1)	send_data_err

Table 2. Endpoints and their corresponding stimulus keys and responses

Sequence	Response	Equivalence	Trace
λ	0		Method
C_f	create_session_err	λ	1c, 1g
C_t	create_session, create_session_ack		1b, 1c, 1d, 1e, 1f, 1g
E	ω		6a
J	ω		6a
R	ω		6a
S	ω		6a
C_tC	ω		6b
C_tE	end_session_ack, clear_flag_and_data	λ	5d, 5f
C_tJ_f	join_session_err	C_t	2b, 2c, 2d, 2e, 5e
C_tJ_t	join_session_ack		2f
C_tR	recv_data_err	C_t	4a, 4c, 4d, 4e
C_tS_f	send_data_err	C_t	3a, 3c, 3d, 3e, 6f
C_tS_t	send_data_ack, store_data, update_flag(1)		3f, 3g, 3h, 6c
C_tJ_tC	ω		6b
C_tJ_tE	end_session_ack		5d
C_tJ_tJ	ω		6d
C_tJ_tR	recv_data_err	C_tJ_t	4a, 4c, 4d, 4e
$C_tJ_tS_f$	send_data_err	C_tJ_t	3a, 3c, 3d, 3e, 6f
$C_tJ_tS_t$	send_data_ack, store_data, update_flag(1)		3f, 3g, 3h, 6c
...			
$C_tJ_tES_t$	send_data_ack, store_data, update_flag(1)		3f, 3g, 3h, 6c, 6e, 6g
...			
$C_tJ_tES_tS_f$	send_data_err	$C_tJ_tES_t$	3a, 3c, 3d, 3e, 6f
$C_tJ_tES_tS_t$	send_data_ack, store_data, update_flag(1)	$C_tJ_tES_t$	3f, 3g, 3h, 6c, 6e, 6f

Table 3. Excerpts from the enumeration table. Canonical sequences are shown in colored rows.

a session (only one user is connected). It is made equivalent to λ. There are two canonical sequences of length two: C_tJ_t and C_tS_t (creating session with a join or send request).

The new canonical sequences are extended, like C_t, with each stimulus. The extended sequences are considered in lexicographical order for response mappings and equivalence decisions. The enumeration process terminates when there are no new canonical sequences to extend, as seen with length 5 sequences ending with $C_tJ_tES_tS_t$.

With the completed enumeration, a Mealy machine is created using canonical sequences as states. Rows of the enumeration table define transitions among states and outputs on the arcs. The Mealy machine for the Data Exchange Controller, in Figure 2, is used to generate test cases and define the test oracle. The Mealy machine is derived manually from the enumeration following a defined procedure [9,15]. A Markov-chain usage model is created by assigning probabilities to every arc from every state.

4.4 Canonical Sequence Analysis and Test Oracle

The completed sequence-based specification defines the controller's test oracle. During testing, both the session's states and controller's outputs are verified. Testing responses is straightforward as HTTP codes or returned output may be tested similarly to unit test. To test the system's internal states, canonical sequences are used. By *Canonical Sequence Analysis* each canonical sequence can be determined using session attributes. Intuitively, each attribute (2nd - 5th column headers) in Table 4 represents a state variable captured by the canonical sequences. Attributes are read using HTTP GET requests to view flag table or session occupancy values.

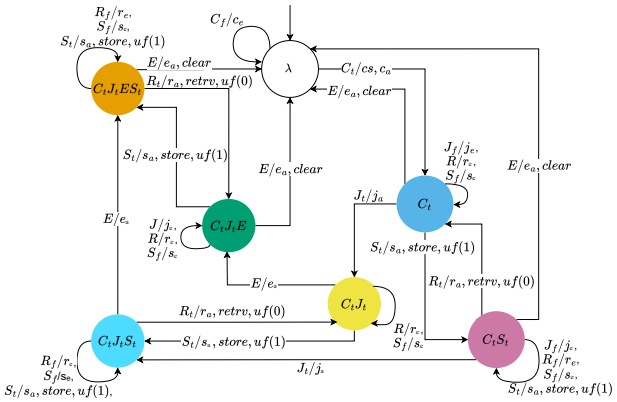

Figure 2. Mealy Machine for sessions in the Data Exchange Service. Shorthand notation is used for stimuli and responses in the form $Stim./Resp.$**.**

Canonical Seq.	Created	Joined	Data Sent	Partial End
λ	0	-	-	-
C_t	1	0	0	-
C_tJ_t	1	1	0	0
C_tS_t	1	0	1	-
C_tJ_tE	1	1	0	1
$C_tJ_tS_t$	1	1	1	0
$C_tJ_tES_t$	1	1	1	1

Table 4. The Canonical Sequence Analysis table represents a list of features that describe the state of the session. '-' means the feature does not apply to the state.

5 Automated Statistical Testing of the Data Exchange Controller

To test the Data Exchange Controller, we implement an automated test workflow using a variety of tools, including the J Usage Model Builder Library (JUMBL) [13], with the Markov-chain usage model written in The Model Language (TML) [12], and a Python program developed to interface with the JUMBL.

5.1 Automated Testing Program

TML is a language for describing Markov-chain usage models. The controller's TML file is used as an input to the JUMBL (our statistical testing tool) for generating test cases. An excerpt of the controller's TML is shown here:

```
1. ($ fill (1) $)
2. model DataExchangeAPI
3.
4. source [lambda]
5.   ($0.01$)  "C_f/c_e"              [lambda]
6.             "C_t/cs, c_a"         [C_t]
7.
8. [C_t]
9.  ($0.1$)    "E/e_a, clear"        [Exit]
10.  ($0.005$) "J_f/j_e"             [C_t]
11.            "J_t/j_a"             [C_tJ_t]
12.  ($0.01$)  "R_f/r_e"             [C_t]
13.  ($0.01$)  "R_t/r_e"             [C_t]
14.  ($0.01$)  "S_f/s_e"             [C_t]
15.            "S_t/s_a, store, uf(1)" [C_tS_t]
```

with `[lambda]` and `[C_t]` on lines 4 and 8 representing two states, and the lines below (5-6, and 9-15) are probabilities, stimuli, responses, and state transitions respectively.

To handle test sequence generation and model and test analyses, we use the JUMBL tool. Model analysis computes statistics of the usage model following standard Markov analysis. These statistics have interpretation in software testing which can be used to validate the model. JUMBL finds session state occurrence and occupancy, showing how often a state is visited or a stimulus is encountered in long-run random testing.

Our tests are automatically generated with three different sampling options: weighted, random, and minimum coverage. Weighted sampling picks the most probable paths in the usage model using the product of arc probabilities. Random sampling uses the probabilities defined on each arc to generate the next stimulus. Minimum coverage creates a set of sequences with minimum total steps to cover each node and each arc of the usage model.

The main automated testing functionality lies outside of the JUMBL tool. Unlike in previous work [8], where we annotate the usage model with testing scripts and then generate executable test cases from the model, we write a Python script to run the software environment, test oracle, and functions interfacing with the JUMBL to automate test case generation, execution, evaluation, and recording of test results.

Sequences are exported from the JUMBL test record and parsed by our Python script, which generates test inputs given by the sequence and verifies outputs associated with each stimulus' response as well as the session states. Pass and fail information, including stop failures, is recorded for every step of the executed test sequence.

When testing is completed, JUMBL runs statistical analysis on the test results. Test case analysis computes statistics of reliability estimates, like Single Use Reliability, and information theoretic measures, like Relative Kullback Discrimination, to assist the test stopping criteria. *Single Use Reliability* is defined as the probability of a randomly selected use being successful. *Relative Kullback Discrimination* reflects if testing approximates the expected use as described by the usage model. These two are among the most important statistics to consider regarding management decisions.

5.2 Results

We apply automated statistical testing to the iterative development of the DES. While following test-driven development, pre-developed unit tests missed bugs resulting from some specific input sequences. The automated test script helps expose bugs not found by unit tests. New unit tests are written to address the conditions found by statistical testing and further analysis is performed to discover more erroneous usage scenarios.

Tests are run on three versions of the Data Exchange Controller. The old version is a prototype to demonstrate the idea of a flexible model-coupling server and was written without formal requirements. The new version is written with requirements derived from the old version, and a newer, fixed version is included with bugs identified and fixed during statistical testing.

The bugs found during statistical testing of the new version are shown in Table 5, with fragments of the failed sequences shown in the first column. Reasons of failure were identified with human inspection.

Our generated test suite includes four min-coverage tests, 200 weighted tests, and $5,000$ random tests. The reliability of the DES is certified by testing until a threshold of 99% Single Use Reliability is achieved. With relatively few stimuli and many tests, some sequences may be repeats or contain repeated components. This is normal for statistical testing because a random sample can contain many tests which are not necessarily unique. In a real-world scenario, the expected use cases are repeated frequently.

Testing results, shown in Table 6, include Single Use Reliabilities, Relative Kullback Discriminants, numbers of stimuli and tests generated/executed/failed. The test analysis report includes node and arc statistics and reliabilities, but they are not included here for brevity. The mismatch between generated and executed stimuli is due to stop failures, where failed stimuli did not modify the session state correctly because of an error.

The fixed version passes every test. The new version passes 90.7%, and the old version passes only 27.2% due to bugs and also requirements changes introduced from the old version to the new version. Using rigorous specification and testing, the Single Use Reliability is improved by 13% over the new version of the controller, and 72% over the old version. The Relative Kullback Discrimination remains low for all three versions, indicating the testing experience is approximating the expected uses. We certify the reliability of the Data Exchange Controller after showing it passing $49,878$ valid and invalid inputs and achieving 99.3% Single Use Reliability.

6 Conclusion and Future Work

In this paper, we certify the reliability of the Data Exchange Controller using sequence-based specification and usage-based statistical testing. Different versions of the Data Exchange Controller are compared to illustrate the effectiveness of our approach for testing stateful HTTP sessions. The newer, fixed version is shown to be reliable and robust while coupling scientific models.

This work uses specialized code to interface with the JUMBL tool. In the future, a generalized framework for fully-automated test case generation, execution, and evaluation can be developed to expedite the implementation of statistical testing regardless of the application. The rigorous software engineering methodologies we use are also applicable to testing other scientific software infrastructure.

Acknowledgments

The authors thank Ayush Lodha and Dr. Yao Liang at Indiana University Indianapolis for an earlier version of the DES software to test. This work is generously funded by NSF under Grant #2209834 and Grant #2209835.

References

[1] A. Bombarda and A. Gargantini. An automata-based generation method for combinatorial sequence testing of finite state machines. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 157–166, Oct. 2020.

[2] R. Chen, F. Li, D. Bieger, F. Song, Y. Liang, D. Luna, R. Young, X. Liang, and S. Pamidighantam. Cyberwater: An open framework for data and model integration in water science and engineering. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM'22, page 4833–4837. ACM, 2022.

[3] E3SM Project. Energy Exascale Earth System Model (E3SM). [Computer Software] `https://dx.doi.org/10.11578/E3SM/dc.20240301.3`, May 2025.

[4] B. Garn, D. E. Simos, F. Duan, Y. Lei, J. Bozic, and F. Wotawa. Weighted combinatorial sequence testing for the TLS protocol. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 46–51, 2019.

[5] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, page 235–244, Antwerp Belgium, Sept. 2010. ACM.

[6] S. Hallé and R. Khoury. Test sequence generation with Cayley graphs. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 182–191, Apr. 2021.

[7] D. R. Kuhn, M. S. Raunak, and R. N. Kacker. Ordered t-way combinations for testing state-based systems. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, page 246–254, Apr. 2023.

Failed Test Fragment	Reason of Failure	Identified Bugs	Implemented Fixes
$C_tJ_tEJ_t$	User was able to join after an end session request when an error should be returned.	The server join function did not check session status to see if a user left.	Server join function was updated to check session status for partial-end.
$C_tS_tR_tR_t$	Second receive request returned data when an error should be returned.	Server receive function only checked if data existed, but did not check relevant flag.	Server receive function was updated to check if relevant flag is 1.
C_f	The function associated with C_f is missing input validation. Session was created with missing information.	Function did not check for missing information.	Input validation added to the function to check request for valid parameters.

Table 5. Test fragments caused errors on the new version. The final stimulus failed during testing.

Total Generated		Old		New		Fixed	
Stimulus/Response	**Generated**	**Executed**	**Failed**	**Executed**	**Failed**	**Executed**	**Failed**
C_f/c_e	56	56	56	56	56	56	0
$C_t/c_a, c_s$	5,204	5,148	0	5,148	0	5,204	0
E/e_a	4,194	4,097	2,351	4,145	0	4,194	0
$E/e_a, clear$	5,204	5,094	70	5,143	77	5,204	0
J_f/j_e	125	120	120	121	0	125	0
J_t/j_a	4,194	4,097	0	4,145	0	4,194	0
J_t/j_e	86	78	78	82	82	86	0
R_f/r_e	465	451	77	456	3	465	0
$R_t/r_a, retrv, uf(0)$	11,143	10,867	1,531	11,005	79	11,143	0
R_t/r_e	502	490	230	493	376	502	0
S_f/s_e	434	424	424	427	4	434	0
$S_t/s_a, store, uf(1)$	18,271	17,865	4,669	18,060	119	18,271	0
Total Stimuli/Responses	49,878	48,787	9,606	49,281	796	49,878	0
Total Tests	5,204	5,204	3,786	5,204	482	5,204	0
Single Use Reliability		0.276368459		0.863924505		0.992865083	
Relative Kullback Discrimination		$35.4876156E-3\%$		$37.2213956E-3\%$		$36.333333E-3\%$	

Table 6. Test results for the three versions of the Data Exchange Controller

[8] L. Lin, J. He, and Y. Xue. An automated testing framework for statistical testing of GUI applications. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 72–79, 2015.

[9] L. Lin, S. J. Prowell, and J. H. Poore. An axiom system for sequence-based specification. *Theoretical Computer Science*, 411(2):360–376, 2010.

[10] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, Jan. 2003.

[11] J. H. Poore, L. Lin, R. Eschbach, and T. Bauer. Automated statistical testing for embedded systems. In J. Zander, I. Schieferdecker, and P. J. Mosterman, editors, *Model-Based Testing for Embedded Systems in the Series on Computational Analysis and Synthesis, and Design of Dynamic Systems*. CRC Press-Taylor & Francis, 2011.

[12] S. Prowell. TML: A description language for Markov chain usage models. *Information and Software Technology*, 42(12):835–844, 2000.

[13] S. Prowell. JUMBL: A tool for model-based statistical testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, Jan. 2003.

[14] S. Prowell and J. Poore. Foundations of sequence-based software specification. *IEEE Transactions on Software Engineering*, 29(5):417–429, May 2003.

[15] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, Reading, MA, 1999.

[16] P. R. Srivastava, N. Jose, S. Barade, and D. Ghosh. Optimized test sequence generation from usage models using ant colony optimization. *International Journal of Software Engineering & Applications*, 1(2):14–28, Apr. 2010.

[17] W. Wang, S. Sampath, Y. Lei, and R. Kacker. An interaction-based test sequence generation approach for testing web applications. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, page 209–218, Dec. 2008.