Precondition Coverage in Software Testing

Cyrille Artho
AIST, Osaka, Japan
c.artho@aist.go.jp

Quentin Gros
University of Nantes, Nantes, France
quentin.gros@etu.univ-nantes.fr

Guillaume Rousset
University of Nantes, Nantes, France
guillaume.rousset@etu.univ-nantes.fr

***Abstract*—Preconditions indicate when it is permitted to use a given function. However, it is not always the case that both outcomes of a precondition are observed during testing. A precondition that is always false makes a function unusable; a precondition that is always true may turn out to be actually an invariant.**

In model-based testing, preconditions describes when a transition may be executed from a given state. If no outgoing transition is enabled in a given state because all preconditions of all outgoing transitions are false, the test model may be flawed.

Experiments show a low test coverage of preconditions in the Scala library. We also investigate preconditions in Modbat models for model-based testing; in that case, a certain number of test cases is needed to produce sufficient coverage, but remaining cases of low coverage indeed point to legitimate flaws in test models or code.

***Keywords*—Unit testing, model-based testing, preconditions, coverage**

I. Introduction

Contracts in software specify under what preconditions a function may be used, such that it guarantees a well-defined outcome (postcondition) [13]. Eiffel [12] introduced such contracts as a language element by distinguishing between preconditions (that must be fulfilled by the caller), postconditions (that are guaranteed by the function), and invariants (consistency conditions that guarantee a well-defined state of a component). Other programming languages like Java use exceptions such as IllegalArgumentException or IllegalStateException to codify preconditions, in addition to assertions, which can represent preconditions, invariants, or postconditions [11]. The library of the Scala programming language [16] features assertions, assumptions (for static verification), postconditions, and the `require` function, which automatically throws an IllegalStateException if a requirement (precondition) is violated [17].

Preconditions are also used in model-based testing. In that domain, many tools use extended finite state machines (EFSM) [9] as a mechanism to specify the behavior of the system under test (SUT). In an EFSM, transitions relate to pairs of states; the SUT must be in the right state for a transition to be executable. In addition to that, a transition function may optionally feature an *enabling function* (a precondition), which must hold for a transition to be executable. ModelJUnit [20] and Modbat [2] are two model-based testing tools that support such enabling functions.

Test coverage is an indicator of defect detection ability. High coverage is difficult to achieve [14]. We argue that preconditions should have full branch coverage (i. e., both the successful and the failing outcomes should be tested). A true outcome implies that the given function can be executed in at least one setting; a false outcome ensures that at least some cases that violate the requirements exist and are caught. In the context of model-based testing, the situation where all preconditions (in the test model) of all outgoing transitions are false, forces a test case to terminate. This situation is not always intentional, so detecting such cases may reveal flaws in a test model.

Therefore, a precondition that is always true or always false indicates a problem. The problem may be that the existing test suite does not cover enough behaviors, so more diverse test cases are needed. However, it may also be the case that a precondition is always true, and hence actually an invariant; in this case it should be encoded as such. If a precondition is always false, the given function is unusable, as the precondition is too strong.

The intrinsic difficulty of achieving very high code coverage [14] may be responsible for the low coverage of such preconditions in current code. In this paper, we present the results of preliminary experiments on the Scala collection library and various Modbat test models. We show that in existing code, failed preconditions are almost never observed during testing. In test models, after a sufficient number of test cases has been generated, preconditions that are always true or always false provide valuable insights into flaws in the model. Corrections entail either a conversion of a precondition to an invariant, or strengthening or weakening a precondition.

The remainder of this paper is organized as follows: Section II shows related work. Our experiments are described in Section III. Section IV concludes and outlines future work.

II. Related Work

In earlier work, we have investigated how parts of a system are never covered because of human bias by the modeler [4], which causes the model to exclude valid test sequences. For object-oriented software, test cases are often designed to cover possible exceptions, but tend to stop at the first exception [8], [19]. This bias was confirmed in our case studies for designing models for network libraries [2], [3]. In this work, we present evidence for a bias for using valid parameters or states over invalid ones, which again shows that incorrect uses of libraries (including multiple incorrect uses) are not sufficiently considered by human testers.

Instead of being poorly tested, a precondition that always holds may simply be too general to be falsifiable. In hardware analysis, the problem of properties being trivially true has been well-known for two decades [7]. So-called *vacuous* properties

Table I. OCCURRENCES OF PRECONDITIONS IN THE SCALA LIBRARY.

	Entire library			Collections		
	Occurrences	Files		Occurrences	Files	
`require` statements	24	16		15	9	
IllegalArgumentException	38	17		22	9	
IllegalStateException	5	4		2	1	
All preconditions (Scala)	67	31	(5 %)	39	18	(6 %)
Eiffel (for comparison)	3318	407	(59 %)	2236	230	(49 %)

include implications of type $a \rightarrow b$, where the antecedent a is never true. Regardless of the value of b, such a property holds. However, because the second part of the formula becomes irrelevant, this case of an "antecedent failure" is likely not what the modeler intended [6].

In software testing, modified condition/decision coverage (MC/DC) and similar test coverage criteria try to ensure that each part of a complex conditional statement is actually relevant for the outcome of a test suite [1]. For each location in the software code where compound conditionals exist, MC/DC demands that, among other criteria, each condition in a decision is shown to independently affect the outcome of the decision [21]. If a condition has no effect on the outcome of a decision, it is likely incorrect (too weak) or redundant. The application of coverage criteria on the model level is emerging work, with only a few relatively simple coverage criteria such as state, transition, and path coverage, being commonly used so far [1]. This work shows that precondition coverage merits being included in such metrics.

III. EXPERIMENTS

We analyze the outcome of preconditions in the regression test suite of Scala 2.11.7 [15] and several transition-based test models written for Modbat [2]. We chose the Scala library because it offers Eiffel-like contracts that can also be used by Modbat models, which itself is based on Scala as well.

A. Scala collection library

The Scala base library contains 579 files in version 2.11.7. Almost half of these (288 files) represent *collections,* algorithmic data structures such as lists, sets, and maps, and functions to access and modify them, such as iterators and filters.

Unfortunately, contracts were introduced relatively late (around version 2.8) [10], when much of the Scala library was already written. Requirements (preconditions) are therefore relatively rare. Table I shows that `require` statements occur only 24 times in the entire library, spread over 16 files.

Other ways to express invalid uses of a function, such as the use of IllegalArgumentException or IllegalStateException, are also rather uncommon (see Table I). We count these constructs together but elide IndexOutOfBoundsException, which is more parameter-specific and occurs 57 times in 41 files (in the collections, 35 times in 19 files). Most of the preconditions are in the collection classes, where 6 % of all files contain at least one such precondition. We compare this to the Eiffel library [12], where the *base* libraries contain such data structures and other helper functions in 468 files, which contain 2236 `require` statements in 230 files. The entire Eiffel library of 682 files, contains 3318 `require` statements in 407 files. In Eiffel, the use of preconditions is therefore quite pervasive while they are still rather scarce in Scala.

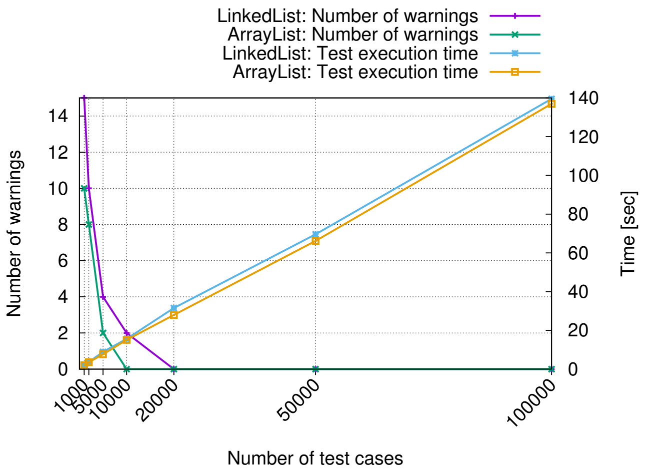

Figure 1. Precondition coverage and test generation time for collection/iterator models.

We instrument the code to log all 39 uses of `require` statements, IllegalArgumentException, and IllegalStateException in the Scala collections. The instrumentation analyzes positive outcomes, where the requirement is fulfilled, and negative outcomes, where a requirement fails. We run the instrumented code with Scala's unit tests (build target `test.junit`), and with all its tests (build target `test`, using a variety of test harnesses).

Table II summarizes the results. Cases where a positive outcome is *not* covered, and cases where negative outcomes have been observed, are shown in bold. Some requirements were never tested at all, neither for the positive or negative case; this concerns 17 out of 39 requirements when running all unit tests, and 11 cases after all tests have been run. Negative outcomes are hardly ever triggered by unit tests (only two cases are covered), while seven cases are covered after all tests have been run.

B. Modbat test models

Modbat [2] is a model-based test tool that allows a user to describe the usage of a system under test (SUT) using extended finite-state machines [9]. Such state machines allow for optional preconditions in a transition, which have to hold for a transition to become executable when the model is in the given state. Preconditions in Modbat models are executed at run-time and evaluated before the body of a transition function is executed. The remainder of the transition typically calls the SUT and verifies the outcome of its operation. Recent versions of Modbat evaluate the outcomes of all executions of each precondition in the model, and issue a warning if the outcome is always the same.

1) Collections: Java collections include lists, sets, and maps, and iterators that can access elements one by one. Iterators can only be used as long as the underlying collection has not been modified. The combination of list modifications and iterator usage yields a fairly complex model [5]. For a model representing lists, the number of warnings drops sharply with an increasing number of tests, as more model states are covered (see Figure 1). Test generation time is linear in the number of tests, as all tests have a similar execution time.

Table II. Test coverage of requirements in Scala collection library.

Class	Method	Unit tests		All tests	
		Passed	Failed	Passed	Failed
Iterator	range	11	0	22	0
	GroupedIterator constructor	419	0	9,587	0
	GroupedIterator.copyToArray	**0**	0	10	**6**
	Leader.finish	1	0	1	0
SeqViewLike	Patched.update	**0**	0	3	0
convert.Wrappers	SetWrapper.constructor	**0**	0	**0**	0
	MapWrapper.constructor	**0**	0	**0**	0
generic.GenTraversableFactory	range	2,149	0	387,227	**1**
generic.GenericTraversableTemplate	transpose	1,342	0	366,759	0
immutable.BitSet	+	4,200	0	6,797	0
	−	716	0	847	0
immutable.NumericRange	check	37,870	**7,106**	37,906	**7,107**
	count	73,560	0	524,485	0
immutable.Range	count	32,346	0	32,359	0
	length	106,341	**705**	8,212,426	**1,122**
	numRangeElements	137,833	0	47,091,134	**196**
	validateMaxLength	120,041	0	119,869,845	0
immutable.StringLike	parseBoolean	**0**	0	**0**	0
	parseBoolean2	**0**	0	**0**	0
immutable.Vector	cleanLeftEdge	**0**	0	**0**	0
	cleanRightEdge	**0**	0	**0**	0
	requiredDepth	112,308	0	712,537	0
	VectorPointer.copyRange	**0**	0	**0**	0
	VectorPointer.getElem	156,632	0	14,016,297	0
	VectorPointer.gotoNextBlockStart	**0**	0	**0**	0
	VectorPointer.gotoNextBlockStartWritable	**0**	0	**0**	0
	VectorPointer.gotoPos	**0**	0	**0**	0
	VectorPointer.gotoPosWritable	**0**	0	**0**	0
mutable.AVLTree	Node.insert	**0**	0	1,027,648	0
mutable.ArrayBuffer	remove	5,588	0	191,100	0
mutable.BitSet	add	10,553	0	1,112,010	0
	ensureCapacity	11,709	0	1,472,708	0
	remove	2,033	0	361,746	0
mutable.LinkedListLike	insert	**0**	0	2	**1**
	tail	15,299	0	520,712	0
mutable.ListBuffer	remove	**0**	0	**0**	**1**
mutable.MutableList	tailImpl	3	0	3,701	0
mutable.ResizableArray	reduceToSize	7,490	0	360,544	0
parallel.mutable.ParArray	constructor	**0**	0	849	0
Total		838,444	7,811	196,319,262	8,434

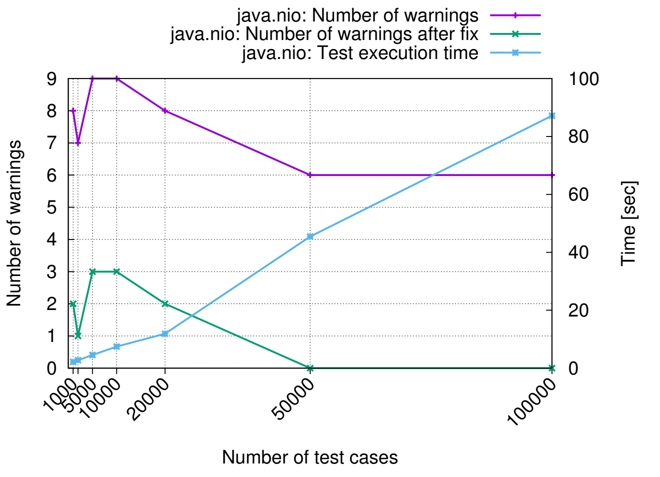

Figure 2. Precondition coverage and test generation time for `java.nio` models.

2) Networking: This model tests java.nio.channels.ServerSocketChannel, a key component for non-blocking network operations in the Java library [18]. This model was created before Modbat issued warnings related to the same outcome of preconditions over all tests [2], [3]. Figure 2 shows the number of warnings issued by Modbat against an increasing number of tests, and the test generation time.

At first, the number of warnings increases as the number of tests grows. This is because with very few tests, model coverage is low, so certain transitions are *never* executed, and hence no warnings are issued. The number of warnings peaks around 5,000–10,000 tests in this case. After that, the number of warnings decreases slightly with more tests, as they reveal new system and model states and therefore cover more precondition outcomes. After 50,000 tests, we hit a fixed point. Note: test generation time increases after 20,000 tests because the system needs time to "recycle" available network ports.

Manual inspection of the warnings shows three reasons for the six warnings (see Table III):

1) Preconditions are always true because they are established in the constructor of the model. In these cases, they can be changed to assertions to guard against changes in the model that would no longer establish these invariants.
2) The precondition is already checked in (a) the caller or (b) the preceding transition. As a model transition is only executed if all preconditions hold, any precondition that is checked twice will always evaluate to *true* when checked more than once, as false outcomes prevent further checks. These redundant checks can also be changed to assertions but may have to be changed back to preconditions if the model structure changes.

Table III. REASONS WHY PRECONDITIONS EVALUATED TO ONLY ONE OUTCOME.

Occurrences	Reason
4	Always true (established in constructor)
1	Precondition already checked by caller
1	Precondition checked in preceding transition

IV. CONCLUSIONS AND FUTURE WORK

Preconditions describe the conditions under which a function may be used. In the Scala library, preconditions are currently not widely used, and poorly tested; some preconditions are never covered, while others are only tested for the successful case. In some of these cases, the test suite is too weak, in others, the preconditions is too general and may never fail, regardless of the input or system state.

In test models, preconditions describe when a test action is executable. If only one possible outcome of a precondition is covered even after many tests, this shows a likely flaw. A precondition that always fails may have to be relaxed to allow it to succeed; a precondition that never fails should probably be encoded as an assertion, to check the expected state of the test model. We found that these flaws exist in real test models.

Future work includes the analysis of mutually exclusive preconditions in test models. If all preconditions of all outgoing transitions of the current state are mutually exclusive, the choice of the next transition in that test model is deterministic. This is often a desirable trait for a test model or at least parts thereof. Validating this trait at run-time may provide further insight into possibly incorrect preconditions.

ACKNOWLEDGEMENTS

This work was supported by *kaken-hi* grants 23240003 and 26280019 from the Japanese Society for the Promotion of Science (JSPS), and *Envoleo* scholarships of the University of Nantes.

REFERENCES

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, USA, 1 edition, 2008.

[2] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Proc. 9th Haifa Verification Conference (HVC 2013)*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.

[3] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proc. 28th Int. Conf. on Automated Software Engineering (ASE 2013)*, Palo Alto, USA, 2013.

[4] C. Artho, K. Hayamizu, R. Ramler, and Y. Yamagata. With an open mind: How to write good models. In *Proc. 2nd Int. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS 2013)*, number 419 in CCIS, pages 3–18, Queenstown, New Zealand, 2014. Springer.

[5] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata. Model-based testing of stateful APIs with Modbat. In *Proc. 30th Int. Conf. on Automated Software Engineering (ASE 2015)*, pages 858–863, Lincoln, USA, 2015. IEEE.

[6] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proc. 31st Conf. on Design Automation (DAC 1994)*, pages 596–602, San Diego, USA, 1994.

[7] I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An industry-oriented formal verification tool. In *Proc. 33rd Conf. on Design Automation (DAC 1996)*, pages 655–660, Las Vegas, USA, 1996.

[8] G. Calikli and A. Bener. Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proc. 6th Int. Conf. on Predictive Models in Software Engineering*, PROMISE 2010, pages 10:1–10:11, New York, NY, USA, 2010. ACM.

[9] K. Cheng and A. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th Int. Design Automation Conference*, DAC 1993, pages 86–91, New York, NY, USA, 1993. ACM.

[10] J. Eichar. Daily Scala: Assert, require, assume, 2010. http://daily-scala.blogspot.jp/2010/03/assert-require-assume.html.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Oracle, 2015.

[12] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, USA, 1992.

[13] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[14] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 291–301. IEEE, 2009.

[15] A. Moors, S. Tisue, J. Zaugg, Ichoran, L. Rytz, V. Ureche, D. Shabalin, E. Burmako, H. Miller, D. Wall, I. Dragos, A. Prokopec, and J. Bogucki. The Scala programming language, 2015. http://www.scala-lang.org/.

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc., USA, 2nd edition, 2010.

[17] Martin Odersky. Contracts for Scala. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Proc. 1st Int. Conf. on Run-time Verification (RV 2010)*, volume 6418 of *LNCS*, pages 51–57. Springer, 2010.

[18] Oracle. Overview (Java platform SE 8), 2015. http://docs.oracle.com/javase/8/docs/api/overview-summary.html.

[19] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *36th Conf. on Software Engineering and Advanced Applications*, pages 286–293. IEEE Computer Society, 2012.

[20] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.

[21] Y. Yu and M. Lau. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *J. Syst. Softw.*, 79(5):577–590, May 2006.