Software Verification Tools

Part I

Peter H. Roosen-Runge

Department of Computer Science
York University
Toronto, Canada

© P. H. Roosen-Runge, 1999, 2000

"It is very difficult indeed to find programs which have been proven correct, and those which are are hailed like freaks in a circus." [Leith, 1990]

0. INTRODUCTION

This text explores the problem of verifying software in terms of a set of simple tools which can be used to symbolically evaluate and prove properties of pieces of programs, represented either abstractly in functional terms, or by actual text. The tools can be thought of as roughly analogous to *spelling and style checkers* in word-processing; they reduce the labor of finding errors and provide some semi-automated aids to making corrections. Like the spelling and style checkers, the use of software verification tools cannot guarantee the absence of errors but they can block certain kinds of errors, and sometimes they can go further by allowing the programmer to derive parts of a program automatically from logical specifications. These specifications provide a representation of the program's intended meaning or goal; they say *what* the program is supposed to do rather than *how* it is to do it. Thus they are easier to understand than the details of the program itself, and can be directly accepted or rejected depending on whether they match or fail to match the requirements of the program's users.

Just as a spelling checker cannot help a writer find something meaningful to say, so verification tools cannot help with the most important aspects of program construction: *what exactly is the program supposed to do? what resources are to be used?* To answer these questions effectively requires an understanding which we cannot expect from a software tool, but the use of the tool can reduce the most immediate sources of errors—mistakes and misunderstandings in the low-level logic, which will not be caught by the compiler and which may indeed go undetected for many executions of the program, until the day arrives with just the right conditions to trigger a "crash".

0.1 Why do we need software verification tools?

Software errors are expensive. It has been estimated that in North America perhaps $3 billion a year is lost in "crashes", with the average cost of a major outage running at $330,000[1]. The airlines estimate that every minute a reservation system is down costs $70,000. But the error rate in programming resists any significant reduction: error estimates for programmers are thought to be about 50 errors for every 1000 lines of code (regardless of programming language), with fully tested market versions running about 3 errors/1000 lines. (Programs written in high-level languages are much shorter than equivalent low-level code, so a high-level language is a much safer programming tool than a low-level language would be; but, at the same time, the use of high-level languages makes feasible the development of much larger, more complex programs than could be attempted with low-level languages, so the error rate/program does not necessarily decrease.)

---

[1] "It has been reported that British businesses suffer around 30 major mishaps a year, involving losses of millions of pounds. The cost of software failures alone in the UK is conservatively estimated at $900 million per year ." [Forester, 1992]

We present a few examples of major software errors to illustrate the very serious consequences which can arise from the smallest of errors in program logic, and the value to be attached to verification tools if they are able to reduce the number of such failures:

The Bank of New York disaster

The following article describes a famous computer "foul-up" triggered by a simple programming error—using a 2-byte counter for the number of messages waiting. The counter had a maximum value of +32,767; adding 1 to this maximum value produced a value which was interpreted by the software as a 0, indicating no messages (in this case, orders for bond trades) were waiting! Note how the original software error became compounded into a further more disastrous error, and the complex way the error effected the environment of interest rates and bond sales.

Quoted from an article on the risks mailing list, now run as the comp.risks Usenet newsgroup:

> "Date: Fri 29 Nov 85 00:43:47-EST
> . . . .
> From the *Wall Street Journal*, Monday 25 November 1985 [quoted without permission],
> A Computer Snafu Snarls the Handling of Treasury Issues by Phillip L. Zweig and Allanna Sullivan Staff reporters of the Wall Street Journal
>
> NEW YORK- A computer malfunction at Bank of New York brought the Treasury bond market's deliveries and payments systems to a near- standstill for almost 28 hours Thursday and Friday. . . . .The foul-up temporarily prevented the bank, the nation's largest clearer of government securities, from delivering securities to buyers and making payments to sellers . . . The malfunction was cleared up at 12:30 p.m. EST Friday, and an hour later the bank resumed delivery of securities.
>
> But Thursday the bank, a unit of Bank of New York Co., had to borrow a record $20 billion from the Federal Reserve Bank of New York so it could pay for securities received. The borrowing is said to be the largest discount window borrowing ever from the Federal Reserve System. Bank of New York repaid the loan Friday, Martha Dinnerstein, a senior vice president, said. . . . Bank of New York said that it had paid for the cost of carrying the securities so its customers wouldn't lose any interest. Bank of New York's inability to accept payments temporarily left other banks with $20 billion on their hands. This diminished the need of many banks to borrow from others in the federal funds market. Banks use the market for federal funds, which are reserves that banks lend each other, for short-term funding of certain operations. The cash glut caused the federal funds rate to plummet to 5.5% from 8.375% early Thursday.
>
> . . . According to Wall Street sources, the malfunction occurred at 10 a.m. Thursday as Bank of New York was preparing to change software in a computer system and begin the days operations. Until Friday afternoon, Bank of New York received billions of dollars in securities that it couldn't deliver to buyers. The Fed settlement system, which officially closes at 2:30 p.m., remained open until 1:30 a.m. Friday in the expectation that technicians would be able to solve the problem.
>
> . . . it seems that the primary error occurred in a messaging system which buffered messages going in and out of the bank. The actual error was an overflow in a counter which was only 16 bits wide, instead of the usual 32. This caused a message database to become corrupted. The programmers and operators, working under tremendous pressure to solve the problem quickly, accidentally copied the corrupt copy of the database over the backup, instead of the other way around."

The all-time most expensive single-character error?

Quoted from comp.risks:

Date: Sat 10 Sep 88 15:22:49-PDT
From: Gary Kremen (The Arb) <89.KREMEN@GSB-HOW.Stanford.EDU> Subject: Soviets See Little Hope of Controlling Spacecraft

According to today's (Saturday, September 10, 1988) New York Times, the Soviets lost their Phobos I spacecraft after it tumbled in orbit and the solar cells lost power. The tumbling was caused when a ground controller gave it an improper command. This has to one of the most expensive system mistakes ever. [estimated at $100,000,000]

Date: Tue, 13 Sep 88 15:20:18 PDT
From: Peter G. Neumann <Neumann@KL.SRI.COM>
Subject: Soviet Mars Probe

The Soviet Mars probe was mistakenly ordered to ‘commit suicide’ when ground control beamed up a 20 to 30 page message in which a single character was inadvertently omitted. The change in program was required because the Phobos 1 control had been transferred from a command center in the Crimea to a new facility near Moscow. ‘The [changes] would not have been required if the controller had been working the computer in Crimea.’ The commands caused the spacecraft's solar panels to point the wrong way, which would prevent the batteries from staying charged, ultimately causing the spacecraft to run out of power.

[From the SF Chronicle, 10 Sept 88, item (page A11), thanks to Jack Goldberg.]

The Ariane 5 failure

Quoted from the report of the Ariane Inquiry Board [Lions, 1996]:

“On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. ,. . . . The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception. . . .The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected. . . .

An underlying theme in the development of Ariane 5 is the bias towards the mitigation of random failure. . . The exception was detected, but inappropriately handled because the view had been taken that software should be considered correct until it is shown to be at fault. The Board has reason to believe that this view is also accepted in other areas of Ariane 5 software design. The Board is in favour of the opposite view, that software should be assumed to be faulty until applying the currently accepted best practice methods can demonstrate that it is correct.
. . . .
R5 . . . .Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment.
. . .
Verify the range of values taken by any internal or communication variables in the software.”

The Phobos and Ariane failures demonstrate two important lessons:

- code cannot be trusted just because it worked in a different context ;
- there is no connection between the scope of the error and the cost of the resulting failure.

Of the many recommendations made by the Ariane Inquiry Board, two are directly relevant to the issue of verification. As we shall see, *identifying implicit assumptions* and *verifying ranges of values* are both problems for which our verification tools can help.

Millions inconvenienced, millions paid in compensation

The following report shows how obscure the source of a major software crash can be, and how unconvincing the inevitable claims are that everything possible is being done to prevent the re-occurrence of such errors. Dijkstra's famous dictum says it all: *Testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence, never their absence.* [Dijkstra, 1972.] And Weinberg's mottoes sharpen the point:

> *"There's always one more bug, even after that one is removed."*
>
> *"An ounce of prevention is worth a pound of cure, but management won't pay a penny for it."* [Weinberg, 1982, p. 83]

Quoted from comp.dcom.telecom:

> Subject: AT&T Crash Statement: The Official Report
> Technical background on AT&T's network slowdown, January 15, 1990
>
> At approximately 2:30 p.m. EST on Monday, January 15, one of AT&T's 4ESS toll switching systems in New York City experienced a minor hardware problem . . . This required the switch to briefly suspend new call processing until it completed its fault recovery action -- a four-to-six second procedure.
>
> . . . messages were automatically sent to connecting 4ESS switches requesting that no new calls be sent to this New York switch during this . . . interval. The switches receiving this message made a notation to show that the switch was temporarily out of service.
>
> When the New York switch in question was ready to resume call processing a few seconds later, it sent out call attempts (known as IAMs - Initial Address Messages) to its connecting switches. When these switches started seeing call attempts from New York, they started making adjustments to their programs to recognize that New York was once again up-and-running, and therefore able to receive new calls.
>
> A processor in the 4ESS switch which links that switch to the CCS7 network holds the status information mentioned above. When this processor (called a Direct Link Node, or DLN) in a connecting switch received the first call attempt (IAM) from the previously out-of-service New York switch, it initiated a process to update its status map. As the result of a software flaw, this DLN processor was left vulnerable to disruption for several seconds. During this vulnerable time, the receipt of two call attempts from the New York switch -- within an interval of 1/100th of a second -- caused some data to become damaged. The DLN processor was then taken out of service to be reinitialized.

> Since the DLN processor is duplicated, its mate took over the traffic load. However, a second couplet of closely spaced new call messages . . hit the mate processor during the vulnerable period, causing it to be removed from service and temporarily isolating the switch from the CCS7 signaling network. The effect cascaded through the network as. . .other switches similarly went out of service. The unstable condition continued because of the random nature of the failures and the constant pressure of the traffic load in the network providing the call-message triggers.
>
> The software flaw was inadvertently introduced into all the 4ESS switches . . .as part of a mid-December software update. This update was intended to significantly improve the network's performance by making it possible for switching systems to access a backup signaling network more quickly in case of problems with the main CCS7 signaling network. While the software had been rigorously tested in laboratory environments before it was introduced, the unique combination of events that led to this problem couldn't be predicted. . . .
>
> On Tuesday, we took the faulty program update out . . and temporarily switched back to the previous program. . . We have since corrected the flaw, tested the change and restored the backup signaling links. We believe the software design, development and testing processes we use are based on solid, quality foundations. All future releases of software will continue to be rigorously tested. We will use the experience we've gained through this problem to further improve our procedures.
>
> . . . Although nothing can be guaranteed 100% of the time, what happened Monday was a series of events that had never occurred before. With ongoing improvements to our design and delivery processes, we will continue to drive the probability of this type of incident occurring towards zero.

The AT&T system failure was triggered by a system upgrade. This is a very common hazard. It is quite difficult to establish through prior testing that an upgrade to a complex systems will not have an unexpected and possibly disastrous effect—as in a recent incident at the Canadian Bank of Commerce, which for two days could not record deposits or payments through its automatic teller machines because, as a newspaper report had it, "attempts by the bank to improve one of their computer systems caused problems with another system, throwing everything out of whack." [Globe & Mail, May 2, 1998].

Testing is not proof

The AT&T report implicitly connects confidence in the correct functioning of its software with the quality of its testing procedures. For some, testing is even confused with *proving* correctness as illustrated by the following posting to comp.risks:

> Date: Fri, 18 Jul 1997 14:10:53 -0400 (EDT)
> From: "Daniel P. B. Smith" <dpbsmith@world.std.com>
> Subject: Unique definition of "proof of correctness"
>
> *Computer Design*, July 1997, p. 38, describes the IDT-C6, a
> Pentium-compatible microprocessor designed by Centaur Technology. "IDT
> claims to have tested the C6 with most modern PC operating systems,
> including Windows 95, Windows 3.1, NetWare, and Solaris for Intel. _The
> complexity and pervasiveness of the Windows operating system generally are
> considered to make up an exacting proof of correctness..._" [emphasis
> supplied]
>
> Give up, folks... the marketers have won. Once they redefine a term, the engineers _never_ win it back...

As we shall see in this text, proofs of correctness are constructed by the application of logic and mathematics to the structure of the program code, not its behaviour when run in a particular environment with particular data. Proof of correctness, i. e., verification, does not involve testing and achieves quite different results.

0.2 Life-threatening software errors

Programmer error in air traffic control software

In the following example [Lamb, 1988], failure to allow for negative values could have had fatal consequences. (Points east of 0° longitude—which is, by definition, the meridian passing through the Royal Observatory at Greenwich, England—can be specified by negative longitudes. In this case, the program apparently ignored the sign of the longitude value.) The program in question was used for air traffic control in the London (England) area by the Civil Aviation Authority (CAA):

> “One of the more startling problems concerned the program's handling of the Greenwich meridian. The National Airspace Package, designed by IBM's Federal Systems division, contains a model of the airspace it controls, that is, a map of the airlanes and beacons in the area. But, because the program was designed for air traffic control centres in the US, the designers had taken no account of a zero longitude; the deficiency caused the computer to fold its map of Britain in two at the Greenwich meridian, plonking Norwich on top of Birmingham. . . .
>
> It’s axiomatic that software has bugs in it.” said Stan Price. He bought the software for the CAA but is critical of the CAA’s failure to use the latest techniques for producing software. Techniques that are lacking, notes Price, include the use of formal methodologies for designing and writing software.”

Ambulance Dispatch System failure

In November 1992, the computerized dispatch system for the London (England) Ambulance Service, broke down suddenly and "locked up altogether", leaving calls for assistance which had already been stored in the system unavailable, and forcing a taxing and cumbersome return to a manual, paper-based dispatch system. Luckily, the breakdown occurred in early morning hours, so that there were no severe consequences for patients.

The cause of the breakdown? Quoting from the subsequent inquiry report ["Report", 1993]:

> 4039 The Inquiry Team has concluded that the system crash was caused by a minor programming error. In carrying out some work on the system some three weeks previously the SO programmer had inadvertently left in memory a piece of program code that caused a small amount of memory within the file server to be used up and not released every time a vehicle mobilization was generated by the system. Over a three week period these activities had gradually used up all available memory thus causing the system to crash. This programming error should not have occurred and was caused by carelessness and lack of quality assurance of program code changes. Given the nature of the fault it is unlikely that it would have been detected through conventional programmer or user testing.

Software that kills

It is remarkable that even the largest and most sophisticated corporations, such as AT&T, which stand to lose so much financially from defective software have invested so little in effective verification tools, but it is even more remarkable that software on which people depend on for their lives (such as the software used to control of the latest "fly-by-wire" models of passenger planes[2]) is often written using low-level programming techniques known to be prone to error, as well as being very difficult to debug, while at the same time more and more reliance is placed on software-based safety mechanisms rather than on simpler but more reliable hardware guards and interlocks. As Robert Baber has remarked, "developing in the traditional way software upon which human life depends is nothing other than high tech Russian roulette." [Baber, 1991, p. 123.] Moser and Melliar-Smith [1990] have commented: "For systems that must be safe, there appears to be no alternative to verification. Testing, fault tolerance, and fault-tree analysis can make important contributions to safety, but they alone are unlikely to achieve the high levels of safety that are necessary for safety-critical applications."

A clear example of the risks of poor programming and verification techniques is the tragic story of the Therac-25—one in a series of radiation therapy machines developed and sold over a number of years by Atomic Energy Canada Limited (AECL). As a direct result of inadequate programming techniques and verification techniques, at least six patients received massive radiation overdoses which caused great pain and suffering and from which three died. The story is a complicated one; it is presented in comprehensive detail in [Levenson, 1993]. A few points are noteworthy:

---

[2] Forester and Morrison [1990] report that errors in the computer-based 'fly-by-wire' system of the US Air Force's Blackhawk helicopter resulted in 22 fatalities in five separate crashes.

The Therac-25 accidents occurred because of coding errors, not, as is more common for software-caused accidents, because of errors in the requirements or specifications. The software itself was never tested or formally analyzed at the module level; indeed, the initial safety analysis ignored the software altogether, even though key hardware interlocks in earlier models had been replaced by software. Even when problems were noticed, the investigators focused on the hardware, apparently preferring to attempt to "fix" the hardware rather than tackle the daunting task of debugging very complex, poorly documented and poorly structured code.

Levenson and Turner warn against simply blaming tragedies such as the Therac-25 accidents on software errors alone—they argue for a deeper analysis of "organizational, managerial, technical" and even "political, or sociological factors" on the grounds that, as we have already observed, there will always be another bug; so focusing only on eliminating individual software errors may blind us to considering the largerenvironment which tolerates and unintentionally furthers the occurrence of such errors with their drastic consequences.

True enough, but one of the underlying causes of unsafe software may be the general reluctance to apply formal methods and verification techniques, usually on the grounds that they are too difficult to automate and hence cannot be applied to "real code". One purpose of this text is to argue by example against the validity of this excuse—to show that through the use of appropriate tools, portions of code can be verified automatically or semi-automatically, or can be generated automatically from logical specifications. Just as we would expect any author who uses a word-processor to use a spelling checker to check for at least those spelling errors which the dictionary can catch, so we should — in the future, if not today — expect programmers to routinely check or generate their code with software verification tools to catch and eliminate at least those errors which the tools can prevent. Even then programmers will not succeed in stamping out that vexatious and stubborn "last bug", but at least its available hiding places will grow smaller, and both money and lives may be saved.

0.3 Tools for logic and tools for correctness

Two kinds of tools are discussed in this text—the first do purely logical calculations; the second kind use the logical calculation tools to check program text for logical correctness.

In the first category are tools which can

- test a propositional function to see if it is a tautology (theorem) in the propositional calculus;

- solve some logic puzzles and word problems automatically when they are translated into an appropriate propositional form;

- simplify propositions involving arithmetic and logical relations into forms whose truth or validity can be automatically checked.

In the second category are tools which

• calculate automatically a logical description of the computation performed by a loop-free code fragment and use this to verify the correctness of the code relative to a goal;

• verify that a functional calculation on an instance of an abstract datatype meets its specification;

• check that the conditions in a loop-free conditional statement allow all branches to be taken;

• automatically calculate a logical description of initial conditions for a code fragment which will cause the fragment to compute a desired final condition;

• test whether the conditions specified for a loop allow the loop to correctly compute the desired final condition from given initial conditions.

The tools are written in the logic-programming language Prolog and use many of the features of the Prolog interpreter for parsing input and displaying output results. The examples shown in the text are output from versions of the tools developed in Quintus Prolog[3], running under the Solaris operating system; they require that a licensed Quintus development system be present on the system. The source code is available for non-commercial purposes in ftp://ftp.cs.yorku.ca/pub/peter/SVT.

Stand-alone versions of the tools that will run under Unix or Windows are easily generated from the Prolog source code, using SWI-Prolog[4]. (This is freely available under a GPL license.) The SWI-Prolog source code is available for non-commercial purposes in ftp://ftp.cs.yorku.ca/pub/peter/SVT/SWI/.

The reader will find some acquaintance with Prolog concepts to be helpful, but the text is intended to be understandable and the tools are usable without any Prolog background.

0.4 An overview of the text

The text is divided into two parts. Part I introduces the basic concepts of propositional logic as a method of description of program states, and presents the tools `tautology` (Chapter 2) and `wang` (Chapter 3) used to check whether a propositional function is a tautology. An additional tool `simplify` is introduced in Chapter 4 to simplify expressions and equalities, and this is combined with `wang` to produce a simple theorem-proving tool `prover` which, while far from complete even for simple arithmetic domains, is transparent and easily extensible. Chapter 5 applies these tools to the verification of expressions on abstract data types, and with the addition of the `induction` tool, it becomes possible to verify automatically some recursive programs expressed in a functional programming language.

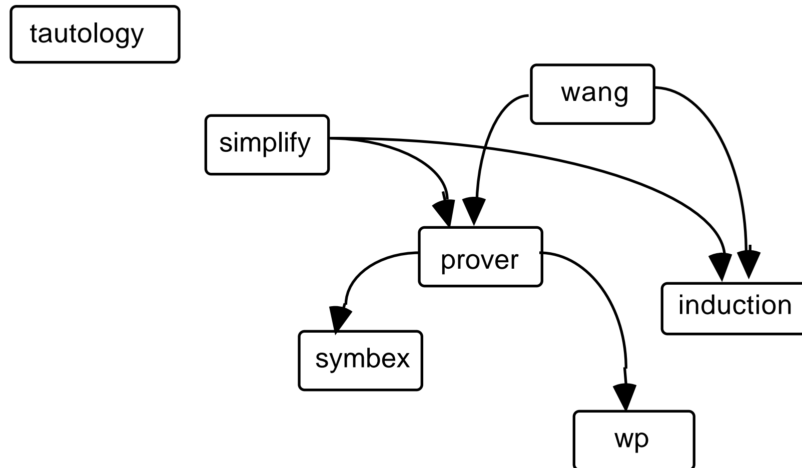
In Part II, we tackle the more difficult problem of code with side-effects—code which alters the state of the computer's memory by assigning values to variables. A more complex application of `prover` is in the verification that the implementation of an abstract datatype is correct (Chapter 6). Chapter 7 introduces the concept of

[3] http://www.sics.se/isl/quintus/
[4] http://www.swi.psy.uva.nl/projects/SWI-Prolog.

specifications and shows how we can automatically prove some properties of specifications such as refinement relations. To give the reader a better sense of why understanding procedural code is difficult, and its verification complex, we examine in Chapter 8 what can be learned from symbolic execution, using the `symbex` tool.

The calculation of preconditions required to achieve a specification, using the `wp` tool, is introduced in Chapter 9 as an alternative to symbolic execution, and this is applied to the verification of code containing a variety of statement types.

Here is how the logic and correctness tools are related:



1. PROPOSITIONS AND PROPOSITIONAL FUNCTIONS

Propositions are statements constructed out of Boolean operators and phrases which can be unambiguously assigned a value of true or false. They are used in the process of checking code fragments to describe the state of a program at a particular point in its computation.

The traditional Boolean operators are: not, and, or, implies and equivalent. These are often represented by ~, , , and in logic and mathematics texts. Other symbols are also used, and various equivalents are found in different programming languages, as shown in the following table:

logic	English	C, Java	Prolog
~, ¬	not	!	\+, not
, .	and	&&	,
, +	or	||	;
	xor	^	
, ,	implies		->
	if		:-
=, ,	iff, equivalent		

(iff is an abbreviation for the bi-conditional *if and only if* .)

We will need to represent propositions in a consistent way using the usual alphabet of ASCII characters available on a keyboard so that we can easily input propositions from the keyboard or a file to a verification tool. We will therefore use the English words not, and, or, implies, and one piece of mathematical jargon, iff, as names for the logical operators[5]. Occasionally it is convenient to use the exclusive-or operation ("either this or that but not both") which we represent by the operator xor.

The simplest propositions are the Boolean constants true, false, and names such as p which are assumed to have either true or false as their values; the next simplest are expressions constructed from Boolean constants and names together with Boolean operations: for example

p or (not r and true).

In such expressions, the truth value of the entire expression is a constant determined by the fixed Boolean values associated with each name. Simple propositions of this sort are rarely of much use as applied to the description of program states; but, as we shall see, propositional *functions*, as opposed to constant propositions, are indeed useful for descriptive purposes, and that is what motivates our use of the propositional calculus in verification.

Propositional functions are functions constructed from propositions in which the variables are treated as the parameters of the function. Every assignment of truth values to the variables produces a value of the function. To distinguish this case from the

---

[5] This approach is used in the Object Constrain Language (OCL) component of the Unified Model Language (UML) standard, with the exception of iff which is represented by = . See [Warmer, 1999].

preceding (where the variables have some fixed value), we will write the Boolean parameters of a propositional function with upper case letters, e. g.:

P implies (not P or R).

In this case, we do not assume that P and R have fixed values.

Propositional functions are more commonly called *propositional formulae*. Strictly speaking, they should be distinguished from propositions like "Shakespeare is alive" or "2 and 2 is 4" which have a fixed truth value. However, we will use the term *proposition*, and *propositional function* or *propositional formula* interchangeably, since the form of the expression and the context will make clear which sense is intended.

1.1 Classifying propositional functions

A good question to ask about a propositional function is "Can it ever have the value false?". (In the example above, setting P = true and R = false produces that result.)

What lies behind the question is the fact that if the answer is "no, it is never false, regardless of the values assigned to its variables", then the function is a ***tautology***; it is a **valid** logical statement; it is a **logical truth**. The way the question was put suggests a boring but workable method of calculating the answer—try successive combinations of true and false values for the variables and for each combination, evaluate the Boolean expression. If the result is ever false, we know the function is not a tautology; otherwise it is. This is the method used by the `tautology` program described in Chapter 2 to check if an input is a tautology.

As we shall see later, the question of whether a propositional function is a tautology is sometimes worth asking with respect to parts of a program. Suppose, for example, a program has an IF-statement of the form

```
if (B) S1; else S2;
```

then if `B` is a tautology, the statement `S2` will never be executed and there is probably an error in the statement of the condition.

The concept of a ***contradiction*** is also useful; a contradiction is a propositional function which is never true, hence

- the negation of a contradiction is a tautology,
- the negation of a tautology is a contradiction.

A propositional function which is not a contradiction is called **satisfiable** since it is at least sometimes true—it has the value true for some assignment of Boolean values to its variables.

Propositional functions which are neither tautologies nor contradictions are called **contingent**; for some values of their variables, they are true and for others, they are false.

In any conditional statement

```
if (B) . . .
```

the condition `B` should be contingent, for if it isn't, we don't need a conditional at all. When `B` is a tautology, we can just delete the **`if (B)`** and the **`else`** clause, if present. If `B` is a contradiction, we can delete the **`if (B)`** and the statement which follows it—in either case the computation performed by the statement is unaffected[6].

To classify a propositional function as a contradiction, satisfiable, or contingent, we need only a method to check for tautology. It can be used to check for the other properties as follows:

- P is a contradiction if not P is a tautology;
- P is satisfiable if not P is not a tautology;
- P is contingent if neither P nor not P are tautologies.

This gives us a first small taste of a method for checking a logical property of a piece of code— extract the conditional expression C from a conditional statement and apply a method for checking tautologies to both C and not C.

Verification scenarios

To make it easier to keep track of the various forms of verification which we will develop in subsequent chapters, we introduce a template for describing compactly what gets verified in a particular verification "scenario" and how:

title: a verification task
context: the type of situation to which the task applies
method: the tools and technique for carrying out the task
background: assumptions or additional knowledge used in the task

For the case of conditional expressions, the scenario can be summarized as follows:

check whether the condition in a conditional statement is dispensable.
context: if (B) S1 else S2
method: test whether either B or not B is a tautology.
background: B is a Boolean expression whose evaluation has no side-effects on either S1 or S2.

Our next verification scenario will show up in Chapter 5; the intervening chapters develop the logical tools which will be needed.

[6] This assumes that the evaluation of B has no side effect on the state of the program.

2. A TAUTOLOGY CHECKER

2.1 Invoking the `tautology` program

The program `tautology` checks propositional functions to see if they are *valid* (tautologies) or *invalid* (non-tautologies.) To run the program, we construct a data file containing propositional formulae constructed from variable names beginning with upper-case letters, the words `true` and `false`, the operators[7] `not`, `and`, `or`, `xor`, `implies`, `iff`, and parentheses. Each formula must end with a period (full stop).

Here is a sample data file:

```
false implies A.
A or B.
This or That.
A and not 2.
A or not 2.
not A or not B implies not(A and B).
false implies AnyProposition.
2.
```

The `tautology` program is invoked with a command of the form

```
%tautology
```

if the input is from `stdin`, or

`%tautology <` *data file*

The output lists each formula and reports whether it is valid or not valid.

[7] The operators are listed in descending order of precedence, using the common concept of precedence in which operators with higher precedence form sub-terms of operators with lower precedence. (Cf. [Gries, 1981, p. 12].). It should be noted that in some versions of Prolog, precedence is defined in reversed order, so that '+' is said to have a *higher* precedence than '*'. This is a more natural definition for operators viewed as functors of terms, since if the term is drawn as an inverted tree, with its principal functor at the top, the principal functor is, in this convention, said to have higher precedence than its sub-terms. See, for example, the Quintus Prolog Reference Manual.

Running `tautology` on the previous sample data file produces the output:

```
% loading file /cs/home/fac2/peter/3111/tautology
% tautology loaded in module user, 0.030 sec 33,728 bytes
Version 2.4, May 14, 1998

false implies A
* Valid.

A or B
* Not valid.
* Counter-example: false or false

A or B
* Not valid.
* Counter-example: false or false

A and not 2
* Not valid.
* Counter-example: false and not 2

A or not 2
* Valid.

not A or not B implies not (A and B)
* Valid.

false implies A
* Valid.

2

!! Non-logical constant in formula.
```

(As a by-product of the way Prolog treats variable names, the variables in the proposition are renamed in the output.)

If a formula is not valid, then there is some assignment of truth values to its variables which makes it false. In this case, `tautology` displays such an assignment as a counter-example to the claim that the formula is valid.

Exercise 2.1: Compare the fourth input (“`A and not 2`”) with the last input. Give a possible explanation for why the occurrence of the non-logical constant is detected in the latter case but not in the former. If you are familiar with Prolog, you can work out the answer from the source code shown in the next section. Otherwise, you will have to make a conjecture.

2.2 Checking the `tautology` program

The `tautology` program uses a set of rules (written in Prolog) which specify when a proposition is false. All possible values of true or false for the variables are tried until an assignment is found which makes the proposition false; if none can be found, the formula is judged to be a tautology. If one is found, its values are substituted into the formula being tested to produce a counter-example.

Here are the main rules[8] used by the program .

```
false('false').
false(not 'true').
false(P iff Q) :- false((P implies Q) and (Q implies P)).
false(P implies Q) :- false(not P or Q).
false(P or Q) :- false(P), false(Q).
false(P xor Q) :- false(not(P iff Q)).
false(P and Q) :- false(P) ; false(Q).
false(not not P) :- false(P).
false(not(P iff Q)) :- false( not(P implies Q)
            or not(Q implies P)).
false(not(P implies Q)) :- false(not( not P or Q)).
false(not(P or Q)) :- false(not P and not Q).
false(not(P xor Q)):- false(P iff Q).
false(not(P and Q)) :- false(not P or not Q).
```

These rules can themselves be read as propositional formulae, with

‘`:-`’ read as ‘if’,
‘`,`’ read as ‘and’ (in the rule for `P or Q`),
‘`;`’ is read as ‘or’ (in the rule for `P and Q`),
and `false` is read as ‘not’.

If the implementation of the `tautology` program is correct, then these rules, interpreted as propositions, must be tautologies; if a rule was not a tautology, then there would some case in which it was false, and it would be a logical error, in that case, to conclude that because the right side was true that the left-side must hold.

So the rules the program uses can themselves be checked by the program! This is illustrated by the last formula in the sample data file given above:

```
not(P) or not(Q)implies not(P and Q).
```

which represents the rule

```
false(P and Q) :- false(P); false(Q).
```

However, using the `tautology` program to check itself will not prove that it is correct; if it were incorrect, it might well incorrectly say its rules were correct. But such self-testing can increase our confidence in the rules, in the sense that if the program were to judge

---

[8] The rules shown (except the one for `xor`) are those used in [Coelho, p. 68].

any of them as *invalid,* then we would know that something was wrong — either in the rule itself or the way it was applied.

The rules have also a less obvious property as propositions: although they are written as implications using Prolog's if operator `:-`, they correspond, in fact, to *equivalences.* This is important, for if they were not equivalences, then we could not conclude that just because the conditions of the rule (the right-hand side) was not satisfied, the left-side was not true. As it is, the rules used by `tautology` are all equivalences, so if a rule fails because the right-side cannot be shown to be true, and no other rule applies to the input, then the program can conclude that the input is never false, and hence is valid.

Exercise 2.2: Use `tautology` to check that each of the rules it uses corresponds to a valid propositional formula.

Exercise 2.3: Use `tautology` to check that replacing the rule

```
false(P implies Q) :- false(not P or Q)
```

with

```
false(P implies Q) :- false(not P), false(Q)
```

is logically justified.

Exercise 2.4: Use `tautology` to show that the rule

```
false(not(P xor Q)):- false(P implies Q).
```

is valid as a propositional formula, but is not usable as a replacement for the existing rule for `not(P xor Q)`.

Exercise 2.5: Could

```
false(P and Q) :-  false(P); false(Q).
```

be replaced by the two rules:

```
false(P and Q) :-  false(P).
false(P and Q) :-  false(Q).
```

(each of which is a tautology) ? Discuss.

2.3 Applications of tautology checking

Many texts on software verification give exercises in which students are asked to compute (by hand) whether or not various propositional functions are tautologies. With a tool such as the `tautology` program, the exercise becomes simply a matter of preparing the input correctly. This may require familiarizing oneself with the author's choice of logical notation.

Examples of different notations are:

(a) (P (Q P)) P
from [Chang, 1973, p. 24].

(b) (E1 = E2) = (E1 E2) (E2 E1)
from [Gries, 1981, p. 20].

(c) (a ¬a) ¬a
from [Hehner, 1988, p. 37].

(Note that `tautology` requires variable names to begin with *upper-case letters*.)

(d) [(p and q r) and (p and not q s)] [p (q r) and (not q s)]
from [Backhouse, 1986, p. 40].

(Note that '[ ]' are not allowable characters in propositional formulae to be input to `tautology`. You also need to know that Backhouse uses 'A B' in a somewhat unusual way to mean that A B is a tautology.)

(e) Here is a 'frightful sentence' from [Manna and Waldinger, 1985, p. 63] whose validity can checked quite quickly by `tautology` once we have figured out how to input it in a form the program will accept:

```
if ((if P1 then (P2 or P3) else (P3 or P4))
        and
    (if P3 then (not P6) else (if P4 then P1))
        and
    not (P2 and P5) and (if P2 then P5)))
then not (if P3 then P6).
```

(f) Some logic exercises require that the variables in a proposition be assigned specific Boolean values. For example, [Backhouse, 1986, p. 29] asks the student to evaluate the truth-value of the proposition:

X and not (not Y or Z) X and Z

under the assumption that "X and Y have the value T and Z has the value F". We can use `tautology` to do this evaluation by rewriting as `implies`, and replacing the variables with the appropriate values `true` and `false`. If the result is "valid", the truth-value of the formula is true, otherwise false.

(g) F or G is equivalent to G if F is unsatisfiable [Lewis, p. 403].

In order to test this, you will have to construct a proposition from it. To do that, you have to decide whether the "if" should be part of the proposition, what the scope of the "equivalent to" should be, and how to represent unsatisfiability in propositional terms.

Exercise 2.6: Use `tautology` to check that whether each of the above examples is a tautology.

Exercise 2.7: Use `tautology` to determine whether each of the following is a tautology, a contradiction, or contingent, (The notation varies in each formula.) .

((a b) c) = (a (b c))

(a (b ¬ a)) (a b)

((p q) r) (p (q r))

Checking satisfiability

As mentioned in **1.1**, satisfiability is tested by showing that the negation of the proposition in question is not a tautology (i. e. , the proposition is not a contradiction.) If `tautology` fails to show that a proposition of the form not P is a contradiction, the list of falsifying values which it computes constitutes a **valuation** which satisfies P.

For example,

```
|: not ((A implies B) implies (B implies not C implies not
A)) .

not (A implies B implies (B implies not C implies not A))
* Not valid.
* Counter-example: not (true implies false implies (false
implies not A implies not true))
```

The satisfying valuation computed for

((A implies B) implies (B implies not C implies not A))

is

A = true and B = false.

Exercise 2.8: Explain why `tautology` doesn't assign a specific value to C in the previous example.

Exercise 2.9: Check that the following proposition is satisfiable, and give a satisfying valuation (with a value for each variable.)

(A C) ((B D) ((A B) C))

2.4 Translating from English to logic

Another kind of exercise in logic involves translating an English sentence into an equivalent propositional form and determining its validity. The trick here is to find the logical connectives in the sentence and the pieces which they connect. Pieces which contain no connective are treated as variables and replaced by capital letters.

For example, a problem due to R. Smullyan and given in [Manna 1980, p. 64] asks:

> Given that the following are true: "I love Pat, or I love Quincy" and "If I love Pat, then I love Quincy", does it follow that I love Pat? Does it follow that I love Quincy?

Let us take 'it follows that' to mean 'the preceding implies that'. Then if we represent "I love Pat" by P, and "I love Quincy" by Q, the problem can be translated into the following questions:

> Does (P or Q) and (P implies Q) imply P ?
> Does (P or Q) and (P implies Q) imply Q?

That is to say, are the propositions

> (P or Q) and (P implies Q) implies P

and

> (P or Q) and (P implies Q) implies Q

always true? The question is now in a form which `tautology` can answer.

Exercise 2.10: "Either the program terminates or the value of *n* is never zero. If the value of *n* is eventually zero then the value of *m* will also be zero. The program does terminate. Therefore the value of *m* will eventually be zero."

Construct a propositional formula which is logically equivalent to the above sentences, treating "either . . or" as an exclusive-or (xor). Use `tautology` to check whether the formula is valid. Show that the implication that *m* will eventually be zero is invalid if "either . . or" is the usual inclusive or.

Exercise 2.11: The following problem is based on [Kogge, 1990, p. 390]. Let
A B represent "If the car has gas, then I can go to the store.";
B C D represents "If I can go to the store and I have money, then I can buy food.";
(D (E F)) G represents "If I have food and either the sun is shining or I have an umbrella, then today I can go on a picnic."

If the above formulae are true, and the car has gas, and I have money and an umbrella, can I go on a picnic?

Show how to use `tautology` to answer the question.

Exercise 2.12: A favorite toy problem in artificial intelligence is “Hunt the Wumpus”[9]. The wumpus hunter is an agent travelling over a grid in which each square may contain a wumpus. If the agent enters a square containing a wumpus, it’s “game over” for the agent. Luckily, if the agent can combine basic propositional reasoning with some sensory information, it can avoid entering a wumpus-occupied square. For if a square contains a stench, then it or one of its four adjacent squares (north, west, east or south) contains a wumpus. Suppose the agent is at square [1,2] and knows from its travels that there is no stench at [1,1], or [2.,1], but there is one at [1,2]:

1,3			
1,2	2,2		
1,1 ok	2,1 ok	3,1	

Let Wij represent the proposition that a wumpus is in square [i, j], and Sij represents a stench at [i, j]. Then

not S21 implies (not W11 and not W21 and not W22 and not W31)

and similar propositions hold for the other squares.

Furthermore,

S12 implies (W13 or W12 or W22 or W11)

Use these and corresponding implications for the other squares, together with the facts not S11, not S21, and S12, to show by means of the `tautology` program that there must be a wumpus in square [1,3].

[9] For more details, see Russell and Norvig, Artificial Intelligence, An Modern Approach, Prentice-Hall: 1995. pp. 153-157, 174-176.

3. CHECKING DESCRIPTIVE STATEMENTS

The `tautology` program has some value for learning the basic concepts of propositional logic; that is why we began with it—its application is easy to understand, and we can look inside and see how it works, without much difficulty. But it has almost no utility for verification problems because it cannot be used to check descriptive propositions which describe program states. (As a result, it is not directly applicable to the problem of checking the condition in conditional statements described in Chapter 1.) The reason is that `tautology` requires that its input consist of propositional functions constructed from Boolean variables represented by upper-case letters. Relational expressions, such as `x > 0`, which are needed to represent a Boolean condition in a conditional statement, or more generally, to describe the state of specific variables at a particular point in a program, cannot be used.

So from now on, we will use a more powerful tool which allows us to check the status of propositions constructed from relational expressions. This program is christened `wang`, after the inventor of the program's algorithm, Hao Wang [Wang, 1960]. For `wang`, the input file contains propositions constructed from **terms** rather than from Boolean variables.

3.1 Terms

A term is a data object structured as a tree, with a symbol assigned to each node. Terms can be represented in many different ways, but typically they are represented as functional expressions $f(t_1, t_2, \ldots, t_k)$, where each t_i is itself a term.

Each node in a term has a fixed number of sub-trees; the number is called the *arity* of the node. If the root node of a term has arity 1, the term can be represented in *prefix* or *postfix* notation[10]; for example:

- t

or

t ! .

If the root node has arity 2, the term can be represented by an *infix* expression; for example:

t_1 and t_2.

The concept of a term is very general. Terms can contain variables, but need not. Often they can be interpreted as expressions which represent a numerical or logical value, but this is not required, and in some cases, terms are just treated as uninterpreted structures used to hold separate items of abstract information.

---

[10] In most programming languages, the representation of a term by a prefix or postfix operator is a matter of syntactic convenience with no semantic implications. A term such as n! is just 'syntactic sugar' for the functional term '!'(n) or perhaps factorial(n). C/Java syntax is unusual and complicated by the presence of two operators (++ and --) which are used in both prefix and postfix modes and mean something different in each case. In this case, `++n` represents one functional term, say, preinc(n), while `n++` represents another, postinc(n).

If the root node of a term is one of the following:

> true, false, not, and, or, xor, implies, iff

then we shall call the term a **logical term**, otherwise, a **non-logical term**.

For example,

> i < n and n < m

is a logical term (the root node is and), and

> 0, i, z(true, false), x=0, p = (q or r)

are all examples of non-logical terms.

In order to describe the state of programs, we will need to use non-logical terms constructed from **relational operators** such as <, , =, , etc. In the verification tools, we will use the ASCII equivalents `<=` and `>=` for and , `=` for =, and `<>` for .

3.2 Invoking the `wang` program

The program `wang` is typically invoked with a command of the form

`%wang <` *data file*

Here is a sample data file:

```
x>0.

(s and q implies r) implies (q implies p or r).

x=2 implies x=2.

0 = 0.
```

and here is the output which results:

```
% loading file /cs/fac/bin/wang
% wang loaded in module user, 0.020 sec 45,860 bytes
Version 1.4, Dec. 12, 1999.

x>0

Counterexample: x>0 is false.
* Not valid.


s and q implies r implies (q implies p or r)

Counterexample: q is true and each of [r,s,p] is false.
* Not valid.


x=2 implies x=2

* Valid.


0=0

Counterexample: 0=0 is false.
* Not valid.
```

The `wang` program does not interpret or evaluate non-logical terms, so it does not assign a truth-value even to such obvious truths as 0=0. In effect, *non-logical terms are treated as complex names for Boolean variables*, and so the program finds the proposition 0=0 to be not valid when treated as a propositional function, just as the proposition x>0 is not a valid formula. We shall see in Chapter 4 how we can add to `wang` the additional mathematical knowledge needed to establish that 0=0 is a tautology.

Like `tautology`, `wang` presents a counter-example if the input proposition is not a tautology by giving an assignment of truth values to the non-logical terms in the input which will make it false.

3.3 Terms and variables in wang

The wang program, like the tautology program, is written in Prolog and uses Prolog's built-in parser to parse non-logical terms in the input. To be properly interpreted by wang, these terms must satisfy the following conditions:

• The only operators allowed are the relational operators:

```
=, <>, <, >, <=, =>,
```

and the arithmetic operators

```
+, -, *, /, **.
```

All other mathematical functions such as integer division (div), remainder (mod), absolute value (abs), etc. are represented by functional expressions. [11]

• expressions associated with complex data structures such as arrays, e. g. x[y+2], are not allowed;

• all variables and function names must begin with a **lower-case letter.**

(Contrast this latter condition with tautology in which Prolog's variable-binding mechanism was used to assign truth variables to the variables in propositional functions, and hence required that the propositional variables be represented by Prolog variables.)

3.4 Solving English-to-logic puzzles

A non-logical term can take the form of a character string constant enclosed in single quotes. This gives us a better way of solving problems such as Exercise 2.2 which involve translating English statements into propositions. Instead of making up variable names to represent the English phrases, we can use the English phrases themselves as non-logical terms, and use wang to test if the resulting formula is a tautology.

As an example, here is the Tardy Bus Problem [Gries, 1993, p. 39] in its English formulation:

"If Bill takes the bus, then Bill misses his appointment, if the bus is late. Bill should not go home, if he misses his appointment and feels downcast. If Bill does not get the job then he feels downcast and should not go home."

Is the conjecture

"Bill gets the job if he misses his appointment and should go home"

[11] The reader may wonder why we don't use the C/Java symbols for equality (==), inequality (!=) or remainder (%). The decision to use different symbols stems from the need to distinguish between the **mathematical** functions which are used descriptively in propositions about program states, and the **programmed** functions used to compute those states. We will have more to say on this in Chapter 7.

true?

To solve this using `wang`, we translate the premises and the conjecture into the proposition

```
('Bill takes the bus' implies ('the bus is late' implies
    'Bill misses his appointment') )
and
('Bill misses his appointment' and 'Bill feels downcast' implies
    not 'Bill should go home')
and
(not 'Bill gets the job'  implies 'Bill feels downcast' and not 'Bill should go home' )

implies

( ‘Bill misses his appointment' and  ‘Bill should go home'  implies
    'Bill gets the job').
```

and if we type it in correctly, `wang` will produce the correct answer.

Exercise 3.1: Find a logical translation for “unless” and use it to translate the statement

“x equals 0 if it is not less than 0, unless it is greater than 0.“

into a proposition.

Exercise 3.2: Rework the ‘picnic’ problem in exercise 2.11, replacing variables with English phrases, and using `wang` rather than `tautology`.

Exercise 3.3: Given the premises of the “Tardy Bus Problem”, use `wang` to check the conjecture that Bill feels downcast if he misses his appointment or the bus is late.

Exercise 3.4: Portia's Caskets Smullyan[1978, p. 60][12] poses a variant of Portia’s problem from Shakespeare’s *Merchant of Venice* : suppose Portia has placed her portrait into one of two caskets, one of which is silver and the other is gold. On the gold casket is inscribed “The portrait is not here”. On the silver casket is inscribed “Exactly one of these inscriptions is true.” Assuming that each inscription is either true or false, use `wang` to determine in which casket the portrait is hidden[13].

Hint: Interpret the gold inscription as not ‘portrait is in gold casket’, and the silver inscription as ‘gold inscription’ xor ‘silver inscription’; construct two implications with the same antecedent expressing the data given in the problem, and with conclusions ‘portrait is in gold casket’ and ‘portrait is in silver casket’. Test each implication with `wang`.

---

[12] See also Backhouse [1986, p. 49].

[13] If we don’t assume this, Portia can place the portrait wherever she wants, since the inscriptions are then just marks from which nothing can be inferred. See [Smullyan, p. 70.]

3.5 Sequents and Wang's algorithm

The `wang` program uses a very different algorithm from the `tautology` program. Rather than searching for a falsifying assignment to the variables of the propositional formula being tested, Wang's algorithm uses a set of rules to rewrite the input formula into simpler and simpler (but equivalent) forms until its truth can be checked 'by inspection'— that is, by a syntactic test on the simplified formula. Wang's algorithm makes no use of truth values at all; it therefore ignores any interpretation which the propositional formula may have. This is an important feature of propositional logic. As long as we are only concerned with the sorts of propositions traditionally of interest to logicians, namely tautologies and contradictions, questions of semantics are irrelevant. All that matters is the formal structure of the proposition—the way it is put together out of variables and logical operations.

The rewrite rules in Wang's algorithm operate on a special form of a proposition known as a *sequent* [14]. Consider propositions with the structure

p_1 and p_2 . . . implies q_1 or q_2. . .

that is, implications in which the antecedent (left-hand side) is a conjunction of propositions and the consequent (right-hand side) is a disjunction. A sequent is a way of rewriting this kind of proposition as stating a relation (which we will denote by **>>**) between the antecedent and consequent, each considered as *sets* of formulae. The and connectives on the left-hand side and the or connectives on the right-hand side are replaced by commas to separate the elements of the sets.

As an example, the formula

p and q implies q or r

has the following form when written as a sequent:

{p, q } >> {q, r}.

A sequent is obviously a tautology if the left-hand and right-hand sides intersect—i. e., if they have a formula in common. (Why?)

Wang's algorithm starts by constructing the sequent

{} >> {*input formula* }

and then repeatedly applies a set of rules to break down the input formula into its constituents, producing one or more new sequents, which are again transformed by the rules, until none of the rules apply. If at any point in the transformation of one of these sequents, it is found that the left side and right side have a term in common, then that sequent is proved to be a tautology.

[14] Sometimes also called a *logical consequence.* See [Mueller, p. 419]. Propositions written in this form are also called *clauses.* Theorem-proving programs typically require the theorem to be proved to be stated in or converted to clause form. See [Richards, p. 93.]

When no further rules apply, the resulting sequents contain only non-logical terms. These sequents can then be easily tested to see if the left- and right-hand sides intersect. If not, the sequent in question is invalid, for if, in a sequent A >> B, all terms are non-logical and A and B have no term in common, then we can assign all the terms in A the value true, so their conjunction is true, and all the terms in B can be assigned false, so their disjunction is false. The sequent is then equivalent to the proposition true implies false, which is false, i. e., A >> B is not a tautology.

The transformation rules used by Wang's algorithm can be stated informally as follows:

Negation

If a sequent contains not P on one side, delete it and add P to the other side:

{not P, . . } >> { . . } is rewritten as { . . } >> {P, . . }
{. . } >> {not P, . . } is rewritten as {P, . . } >> {. . }

Conjunction

Replace a conjunction on the left-hand side of a sequent by its conjuncts:

{P and Q, . . } >> { . . } is rewritten as {P, Q, . . } >> { . . }

Replace a sequent with a conjunction on the right-hand side by two copies in which the conjunction is replaced by each of its conjuncts:

{ . . } >> {P and Q, . . } is rewritten as
{ . . } >> {P, . . }
and
{ . . } >> {Q, . . }.

The propositional form of this rule is:

```
(R implies (P and Q) or S) iff
        ((R implies P or S)
        and
        (R implies Q or S))
```

which can easily be shown to be a tautology.

Implication

To eliminate an implication P implies Q in a sequent, replace it by not P or Q.

Equivalence

To eliminate an equivalence (iff) in a sequent, the equivalence is replaced by its definition in terms of implications.

For example,

{P iff Q, . . } >> { . . } is rewritten as

{(P implies Q) and (Q implies P), . . } >> { . . }

Exclusive-or

To eliminate an exclusive-or P xor Q in a sequent, replace it by not (P iff Q).

A more detailed discussion of Wang's algorithm expressed in traditional logic notation, together with Lisp and Prolog implementations, is to be found in [Mueller, 1988, 419-449]. A brief discussion and Prolog implementation is also found in [Sperschneider, 1991, p. 209-210].

Exercise 3.5: Show that the first of the negation rules is logically justified in the sense that

((not P and Q)) implies R iff (Q implies P or R)

is a tautology.

Exercise 3.6 Express the second negation rule as a propositional formula and show it is a tautology.

Exercise 3.7:

(a) Develop rules to eliminate disjunctions from a sequent, paralleling those for conjunctions;

(b) construct propositional formulae to correspond to these rules and show they are tautologies.

3.6 Performance issues

The programs `wang` and `tautology` use very different methods to decide whether a formula is a tautology, and this is reflected in very different running times for the same formulae. `tautology` can only establish that a formula is a tautology by checking that *no* assignment of truth values to the variables yields false, so if a valid formula has n variables, there are 2^n cases which have to be checked. For `wang`, on the other hand, the running time depends not on the number of variables, but on the number and complexity of the terms in the formula. For example, if a formula has only non-logical terms and is already in sequent form:

P_1 and P_2 and . . . P_k implies Q_1 or Q_2 or . . . Q_m

then the time to determine whether it is a tautology is, in the worst case, the time required to compare each P_i with all the Q_j , which is proportional to k*m. The comparison is complicated however by the fact that for both algorithms the running time may be much less than the worst case — `tautology`'s algorithm can, for example,

determine that a conjunction is invalid as soon as it finds a false conjunct ; the remaining conjuncts do not need to be determined. On the other hand, Wang's algorithm can determine that a formula is valid as soon as it detects an overlap between left and right sides of the sequent, even before the sequent has been completely reduced to a list of non-logical terms.

The `tautology` program is short and easy to understand; the `wang` program is longer and much more complicated since its transformation rules involve various set manipulations. (Actually, to save the time involved in maintaining sets of terms with no duplicates, `wang` just maintains sequents as a pair of lists which may include duplication of terms.) In general, the greater complexity of Wang's algorithm pays off in performance as shown by the following examples, using `/usr/bin/time` to get an approximate running time on a Sun workstation: (Only the "real " or program execution time is shown.)

```
% /usr/bin/time tautology < taut.test1
Version 2.4, May 14, 1998

A iff B iff A iff B iff A iff B iff A iff B iff A iff B iff
A iff B iff A iff B iff A iff B iff A iff B iff A iff B iff
A iff B iff A iff B
* Valid.

real     6:19.0

----------------------------------------

% /usr/bin/time wang < wang.test1
Version 1.4, Dec. 12, 1999.

a iff b iff a iff b iff a iff b iff a iff b iff a iff b iff
a iff b iff a iff b
iff a iff b iff a iff b iff a iff b iff a iff b iff a iff b

* Valid.


real     1:11.9

----------------------------------------
```

```
% /usr/bin/time tautology < taut.test2
Version 2.4, May 14, 1998

A iff B iff A iff B iff A iff B iff A iff B iff A iff B iff
A iff B iff A iff B
iff A iff B iff A iff B iff A iff B iff A iff B iff A iff C
* Not valid.
* Counter-example: false iff true iff false iff true iff
false iff true iff false iff true iff false iff true iff
false iff true iff false iff true iff false iff true iff
false iff true iff false iff true iff false iff true iff
false iff false

real     1:11.4

---------------------------------------

% /usr/bin/time wang < wang.test2
Version 1.4, Dec. 12, 1999.

a iff b iff a iff b iff a iff b iff a iff b iff a iff b iff
a iff b iff a iff b
iff a iff b iff a iff b iff a iff b iff a iff b iff a iff c

Counterexample: b is true and each of [a,c] is false.
* Not valid.

real       22.8
```

The particular type of proposition used in the above examples is stressful for both algorithms. In the case of `tautology`, the search for a falsifying assignment of truth values is not very intelligent and in these examples leads to a very large amount of backtracking and retrying; in `wang`, each iff operation doubles the number of sequents that have to be checked, causing an exponential growth in the running time.

But interestingly enough, there is a special tautology test for just these sorts of propositions whose running time is *linear* in the length of the proposition and hence very efficient. The test is just to check the parity of the number of occurrences of every variable in the formula, since a formula which consists of only bi-conditionals is a tautology if and only if each propositional variable occurs an even number of times[15]. The moral is that a problem which may be very difficult for a general algorithm may be almost trivial for a special-purpose algorithm designed for just that case.

Exercise 3.8: Knights and Knaves [Smullyan, 1978, p. 22]. Suppose knights always tell the truth and knaves always lie. We encounter A and B, each of whom is either a knight or a knave. A says “I am a knave but B isn’t.” Compare the performance of `wang` and `tautology` in determining whether A is a knight or a knave, and the same for B.

[15] [Enderton, 1972, p. 39]

Exercise 3.9: Construct an equivalence which expresses the associativity of iff and show that it is a tautology. Demonstrate the associativity of iff by showing that the different parenthesizations of a iff b iff b iff a are all valid.

3.7 The Deduction Theorem

The meaning of the term 'theorem' is not *a proposition which is true* but *a proposition which has a valid proof* — that is, a sequence of statements which begins with the hypothesis or premise of the theorem and ends with its conclusion, in which each statement is derived from the previous statements in the sequence by means of a valid **inference rule**, such as *modus ponens* :

> from the premises
> p
> and
> p implies q,
> infer q.

To be a valid inference rule, the rule must have the property that it never allows a false statement to be derived from a true one. This property is mirrored in propositional logic by the relation of implication: a true statement can never imply a false one.

All the theorems of propositional logic are expressible as tautologies and vice-versa. So if "P implies Q" is a tautology then "P implies Q" is a theorem and Q can be inferred from P. This result is called the Deduction Theorem (see [Gries, 1981, p. 36].) It is actually an example of a **metatheorem**, since it is a theorem about the properties of theorems.

We can use the Deduction Theorem to side-step the problem of finding a proof in propositional logic for a purported theorem. Instead of looking for a proof, we convert the theorem into an implication of the form p implies q and check if the implication is a tautology. If it is, then the Deduction Theorem tells us a proof exists. If the implication is not a tautology, then for some values of its variables, it must be possible for p to be true while q is false. So there could be no valid inference of q from p and p implies q cannot be a theorem.

Exercise 3.10: Reformulate each of the following inferences as an implication and use wang to show that , on the basis of the Deduction Theorem, each of the following inferences is correct:

(a) from the two premises p implies q, and q implies r, one can correctly infer the proposition p or q implies r.

(b) from not (p and q) and q, one can infer not p.

(c) [Dromey, 1989, p. 54] e implies d follows from the hypotheses:

a implies (b implies c and d)
not e or a

b
e.

Exercise 3.11: Use `wang` to show that each of the following inference patterns is incorrect:

(a) from p implies q and q, one can infer p.

(b) from p or q and q, one can infer not p.

(c) if (p and q) and not q are not both true, then one can infer p.