Testing dk.brics.automaton and dk.brics.string

Marlin Roberts
marlinroberts@u.boisestate.edu
Boise State University

Kenny Ballou
kennyballou@u.boisestate.edu
Boise State University

Abstract

The `string` [9] and `automaton` libraries were examined and tested to study and perform property verification. Various assertions and code instrumentation were added to the two libraries to verify several state and sequential properties. Several possible bugs and recommendations will be discussed as a result of this examination.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Software verification and validation**.

ACM Reference Format:
Marlin Roberts and Kenny Ballou. 2020. Testing dk.brics.automaton and dk.brics.string. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/0000000000

1 Introduction

The Java String Analyzer [9] tool consists of two main libraries, `string` and `automaton`. The `automaton` library implements a deterministic and non-deterministic finite-state automata with an unrestricted set of regular expression operators. The `string` library uses the `automaton` library to model various Java `String` [3] and `StringBuilder` [4] operations.

The two libraries were examined, instrumented, and tested to verify various program properties. Several tools were used to perform the verification, chiefly, Java `Assertions` [8], `JUnit` [6], and `JavaMOP` [5].

The following is a brief overview of the paper. First the methodology of the approach to property discover, property verification, and program testing (2). Next, the program properties examined (3). Then, the various aspects of testing the two libraries (4). Before concluding, the analysis and results of our program verification study (5).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/0000000000

2 Methodology

2.1 Property Discovery

Three approaches were used to identify interesting properties: Reading documentation within the source code or in `README` files, examining the source code directly, and observing event sequences. Each approach identified interesting properties.

The `automaton` package contained comments that directly referenced the state of a returned automata, but few were found in the string operations with similar detail. Automata operations that returned determinized or minimized automata were usually annotated with comments indicating so. Some properties were determined by the documented purpose of the string operations, which is to model the behavior of Java `String` and `StringBuilder` operations. The expected behaviors of Java `String` and `StringBuilder` operations are available from online sources [3, 4]. These behaviors were examined to determine additional properties that related directly to the correct operation of the software under test.

Examination of the source code provided several interesting properties. An example is the discovery of an assertion in one of the string operations, which asserted that the input automata was deterministic. This shows that the programmers expected a certain set of conditions to exist even if they did not explicitly indicate so in the documentation. Interestingly, test cases that should have provided a deterministic automata failed this assertion and led to the conclusion that some operations for building automata might produce automata with incorrect or inconsistent properties.

Observing event sequences during preliminary testing showed interesting sequential properties. For instance, a majority of the string operations ended with calls to `Automaton.clearHashCode()` followed by `Automaton.recomputeHashCode()`, while one did not. This indicated that static checks should be put in place to determine if the hash code of returned automata was correct, however, this proved problematic due to the visibility of the instance variable representing the hash code and the behavior of the method that returns the hash code. Sequence checking was then put in place in order to observe this property to the extent possible. Section 3, Properties, contains the complete details of identified properties.

2.2 Property Checking

The primary methods for checking state or static properties are Java Modeling Language [7] (JML) and Java assertions [8]. JML has significant advantages over assertions, including the

ability to easily compare *pre* and *post* conditions of variables and results. Using JML requires the use of a specific Java compiler that translates the JML annotations into regular Java bytecode for execution. Attempting to use this compiler exposed significant issues that prevented successful compilation. While workarounds did produce a compiled project, further errors were encountered during execution that made continued use of JML too problematic.

Java assertions were then implemented in order to check the state properties. The assertions are located in test cases where possible, in order to minimize intrusion into the classes being observed. However, in cases where properties were not visible to the test cases, assertions were placed in the source code. Results of static property checking are discussed in Section 5.

Checking for sequential properties was accomplished with the Monitoring-Oriented Programming (MOP) tool, Java MOP [5], along with the AspectJ [1] compiler. Checks provided by these tools "cut-across" classes within the software under test which allows observation of events generated anywhere in the source code. These tools were initially used to identify interesting sequential properties by defining a wide range of possible events and observing the sequences produced during preliminary testing. After these properties were identified, the MOP definition files were modified to observe only the events related to these properties. Logic was introduced to check for correct or incorrect sequences. Results of sequential property checking are discussed in Section 5.

2.3 Static Analysis

Static analysis techniques were applied using Coverity [2]. This online static analysis tool provides the ability to run a variety of checks that are provided in sets called checkers. Checkers are specific to a source language and are further targeted at specific program types. For instance, a Java program that provides a web service may have a checker that examines the source for specific issues and code smells that commonly appear in web services code. The source code is prepared for analysis by performing a build using supplied tools that records the build operation and results. This information is then submitted for analysis.

An abundance of false positives tend to be a significant issue when using static analysis tools. False positives are code problems identified by static analysis tools, but in fact are not. Tools like Coverity allow for analysis tuning, which is the process of changing the scope of the analysis or the definition of what constitutes an actual problem.

Results of the static analysis are discussed in Section 5.

3 Properties

The majority of the properties identified for the string operations involve the ability of the operation to match the behavior of the Java String or StringBuilder operation it models. Due to the large number of operations and the related behaviors, this set of properties was treated as an opportunity to apply some of the regression testing techniques, with the idea that the operations likely conform to those behaviors now and tests should be available to make sure they continue to operate correctly. Details of this process is provided in Section 4.

Properties identified for the `automaton` package relate to the structure and validity of the automata they return. Some of these properties could be checked by using available accessor methods, such as whether or not automata are deterministic being available through the `Automaton .isDeterministic()` method. Other properties, such as whether or not an automata is minimal, did not have a readily accessible flag for testing. Various approaches were discussed as to how a property like this could be tested. One approach would be to make a clone of a given automata that should be minimal, minimizing it and comparing it with the original. If the two were equivalent, then the original automata was likely minimal to begin with. Another approach was to monitor sequences of events in order to determine if the automata had the minimize operation performed on it before being returned. In this particular case it was decided that monitoring the sequence of method calls was a less intrusive approach, so this property is tested as a sequential property.

String Operation Behavior. String operation behavior properties are verified by a set of tests that store the current test output for comparison for future executions. There are 23 operations with associated tests for each. The string operation classes are listed in Table 1.

`CharAt1`	`Postfix`	`Replace5`	`Split`
`CharAt2`	`Prefix`	`Replace6`	`Substring`
`Contains`	`Replace1`	`Reverse`	`ToLowerCase`
`Delete`	`Replace2`	`SetCharAt1`	`ToUpperCase`
`DeleteCharAt`	`Replace3`	`SetCharAt2`	`Trim`
`Insert`	`Replace4`	`SetLength`	

Table 1. String Operations

String Operation in Sequence. String operations should be able to be used in sequence, i.e., operations can be chained together, with the automaton returned being used as input to the next in sequence. Assumptions about the returned automata may not hold between operations in the sequence, worse, they may not be detected by property checks. Several tests were created to take this possibility into account. However, due to the large number of possible combinations only a small subset of these tests were implemented.

Returned Automata Minimal. Several methods in the automata class were commented as returning minimal automata. These methods were tested in a separate JUnit test case, as determining if an automata is minimal required several steps. These methods are listed in Table 2.

MinimizationOperations.java	
minimize()	minimizeHuffman()
minimizeValmari()	minimizeBrzozowski()
minimizeHopcroft()	

Table 2. Minimal Methods

Returned Automata Deterministic. Nine methods were commented as returning deterministic automata. Since this property could be tested by accessing the `isDeterministic()` method, all of them were instrumented with assertions. These methods are listed in Table 3.

BasicAutomata.java	
makeEmtpyString()	makeCharSet()
makeAnyString()	makeString()
makeAnyChar()	makeStringUnion()
makeChar()	makeStringMatcher()
makeCharRange()	

Table 3. Deterministic Methods

Automata Arguments Immutable. Automata have an instance variable that indicates whether or not an operation is allowed to change it. By default, this variable is set to false, which should indicate to string operations that the automata is to be considered immutable. It was found that most operations do not check this flag, but do in fact clone arguments in order that they remain unchanged inside the method. Considering this, a desirable property for all string operations is to treat all automata arguments as immutable. Checks were placed in the JUnit test cases to capture the state of arguments passed to operations in order to compare them after an operation completed.

Automata Hashcodes Set. Each automaton contains a basic hashcode that is a function of the number of states and transitions comprising the automaton. It is used as part of the equality comparison, and regardless if the actual state and transitions in two automata are identical, a hashcode inequality will cause the comparison to return false. Since the hashcode accessor method recalculates a cleared hashcode it was difficult to determine if the code was correct. Checks for a hashcode of `0` were implemented as assertions in the automata operations but sequential checks were implemented to look for the `Automaton.recalculateHashCode()` method call after an occurrence of `Automaton.clearHashCode()`.

Consistent Minimization Algorithm. The automaton class provides three different minimization algorithms. A flag is set in the automaton that indicates which algorithm to use when `Automaton.minimize()` is called. Automata with different minimization algorithms were considered to be usable together as arguments to a single operation. Since there is only one minimal representation of a deterministic automata, each algorithm should produce identical results.

Automata Set Operations Correct. The basic set operations of union, intersection and minus are used repeatedly throughout the string operations. These operations have well defined behaviors that can be verified.

Argument Automata Cloning (MOP). Automata passed as arguments should not be changed. Immediate cloning inside the `op()` method protects the arguments.

Events	`UnaryOperation.op()`
	`BinaryOperation.op()`
	`Automaton.clone()`

Regular Expression	`c* o c+`

Table 4. Argument Automata Cloning

Determinize Before Minimize (MOP). Automata should be deterministic before they are minimized. Calling `determinize()` before `minimize()` ensures this.

Events	`Automaton.determinize()` `Automaton.minimize()`
Regular Expression	`(d | d m)*`

Table 5. Determinize before Minimize

Recompute Hashcode after Minimize (MOP). Minimize calls do not have a specific `clearHashCode` call within them. A call to `recomputeHashCode()` ensures it is correctly set.

Events	`Automaton.minimize()` `Automaton.recomputeHashCode`
Regular Expression	`(r | m r)*`

Table 6. Recompute Hashcode after Minimize

Recompute Hashcode After Clearing Hashcode (MOP). The hashcode needs to be recomputed after it is explicitly cleared.

Events	`Automaton.clearHashCode()` `Automaton.recomputeHashCode()`
Regular Expression	`(r | c+ r+)*`

Table 7. Recompute Hashcode After Clearing Hashcode

Restore Invariant After State Access (MOP). Modifying states or transitions requires a call to `restoreInvariant`. If the states are accessed, it is possible they have been modified.

Events	`Automaton.restoreInvariant()` `Automaton.getStates()`
Regular Expression	`(rI* | gS+ rI+)*`

Table 8. Restore Invariant after State Access

4 Testing

The software under test in the project is primarily for use in a program used for analysis of string constraints in Java, the Java String Analyzer (JSA) [9]. The string operations can also be used to implement other string constraint analysis programs, and the automata package can be used to implement automata outside of the string constraint analysis context. Therefore, no single interface or program was available for us to test. Testing was accomplished by creating JUnit tests that executed string operations in isolation or as part of a sequence. In order to more thoroughly test the automaton package, JUnit tests were created that operated directly with the automata instead of through the string operations.

4.1 Test Cases

Each string operation is designed to modify an automata which represents a string or set of strings. String operations extend either the `UnaryOperation` or the `BinaryOperation` classes. These classes provide a single public method, `op()`, that takes one or two automata as arguments. Arguments other than automata are passed to the operation through the constructor. String operations take a variety of other argument types such as char and string in this manner. The string operations are used by instantiating an operation with arguments followed by calling the `op()` method with the target automaton as arguments. The return type of `op()` is Automaton.

In order to fully test the operations, characteristics of an automata such as length, alphabet and whether or not they accept the empty string or are empty were determined. These characteristics were used in place of input parameters. Individual automata were created with characteristic combinations determined using two tools, Test Specification Language (TSL) [11] and Combining Arrays through Simulated Annealing (CASA) [10]. These characteristics are shown in Table 9

Length	MIN	4 characters long
	MED	100 characters long
	MAX	200 characters long
Alphabet	MIN	4 characters
	MED	26 characters
	MAX	All characters
Empty String	YES	Accepts empty string
	NO	Rejects empty string
Empty	YES	Rejects all strings
	NO	Accepts some strings

Table 9. Automata Characteristics

TSL applies the Category Partition Method in order to determine test case parameter values. Parameters are first categorized and further partitioned within each category. These are then analyzed by the TSL tool and a comprehensive list of parameter value combinations produced. Using the categories and partitions except the Empty category defined in Table 9 as input to TSL generated 18 combinations. The Empty category was implemented as a separate, empty automaton since being empty precluded having any other property.

CASA applies Combinatorial Interaction Testing (CIT) as a way of reducing the number of required parameter value combinations. Research has shown that a majority of errors are the result of pairs of input parameter values. This means that it is possible to find a majority of errors by testing all possible pairs of input parameters. Using CASA with the categories for length, alphabet and empty string produced 9 combinations. These were matched to the corresponding TSL cases and 9 automata definitions were created. Each string operation was tested with automata that fit these 9 definitions.

A similar process was attempted in order to create the string operation sequences for testing. Due to the large number of string operations the number of test cases required to achieve pair-wise testing in a sequence of 3 operations was 529 test cases. Two of these combinations were implemented, but the length of time required to run them on the various automata types proved prohibitive.

Input and output files from the TSL and CASA processes are provided in the TSL and CASA directories in the repository.

4.2 Testing Correctness

Testing the correctness of the string operations was treated as an opportunity to create a set of regression tests rather than implementing specific tests for each string operation that targeted a specific behavior of a Java string operation. In order to create these regression tests, correct behavior is

assumed, and tests are created that are intended to verify continued correct behavior.

In order to create the oracle of correct results, the ability to save the resulting automata from all string operation tests was built into the JUnit tests. The resulting automata are serialized and stored in files under a specific directory. The tests can be run in a mode that creates the test result files or uses the result files to verify correct behavior.

5 Results

5.1 Coverage

All but three of the string operations achieved 100% coverage in the `op()` method. The three operations where 100% coverage of the `op()` method was not achieved expose weaknesses in our choice of inputs. In the case of `ToUpperCase`, there is a branch that is taken only with extended character sets. The use of these character sets was not considered when determining the characteristics of input automata. There are two special cases in `Contains` that were not covered with the input combinations: An input automata that is empty, and an input automata that is infinite. Finally, the `Replace6` operation has a special case in which the target of the replace is an empty string.

The `op()` method in `Replace5`, `CharAt2`, and `Split` consist of a single statement that executes other operations. This led to low instruction and line coverage for these operations.

Summarized coverage results are presented in Table 10. A full coverage report in `HTML` format is provided in the `coverage` directory in the repository.

Class	Instruction	Branch	Line	op()
Automaton	74%	77%	74%	NA
MinimizationOperations	99%	93%	97%	NA
CharAt1	92%	92%	90%	100%
CharAt2	50%	NA	40%	100%
Contains	86%	79%	76%	88%
Delete	89%	100%	83%	100%
DeleteCharAt	89%	100%	82%	100%
Insert	86%	100%	77%	100%
Postfix	77%	100%	69%	100%
Prefix	77%	100%	69%	100%
Replace1	65%	60%	69%	100%
Replace2	64%	57%	62%	100%
Replace3	62%	50%	62%	100%
Replace4	81%	100%	72%	100%
Replace5	32%	NA	38%	100%
Replace6	80%	72%	84%	98%
Reverse	88%	100%	81%	100%
SetCharAt1	75%	75%	73%	100%
SetCharAt2	89%	100%	81%	100%
SetLength	81%	100%	76%	100%
Split	50%	NA	40%	100%
Substring	81%	100%	74%	100%
ToLowerCase	83%	90%	77%	100%
ToUpperCase	64%	71%	63%	72%
Trim	96%	87%	94%	100%

Table 10. Code Coverage

5.2 Static Analysis

Coverity identified 12 defects in 2 categories and a defect density of 0.63. These were easily examined and classified without the need for further tuning. A general observation is that the default Java checks appear to be very conservative in what they identify as problems, which could be an attempt to avoid large amounts of false positives that require verification by developers. The most interesting of these defects were resource leaks identified in the serialization and deserialization of several classes in the automaton package. These defects could have been the source of specific problems that occurred when attempting to serialize and deserialize automata. Repeated serialization and deserialization of large automata led to a stack overflow exception. Examination of the stack showed that the error happens at a point when input and output streams are in use.

5.3 Hash Code Not Set

Sequential checks put in place to observe the sequence of `clearHashcode()` and `recomputeHashcode()` calls identified a string operation that failed to correctly set the hash code before returning. The accessor method for the hashcode will recompute a hashcode of zero before returning a result. This behavior masks the fact that an operation failed to set the hashcode correctly. However, the instance variable that holds the hashcode in the automaton class is marked package private, which gives code inside the automaton package visibility to the value of this variable. It is possible, therefore, that code exists within the automaton package that accesses this value directly and could possibly obtain an incorrect hashcode.

5.4 Assertion Failure

Certain automata failed a deterministic assertion check in the string operation `Replace6()`. This assertion was present in the source code before analysis began. The assertion asserts that an automata passed as an argument is deterministic after it was cloned. It is difficult to say if the intent was to make sure that an argument was deterministic or that the clone operation returned a deterministic automata. In either case, it appears that some automata operations that are used to return new automata or modify existing ones do not return deterministic automata. In this case the functions related to repeating a given automata a number of times in order

to make one that accepts longer strings does not return a deterministic automata when it appears it should.

5.5 Determinize and Minimize Calls

The sequence of determinizing an automaton before minimizing was difficult to observe through MOP. A call to `determinize()` is the first statement within the `minimize()` method. While this would normally ensure that an automata was deterministic before the minimization occurs, it also led to multiple instances of a call sequence such as `determinize()-minimize()-determinize()`. This is not necessarily a defect, but does indicate that the performance of the libraries could be improved with careful consideration of how the determinization and minimization occur.

6 Conclusion

The `automaton` and `string` libraries tend to be very robust, as expected from a mature pair of libraries. There are certain issues of this age and the overall maintenance, but this did not impact the overall examination to a large degree.

There are, however, several standout results of the two libraries. There is a bug in the handling of hash codes. Furthermore, there is a defect in the instance variable accessor modifiers that may allow more problematic code in future versions. Some of the assumptions that are encoded by the assertions do not universally hold for all combinations of automata and operations. Overall, a deeper inspection of the method call sequence may yield further improvements of both code organization and performance.

References

[1] AspectJ. https://www.eclipse.org/aspectj/. Online, Accessed 2 May 2020.

[2] Coverity scan static analysis. https://scan.coverity.com/. Online, Accessed 2 May 2020.

[3] Java String. https://docs.oracle.com/javase/8/docs/api/java/lang/String.html. Online, Accessed 1 May 2020.

[4] Java StringBuilder. https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html. Online, Accessed 1 May 2020.

[5] JavaMOP. http://fsl.cs.illinois.edu/index.php/JavaMOP4. Online, Accessed 22 February 2020.

[6] JUnit. https://junit.org/junit4/. Online, Accessed 1 May 2020.

[7] OpenJML. https://www.openjml.org/. Online, Accessed 22 February 2020.

[8] Programming with assertions. https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html. Online, Accessed 22 February 2020.

[9] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[10] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.

[11] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, Jun 1988.