www.irss.academyirmbr.com July 2018
International Review of Social Sciences Vol. 6 Issue.7

Condition Simplification by Logic

SADIA TARIQ
MS Scholar

SAJJAD ABBAS
PhD Scholar
Barani Institute of Sciences, Sahiwal Campus
Email: sajjad.ranaswl@yahoo.com

HABIB ULLAH
MS Scholar

Abstract

Formal specification of software is the major concern for every software client nowadays. No organizations try to get the reliable and accurate software solutions. The one of the best way to get maximum accuracy in solutions is achieved by the techniques used for formal specification of software engineering. Propositional Logic (PL) is the one the simplest and oldest techniques with its grounds in mathematics to formally specify a program with the using of Hoare's Triple. Propositional logic is the branch of mathematics which is being used by many of science disciplines for formal specification of real time problems. Every problem which exists can be stated in any English like language can also be transferred using propositional logic and its validity and invalidity can be checked, verified and proved using laws and logical equivalence of Propositional Logic. PL provides us verity of rules that can be applied into computer programming to formally specify a computer program and provide several mechanisms for checking their valid and proof. This paper will present a unique and different use of propositional logic, its laws, and especially the logical equivalence in the processing of Software development life cycle. We have used Pl in coding part of software for condition simplification. Conditions are the primitive and basic parts of software code and it enable to get a simplified version of complex conditions prior to code it; it will surely give us simplification of code. With the help of code examples it will be proved that how propositional logic can be used for the condition simplification, and hence reduction of code size and logic circuitry of a computer program.

Keywords: *Logic, Laws of Logic, Logical Equliances, AND, OR, NOT, Implifciation and Bi-conditioanl implification.*

Introduction

Formal methods are an important part of the students' education as it helps to train them to treat the specification of software in software development life cycle (Colin J Burgess). In graduate courses, we have had students read the literature on formal methods usage, including practical applications (A. K. Sobel, 2002), educational benefits (J. Hatcliff , 1990) , and the "myth buster" papers that debunk popular misconceptions about formal methods (J. P. Bowen, 1995). Formal methods are mathematical and procedural techniques which are used to specify complex systems formally in terms of mathematical entities. Formal specification makes it possible and reliable to verify the accuracy of even more rigorous and complex systems using these formal models. This proves that accuracy testing is conducted in more empirical fashion (kneethe H. Rosen, fifth edition). With the arrival of Java language, an progression

expansion started in formal specification that was the beginning of several Java-specific notations. These include iContract, J@va, JASS, jContractor, and JML (D. Bartetzko , 2001).

Formal Specification is the first phase of formal modeling of a complex system. Use of formal methods and formal specification is now heavily adopted by software engineers to make a 100% accurate applications. formal methods can be used to describe any approach which utilizes a formal specification language and specifies the role of that formal specification during the software development life cycle (Colin J Burgess). Some software applications demand hundred percent accuracy for example, air ticketing system, missile embedded systems and any real time applications. There are many examples of software bugs which yield a loss of budgets and even human lives. In the process of writing formal specification of software, an engineer thoroughly expresses this software in terms of formal methods. Propositional logic, Object Constraint Language, Algebraic expression, Z specification and many others formal methods are examples of formal methods. The formal modeling languages or we can say that specifications use fixed grammars to provide an additional benefit of modeling complex software systems against some predefined types provided in programming languages. The expressing of formal specification is a system that is same one as we translate English problems statement into Propositional Logic constructors or into Algebraic notation.

Propositional logic is the building block for computer. In the early days of computing, computer was known as logic machine. In fact, Propositional logic participate an important role in computer design, programming and use of computer (Van Chan Ngo, 2012). Rules of logic can specify the mathematical statements in a definite manner. For example “An integer exists which is not the sum of two other integers” (M.G. Meulen, 2010). Another example is “the sum of any positive n positive integer is always n (n+1)/2”. Propositional logic is a basis for mathematical and computing nature of sciences. Since computer is based on Boolean algebra, a mathematical branch, hence PL has important practical applications in the system development life cycle, system specification, computer programming and designing of programming language (M.G. Meulen, 2010). Formal specification using logic and other mathematical techniques can be applied to compiler construction as well (M. Collins, 1998). Propositional logic is one of the oldest mathematical formal methods used to specify the Programs.

Statements within a first order logic can easily be transferred into computer programs (Walton, 1954). In this paper we would start with the introduction to Propositional Logic, its rules of inference, proof using laws of logic and laws of logical equivalence to their application in the coding phase of a program life cycle. A branch of logic is predicate logic which is sometimes refers to programming language (Walton, 1954).

In the analysis section we have discussed some code of programs which have nested conditions and conjunct conditions, and then with the help of PL, we have tried to simplify them. This will also improves the quality of software as quality should be proven correct with variety of techniques (M. H. Van EMDIN, 1976).

Related Work

The problems involved in the systematic development of professional software are numerous and hard to tackle. Any formal specification language can be used to describe the desired behavior of the software in abstract manner (J. Hatcliff, 2012). A lot of work has been done in development of formal methods and we have a great number of lists of these methods. But the motivation to use these models is increasing now day by day. Many big incidents in the history of world that caused human to die have been observed only due to software bugs. The reason behind that was the lack of use of formal methods and formal specifications.

Nico Plat in his thesis work done describe a convenient model for transformation of problem into software. This model provides a framework, that is used to identify defects in the process and solutions for these

defects. This model was based certain assumption form the motivation taken from use of formal specifications.

M. Collins in his work about Formal Methods in back 1998 mentioned that "Simple propositions expressed by Propositional Logic automatically provide help for the verification of more complex theorems" (Kneethe H. Rosen, fifth edition). Since propositions used in Algebraic expressions are literals and could be represented by single identifiers.

Gene Fisher and Corrigan Johnson worked in the same domain and proposed a tool "Spest". This tool is used to generate unit testing code from a formal program specification (Gene Fisher). They proved that it can be used handle some of the difficulties while testing and it is easy to produce readable test code that can be extendible too. Other directly related tools for test generation tools are JMLUnit (Y. Cheon, 2002) and JMLUnitNG (D. M. Zimmerman, 2010) that have been developed for JML. The Spest system with its easy and stand-alone graphical user interface make it easy for its users to select code files and generate the tests. Then these tests code uses the TestNG library (C. Beust, 2007).

The very first authentic work in the direction of program construction for calculating implementation from specification is being done by Roland Backhouse, the author of world famous book "Program Construction". http://www.cs.nott.ac.uk/~rcb/papers/papers.html Here is the list of world famous publications regarding formal specifications (Roland Backhouse). Roland used logic for his all kinds of research related to program construction. In this famous book "Program construction" he explained logic with reference to computer programs, Hoare's triple and formal specifications.

Proposition: A statement that can prove to be true or false is called a proposition. For example, "x+y=3" can be proved true or false if we know the values of x and y variables in this expression. Proposition are denoted by letters when they are manipulated and the truth value of a proposition is denoted by T if it is true and F if it is False.

Propositional logic: The branch of logic which deals with these propositions is called Propositional calculus or Propositional Logic.

Compound Proposition: when two or more propositions are joined using conjunction or disjunction operators then these are called compound proposition. There are three basic operation for a proposition, AND, OR and NOT operator. For example "Today is Sunday and it rains usually in the winter". Let *P* denotes the statement "Today is Sunday", *Q* denotes the statement "it rains usually in the winter" then the compound proposition can be represented as;

P ^ Q (^ is used to denote the AND logic operator)
Here are truth tables for AND, OR, and NOT operators.

TABLE 1 NOT LOGICAL OPERATOR

P	**~P**
T	F
F	T

TABLE 2 AND LOGICAL OPERATOR

P	**Q**	**P ^Q**
T	T	T
T	F	F
F	T	F
F	F	F

TABLE 3 OR LOGICAL OPERATOR

P	Q	P v Q
T	T	T
T	F	T
F	T	T
F	F	F

Conditional Statement: These statement are called sometime the implication, Let P and Q are two implications the Implication defines the logic relationship between them as mentioned in the following truth table. This logic relation and law is equal to if statement of a programming language, if P then Q. Its truth table value is false when only Q is false and P is true and otherwise true for all other combination.

TABLE 4 P -> Q (IMPLICATION)

P	Q	P-> Q
T	T	T
T	F	F
F	T	T
F	F	T

Bi-conditional Statement: If P implies Q and Q also implies P then we can say that this is a bi-conditional statement. In P ↔ Q, proposition is true when P and Q have same values otherwise false.

TABLE 5 P -> Q (IMPLICATION)

P	Q	P ↔ Q
T	T	T
T	F	F
F	T	F
F	F	T

Propositional Equivalence: In computer programming sometimes we have many complex if and conditional structures, examples have been displayed in the next section, we need to replace the condition with some simple and equal conditions here we need Propositional equivalence to impart his role. If two simple or compound propositions have same truth values then they are said to be the logically equal to each other and we can replace them with each other. For Example let consider the following two equal compound proposition, we have proved here with the help of truth table, it can be proved using many other ways as well.

P-> Q = ~P v Q

TABLE 6 OR LOGICAL OPERATOR

P	Q	~P v Q
T	T	T
T	F	F
F	T	T
F	F	T

From table 4 and table 6 it is apparent that above propositions are logically equivalent; the truth values are same for both the compound proposition.

This logical Equivalence is extremely a constructive element of formal methods by which we can specify a program and we can simplify a computer program's code significantly. Logical Equivalence is commutative, associative, distributive, and symmetric in nature.

Tautology: A compound proposition formula is valid which is always true no matter what the values of its propositions that formed this is called tautology. For example P v ~P is always true (can be shown in Truth table).

Contradiction: A compound proposition which is always false no matter what the values of its propositions that shaped this is called contradiction. For example P ^ ~P is always false (can be shown in Truth table).

Logical Equivalence (LE): Now it is easy to define Logical Equivalence that if P and Q are two Propositions and their bi-conditional is tautology then these are logically equivalence. If every model of a Preposition P is a also a model of Proposition Q then P ≡ Q holds (M. Collins , 1998) and they can be written as;

If **P ↔ Q=T (Tautology)**

Then P and Q have logical equivalence. Symbolically we write them as P ≡ Q. For example the DeMorgan Laws have logical equivalence. Here is the table that shows the results.

TABLE 7 PROOF OF DEMORGAN LAW (LE)

P	Q	PvQ	~(PvQ)	~P	~Q	~P^~Q
T	T	T	F	F	F	F
T	F	T	F	F	T	F
F	T	T	F	T	F	F
F	F	F	T	F	T	T

Here we can see that Logical equivalence holds for DeMorgan's Laws. This LE has some basic properties in nature which are as follows. The main objective of these rules is to provide the semantics to prove all valid and statements and their entailments.

Commutative Laws

$p \wedge q \equiv q$
$p \vee q \equiv q \vee p$

Associative Laws

$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
$(p \vee q) \vee r \equiv p \vee (q \vee r)$

Distributive Laws

$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$

Identity Laws

$p \wedge t \equiv p$
$p \vee c \equiv p$

Negation Laws

$p \vee \sim p \equiv t$
$p \wedge \sim p \equiv c$

Double Negation Law

$\sim(\sim p) \equiv p$

Idempotent Laws

$p \wedge p \equiv p$
$p \vee p \equiv p$

DeMorgan's Laws

$\sim(p \wedge q) \equiv \sim p \vee \sim q$
$\sim(p \vee q) \equiv \sim p \wedge \sim q$

Universal Bound Laws

$p \vee t \equiv t$
$p \wedge c \equiv c$

Absorption Laws

$p \vee (p \wedge q) \equiv p$
$p \wedge (p \vee q) \equiv p$

Negation of t and c

$\sim t \equiv c$
$\sim c \equiv t$

Analysis

This is our main section where I have tried to explain the use of logic and its laws to simplify the conditions used in a program coding. I used above stated logic laws and logic rules to carry out the condition simplification proves. Figure-1 shows the flow of work done in this simplification. For the first part I haves proved my code simplification proves with the help of two examples which might be used as basic to extend this research for future work to make it more clear and workable.

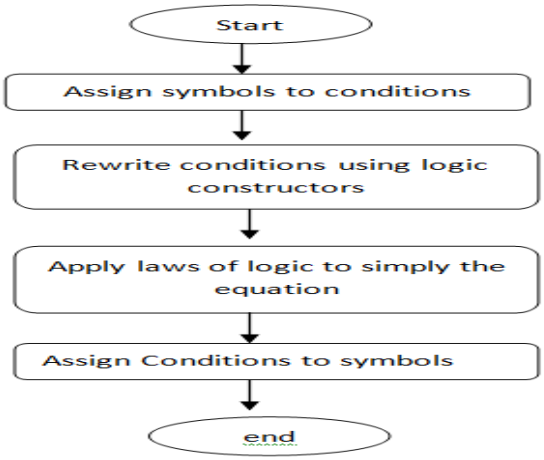

FIGURE-**1**

From Figure-1 it is apparent that it is a simple straight forward process to simplify the complex conditions in coding to simpler one. The key point is to precisely and accurately using transformations from conditions to symbols and converse and the application of laws of logic.

In the preceding both examples it can be observed how these activities are being carried out in sequence and the resultant conditions are so short and cute to handle.

Example 1:

if ((stFine > stRegFee) and not ((stFine > stRegFee) and (srEnrollment <1500)))

```
        statement(s);
else
        statement(s)
```

Assign Symbols to Conditions

Let we translate above stated if conditions into propositional logical statements, as mentioned in introduction section that we use letter to denote proposition then Let

P = stFee > stRegFee
Q= stEnrollment <1500

Rewrite conditions using logic constructors

We can write it as

if ((stFine > stRegFee) and not ((stFine > stRegFee) and (stEnrollment <1500)))

≡(P ^ ~(P^Q))

Apply laws of logic to simplify the equation

Now with the help of above discussed rules of equivalence and laws of logic we can simple this logical statement as follows

≡ (P ^ ~P v ~Q)
≡ (P ^ ~P) v ~Q)
≡C v ~Q
≡~Q
≡ if (Not(stEnrollment < 1500))

Assign Conditions to symbols

Hence simplified version of the above stated conditions in the example will be
if (Not(stEnrollment < 1500))

```
statement(s)
else
statement(s)
```

Now we will observe logical equivalence of simplified condition with its previous version from where we had started. With the help of truth table, the simplest proving technique we can observe that

TABLE 8 PROOF FOR EXAMPLE 1 (LE)

P	Q	~P	~Q	P^~P	(P v ~(P ^Q)	~Q
T	T	F	F	F	F	F
T	F	F	T	F	T	T
F	T	T	F	F	F	F
F	F	T	T	F	T	T

We can see that in the last two highlighted columns we obtain same values, hence the above proof is correct and we can replace the conditions already mentioned in the if() to this simple version. ;

Example 2:

This real life example is taking for a University registration system in which the following is the criteria for allotting project or internship to a student. A student can be allotted project if and only if it fulfills any one of following conditions;

- He has passed 15 credit hours and CGPA is greater than or equal to 1.75.
- He has not passed 15 credit hours but CGPA is greater than or equal to 1.75.
- He has passed 15 credit hours but CGPA is not greater than or equal to 1.75.

```
If((crPassed=15 && cgpa>=15) || (crPassed!=15 && cgpa>=15) || (crPassed=15 && cgpa<15)) then
Alotporject()= “Yes”;
Else
Alotproject()= “No”
```

Let we define proposition as;

P denotes “Pass credit hours equal to 15”
Q denotes “CGPA >= 1.75 “

- He has passed 15 credit hours and CGPA is greater than or equal to 1.75= (P^Q).
- He has not passed 15 credit hours but CGPA is greater than or equal to 1.75= (~P^Q).
- He has passed 15 credit hours but CGPA is not greater than or equal to 1.75=(P^~Q).

By combing these using with OR operator we have (as is mentioned anyone can be fulfilled) hence the compound proposition becomes:

If((crPassed=15 && cgpa>=15) || (crPassed!=15 && cgpa>=15) || (crPassed=15 && cgpa<15))

≡(P ∧ Q) ∨(~P ∧ Q) ∨ (P ∧~ Q)

Now using laws of logic and equivalence we try to simplify it

≡(P ^ Q)□ v (~P ^ Q) v (P ^ ~Q)
≡ (P v ~P)□ ^ Q v (P ^ ~ Q)
≡ t ^ Q v (P ^ ~Q)
≡ Q v (P ^~ Q)
≡ (Q v P) ^ (Q v ~ Q)
≡ (Q v P) ^ t
≡Q v P
≡P v Q
≡If(crPassed=15 || cgpa>=1.75)

Conclusion

As shown in the above examples it is clear that if we have multiple conditions in a source code with conjunction or disjunction operator then by using laws of logic or using truth table we can simplify the code and hence decrease code size and complexity, which will surely decrease the defect rates per Kilo Line Of Code. Use of Formal Specification certain applications looks authentic but is future to spread in large spectrum of applications is still under conditions (Colin J Burgess).

References

An introduction to laws of thought, Walton, London 1954.

A. K. Sobel and M. Clarkson. Formal methods application: An empirical tale of software development. IEEE Transactions on Software Engineering, 28(3):308–320, 2002.

A. Hall. Seven myths of formal methods. IEEE Software, 12:11–19, 1990. [12] J. Hatcliff, G. T. Leavens, K

C. Beust and H. Suleiman. Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley Professional, 2007.

C. Jaspan, M. Keeling, L. Maccherone, G. L. Zenarosa, and M. Shaw. Software mythbusters explore formal methods. IEEE Software, 26(6):60–63, 2009.

D. Bartetzko, C. Fischer, M. M¨oller, and H. Wehrheim. Jass – java with assertions. Electronic Notes in Theoretical Comp. Sci, 55(2):103–117, 2001.

Discrete mathematics by kneethe H. Rosen, fifth edition.

D. M. Zimmerman and R. Nagmoti. Jmlunit: The next generation. In Proc. of the 2010 Intl. Conference on Formal Verification of Object-oriented Software, FoVeOOS'10, pages 183–197, Berlin, Heidelberg, 2011. Springer-Verlag.

Experiments with Formal Methods in Software Engineering….http://nicoplat.com/sites/default/files/nicoplat-phd-thesis.pdf

Formal Methods, M. Collins, Carnegie Mellon University 1998.

Formal Verification of Transformations on Abstract Clocks in Synchronous Compilers, Van Chan Ngo, Sep 2012.

G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: A behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes, 31(3):1–38, May 2006.

http://delivery.acm.org/10.1145/2900000/2899424/p224-fisher.pdf?ip=43.245.8.26&id=2899424&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=653447577&CFTOKEN=77771288&__acm__=1470706360_63c91dc6e8eb6135e67ca417b916b6e1

I. B. Bourdonov, A. V. Demakov, A. A. Jarov, A. Kossatchev, V. V. Kuliamin, A. Petrenko, and S. V. Zelenov. Java specification extension for automated test development. In Revised Papers from the 4th Intl. Andrei Ershov Memorial Conference on Perspectives of System Informatics: Akademgorodok, Novosibirsk, Russia, PSI '02, pages 301–307, London, UK, UK, 2001. Springer-Verlag.

J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Muller, ¨ and M. Parkinson. Behavioral interface specification languages. ACM Comput. Surv., 44(3):16:1–16:58, June 2012.

J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. IEEE Software, 12(4):34–41, 1995.

Logic and formal verification by Jeremy Avigad, June 2009.

M. Karaorman and P. Abercrombie. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. Formal Methods in System Design, 27:275–312, 2005.

Making Formal Methods More Relevant to Software Engineering Students via Automated Test Generation Gene Fisher Department of Computer Science California Polytechnic State University San Luis Obispo, CA 93407 gfisher@calpoly.edu Corrigan Johnson Department of Computer Science California Polytechnic State University San Luis Obispo, CA 93407 johnsoncorrigan@gmail.com

Philosophy 220A: Symbolic Logic-I, Department of Philosophy, University of British Columbia, and September 6, 2004.

R. Kramer. icontract - the java(tm) design by contract(tm) tool. Technology of Object-Oriented Languages, Intl. Conference on, 0:295–308, 1998.

Roland Backhouse http://www.cs.nott.ac.uk/~rcb/papers/papers.html

Semantics of predicate logic as a programming language, M. H. Van EMDIN Scotland. 1976.

T. Margaria and B. Steffen, editors. ISoLA'12: Proc. of the 5th Intl. Conference on Leveraging Applications of Formal Methods, Verification and Validation: Applications and Case Studies - Volume Part II, Berlin, Heidelberg, 2012. Springer-Verlag.

The Role of Formal Methods in Software Engineering Education and Industry Colin J Burgess University of Bristol Department of Computer Science University of Bristol Bristol, BS8 1TR England.

Verification of PLC source code using, propositional logic, M.G. Meulen, April 2010.

Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In Proc. of the 16th European Conference on Object-Oriented Programming, ECOOP '02, pages 231–255, London, UK, UK, 2002. Springer-Verlag.