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

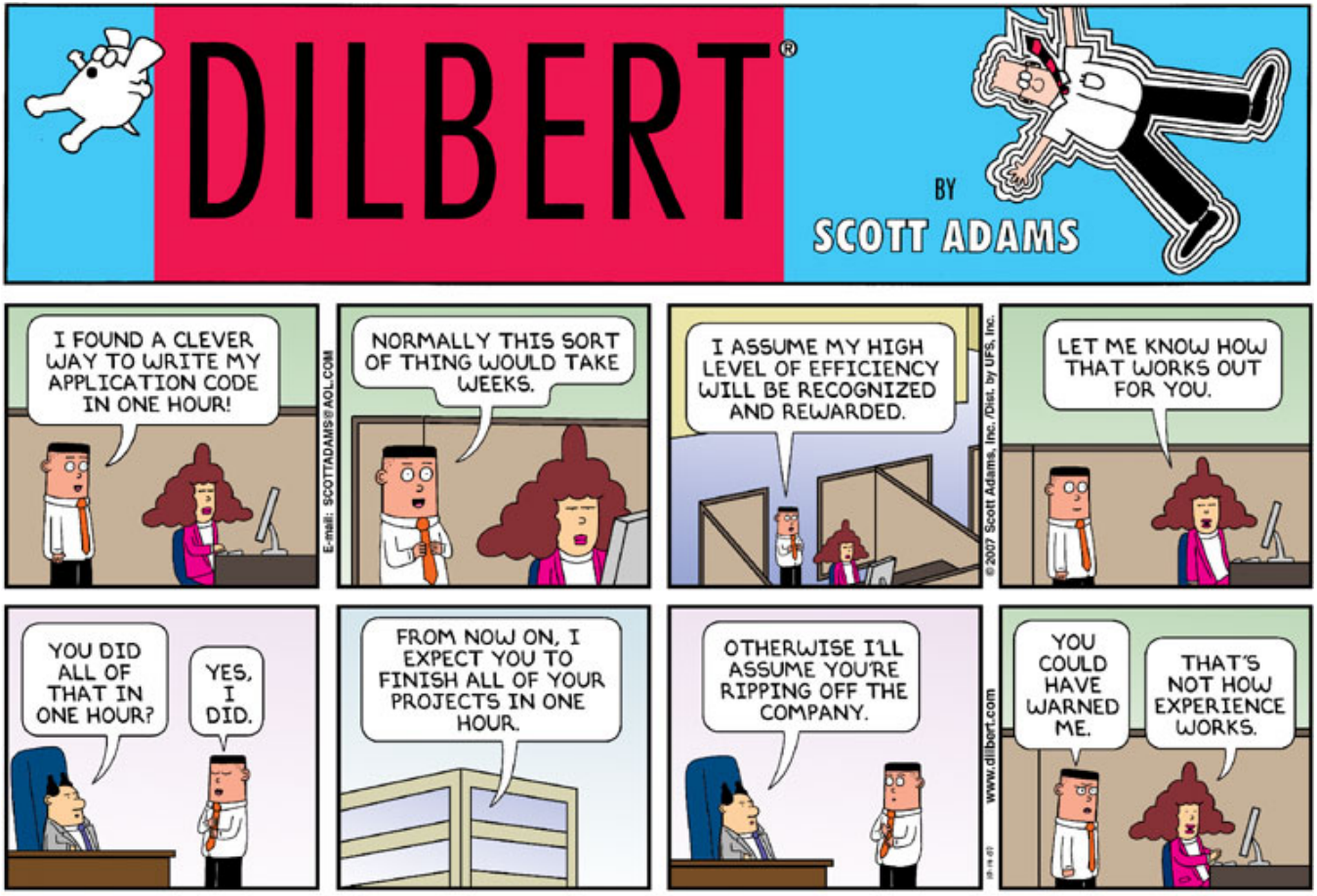
Jan van Eijck

jve@cwi.nl

Master SE, September 15, 2010

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.



What is the Use of Formal Methods?

- Why are formal methods useful for software engineering?
- Why are formal methods useful for software testing?
- Isn't software testing supposed to be practical?

A Letter I Received a Week Ago

```
from Tobias Schoofs <tobias.schoofs@gmx.net>
reply-to tobias.schoofs@gmx.net
to Jan.van.Eijck@cwi.nl
date Thu, Sep 9, 2010 at 9:05 PM
subject Exercise Solutions for "Haskell Road"
```

Dear Jan,

I have just started to read the "Haskell Road" and would be interested in getting the solutions to exercises.

Perhaps a word on my motivation to study your book: I am working for a software company (www.gmv.com) that is active in the area of critical software development. I am personally responsible for research programmes in the area of aeronautical and space on-board systems.

There is growing interest in the industry in what is often called "formal

methods". We have some experience with such formal methods, mainly by using commercial/open source tools (frama-c, Rodin, etc.). (If you are interested I can send you some information.) However, tools are still far from being perfect, costly or time-intensive, buggy or too limited.

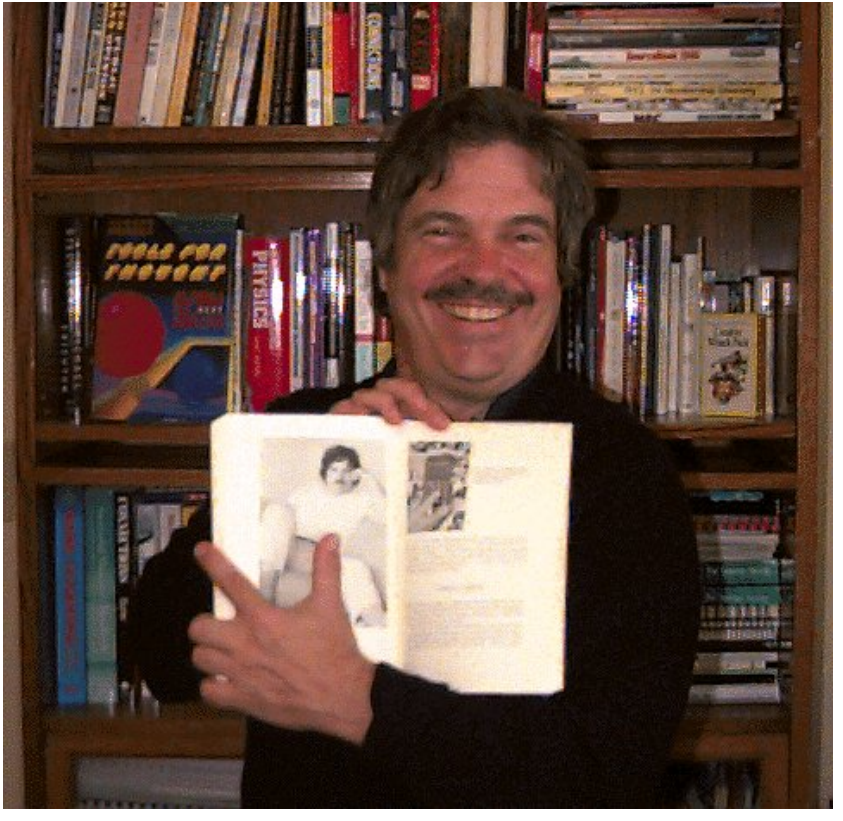
An alternative could be a kind of "grassroots" approach that aims at sharpening the understanding of programmers for their code, or, in other words, by teaching programmers to formally reason about the algorithms and data structures they tend to use by intuition. I am planning to create a course for this purpose (for the moment just for internal use). The method to use would be Haskell; the "Haskell Road" looks very promising as a possible outline for this course.

Thank you in advance.

Best regards,

Tobias

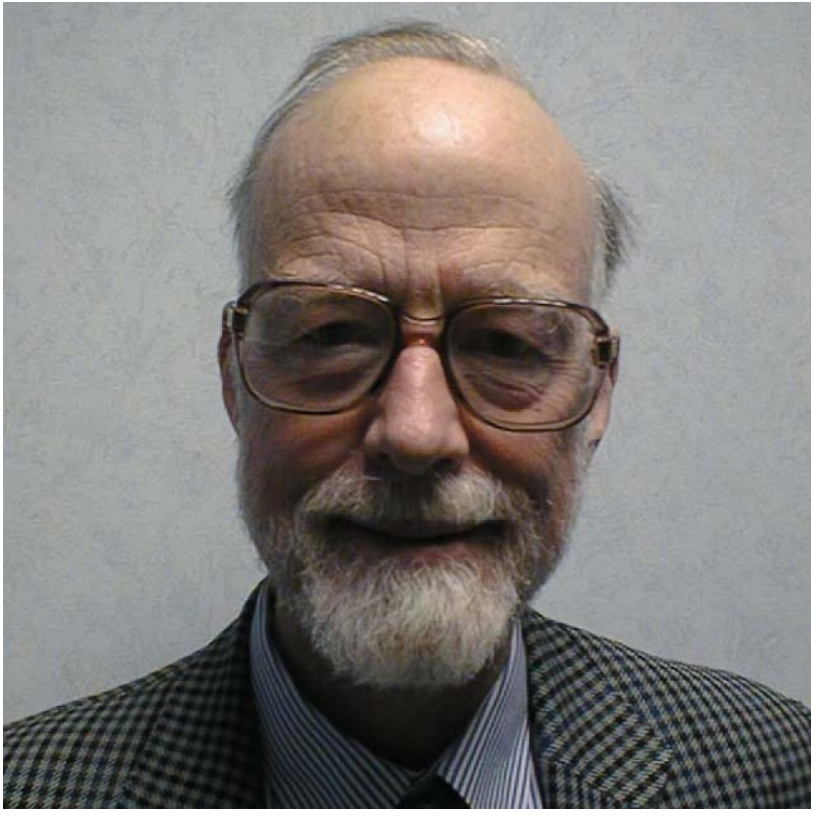
Alan Kay about Software ‘Engineering’



> "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]

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 Zeller [2005], 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)
- Making the designer of the software aware of what the software is supposed to do? (Tony Hoare)

What is Software Testing?

Five 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.
- Testing the methods by which the software was constructed.

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.

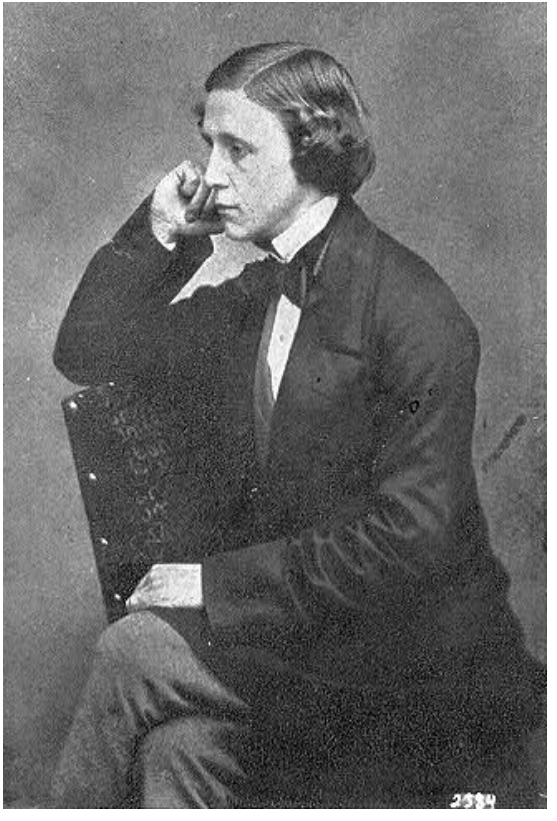
The reason for this is that partition testing would work better if we knew a priori that some subdomains contained more defects.

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 say 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?

A Puzzle About Probability



An urn contains a single marble, either white or black. Mr A puts another marble in the urn, a white one. The urn now contains two marbles. Next, Mrs B draws one of the two marbles from the urn. It turns out to be white. What is the probability that the other marble is also white?

The Monty Hall Puzzle

Probabilistic Functional Programming

See http://web.engr.oregonstate.edu/~erwig/pfp/

```
MontyHall> eval stay
Lose  66.7%
 Win  33.3%

*MontyHall> eval switch
 Win  66.7%
Lose  33.3%
```

Risk and Probability

Someone wants to borrow money from you, say 300 euros. He promises to pay you back in one year.

You know that there is a $\frac{1}{4}$ chance that he will not keep his promise.

How much should you ask him to pay you back in one year to compensate for the risk?

Risk and Probability 2

That was not very realistic.

Someone wants to borrow money from you, say 300 euros. He promises to pay you back in one year.

You don't know what the risk is that he will not keep his promise.

Suppose he has borrowed money from you twice before, and he has paid it back both times.

Can you still calculate how much you should ask him to pay you back in one year to compensate for the risk?

Relevance for Software Testing

http://journals.pepublishing.com/content/d507776477w13026

Using Bayesian statistics to support testing of software systems Journal Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability Publisher Professional Engineering Publishing ISSN 1748-006X (Print) 1748-0078 (Online) Issue Volume 221, Number 1 / 2008 Pages 85-93 DOI 10.1243/1748006XJRR2

Very Short Intro to Probability and Statistics

Suppose someone invites you to toss a coin repeatedly, and assures you that the coin is fair. You start tossing and this is what you get:

hhhhth

Do you still believe the coin is fair? You go on tossing, and this time you get:

hhthhhhhhthhh

At some point your initial belief that you are tossing a fair coin will be shaken. But why? The outcome of

hhhhthhhthhhhhhthhh

is just as likely as any other possible outcome . . . Or isn't it?

Laws of Conditional Probability

Let H be an event with positive probability. Let A be any event. Then we define:

$$P(A|H) = \frac{P(AH)}{P(A)}.$$

From this:

$$P(AH) = P(A|H) \cdot P(A).$$

Suppose $H_1, \ldots, H_n$ are mutually exclusive events, and their union is the whole sample space Ω. That is, one of the H_i necessary occurs. Then we have for any event A:

$$A = AH_1 \cup AH_2 \cup \cdots \cup AH_n.$$

Since the AH_i are mutually exclusive, their probabilities add:

$$P(A) = \sum_{j=1}^{n} P(A|H_j) \cdot P(H_j).$$

For the special case of H_j we have:

$$P(H_j|A) = \frac{P(AH_j)}{P(A)}.$$

Expanding $P(AH_j)$ and $P(A)$, we get:

$$P(H_j|A) = \frac{P(A|H_j) \cdot P(H_j)}{\sum_{i=1}^{n} P(A|H_i) \cdot P(H_i)}.$$

This is called Bayes' Law.

Applying Bayes' law to the problem of checking whether a coin is fair: http://en.wikipedia.org/wiki/Checking_whether_a_coin_is_fair.

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 test- ing 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. Learning functional programming is another.

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?

Relations

Usually, we are not only interested in meaningful collections of objects but also in the ways in which objects are related to each other.

Everyday examples of how objects can be related: motherhood, being married, or being the brother of someone's sister-in-law.

Mathematical relations: relations between numbers, like divisibility or being twin primes.

Formally, a relation between two sets A and B is a collection of ordered pairs (a, b) such that $a \in A$ and $b \in B$.

An ordered pair is a collection of two distinguishable objects, in which the order plays a role.

(Bonnie, Clyde) is the ordered pair that has Bonnie as its first element and Clyde as its second element.

The notation for the set of all ordered pairs with their first element taken from A and their second element taken from B is $A \times B$. This is called the Cartesian product of A and B.

A relation between A and B is a subset of $A \times B$.

Example: Talking about Chess

The Cartesian product of the sets $A = \{a, b, \ldots, h\}$ and $B = \{1, 2, \ldots, 8\}$ is the set

$$A \times B = \{(a, 1), (a, 2), \ldots, (b, 1), (b, 2), \ldots, (h, 1), (h, 2), \ldots, (h, 8)\}$$

of chess positions.

If we multiply the set of chess colours $C = \{\text{White}, \text{Black}\}$ with the set of chess figures,

$$F = \{\text{King}, \text{Queen}, \text{Knight}, \text{Rook}, \text{Bishop}, \text{Pawn}\},$$

we get the set of chess pieces $C \times F$.

If we multiply this set with the set of chess positions, we get the set of piece positions on the board, with $(\text{White}, \text{King}, (e, 1))$ indicating that the white king occupies square $e1$.

To get the set of moves on a chess board, take $((C \times F) \times ((A \times B) \times (A \times B)))$, and read $((\text{White}, \text{King}, ((e, 1), (f, 2))$ as 'white king moves from $e1$ to $f2$', but bear in mind that not all moves in $((C \times F) \times ((A \times B) \times (A \times B)))$ are legal in the game.

Exercise 1 Take A to be the set $\{\text{Kasparov}, \text{Karpov}, \text{Anand}\}$. Find $A \times A$.

$A \times A$ can also be written as A^2.

Binary Relations, Ternary Relations, . . .

Sets of ordered pairs are called binary relations.

Sets of ordered triples are called ternary relations.

An example of a ternary relation is that of borrowing something from someone. This relation consists of triples, or: 3-tuples, (a, b, c), where a is the borrower, b is the owner, and c is the thing borrowed.

In general, an n-ary relation is a set of n-tuples (ordered sequences of n objects). We use A^n for the set of all n-tuples with all elements taken from A.

Unary Relations: Properties

Unary relations on A are just subsets of A.

They are also called properties.

A property can always be represented as a set, namely the set that contains all entities having the property.

For example, the property of being divisible by 3, considered as a property of integer numbers, corresponds to the set

$$\{\ldots, -9, -6, -3, 0, 3, 6, 9, \ldots\}.$$

Composition of Relations

An important operation on binary relations is composition.

If R and S are binary relations on a set U, i.e., $R \subseteq U^2$ and $S \subseteq U^2$, then the composition of R and S, is the set of pairs (x, y) such that there is some z with $(x, z) \in R$ and $(z, y) \in S$.

Notation $R \circ S$

Example: the composition of $\{(1, 2), (2, 3)\}$ and $(2, 4), (2, 5)\}$ is $\{(1, 4), (1, 5)\}$.

Exercise 2 What is the composition of $\{(n, n + 1) \mid n \in \mathbb{N}\}$ with itself?

Relational Converse, Symmetry

Another operation on binary relations is converse.

If R is a binary relation, then $R^{\smile}$ is the relation given by $R^{\smile} = \{(y, x) \mid (x, y) \in R\}$.

The converse of the relation 'greater than' on the natural numbers is the relation 'smaller than' on the natural numbers. If a binary relation has the property that $R^{\smile} \subseteq R$ then R is called symmetric.

Exercise 3 Show that it follows from $R^{\smile} \subseteq R$ that $R = R^{\smile}$.

Identity Relation, Reflexivity

If U is a set, then the relation $I = \{(x, x) \mid x \in U\}$ is called the identity relation on U.

If a relation R on U has the property that $I \subseteq R$, then R is called reflexive.

The relation $\leq$ ('less than or equal') on the natural numbers is reflexive, the relation $<$ ('less than') is not.

Transitivity

A relation R is called transitive if it holds for all x, y, z that if $(x, y) \in R$ and $(y, z) \in R$, then also $(x, z) \in R$.

To say that the relation of 'friendship' is transitive boils down to saying that it holds for anyone that the friends of their friends are their friends.

Exercise 4 Which of the following relations are transitive?

1. $\{(1, 2), (2, 3), (3, 4)\}$.
2. $\{(1, 2), (2, 3), (3, 4), (1, 3), (2, 4)\}$.
3. $\{(1, 2), (2, 3), (3, 4), (1, 3), (2, 4), (1, 4)\}$.
4. $\{(1, 2), (2, 1)\}$.
5. $\{(1, 1), (2, 2)\}$.

Exercise 5 Check that a relation R is transitive if and only if it holds that $R \circ R \subseteq R$.

Exercise 6 Can you give an example of a transitive relation R for which $R \circ R = R$ does not hold?

Functions

Functions are relations with the following special property: for any (a, b) and (a, c) in the relation it has to hold that b and c are equal.

Thus a function from a set A (called domain) to a set B (called range) is a relation between A and B such that for each $a \in A$ there is one and only one associated $b \in B$.

A function can be viewed as a mechanism that maps an input value to a uniquely determined output value.

Functions are important because they allow us to express the concept of dependence.

Extensional View of Functions

Functions can be seen as sets of data, represented as a collection of pairs of input and output values. This tells us something about the behaviour of a function, i.e. what input is mapped to which output. As an example, consider the following conversions between temperature scales.

Kelvin	Celsius	Fahrenheit	
0	-273.15	-459.67	(absolute zero)
273.15	0	32	(freezing point of water)
310.15	37	98.6	(human body temperature)
373.13	99.98	211.96	(boiling point of water)

Intensional View of Functions

Another way to look at functions is as instructions for computation. This is called the intensional view of functions.

In the case of temperature conversion the intensional view is more convenient than the extensional view, for the function mapping Kelvin to Celsius can easily be specified as a simple subtraction

$$x \mapsto x - 273.15$$

This is read as 'an input x is mapped to x minus 273.15'. Similarly, the function from Celsius to Fahrenheit can be given by

$$x \mapsto x \times \frac{9}{5} + 32$$

Calculation as Rewriting Expressions

If we have a temperature of 37 degrees Celsius and want to convert it to Fahrenheit, we replace x by 37 and compute the outcome by multiplying it with $\frac{9}{5}$ and then adding 32.

$$37 \times \frac{9}{5} + 32 \rightarrow 66.6 + 32 \rightarrow 98.6$$

This shows that the intensional view of functions can be made precise by representing the function as an expression, and specifying the principles for simplifying (or: rewriting) such functional expressions.

Rewriting functional expressions is a form of simplification where part of an expression is replaced by something simpler, until we arrive at an expression that cannot be simplified (or: reduced) any further. This rewriting corresponds to the computation of a function.

Function Composition

Functions can be composed, as follows. Let g be the function that converts from Kelvin to Celsius, and let f be the function that converts from Celsius to Fahrenheit. Then $f \cdot g$ is the function that converts from Kelvin to Fahrenheit, and that works as follows.

First convert from Kelvin to Celsius, then take the result and convert this to Fahrenheit. It should be clear from this explanation that $f \cdot g$ is defined by

$$x \mapsto f(g(x)),$$

which corresponds to

$$x \mapsto (x - 273.15) \times \frac{9}{5} + 32$$

Exercise 7 The function $s : \mathbb{N} \to \mathbb{N}$ on the natural numbers is given by $n \mapsto n + 1$. What is the composition of s with itself?

Characteristic Functions

The characteristic function of subset A of some universe (or: domain) U is a function that maps all members of A to the truth-value **True** and all elements of U that are not members of A to **False**. E.g. the function representing the property of being divisible by 3, on the domain of integers, would map the numbers

$$\ldots, -9, -6, -3, 0, 3, 6, 9, \ldots$$

to **True**, and all other integers to **False**. Characteristic functions characterize membership of a set. Since we specified relations as sets, this means we can represent every relation as a characteristic function.

Exercise 8 $\leq$ is a binary relation on the natural numbers. What is the corresponding characteristic function?

Equivalence

An equivalence relation on A is a binary relation on A that is reflexive, symmetric and transitive.

Exercise 9 Let $f : A \to B$ be a function. Show that the relation $R \subseteq A^2$ given by $(x, y) \in R$ if and only if $f(x) = f(y)$ is an equivalence relation on A.

First Order Logic: Syntax, LAI Section 6.2

Grammar for the language of predicate logic (see LAI, page 139):

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

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: LAI Section 6.3

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: LAI pages 150–152.

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.

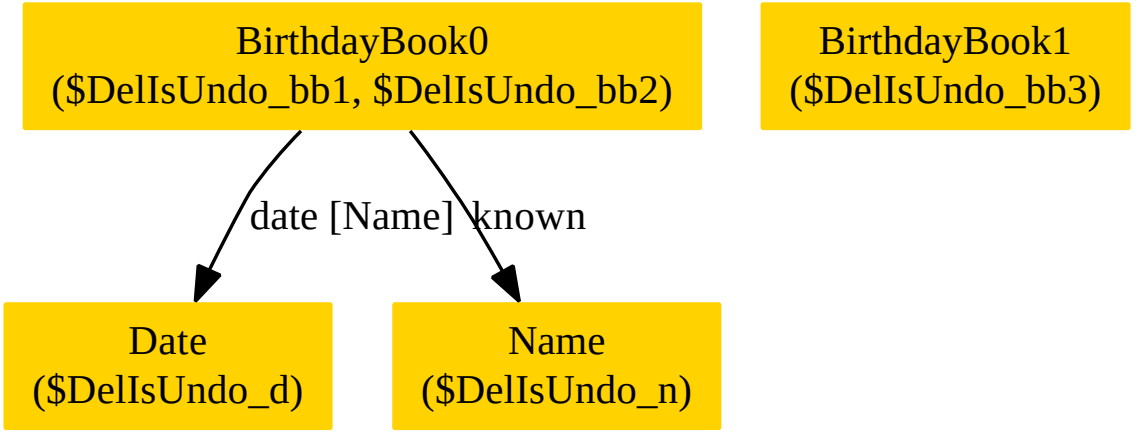
Will this succeed or fail?

The second check checks the predicate DelIsUndo, 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
```



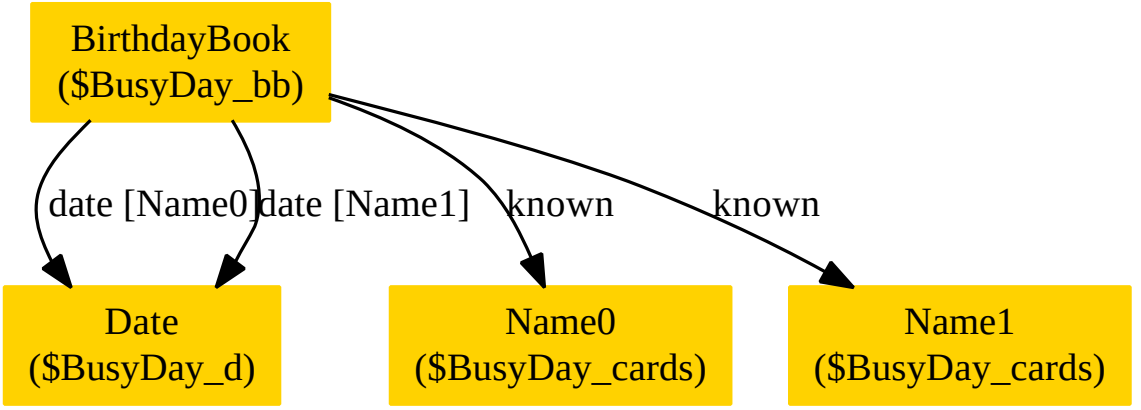
Predicates defined with Formulas

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

What does this say?

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

Example


BirthdayBook
($BusyDay_bb)
date [Name0]
date [Name1]
known
known
Date
($BusyDay_d)
Name0
($BusyDay_cards)
Name1
($BusyDay_cards)

References

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.