

About Testing and Specification . . . and about First Order Logic

Jan van Eijck

jve@cwi.nl

Master SE, November 5, 2008

Abstract

Introduction to a number of issues related to testing and specification.

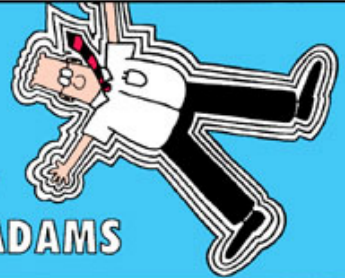
Brief review of first order logic.

Use of first order logic for specification, in the specification tool **Alloy**.



DILBERT[®]

BY
SCOTT ADAMS



Alan Kay about Software ‘Engineering’

“If you look at software today, through the lens of the history of engineering, it’s certainly engineering of a sort—but it’s the kind of engineering that people without the concept of the arch did.

Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.”

Alan Kay in an interview

Alan Kay is one of the designers of Smalltalk, and winner of the Turing Award 2003. (Ruby is considered by some to be a modern version of Smalltalk.)

Tony Hoare about Software Engineering

Just recently, I have discovered that an early advocate of the assertional method of program proving was none other than Alan Turing himself. On June 24, 1950 at a conference in Cambridge, he gave a short talk entitled “Checking a Large Routine” which explains the idea with great clarity. “How can one check a large routine in the sense of making sure that it’s right? In order that the man who checks may not have to difficult a task, the programmer should make a number of definite **assertions** which can be checked individually, and from which the correctness of the whole program easily follows.”

Tony Hoare (winner of the Turing Award 1980) in his Turing Award lecture.

About the design of the successor of Algol 60, in the same lecture:

[. . .] I gave desperate warnings against the obscurity, the complexity, and overambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are **obviously** no deficiencies and the other way is to make it so complicated that there are no **obvious** deficiencies.

Hoare [1981]

Hoare on the Need of Mathematical Abstraction

The absence or even conscious avoidance of mathematical abstraction in programming education explains why many programmers have often been regarded more like craftsmen or technicians than engineers. They are wonderful people, with experience and skills greatly to be admired and valued. But they work best in isolation on self-contained tasks. They have no language to discuss, explain and justify their work to their colleagues and superiors. Documentation is their bane. They do not read the technical literature to keep abreast of their field. On promotion, they find it difficult to maintain intellectual control of the work of their teams.

Hoare [1999]

Testing expected to be easy

Testing should be the aspect of software engineering with the securest basis (Dick Hamlet):

- your **program** is given.
- you know which outcomes to expect for some finite collection of input values. This counts as a specification of your test.

The truth of the matter is that the foundations of the practice of testing are very muddy.

Terminology: Bugs, Faults, Defects

[. . .] the word **bug** suggests something humans can touch and remove — and are probably not responsible for. This is already one reason to avoid the word **bug**. Another reason is its lack of precision. Applied to programs, a bug can mean:

- An incorrect piece of **program code** (“This line is buggy”)
- An incorrect **program state** (“This pointer, being null, is a bug”)
- An incorrect **program execution** (“The program crashes; this is a bug”)

Andreas Zeller, **Why programs fail**, Ch 1.

Improved Terminology

The following terminology is more precise [Zeller \[2005\]](#)

Defect An incorrect program code

Infection An incorrect program state

Failure An observable incorrect program behaviour

Now we can say that defects cause infections which lead to failures.
Or conversely: failures allow us to track infections, which lead us to defects that we can fix.

What makes a test a good test?

Answer depends on the test purpose:

- Check whether the program meets the test? (naive view)
- Exercise software to reveal faults? (Myers, [The Art of Software Testing Myers \[1979\]](#))
- Finding a measure for the dependability of the software under scrutiny? (Dick Hamlet)

A quote from the philosopher Ludwig Wittgenstein

“Someone does not trust a message in a newspaper. He runs out and buy another copy of the same newspaper and is relieved to find confirmation . . .”

Software testing is often like this . . .

What is Software Testing?

Four ideas about the essence of software testing:

- Finding defects through hard work
- Estimate probabilities of program failures.
- Increasing software reliability, where reliability is taken to be the probability of correct behaviour in situations with given conditions of use, and well-delimited periods of time.
- Establishing a measure for reliability: development of a measure for our trust in the correctness of software.

The Analogy with Fishing

Compare:

I've searched hard for defects in this program, found a lot of them, and repaired them. I can't find any more, so I'm confident there aren't any.

with

I've caught a lot of fish in this lake, but I fished all day today without a bite, so there aren't any more.

Quantitatively, the fishing argument is much the better one: a day's fishing probes far more of a large lake than a year's testing does of the input domain of even a trivial program.

Dick Hamlet, Foundations of Software Testing [Hamlet](#) [1994a]

Testing in Practice

Idea of **coverage**: The quality of a test is a function of how well the test 'covers' the program (or the specification).

- Functional coverage: based on specification
- Control coverage: based on program structure

Alternatives:

- Formal development methods: towards provably correct software.
- Formal specification with automated testing
- Software inspection.

In practice, often a mix of these techniques is used.

Partition testing versus random testing

The input domain is split into subdomains covering the whole domain. Black box testing, white box testing, path-coverage structured testing, ... These are all forms of partition testing.

Statistic analysis of the difference between partition testing and random testing revealed that partition testing

- hardly reveals any more failures, and
- hardly increases the probability to find a specific failure.

The reason for this is that partition testing would work better if we knew a priori that some subdomains contained more defects.

In practical situations it is often unknown 'where the bugs are'.

Random testing: when is it appropriate?

- Random testing is in fact: taking samples from the space of possible input values of the program, and observing the results.
- Are tests statistical samples of expected behaviours for given inputs?
- Opponents of this view point out that software defects are **reproducible**, in contrast with the influence of accidental physical circumstances.
- It does make sense to day that a dyke is designed with a risk of one flooding in a thousand years in mind.
- What does it mean to say that a software system is designed with the risk of one **defect** in 10.000 code lines in mind. Or with a risk of one **infection** in 10.000 instructions in mind? Or with a risk of

one **failed output** in 10.000 possible outputs? Or with a risk of one **failure** in 10.000 hours of uptime?

Hoare on the Purpose of Testing

Philosophers of science have pointed out that no series of experiments, however long and however favourable can ever prove a theory correct; but even only a single contrary experiment will certainly falsify it. And it is a basic slogan of quality assurance that "you cannot test quality into a product". How then can testing contribute to reliability of programs, theories and products? Is the confidence it gives illusory?

The resolution of the paradox is well known in the theory of quality control. It is to ensure that a test made on a product is not a test of the product itself, but rather of the methods that have been used to produce it — the processes, the production lines, the machine tools, their parameter settings and operating disciplines. If a test fails, it is not enough to mend the faulty product. It is not enough just to throw it away, or even to reject the whole batch of products in which a defective one is found. The first principle is that the whole production line must be re-examined, inspected, adjusted or even closed until the root cause of the defect has been found and eliminated.

Hoare [1996]

Hoare on the Value of Testing

The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code. Programmers who consistently fail to meet their testing schedules are quickly isolated, and assigned to less intellectually demanding tasks. The most reliable code is produced by teams of programmers who have survived the rigours of testing and delivery to deadline over a period of ten years or more. By experience, intuition, and a sense of personal responsibility they are well qualified to continue to meet the highest standards of quality and reliability. But don't stop the tests: they are still essential to counteract the distracting effects and the perpetual pressure of close deadlines, even on the most meticulous programmers.

Hoare [1996]

Informal Specification Versus Formal Specification

- Informal specification: natural language, or drawings
- Formal specification: formal language, or language of drawings
- Examples of formal languages:
 - Logical languages: First order logic, Higher order logic, ...
 - Picture languages: UML, ...
- What makes a formal language formal? Precise definition of what each language construct means.
- Expressive power versus tool support: more expressive = more difficult to check or reason with.

Importance of Declarative Thinking

declarative thinking thinking in terms of states of affairs. Key question: what is the case?

operational thinking thinking in terms of actions, changes in computer memory, etc. Key question: what happens?

States of affairs are **conceptually simpler** than actions.

Learning (predicate) logic is one way of learning how to think declaratively.

How does declarative thinking help to become better at testing?

Java programmers are often very poor at declarative thinking. Haskell programmers are often very good at it. Why?

First Order Logic: Syntax, LICS Section 2.2

Grammar for the language of predicate logic (see LICS, page 90, 91):

$$t ::= x \mid c \mid f(t, \dots, t)$$
$$\varphi ::= P(t_1, \dots, t_n) \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid (\forall x\varphi) \mid (\exists x\varphi)$$

Free and bound variables in a formula φ .

Substitution of a term t for a variable x in φ . Notation $\varphi[t/x]$.

t is free for x in φ

Alphabetic variant of a formula φ .

Semantics: LICS Section 2.4

Universes.

Models of predicate logic.

Why does the truth table method from propositional logic fail for predicate logic?

Notion of a **lookup-table** or **environment** for a universe.

Truth definition: LICS page 128.

Semantic entailment.

The treatment of equality.

First Order Logic and Alloy

- Alloy based on First Order Logic + Relational Operations
- Check only possible for small domains
- Small domain hypothesis: 'most design errors show up in small domains'
- Abstract level of representation
- No theorem proving needed!
- Form of automated testing of specifications
- Check out the Alloy homepage <http://alloy.mit.edu> for quick-start guide and further tutorial material.

From the FM 2006 Alloy Tutorial

- Alloy = logic + language + analysis
- logic: first order logic + relational calculus
- language: syntax for structuring specifications in the logic
- analysis: bounded exhaustive search for counterexample to a claimed property using SAT

First Order Signatures

First order model (or: predicate logical model) consists of

- **objects** of various kinds (the **domain of discourse**),
- predicates (properties of objects),
- relations between objects.
- These together are called the **signature**.
- Hardest thing to understand about formal specification: **nothing is assumed except what is stated in the specification**.

Example: Birthday Book; Signature

This example is in Alloy 4.1, directory `models/examples/toys/birthday.als`

```
sig Name {}
```

```
sig Date {}
```

```
sig BirthdayBook {known: set Name, date: known -> one Date}
```

- Domain of discourse: **Name** objects, **Date** objects, **BirthdayBook** objects.
- Predicate **known**, relation **date**.
- **known** denotes a set of names
- **date** relates known names to single dates.

Predicates

```
pred AddBirthday (bb, bb': BirthdayBook, n: Name, d: Date) {  
  bb'.date = bb.date ++ (n->d)  
}
```

```
pred DelBirthday (bb, bb': BirthdayBook, n: Name) {  
  bb'.date = bb.date - (n->Date)  
}
```

Adding a birthday: changes an old birthday book **bb** to a new one **bb'**, by adding a pair consisting of a name **n** and a date **d** (the birthday of the person named).

Deleting a (person and his/her) birthday: changes an old birthday book **bb** to a new one **bb'**, by removing the pair of a name **n** and its birthday.

More Predicates

```
pred FindBirthday (bb: BirthdayBook, n: Name, d: lone Date)
d = bb.date[n]
}
```

```
pred Remind (bb: BirthdayBook, today: Date, cards: set Name)
cards = (bb.date).today
}
```

FindBirthday finds someone's birthday in a birthday book.

Remind: who should get birthday cards? Predicate that relates a birthday book **bb** to a date **today** and a set of **cards** (names of those whose birthday is today).

One More Predicate

```
pred InitBirthdayBook (bb: BirthdayBook) {  
  no bb.known  
}
```

InitBirthdayBook: predicate that specifies an empty birthday book.

Assertions

```
assert AddWorks {  
  all bb, bb': BirthdayBook, n: Name,  
    d: Date, d': lone Date |  
      AddBirthday (bb,bb',n,d)  
      && FindBirthday (bb',n,d') => d = d'  
}
```

AddWorks asserts that **AddBirthday** works as it should: if you add an entry, then look it up, you get back what you just entered.

Assertions, ctd

```
assert DelIsUndo {  
  all bb1,bb2,bb3: BirthdayBook, n: Name, d: Date |  
  AddBirthday (bb1,bb2,n,d) && DelBirthday (bb2,bb3,n)  
    => bb1.date = bb3.date  
}
```

DelIsUndo asserts that performing **DelBirthday** after **AddBirthday** undoes it.

Checking Assertions

check AddWorks for 3 but 2 BirthdayBook

check DelIsUndo for 3 but 2 BirthdayBook

The first check checks the predicate **AddWorks**, up to a maximum of 3 for every kind of thing in the signature except BirthdayBook, for which the maximum is 2.

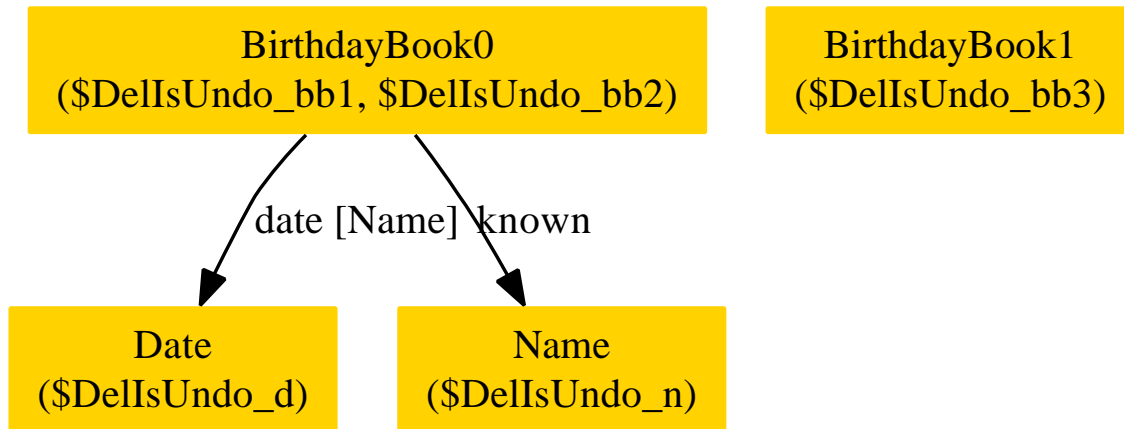
Will this succeed or fail?

The second check checks the predicate **DellsUndo**, up to a maximum of 3 for every kind of thing in the signature except BirthdayBook, for which the maximum is 2.

Will this succeed or fail?

Counterexamples

check DelIsUndo for 3 but 2 BirthdayBook



Predicates defined with Formulas

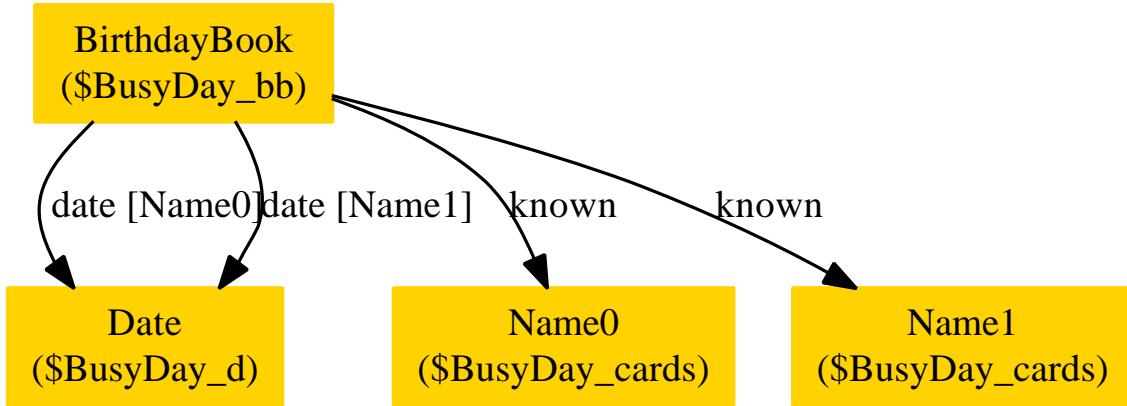
```
pred BusyDay (bb: BirthdayBook, d: Date){  
  some cards: set Name | Remind (bb,d,cards) && !lone cards  
}
```

What does this say?

Can we find an example of this, in a small birthday book?

```
run BusyDay for 3 but 1 BirthdayBook expect 1
```

Example



A Puzzle About Termination

Does the following Ruby program always terminate?

```
def run k
  raise "argument < 1" if k < 1
  return [k] if k == 1
  return [k] + run(k/2) if k.modulo(2) == 0
  return [k] + run(3*k + 1)
end

print "Give positive whole number: "
n = gets.to_i
p (run n)
```

Sample Runs

```
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 7
[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8,
4, 2, 1]
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 8
[8, 4, 2, 1]
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 11
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

```
lucht:~/courses/testing2007/notes jve$ ruby syracuse.rb
Give positive whole number: 27
[27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322,
161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103,
310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395,
1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251,
754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958,
479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288,
3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160,
80, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Does this program always terminate? Can you test this?

What does the example tell us about the power of testing?

References

- Dick Hamlet. Foundations of software testing: Dependability theory. In [Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering](#), 1994.
- Dick Hamlet. Random testing. In [Encyclopedia of Software Engineering](#), pages 970–978. Wiley, 1994.
- C.A.R. Hoare. The emperor's old clothes. [Communications of the ACM](#), 24(2):75–83, 1981.
- C.A.R. Hoare. How did software get so reliable without proof? In [FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods](#), pages 1–17, London, UK, 1996. Springer-Verlag. Keynote Address.

C.A.R. Hoare. Software: Barrier or frontier? Technical report, Oxford University Computing Laboratory, 1999.

G. Myers. **The Art of Software Testing**. Wiley, New York, 1979.

Andreas Zeller. **Why Programs Fail: A Guide to Systematic Debugging**. Morgan Kaufmann, 2005.