# Topics in Software Design
# Volume 1

Emil Sekerinski (Ed.)

SQRL Report 34
McMaster University

June 2005

# Contents

# Introduction

This collection of papers is produced by participants of the graduate course CAS 703 Software Design, winter term 2004/05. The course was divided into two parts. In the first, participants and the instructor gave seminars on fundamental topics in software design. For the record, these were:

1. Program Verification, Tim Paterson
2. Program Modularization, Magdin Stoica
3. Abstract Programs, Emil Sekerinski
4. Stepwise Refinement, Upasana Pujari
5. Exception Handling, John Xu
6. Testing, Ning Zhou
7. Program Documentation, Zhuo Zheng
8. Structured Input and Output, Ning Liu
9. Reactive Systems, Nima Dezhkam
10. Object-Oriented Modelling, Shu Wang
11. Problem Solving, Huarong Chen
12. User Interfaces, Wen Yu
13. Requirements Documenation, Reza Sherafat
14. Functional Specifications, Michael Kucera
15. Concurrent Programming, Gabriel Indik
16. Design Patterns and Software Architecture, Ed Sykes
17. Configuration Management, Mark Pavlidis
18. Program Transformation, Olivier Dragon
19. Development Process, Ramez Mousa

For the second part, students selected a more advanced topic for which they reviewed the literature, gave a presentation, and wrote a paper. This report consists of those papers, in order of presentation. The topics are all on "emerging" issues in software design. Some of the articles are surveys and some develop new ideas; they are all beyond the material found in textbooks on software design. The diversity of themes and the dedication of the authors makes this collection an enjoyable read and insightful read! The collection is recommended to anyone who likes to deepen their understanding of, or engage in research in the respective topic.

Emil Sekerinski
June 2005

# Chapter 1

# Tim Paterson: Decision Procedures

Decision procedures are a class of algorithm for determining the validity of sentences in various mathematical languages, and are an integral part of modern theorem provers and systems of program verification. In this paper, we explore some useful and well-known decision procedures, provide a short summary of some techniques for combining and augmenting decision procedures in ways that are suitable for automated reasoning, and examine the complexity and tractability of the various algorithms.

## 1.1 Preliminaries

Decision procedures date back more than 70 years to the work of Presburger, who gave a procedure for determining the validity of sentences over the integers (with addition). They form the core of all modern theorem provers and automatic program verifiers, and are useful for their ability to perform repetitive analysis of simple (and not-so-simple) formulas.

A decision procedure works over a *language*, although we will see that two languages can be combined and a single decision procedure can be derived to work over their union. These languages may be arithmetic in nature, or they may deal with mathematical objects, for instance, stacks or arrays. It is also convenient to speak of the *sorts* that a decision procedure works over, where each *sort* is an element of the language's domain. As an example, when the language is Presburger Arithmetic, there is a single sort: the integers.

Before a decision procedure can be developed, however, the fundamental question of decidability must be addressed.

## 1.2 Decidability

The question of decidability is one of the most basic and profound in all of theoretical computer science. Certain problems, such as the Halting Problem, can be shown to be undecidable. That is, for any algorithm that purports to solve the problem, there exist inputs which will cause it to loop forever. A more thorough treatment of this can be found

in [10]. Proofs of undecidability are outside the scope of this paper, but we will see a proof of decidability[1] when we examine Cooper's algorithm.

For our purposes, therefore, we must limit ourselves to problems which are decidable. Fortunately, many interesting problems are decidable.

## Natural numbers

Also known as Presburger Arithmetic, the theory of Natural numbers can be stated as the set of all sentences over the non-negative integers with addition. Formally, $\mathcal{L}_{\mathbb{N}} = (\Gamma_{\mathbb{N}}, T_{\mathbb{N}})$ where $\Gamma_{\mathbb{N}} = \{(0,1), +, =\}$ is the set of constants, functions and predicates, and where $T_{\mathbb{N}}$ is:

- $\forall x. \neg 0 = x + 1$
- $\forall x, y, \neg(x = y) \Rightarrow \neg((x + 1) = (y + 1))$
- $\forall x, y. (x + y) + 1 = x + (y + 1)$
- $\forall P. (P(0) \wedge \forall x. (P(x) \Rightarrow P(x + 1)) \Rightarrow \forall x. P(x)$

This is a decidable problem (as we will see). This means that the truth of any statement involving addition and equality over the integers can always be determined. One example of such a sentence is

$$\forall x. \neg \exists w. 2 * w = x \Rightarrow \neg \exists z. 2 * z = 3 * x$$

which says that if $x$ is odd, then $3x$ is odd. This is true.

If we include a second function symbol $*$, representing multiplication[2], the resulting theory is Peano Arithmetic. The undecidability of Peano Arithmetic is one of the most famous results in discrete mathematics of the twentieth century[3].

## Reals

Interestingly enough, if we expand Peano Arithmetic further, we obtain another decidable problem. The theory of the Real numbers is obtained by taking Peano Arithmetic, but by then allowing variables to be chosen from the set of Real numbers, rather than the natural numbers. The ability to decide this new problem can be seen by considering statements like

[1] In general, to show that something is decidable it suffices to give an algorithm which decides it. To show undecidability, however, is more difficult, and usually relies on a proof-by-contradication involving a known undecidable problem

[2] Although the previous example contained multiplication, it was only multiplication by a constant, and can be seen as a shorthand way of writing $x + x$ or $x + x + x$. The undecidability of Peano Arithmetic stems from it's ability to multiply variables by other variables

$$\forall x.\exists y.x = y + y$$

which is untrue if $x$ and $y$ are chosen from the natural numbers, but obviously true if they are reals.

Several other theories can be shown to be decidable. Manna and Zarba [6] mention several other theories that are decidable involving more complicated data structures such as lists, sets, and arrays. One simple, yet interesting theory is that of equality.

## Equality

The theory of equality has no axioms, and simply asserts the trivial equality relationship in which everything is equal to itself, and nothing else. Proofs in the theory of equality typically involve repeated substitution of equals-for-equals, although the difficulty arises in knowing which subsitutions to make. Take the following example [9] in which arrays (modelled as functions) are permitted. To prove the tautology:

$$(I = J \wedge K = L \wedge A[I] = B[K] \wedge J = A[J] \wedge M = B[L]) \Rightarrow A[M] = B[K]$$

we can follow this line of substitutions:

| | |
|---|---|
| A[M]= | |
| A[B[L]] | (from M = B[L]) |
| A[B[K]] | (from K = L ) |
| A[A[I]] | (from A[I] = B[K]) |
| A[A[J]] | (from I = J) |
| A[J]* | (from J = A[J]) |
| A[I] | (from I = J) |
| B[K] | (from A[I] = B[K]) |

Note that on the line marked *, if we had done the replacement in the other direction, a potentially infinite chain of A[A[...[A[J]]]..] might have resulted. Clearly, even simple applications of decision procedures have the possibility of being undecidable.

The theory of equality is interesting, because in its basic form it is undecidable [3], but if we restrict ourselves to sentences without quantifiers, then it becomes decidable.

Quantifiers are interesting because decision procedures for both the integers and the reals generally begin with a phase of quantifier elimination. It is illustrative to examine both the abilities and limitations of quantifiers.

## 1.3 Quantifier Elimination and Practical Decision Procedures

There are two basic types of quantification, universal and existential. Universal quantification is denoted $\forall x$, and the sentence $\forall x.A(x)$ would be true if $A$ evaluated to *true* for every possible value of $x$. Likewise for the existential case, $\exists x.A(x)$ is true if some value of $x$ makes $A$ true.

Although it may not seem obvious at first, quantification is a valuable tool for shortening formulas. A given formula can be cut down by a factor which is exponential in the number of quantifiers. For example (if variables range over $\{0, 1\}$):

$$\forall x, \exists y(A(x, y)) \Leftrightarrow (A(0, 1) \vee A(0, 0)) \wedge (A(1, 1) \vee A(1, 0))$$

However, this savings in space comes with a consequent increase in time when a decision procedure encounters a quantifier. For this reason, decision procedures often begin by eliminating quantifiers. One (naive) way to do this is to expand quantifiers as shown above. However, this leads to an exponential increase in the size of the formula, and is only possible if the quantified variables are chosen from a finite domain. It is impossible to use this technique to expand a quantifier over the reals or integers.

### Fourier-Motzkin Variable Elimination

One famous algorithm for quantifier elimination over the real numbers[3] is Fourier-Motzkin variable elimination [8]. For simplicity, $\forall x.A(x)$ will be replaced everywhere by $\neg\exists x.\neg A(x)$, and $\geq$ and $>$ will be replaced with $\leq$ and $<$ where convenient.

The process begins from the middle of a formula and works outwards. This means that, at every step, a formula of the form:

$$\exists x.A(x)$$

where $A(x)$ is quantifier-free, is converted to an equivalent formula with no quantifiers. From this, a simple inductive argument suffices to show that all quantifiers can be eliminated from a formula.

The actual elimination rests on four equivalences:

$$(\exists x.c \leq ax \wedge bx \leq d) \equiv bc \leq ad$$

$$(\exists x.c < ax \wedge bx \leq d) \equiv bc < ad$$

[3]Here, the Theory of Reals has been augmented with $<$ and $\leq$ predicates, but questions of decidability are unchanged

$$(\exists x.c \leq ax \wedge bx < d) \equiv bc < ad$$

$$(\exists x.c < ax \wedge bx < d) \equiv bc < ad$$

The validity of any of these is simple to demonstrate. For the first: $\Rightarrow$ Assume that for some $xc \leq ax$ and $bx \leq d$ then $bc \leq abx$ and $abx \leq ad$, so by transitivity we obtain $bc \leq ad$. $\Leftarrow$ Assume that $bc \leq ad$. Then $c \leq a\frac{d}{b}$ and $b\frac{d}{b} \leq d$, so we take $x = \frac{d}{b}$. The others can be similarly shown to be correct.

This procedure works by isolating each variable and determining a set of constraints that fit the patterns given above. Then, the variable and its quantifier are eliminated, leaving behind an equivalent formula with one fewer quantifier. Once all variables and quantifiers have been eliminated, the resulting formula contains only numbers, functions, and predicate symbols, (e.g. $5 < 3 \wedge 3+4 = 7$) and its truth can be trivially determined (in this case, $false$). In purely existential cases, a process of backwards substitution can be used to produce a statisfying assignment to a true sentence.

However, this process is very inefficient. On a change of quantification, the quantified formula must be changed to disjunctive normal form, which takes time $O(2^n)$[4].

On top of the possible cost in converting to DNF, a formula with $m$ quantifiers and $n$ constraints could require the solving of $\frac{n^{2^m}}{4^m}$ constraints [8].

## Example

An example, also from Norrish [8], is helpful. Starting from the forumla:

$$\forall x.20 + x \leq 0 \Rightarrow \exists y.3y + x \leq 10 \wedge 20 \leq y - x$$

we rearrange to form:

$$\forall x.20 + x \leq 0 \Rightarrow \exists y.20 + x \leq y \wedge 3y \leq 10 - x$$

then eliminate $y$:

$$\forall x.20 + x \leq 0 \Rightarrow 60 + 3x \leq 10 - x$$

and rearrange again:

$$\forall x.20 + x \leq 0 \Rightarrow 4x + 50 \leq 0$$

then change the universal to an existential quantifier:

[4]The easiest way of showing this is to note that an equivalent NDF formula can be constructed by forming a disjunction of all the satisfying truth assignments, of which there are $O(2^n)$

$$\neg\exists x.20 + x \leq 0 \wedge \ \neg(4x + 50 \leq 0)$$

and finally rearrange and eliminate $x$:

$$\neg(-50 < -80) \equiv true$$

The Fourier-Motzkin procedure fails for integers. On the integers, we must use an alternate scheme, of which there are several. One of the most famous is due to Cooper.

## Cooper's Algorithm

Cooper's Algorithm [7] constitutes a decision procedure for the Integers. Like Fourier-Motzkin variable elimination, it begins with a process of quantifier elimination.

The initial step is to normalize the input. This is in a similar fashion to above, changing $>$ and $\geq$ to $<$ and $\leq$, changing $\forall$ to $\neg\exists$, and removing other complicated expressions. Next, all instances of a quantified variable must be isolated, and multiplied by the least common multiple of all the coefficients of that variable. This lets you 'factor out' the coeffiecient and express it as a divisibility constraint. For eample (where $a|b$ should be read $a$ divides $b$)

$$\exists x.P(x) \wedge Q(x)$$

becomes

$$\exists x.P(ax) \wedge Q(ax) \wedge (a|x)$$

where $a$ is the least common multiple of the coefficients of $x$ in $P$ and $Q$. Then we can remove the quantifier on $x$.

First, we check if any equalities allow variables to be eliminated trivially. If, for example, we have:

$$\exists x.P(x) \wedge x = a$$

we can change it to

$$\exists x.P(a)$$

and throw away the quantifier (since there are now no quantified occurrances of $x$):

$$P(a)$$

Once this has been done, the 'real' quantifier elimination can begin. The full procedure is beyond the scope of this paper, but we can sketch an outline. First, if

$$\exists x.P(x)$$

is true then either $P$ is true for some minimum value of $x$, or else for any value of $x$, no matter how small, $P$ is true. Cooper's Algorithm splits up these two cases and eliminates the quantifier by constructing similar constraints on $x$ as in the Fourier-Motzkin procedure (see [7], or [4] for more detail).

Cooper's Algorithm follows the work of Presburger, and in doing so provides an alternate proof of its correctness.

## 1.4 Combination of Decision Procedures

Often it is necessary to work with sentences that are not expressible in a single language alone. For example, the conjunction

$$\begin{aligned}\Gamma \;=\; & \{f(f(x)-f(y)) \neq f(z),\\ & x \leq y,\\ & y+z \leq x,\\ & 0 \leq z\}\end{aligned}$$

involves the theory of Reals and the theory of Equality. In such cases, some enhancement of a decision procedure must take place. These enhancements typically take one of two forms: augmentation, or combination.

Augmentation of decision procedures was started in 1988 by Boyer and Moore, and is typically concerned with adding features to a single decision procedures to allow it to tackle undecidable extensions to decidable theories [5]. These new features include rewriting techniques, invocation of various lemmas, and other aspects of heuristic theorem provers.

Alternatively, it is possible that we will want to prove a collection of theorems which span two (or more) theories. That is, a significant number of the functions and predicates contained in the sentences are distributed across multiple theories. In this case, it is necessary to combine the appropriate theories and work with the new, larger theory when obtaining our proofs or refutations.

The combination of decision procedures poses special problems if the signatures of the two theories contain duplicated constants, functions or predicates which conflict In fact,

dealing with overlapping signatures is so difficult a problem that it is only now starting to attract the attention of researchers [6].

This lack of progress is representative of the field at large. The best example of this is that the state-of-the-art is a procedure which is more than 25 years old.

## Nelson-Oppen Method

The Nelson-Oppen method combines two or more decision procedures into a single procedure which decides sentences built from their union. When the formulas in question span more than two theories, the Nelson-Oppen method deals with them in a pairwise fashion. There are two versions described by Manna and Zarba [6], one deterministic, one non-deterministic. The following deals with the deterministic procedure, which, although more complicated, is also more suitable for practical implementation.

The Nelson-Oppen method is correct only when it is applied to *stably infinite* theories. A theory is stably infinite if all sentences constructed from it can be satisfied by an interpretation with an infinite domain. For example, the theories presented above are stably infinite. This is trivially true for the integers and reals, and the theory of equality is satisfied by taking any interpretation and adding an infinite number of new members to the domain. An example of a non-stably-infinite theory would be the following:

$$\Gamma = \{\forall x.x^2 = 25\}$$

Any interpretation satisfying a formula in this theory can have cardinality no greater than 2 (since $\Gamma$ implies that either $x = 5$ or $x = -5$).

### Variable Abstraction

The Nelson-Oppen method begins with a variable abstraction phase. Let $\Sigma_1$ and $\Sigma_2$ be two decision procedures. The first step is to transform the formula $\Gamma$ into a conjunction of formulas $\Gamma_1 \bigcup \Gamma_2$, such that $\Gamma_i$ contains only literals from $\Sigma_i$ and that the new conjuction is equisatisfiable[5] with the original formula.

Then, for every term of the form:

$$f(t_1, \ldots, t, \ldots, t_n)$$

or

$$P(t_1, \ldots, t, \ldots, t_n)$$

where $f$ or $P \in \Sigma_1$ and $t \in \Sigma_2$ (or vice versa), introduce a new variable $w$,

[5] $A$ and $B$ are *equisatisfiable* if $A$ is satisfiable $\Leftrightarrow$ $B$ is satisfiable

$$f(t_1, \ldots, w, \ldots, t_n)$$

(similarly for $P$), and introduce the new equality $w = t$. Lastly, for all equalities,

$$s = t$$

where $s \in \Sigma_1$ and $t \in \Sigma_2$, introduce another new variable $w$, and add the equalities

$$s = w, t = w$$

When this process finishes, the $\Gamma_i$ have been separated from one another, and we can proceed to the satisfiability test.

**Equality Propagation**

In the variable abstraction phase, we produced two groups of formulas $\Gamma_1 \bigcup \Gamma_2$. In the equality propagation phase, we build a set of equalities among the shared variables of $\Gamma_1 \bigcup \Gamma_2$. The procedure works as follows. Begining with:

$$< \Gamma_1, \Gamma_2, \{\} >$$

we select a new equality, $x = y$ where $x$ and $y$ are in the set of shared variables of $\Gamma_1 \bigcup \Gamma_2$. If $\Gamma_1 \bigcup x = y$ or $\Gamma_2 \bigcup x = y$ is unsatisfiable, then we close this branch of investigaion and continue. If at any point, we find that all branches are closed, we conclude that the initial formula is unsatisfiable. If, instead, we reach a stage where one branch is not closed, and no futher equalities are up for consideration, we have a satisfying equivalence relation over the shared variables of $\Gamma_1 \bigcup \Gamma_2$. There are a large[6], yet finite, number of equivalence relations over a finite set of variables, so this process must eventually come to an end.

**Example**

As an example, we will follow the deterministic Nelson-Oppen method through the example above in (1.1). This example is from [6]. First, during the variable abstraction phase, we obtain:

$$\Gamma_1 = \begin{pmatrix} x \leq y, \\ y + z \leq x, \\ 0 \leq z, \\ w_3 = w_1 - w_2 \end{pmatrix} \quad \Gamma_2 = \begin{pmatrix} f(w_3) \neq f(z), \\ w_1 = f(x), \\ w_2 = f(y) \end{pmatrix}$$

[6]In fact, the number of equivalence relations is given by the Bell numbers [1], which are super-exponential, so 'large' is an understatement.

where $w_1$, $w_2$, and $w_3$ are new variables. Note that $\Gamma$ is indeed unsatisfiable, because the last three clauses assert that $x = y$ and $z = 0$, but then the first states that $f(0) \neq f(0)$. There are 203 possible equivalences [1] over the shared variables, so we will just follow one that shows the unsatisfiability of $\Gamma$.

$$< \Gamma_1, \Gamma_2, >$$

$$< \Gamma_1, \Gamma_2, x = y >$$

$$< \Gamma_1, \Gamma_2, x = y, w_1 = w_2$$

$$< \Gamma_1, \Gamma_2, x = y, w_1 = w_2, z = w_3 >$$

$$false$$

## 1.5 Complexity and its Ramifications

In general, decision procedures for the reals and the integers are marked by extremely poor worst-case complexity. Cooper's algorithm has worst-case complexity $2^{2^{2^n}}$, and any decision procedure which involves a conversion to disjunctive normal form will have exponential worst-case complexty for the conversion process alone.

Bundy et al. [2] have shown that Cooper's Algorithm is strongly affected both by the size of a formula (in number of terms) and also by the number of different variables in a formula. Furthermore, Cooper's Algorithm performs much worse on invalid forumlas than on valid ones.

Further development into software for theorem proving must work within these theoretical bounds, and concentrate instead on more efficient techniques to reduce the hidden constants. Some of the ways that this can be done include caching of previous results to avoid reevaluation of subproofs, partial evaluation to 'short-circuit' larger formulas, using various techniques to narrow the space over which solutions are searched for, or by appealing to one of a trusted set of inference rules[7]. In some situations, it may be that applications can be limited in such a way that the worst cases are avoided, while all the useful operations can still be performed.

## 1.6 Concluding Remarks

The poor performance of decision procedures is dismaying, but does not completely invalidate them as useful tools of program verification. Although they are theoretically intractable, research is focused on making them more practically applicable. Considerable time savings can be achieved if hidden constants can be lowerd, or if worst-case performance can be avoided through clever shortcuts and other techniques. More disappointing is the lack of

[7]Thanks is due here to personal communication with Dr. William Farmer

theoretical breakthroughs, as evidenced by the fact that the algorithms discussed in this paper are all at least 30 years old. However, these facts do not render decision procedures useless.

As we have seen, there are useful decision procedures for many of the first-order logical theories that may arise in program verification. Techniques for deciding the integers (without multiplication) and the reals date back as far as 1929. These decision procedures are easily implementable, sound, and complete, and form the core of many modern theorem proving and program verification systems.

Furthermore, for problems which span more than one theory, it is possible to combine decision procedures to solve them. This ability is particularly useful in practical systems, as it allows theories to be developed easily in isolation, and used in a variety of real-world verification problems.

Improvements to these decision procedures, both theoretical and practical, may seem distant or unlikely, but the potential benefits make them worth persuing.

# Bibliography

[1] E.T. Bell. Exponential numbers. *American Mathematical Monthly*, 41:411–419, 2002.

[2] A. Bundy, I. Green, and J. Predrag. A comparison of decision procedures in Presburger Arithmetic. Proceedings of VIII International Conference on Logic and Computer Science (LIRA '97) pp: 91-101, Novi Sad, September 01-04, 1997.

[3] A Church. A note on the Entscheidungproblem. *Journal of Symbolic Logic*, 1:101–102, 1936.

[4] D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[5] P. Janicic and A. Bundy. A General Setting for Flexibly Combining and Augmenting Decision Procedures. *Journal Automated Reasoning*, 28(3):257–305, 2002.

[6] Zohar Manna and Calogero G. Zarba. Combining decision procedures. In *Formal Methods at the Cross Roads: From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 381–422. Springer, 2003.

[7] Michael Norrish. Cooper's algorithm as a derived rule in HOL. http://users.rsise.anu.edu.au/~michaeln/pubs/cooper-slides.ps.gz, 2000.

[8] Michael Norrish. Arithmetic decision procedures: a simple introduction. Automated Reasoning Group's Logic Summer School, 2003.

[9] R. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21:583–585, 1978.

[10] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publishing, 1996.

# Chapter 2

# Magdin Stoica: Software Estimation—An Engineering Approach

This paper presents what I believe to be a novel approach to software estimation, one that promises to provide a formal foundation in a domain where empirical studies are considered current state of the art. I start by motivating the need for a more formal approach to software estimation followed by a short summary of the current software estimation techniques. The paper applies control engineering theory to the process of software estimation. Furthermore it contrasts this approach to current mathematical approaches and motivates why control theory provides a much more suitable framework for analyzing and formalizing aspects of software estimation.

## 2.1 Introduction

If you are a software developer, motivating why we need better estimation seems like a moot point. There is great chance that throughout your career you have worked in at least one project that had you spend weekend after weekend, night after night, month after month in the office trying to achieve an impossible deadline. When you work on a medium sized project you have a 75% chance that you will experience excessive schedule pressure. If you work on large project your "chances" increase to 90% [4]. By definition this was termed by Yourdon as a *death march* [6]. Even if you've never heard of this term before you understand it immediately and completely. You have also been asked to use you gut feel to estimate software.

If you are a manager, you are probably frustrated with the unreliability of existing estimation techniques to a point where nothing seems to be worth using. The time and effort you put into improving your team's estimation skills doesn't seem to pay off. In the end, Murphy's

Law of software project planning always prevails: "*after careful planning it only takes twice as long to complete as expected, compared to three times for no planning whatsoever.*" [2]

If you are a user, you got used by now with computers "freezing" and applications crashing. But lately, you are probably wondering how come your TV seems to lock sometimes and why is the customer support representative insist on you rebooting your cable box – whatever that means; or why your phone sometimes dials the wrong number or drops calls and why your car simply refuses to start or tries to go in reverse while you are on the highway driving 100 km / hour. You might be waiting for an increase in pension that cannot come because the software system cannot handle such an increase. Or you may be paying too many taxes because the government looses billions of dollars on runaway software intensive systems.

There are many reasons why software projects go wrong but the number one cause was identified to be optimistic estimation [5].

## 2.2 A mathematical approach

### The empirical stage

Today, software estimation is in the empirical stage. As in any other domain, scientists perform measurements to try and come up with the equations defining the domain behaviour and rules of operation. Computer scientists are following the path of their ancestors. Measure, perform experiments, try to generalize a result from those experiments and express these results into a formal, mathematical theory.

The results of this approach are estimation models which define software project parameters and equations that relate parameters and estimation. Estimation models have tens of parameters like type of application, team size, team experience, productivity, expected turnover, computing platform, programming language, process used etc.

The mathematical approach is to define one or more estimation functions which given the project parameters as inputs will output the estimation of software cost and / or schedule duration.

### Basic estimation equations

Most estimation equations attempt to first estimate the project size and than relate size to cost and schedule. This is so because size can be easily measured post-mortem and since we are in the empirical phase this makes perfect sense. We keep measuring projects, we know how much they cost and how much time it took to build them and having these measurements we are trying to come up with the equation that relates the two.

For example the equation relating the actual schedule duration to the estimated effort is given by:

$$schedule\_in\_months = 3.0 \times \sqrt[3]{man\_months\_effort}$$

Notice in the above equation the 3.0 factor. This was computed completely empirically. In fact, opinions vary about whether the factor should be 3.0, 4.0 or 2.5 [1].

Sometimes equations are complemented by table containing the variations of possible factors. For example, Capers Jones came up with the following equation used for computing schedule duration based on function-point count:

$$schedule\_in\_months = (function\_point\_count)^{\alpha}$$

where $\alpha$ is influenced by two factors: the type of software and the organization productivity and is given in tabular format as in Figure 1.

| Type of Software | Best in Class | Average | Worst in Class |
|---|---|---|---|
| System Software | 0.43 | 0.45 | 0.48 |
| Business Systems | 0.41 | 0.43 | 0.46 |
| Shrink-wrap | 0.39 | 0.42 | 0.45 |

Figure 2.1: First Order Estimation [1]

Sometimes estimation equations are given completely in tabular form a technique called blue-book estimation, a term used in construction [3]. An example of an equation given in tabular form relating product estimated size to schedule duration is given in Figure 2.

| **Estimated Size** (LOC) | **Schedule Duration** (calendar months) | | |
|---|---|---|---|
| | System Software | Business System | Shrink-Wrap Products |
| 10,000 | 10 | 6 | 7 |
| 25,000 | 15 | 9 | 10 |
| 50,000 | 20 | 11 | 14 |
| 70,000 | 23 | 13 | 16 |
| 100,000 | 26 | 16 | 18 |
| 200,000 | 35 | 20 | 24 |

Figure 2.2: Tabular Form Equation [1]

## Estimation models

The basic equations are simplistic approaches that can be used to obtain only rough estimates. For more precise estimates one would use an estimation model. One of the most used estimation models is COCOMO created by Barry Bohem. In this model equations are tuned not just by type of application and productivity but by many factors. In COCOMO II equations include two types of factors: scale drivers and cost drivers. Scale drivers influence equations exponentially while cost drivers are only multipliers [7].

Scale drivers used in COCOMO II are:

- Precedentedness
- Development Flexibility
- Architecture / Risk Resolution
- Team Cohesion
- Process Maturity

There are many cost drivers included in COCOMO II and they are divided into four categories:

- Personnel Factors (e.g. experience, capability, continuity)
- Product Factors (e.g. reliability requirements, reusability requirements)
- Platform Factors (e.g. timing constraints, size constraints, type of platform)
- Project Factors (e.g. use of tools, multi-site development)

COCOMO II estimates based primarily on a given size estimation measured in lines of code. As such, two of the most fundamental equations of the COCOMO II model are:

i) Effort estimation equation given by:

$$man\_month\_effort = 2.94 \times eaf \times (project\_size\_estimation)^{E}$$

where *eaf* is effort adjustment factor derived from cost drivers and E is the exponential factor derived from the scale drivers.

ii) Schedule duration equation given by:

$$schedule\_in\_months = 3.67 \times (man\_month\_effort)^{SE}$$

where *SE* is the schedule exponential factor derived from the scale drivers

From these two equations the staff requirements can be simply derived by dividing the effort into the number of months:

$$average\_staff\_requirements = \frac{man\_month\_effort}{schedule\_in\_months}$$

Estimation models are more customizable than basic equations thus applicable to a larger variety of projects. COCOMO II acknowledges the many factors a software schedule can be influenced by and attempts to consider them in its equations. Fundamentally however estimation models are not different than the basic equations or blue-book type estimations. They too are empirical and they too cannot possibly cover all things that can go wrong in a project. In addition they all start from an estimation for which they offer almost no support other than empirical data: the estimation of size. Even worse, COCOMO II uses lines of code as means to estimate size which were proven to be a flawed measurement for many reasons: code doesn't exist in the early stages of development, LOC penalize high-level languages and they cannot be used to measure productivity as the cost per lines of code often increases the more productive the team is [2].

### Overcoming limitations

Is our limited capacity to estimate caused by a lack of estimation models or equations? It is my opinion that this is not the case. COCOMO I and its successor COCOMO II, are detailed and thorough and they are the stepping stone of software estimation.

The limitations come from the mathematical approach we are taking in software estimation. We will never be able to find enough equations, factors and drivers to account for each an every problem a software project might. Even if we could find a complete model, a software project is not static. It is a continuously changing dynamic system and thus all the factors and drivers we set when we estimate, change with it – continuously.

To overcome these limitations we must change our fundamental approach to estimation.

## 2.3 An engineering approach

There is a class of systems in nature which behave just like software development projects do. They are dynamic, affected by continuously changing factors, too difficult to predict and sometimes simply unavailable for analysis. They are called dynamic systems and an entire engineering domain is dedicated to analyze them and controlling them: control systems engineering.

### Control systems - a simple example

The cruise control we enjoy in almost any car is a control system. The cruise control is used to control the speed of the car and keeping it constant, equal to a desired *set point* – the speed you want to travel with.

The theory of operation is incredibly simple: the cruise control receives as input the current speed of the vehicle from a sensor and depending on the value of the current speed it either slows down the car if the speed is above the set point or it accelerates if the speed is below it.

Imagine a mathematical approach to solving this problem: trying to come up with a *static* equation that gives an estimation of desired acceleration or breaking levels such that speed stays constant around a desire set point. Using this equation the car would know ahead of time the amount of acceleration or break to apply at any point in time.

The first question you would ask when attempting to solve a mathematical problem is "what do I know". What is the hypothesis of the problem? Can we assume something about the type of road, the road conditions, the type of car, the driving skills of the driver, how many passengers are in the car, traffic? The answer is NO. We want a model that would work for any conditions, any time. I can almost hear the answer in your mind – it's impossible. How can you take in consideration all the possible factors a vehicle's speed is affected by? Impossible! I couldn't agree more.

Now imagine that instead of trying to find another approach to solve our cruise control problem we start measuring the speeds of every car out there in all kinds of road conditions, times of day etc. We measure hundreds of thousands of cars and then we try to extrapolate from the data an equation that would likely apply to any car and any condition. But there are always new type cars, new types of tires, new roads, and new types of fuels. So we need to perform all measurements all over again to include these new parameters and we need to come up with new equations.

One can see how incredibly difficult this task would be if not impossible. Even if we could derive such a mathematical model, as soon as a new factor would come into play the estimation would fail and our cruise control would malfunction.

Let's recap how a cruise control system *actually* works:

- measure current speed
- if current speed is below set point, accelerate
- if current speed is above set point, break

The cruise control system is shown using a typical control systems block diagram in Figure 3.

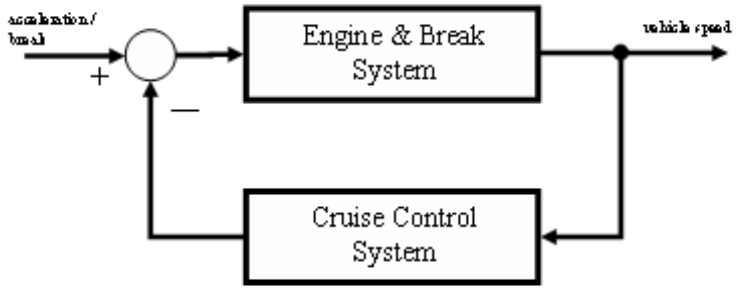



Figure 2.3: Cruise Control System Block Diagram

## Software Estimation System

It is obvious that a mathematical approach to solving the cruise control problem is fundamentally flawed. This is because it is not a problem of estimation; it is a problem of control. What if software development systems are the same? What if we cannot estimate them? What if we can only control them? How much simpler and more predictable would software estimation be if we take a control approach?

To apply control engineering principles we must be able to draw a similar block diagram for our software development system. Thus we need to define at least:

- the system itself, also called plant in control engineering terminology
- the output of the system
- the set point
- its input
- state variables

**Plant**. A software estimation system is made of the software to be built, people building the software, the process they are using along with the tools. Last but not least the organization itself is also part of this system.

**Output.** The output of the system is the estimation accuracy. We define the estimation accuracy as:

$$ea_d \equiv \frac{estimation\_of\_schedule\_duration}{actual\_duration}$$

In this case we call it *duration estimation accuracy.* Depending what we are trying to estimate we might define the estimation accuracy in terms of project cost.

$$ea_c \equiv \frac{project\_cost\_estimation}{actual\_cost}$$

**Set point**. In control system the set point is the level of output we are trying to stabilize the system at. For the cruise control system for example it is the speed of the car we want to travel with. In our case the set point is as follows:

$$sp \equiv 1$$

What we are trying to achieve is perfect estimation accuracy. If we would have that our estimation would be equal to the actual duration (or cost) and therefore it would be equal to one.

**Input.** The input of the system will consist of software artefacts. Of course the types of artefacts vary depending on the phase in the development life-cycle. In requirements analysis the inputs would be use-cases, maybe tables depending on the requirements methodology. In some cases prototypes would be an input in this phase too. During design the input may consist of classes, diagrams, requirement changes, more prototypes. During implementation we have code which can be measured in terms of lines of code or classes or functions. During testing we have test cases, defect correction, changes. Throughout the life-cycle the project plan will always be an input since it will contain the projections for schedule and cost.

**State variables.** Although not visible on the block diagram control systems work by monitoring the state of the system as well as the output and modifying the states. The state the estimation system is made of all the factors present in current estimation models. This is where we can use estimation models – to model the estimation system, of the plant. Figure 4 depicts the estimation system as a control system.

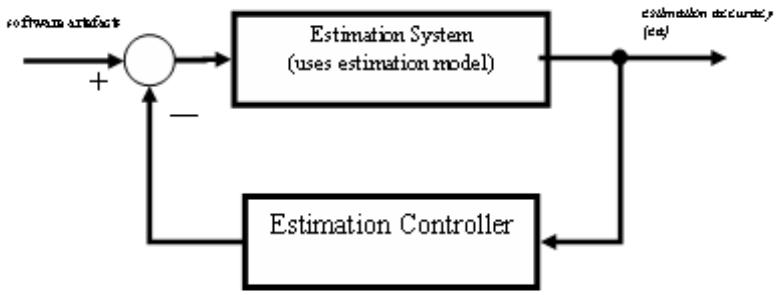



Figure 2.4: Estimation System

Size of the system for example is a state and not an output. Although all estimation models start from estimation of size nobody is really interested in how many lines of code or function points are in system. What we are always interested in are when the system will be ready and how much will it cost. This matches the description of a system's state. It is used by the system, it is used to control the system, it influences the output but it is not an output.

## Controlling the estimation accuracy

Control engineering provides the perfect framework for formally defining control policies that will lead to reliable estimations. Notions like controllability, stability, observability, system transfer function with its zeros and poles, all provide an insight into what we need to analyze in order to be able to estimate effectively.

**Measurements.** While measurement is heavily advocated as one of the main components of any estimation effort it is not usually done. Regardless of estimation accuracy a company should be able to learn from its successes and failures. Without measurements this is impossible and in fact in many cases nobody really knows the cost even for a completed system nor does anybody know the actual schedule duration (i.e. including all the overtime). The lack of measurement is exactly why a successful death march will only make matters worse by transforming poor estimation practices into company policies.

In general companies understand that you cannot optimize what you cannot measure. If we establish the fact that estimation cannot be calculated it can only be controlled (optimized) than measurement might be seen as what it is – an absolute necessity.

Control systems cannot exist without measurement. Imagine how would the cruise control system work if there wouldn't be any sensor to sense the current speed of the car? How would it know when to accelerate, when to break and when to simply stay put? Without sensors, without measurements, control systems cannot work – they are fundamental to their theory of operation. When seen in this light it is obvious that without measurements no estimation control can be performed and therefore an accurate estimation cannot be obtained.

**Stability.** In control systems stability is decided by analyzing the *transfer function* of the system – the function that relates its inputs to its outputs. When a system is unstable there exist certain inputs that cause the system to have an unbounded output and therefore fail. The transfer function is usually given as a quotient of two polynomials and the roots of the denominator are called the poles of the system. When the transfer function has positive poles the system is unstable.

An estimation system is definitely unstable. While we cannot characterize it through a simple transfer function we know very well what the *unbounded output* looks like - an estimation that keeps increasing to a point where the project is cancelled.

**Controllability.** A system is controllable if through feedback control, its unstable (positive) poles can be stabilized (can be moved over to the left plane) such that when controlled, the system is stable.

An estimation system is controllable. We know that because there are companies out there who can estimate software development system with 3% error [4].

So how do we control the estimation system? In control theory there are two fundamental types of controllers:

1. proportional controller
2. proportional-integral-derivative controller (PID)

And they all have an equivalent in current estimation techniques.

**Proportional Estimation Controller.** Proportional estimation controller would multiply the estimation by a certain constant factor. You try and estimate the best you can with the information you have and then you multiply everything by a factor 1.5 or 2 and you get the "controlled" estimation. Proportional controller is a very basic type of control which is again true in the case of an estimation system – it only works if you are always wrong by the same factor.

**PID Estimation Controller.** By adding a derivative and the integral components the controller is able to anticipate future errors with a greater precision. By checking the current error vs. the last error the controller knows if the error is getting bigger or smaller and knows how to modify the output appropriately. In a similar fashion a recommended re-estimation technique it to multiply the estimation by the same percentage as the last error. If let's say in the first task you are wrong by 50% then you multiple the entire schedule by 50%. If in the second tasks (which is not estimated to take 50% more) you are wrong by -25% (you do better than estimated) then you multiply the entire schedule by -25%. This is a much better choice then simply adding the error to the schedule which is what is usually done – when the first task is 100% underestimated and it takes two weeks instead of one than the extra week is added to the schedule which is a negligible percentage of the whole schedule.

**Estimation Performance.** In control theory the performance of a system is given in terms of its response to several factors. The typical output of a controllable system is given in Figure 5. All these factors have clear equivalent notions in a development system.

The *maximum overshoot* is the maximum estimation error and the Peak Time is the time it takes to get there. When multiplying all tasks of the schedule by the estimation error you made on the last task you are bound to get an overshoot. It is unlikely (although not impossible) the each and every one of the tasks are overestimated. Usually smaller tasks are underestimated while bigger tasks are overestimated. If you thought a task is going to take an hour and it took eight it means you need to multiply the entire schedule by eight. This will most likely generate an overshoot.

The *settling time* is the time it takes the system to start operating in the specified range. When building a new system the estimation is very rough. It takes some time until developers become familiar enough with the system being built so that they can estimate properly. It also takes some time for a new team to gel and for the project to "find its rhythm". This time is the settling time of the estimation system.

The *rise time* is the time to get into 90% of the desired output. This too is extremely important to estimation systems. Some projects start as simple projects. While work is done

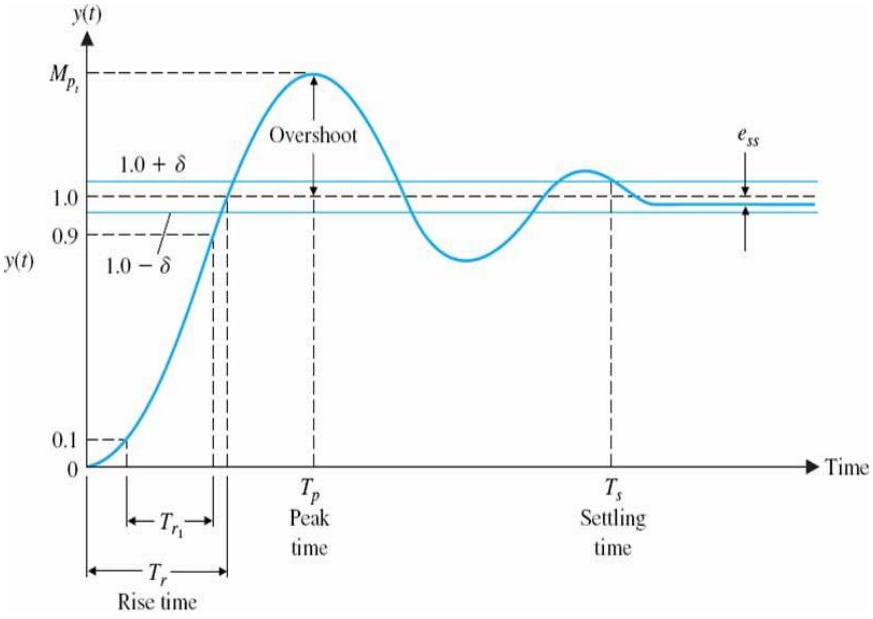



Figure 2.5: Output of a controllable system

developers start to realize their complexity. The estimation keeps increasing and increasing and the project is cancelled because there doesn't seem to be any end in sight. Minimizing the rise time of the estimation system means we get quicker to an estimation that is closer to reality. As with other control systems minimizing rise time means increasing the overshoot – if you hurry to get to your set point chances are you will go beyond and need to come back.

Most development teams are afraid to overshoot their estimation which leads to an output that has little or no overshoot but a rise time equal to the settling time. That is precisely when projects get cancelled. Not because they were costing too much, not because they were taking too long but because the rise time of the estimation was too slow and management couldn't possibly know if the project will ever finish.

## 2.4 Conclusions

Working with estimation as we would with a regular control system provides us with a formal, proven framework of analyzing and optimizing the process of estimation software schedules and costs.

The notion of controlling vs. estimation is not new. In "Principles of Software Engineering Management" Tom Gilb argues this exact point stating that iterative development provides the means of calibrating the estimation based on real data rather than empirical data. In "Controlling Software Projects: Management, measurement & estimation" Tom DeMarco also advocates replacing estimation with projection, measuring and re-estimating as often as

possible. He also advocates separating the estimation and measurement function from the development function as means to reduce interference.

This paper's contribution is applying control theory to explain the natural instability of the estimation process, the need to actively control the estimation process in order to obtain a stable estimation and ways to asses the performance of an estimation process through parameters used in assessing the performance of real-life control systems.

Instead of continuing the mathematical approach followed until now it is time we recognize its limitation and use a well researched, well understood and formally defined area of control engineering to exit once and for all the empirical stage we seemed to be stuck at.

# Bibliography

[1] Steve McConnell. *Rapid Development*. Microsoft Press, 1996.

[2] Brian Dreger. *Function Point Analysis*. Prentice Hall, 1989.

[3] Tom DeMarco. *Controlling Software Projects: Management, Measurement & Estimation*. Yourdon Press, 1982.

[4] Steve McConnell. *After the Gold Rush. Creating a True Profession of Software Engineering*. Microsoft Press, 1999.

[5] Robert Glass. *Frequently Forgotten Fundamental Facts about Software Engineering*. IEEE Software, May/June, 2001.

[6] Edward Yourdon. *Death March*. Yourdon Press, 1999.

[7] Overview of COCOMO. http://www.softstarsystems.com/overview.htm.

# Chapter 3

# Hossein Safyallah: Survey of Dynamic Analysis Techniques

Dynamic analysis is a process of using dynamic and behavioral information of the software system to address its problems such as program profiling, software architecture recovery, and program verification. Dynamic analysis constructs a new model by changing the data gathering phase of the program analysis to solve the problem and constructing a new model based on the new extracted data. In this paper, we first introduce the dynamic and static analysis procedures, and how they work together to solve such problems. Further, we take a close look at some specific problem areas: program analysis, reverse engineering, and program verification, and how their problems can be solved using dynamic analysis procedure as an auxiliary or pivotal method.

## 3.1 Introduction

Software systems need to be analyzed during their lifetime in different aspects. In the task of software development, software should be investigated to reveal its bugs, errors and mal-functions. During software testing phase, software code is studied to find if a desired test suite is complete or not; non-functional requirements such as performance are also measured in this phase. On the other hand, a program might be studied for its code to be verified. In addition to forward engineering, program analysis also plays an important role in reverse engineering. There, to find the software architecture, software code, its documents, and its runtime behavior are analyzed. Program analysis can be seen in two different views, static and dynamic analysis. Static analysis investigates the program code. However, dynamic analysis is focused on the behavior of the program at its runtime. These behaviors include program output and program states during runtime, (e.g., variable values, class instantiations, and function invocation sequences).

Static analysis of a software examines its code and builds a model of program states which should be valid for all possible program executions. Therefore, a typical static analysis has the following properties; it is sound , i.e., the results of the analysis is true no matter of the

program input or the situation in which the program is run. Also, it is conservative, i.e., this analysis reports weaker properties than those which may in fact be true. For instance, if the program code contains iterative statements, a conservative assumption is an assumption which is true when the statement is executed once, several times or not at all.

Making a sound model turns the static analysis to an imprecise and time consuming analysis. It is a time consuming procedure, since building such a huge state space containing all possible behaviors takes time. Also, it is an imprecise analysis, because there are many possible executions, and the analysis must keep track of multiple different possible states, but it is usually not reasonable to consider every possible run-time state of the program. Therefore, static analysis works by using an abstarcted model of program state that loses some information, however; it is more compact to manipulate. As a result, it turns to an imprecise analysis with more approximate assumptions and abstractions [2][3].

Dynamic analysis operates by executing a program and observing the executions. Unlike static analysis, which is a compile time analysis, dynamic analysis is a runtime analysis. It acts on actual observation of the system, which in turn makes it totally valid for that specific execution. Moreover, dynamic analysis is precise because no abstraction needs to be done, and actual behavior of the system is observed instead of an abstract model of it. This fact makes the dynamic analysis a faster analysis, as fast as a program execution [2][3].

## 3.2 Dynamic Analysis

Generally, analysis contains of at least two obvious phases, data gathering and processing. In dynamic analysis, data gathering phase is performed by running the program and extracting the desired data from the running software.

Gathering data from the running software makes this analysis a dynamic one which may vary from run to run. It also makes the analysis to be dependent on a specific scenario. Thus, data gathering phase of a dynamic analysis has the following properties: a single run is insufficient, and a complete or semi-complete set of scenarios, which reveals all aspects of the software program, should be applied.

Applying a set of scenarios is a challenge in dynamic analysis. First of all, a well-selected set of test cases should be made, and then the software system should be modeled by the relation between its input scenarios and desired output. As stated in the previous section, desired output of the system might differ from the actual output. In one application, variable values, which was captured in entry and exit points of each function, make the desired output. In others, the sequence of invoked functions through a run makes the desired output.

Managing consistency and accuracy among the extracted data is another challenge in application of dynamic analysis. There, handling a huge amount of extracted data and making the proposed domain model are major problems. Therefore, an intermediate phase of knowledge extraction applies before the analysis phase. Knowledge extraction is a process in which data is being cleaned (by deleting noise and redundancies), and hidden relationships (among data) are being discovered. This phase is usually done by applying data mining algorithms or pattern matching algorithms to the extracted data.

Application of dynamic analysis in software engineering could be categorized in three different categories: program analysis, reverse engineering, and program verification. Program analysis involves with performance and test coverage measurement, in which both have a dynamic nature and reveal just in run time behavior of the system, and memory leak detection which could be done both statically and dynamically. Program verification and validation is another area in which information of running software helps the proposed task done easier. In reverse engineering also new approaches of using behavioral information of program code is used in order to discover the architecture of the legacy systems [4]. Unlike traditional methods of reverse engineering which only involve software code, now dynamic data which could be obtained from a running software is mined to discover new views of a software system.

## 3.3 Program Analysis

In this section we discuss the application of dynamic analysis in program analysis. Our discussion in this section is divided into three different parts and in each part one method or application of dynamism in program analysis is discussed in details. In the first part we mention the *Profiling*, a method for identifying performance bottle-necks, and then a common implementation of this technique, *gprof*. In the second part *Code Coverage Analysis*, a technique for qualifying test suites, is discussed in details. And finally we review the *Memory Leak Detection* as another technique in program analysis.

### Profiling

As stated in [5] *profiling* is a series of techniques for gathering information about the behavior of a program during execution, and specially for recording the amount of time spent in each part of the program. *Profiling* is a technique for identifying performance bottle-necks and measures the time spent in each subroutine as the program runs. Each profiler has two distinguished phases, gathering profile data and data presentation.

Data gathering phase in profiling methods is almost done by two different methods: *Instrumenting* and *Sampling*. Instrumenting profilers insert special piece of code at entry point and exit point of each function and record start time and end time of the subroutine as it runs. With this information the actual time spent in each subroutine is measured. Instrumenting profilers impose an overhead to each instrumented subroutine, which is the amount of time spent in the instrumenting code itself at start and end point of each subroutine, which should be measured separately and subtracted from the amount of time spent in each subroutine. Even considering this point there still remains another point of interest with this method as stated in [6] as bellows:

> However, when a routine is very short, another effect due to the instrumentation becomes important. Modern processors are quite dependent on order of execution for branch predictions and other CPU optimizations. Inevitably, inserting a

> timing operation at the start and end of a very small routine disturbs the way it would execute in the CPU, absent the timing calls. If you have a small routine that is called millions of times, an instrumenting profiler will not yield an accurate time comparison between this routine and larger routines. If you ignore this, you may spend great deal of effort optimizing routines that are not the real bottlenecks.

On the other hand, *sampling* profilers perform their task without modification of the application under profiling and all profiling work is done outside the application's process. In order to do that these profilers record the currently executed instruction as CPU is interrupted to do context switching. So the whole operation of the profiling is done by copying the content of the *Program Counter* register to the memory and at the end *frequency* of executing a specific line of code or a subroutine is computed over the part or entire program run time. Here the overhead imposed by the profiler is negligible but profiler just tells what routine is executing currently, not any information about where it was called from. So unlike *instrumenting* method it cannot give call-graph information about the profiled program [6]. A call-graph is a directed graph in which nodes are program subroutines and there is an directed edge between two node if and only if one was called from the other.

The *gprof* is a common Unix based profiler tool which uses both instrumentation and sampling as its data gathering phase. The *gprof* counts the number of calls of each subroutine and charges its time to the callers in proportion to the number of calls they make. In fact *gprof* is a compiler assisted tool and the program under analysis should be compiled with special options in order to be get instrumented. The instrumented version of the program now invokes a call to an special function, *mcount*, as its first operations in each subroutine in order to record some information about where it was called from. Sampling is also done by watching the program counter register at some predefined frequencies [7]. The *gprof* provides its output in the following forms [1]:

- The *flat profile*, which shows how much time your program spent in each function, and how many times that function was called.
- The *call graph*, which shows, for each function, which functions called it, which other functions it called, and how many times. There is also an estimate of how much time was spent in the subroutines of each function.
- The *annotated source listing*, which is a copy of the program's source code, labeled with the number of times each line of the program was executed.

## Code Coverage Analysis

As stated in [8] *code coverage analysis* is a technique of test suite qualification. This qualification is done by finding areas of a program which are not exercised by a set of test cases as

[1] Information is provided by the *gprof* Unix manual page

```
int* p = NULL;
if ( condition ) p = &something;
*p = 123;
```

Figure 3.1: Deficiency of line coverage technique regarding conditional statements

well as identifying redundant test cases that do not increase the coverage. The granularity at which *code coverage analysis* is performed varies in a large range which *line*, *branch*, *function entry*, *loop*, and *race* are some examples of it.

The *line coverage* is a technique in which running program is observed as each line of code is executed through an entire program run or not. This kind of coverage benefits the simplicity of implementation, but has some serious disadvantages regarding conditional statements, e.g., in code snippet of Figure 3.1 the coverage would always reported as complete regardless of the *condition* value. A common implementation technique of line coverage is to put a breakpoint on every line of code and record that line whenever that breakpoint is hit.

Unlike the *line coverage* technique the *branch coverage* checks whether all `if` statements have been taken in both the `then` and `else` directions [9]. Other than *line coverage* and branch coverage which engage largely in the source code there is another kind of coverage measurement with a larger granularity. The *function entry* coverage measures if each function is executed or not, and so it provides a higher level of information. The *loop coverage* is another technique in coverage analysis which determines whether each loop body executed zero times, exactly once, and more than once. And finally *race* coverage reports whether multiple threads execute the same code at the same time. In fact *race* coverage is a good technique to discover failures in synchronized resource access [8].

## Memory Leak Detection

One other aspect which its problems is addressed in program analysis is *memory error detection*. Memory errors are categorized in memory leaks, read of un-initialized memory, access to de-allocated memory, and access out of bound. Memory leak is a memory management failure to release an allocated memory block. There are two types of memory leaks, physical and logical memory leak. A logical memory leak occurs when an allocated block of memory is not used through the whole lifetime of the program. A physical memory leak occurs when there exists no pointer to an allocated block of memory, a memory access problem which could not occur in programming languages such as Java or C# unless the garbage collector does not perform correctly. One common method in memory leak detection is the use of reference counting. In this method a reference counter for each block of memory is maintained and garbage collecting algorithms such as mark-and-sweep are used in order to do the task of detection. In order to keep track of memory block information, dynamic instrumentation could be used at machine code level, a method which is used in [10]. Dynamic instrumentation makes the proposed method completely language independent. Also it gives the ability

to show where the leaked blocks were allocated, lost and where the last references to these blocks were created. The proposed method to handle memory leaks in [10] is as follows:

> For each allocated block, quite a bit of information is recorded. First of all, the call stack at the time of allocation is stored. Next, we also give each allocated block a reference count, a unique identifier (called a *block id*), and a *usecount*. This last field keeps track of how many times a reference to said block has already been created and will allow us to detect stale references.

Other kind of memory access errors could be resolved in similar methods as discussed above.

## 3.4 Reverse Engineering

Reverse engineering is a research area intended to recover design decisions which were taken in the software development time. This process is consisted of three different phases: data gathering, fact extraction, and analysis. Traditionally data gathering phase in reverse engineering was performed using the program source code as the major source of information about the software system, which made the reverse engineering a static analysis. Dynamic analysis contributes to reverse engineering in its data gathering phase. Run time information and behavioral information about the code now is obtained by instrumenting the software system. In this chapter three different dynamic methods in reverse engineering is presented. First *behavior recovery* is described as a new method of discovering the dynamic components of the software system, then a method of discovering the usage scenarios from runtime behavior of the system is described, and finally *architecture recovery* of object-oriented applications is presented as a reverse engineering method for future legacy applications, i.e. future legacy system are today's object-oriented systems.

### Behavior Recovery

Program output, state transition and execution trace are three ways in which software (dynamic) behavior could be described. Dynamic components in a software could then be defined as a set of functions which are working cohesively in order to perform a specific task. *Software behavior recovery*, the act of constructing dynamic components of a system by inspecting its behavior, is performed by inspecting a set of program execution traces and constructing dynamic components of that program by use of a set of data mining algorithms. A sequential pattern mining algorithm based on AprioriAll [12] is used to find the common patterns, a sequence of function invocations, which exist among execution traces. Then based on these patterns a degree of similarity between each function pair is determined. Finally a clustering algorithm is applied to find the major components of the system.

Data gathering phase of *behavior recovery* is performed using an instrumenting tool, namely *Aprobe*, which is a binary level software instrumentation tool. The instrumented

software system then constructs a dynamic call graph of its actual function invocations through its run. The chronological order of invoked functions which can be obtained by traversing the call graph in depth first search manner then is adopted as input data to be mined in order to find the common patterns of functions later [11].

## Use Case Recovery

Stroulia et al. [14] address the problem of legacy system software requirements loss by means of dynamic analysis. Software requirements loss is a common problem which legacy systems suffer from. As software systems get regularly maintained through out their lifetime, bugs fixed or new features added to the system, the documentation of their requirements become obsolete or get lost. They develop an interaction pattern miner to recover functional requirements as usage scenarios. Their method analyzes traces of the run time system user interaction to discover frequently recurring patterns. These patterns show the functionality currently exercised by the system users and represented as usage scenarios [14]. The discovered scenarios provide the basis to migrate the under study legacy system into web accessible components.

This method of recovering the system usage scenarios has similarities with the behavior recovery method discussed in the previous section. In both, analyzing the dynamic behavior of the system leads to discovering a view of the system. Here the dynamic data which the actual pattern mining is intended to mine is snapshots of the legacy system. These program snapshots are captured through running the system under a set of specific scenarios. However, in behavior recovery sequences of subroutine invocations was the data under investigation.

## Object-Oriented Applications Architecture Recovery

System functionality of object-oriented applications is provided by cooperation of interaction of objects and methods [4]. Discovering this functionality from the source code is hard or even infeasible. Moreover object-oriented features such as *polymorphism* and *inheritance* make this task even harder. *Polymorphism* makes it difficult to determine which method is actually executed at runtime. *Inheritance* means that each object is not only defined in its class, but also in each of its super-classes.

Dynamic analysis contributes to recovering the architecture recovery of object-oriented applications in data gathering phase as well as other problem areas. Dynamic information such as class loading, object creation, and method invocations are captured by instrumenting the program. Then classes and their runtime relations are displayed as a dynamic dependency graph (DDG). A DDG is a graph where its nodes represent classes or objects and edges represent relations (e.g., method invocation) between two objects or between a static class and an object [13]. In order to find the dynamic structure of the system a clustering algorithm is used to partition the DDGs. This is a similar method to static architecture recovery of the legacy systems. In [1] a similar approach is adopted to parse the source codes of a legacy

system and extract source graph of the code. Then applying clustering algorithms leads to finding the system modules and their relationships.

## 3.5 Program Verification

This section describes the application of dynamic analysis in program verification. Program verification is a formal method for proving the program's consistency with a formal specification. This method is mainly done with use of model checkers and theorem provers. Model checking is an exhaustive search through an entire state space. This search can only be done when the state space has a finite size, i.e., when the number of processes in the system or possible variable values increase the analysis becomes time consuming or even infeasible. Theorem provers on the other hand manipulate logical formulas so they scale to unbounded processes and variables. Program verification is mainly done with static analysis of the program code, dynamic analysis instead helps making some human intensive parts of this process done easier [15].

Program verification using dynamic analysis is mainly done in three steps; dynamically discovering likely program invariants, run the program under a specific test suite, and use of discovered invariants as lemmas for theorem prover. In the rest of this section each step is described in more details.

### Dynamically Discovering Likely Program Invariants

Dynamically discovering the program invariants is a process in three steps; instrumentation, running the subject system over a test suite, and inferring invariants [17].

Invariants could be detected at specific points of the program such as procedure entries and exits. By instrumenting the program, variable values at these specific points could be sampled. Then a process of invariant inference is done for finding unary, binary, and ternary invariants. Now each tuple of variables up to arity 3 is checked to find if it is a potential invariant based on a list of predefined invariants or not. Examples of predefined invariants are as bellows:

- constant value
- small value set, i.e., the variable only takes a small number of different values.
- range limits: the variable $x$ has following property: $x \in [a \dots b]$.
- linear relationship: $y = ax + b$

The extracted invariants by this method are likely invariants and should be proved with other methods, because they are all inferred by inspecting the values of a running program. Since this observation is not complete at all so the results can not generalize to all situations.

### Program Execution

The accuracy of the likely invariants detected in previous step highly depends upon the test suite used for running the program. Therefore, generating suitable test cases that support accurate detection of program is crucial in this problem. Test data can be generated randomly or using grammars.

Randomly generated test suites have pour coverage and failed to execute many portions of a program. Grammar based test suites on the other hand is a black box approach and in general can fail to cover a significant part of the program. An approach here is that to augment the current test suites using feedbacks of running the dynamic analysis. In this approach dynamic analysis may be run using a set of test suites and then detected program invariants can be used in order to make the previous test suite more accurate [16].

### Program Verification Using Likely Program Invariants

Likely program invariants is now suggested as some lemmas to help proving the final goal of the system. In fact lemmas take the place of substantive human input to theorem prover, which greatly eases proving program properties [15].

## 3.6 Conclusion

In this paper, various methods in which dynamic analysis contributes to software development process was studied. In all methods, dynamic analysis techniques made the whole analysis a precise one by taking the actual states of the running program instead of an abstract model of it. Instrumenting phase of all described methods addressed only the data of interest. Thus, the amount of effort to solve the problem was extensively reduced. A chief disadvantage of all dynamic analysis methods is that their results may not generalize to future executions. Therefore, other methods are required to prove the validity of the performed job.

Static and dynamic analysis can enhance each other by providing information that would be otherwise unavailable. Each of these analysis systems can address a certain view of the subject system and they may complete each others' results. Moreover, one can provide intermediate information or guidelines to ease another's process.

# Bibliography

[1] K. Sartipi. Software Architecture Recovery based-on Pattern Matching. Ph.D. Thesis, School of Computer Science, University of Waterloo, 2003.

[2] M. D. Ernst. Static and dynamic analysis: synergy and duality. ICSE Workshop on Dynamic Analysis, Portland, Oregon, USA 2002.

[3] C. Steindl. Static Analysis of Object-Oriented Programs. 9th ECOOP Workshop for Ph.D. Students in Object-Oriented Programming, Lisbon, Portugal, June 14-15, 1999.

[4] T. Richner, S .Ducasse. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. In Proc. of the Int. Conf. on Software Maintenance, pp. 13-22, Oxford, England, 1999.

[5] J. M. Spivey. Fast accurate call graph profiling. Software-Practice and Experience, v.34 n.3, p.249-264, March 2004.

[6] AutomatedQA. The Truth about Profiling. Technical Paper, http://www.automatedqa.com/techpapers/profiling.asp.

[7] J. Fenlason and R. Stallman. GNU gprof, The GNU Profiler. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/html_node/gprof_toc.html, 1998.

[8] S. Cornett. Code Coverage Analysis. http://www.bullseye.com/coverage.html, July 2004.

[9] B. Marick. A Buyer's Guide to Code Coverage Terminology. http://www.testing.com/writings/coverage-terminology.html.

[10] J. Maebe, M. Ronsse, and K. De Bosschere. Precise detection of memory leaks. Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004), pp. 25-31, 2004.

[11] K. Sartipi, H. Safyallah, and N. Dezhkam. Multiview Architectural Reconstruction Environment to Enhance an Evolving Software System. submitted for WCRE 2005, 2005.

[12] R. Agrawal, R. Srikant. Mining Sequential Patterns. In Proc. of the 11th Int. Conf. on Data Engineering, pp. 3-14, IEEE Comp. Soc. Press, 1995.

[13] J.Gargiulo, S.Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 244-251, Buenos Aires, Argentina, June 2001.

[14] E. Stroulia, M. El-Ramly, and P. Sorenson. From Run-time Behavior to Usage Scenarios: An Interaction-Pattern Mining Approach. Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, July 23-26, 2002.

[15] T. Ne Win, M. D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report, (Cambridge, MA), 2002.

[16] N. Gupta. Generating Test Data for Dynamically Discovering Likely Program Invariants. ICSE 2003 Workshop on Dynamic Analysis co-located with International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA, May 3-10, 2003.

[17] M. D. Ernst. Dynamically Discovering Likely Program Invariants. Ph.D. Thesis, University of Washington, 2000.

# Chapter 4

# Upasana Pujari: Comparison of Formal Developments of Concurrent Programs

In this report we are comparing the formal development of concurrent programs in three different approaches - $Refinement$ approach, $Atomicity\ refinement$ approach and $Verification$ approach. The three approaches are being studied and compared with the help of an example in each approach.

## 4.1 Preliminaries

**Action.** An action $A$ is of the form

$$c \rightarrow B$$

where $c$ is the guard of the action and $B$ is the body of the action.

The body of the action is executed whenever the guard of the action is true. Unlike methods, actions are not called. The guard of the action A is also written as $gA$, so that $gA = g(c \rightarrow B) = c$ The body of the action A is also written as $sA$, so that $sA = s(c \rightarrow B) = B$

Actions are atomic, i.e., when an action is executing, no other action in the system can interfere with its execution. Two or more actions can execute in parallel if there are no shared variables between them.

**Action System.** An action system $S$ with global variables $z$ is defined as follows

$$S \quad = \quad \textbf{var}\ x; x := x_0; \textbf{do}\ A_1 [\!] ... [\!] A_m\ \textbf{od}$$

where,

$x$ denotes the local variables of the action system.

$x := x_0$ denotes the initialization of the local variables $x$

$x_0$ is the initial value of $x$.

$A_i, i \in \{1, ..., m\}$ denotes the actions of the action system.

Each action $A_i$ is of the form $A_i = gA_i \rightarrow sA_i$

In the definition of action system, the symbol $[\!]$ denotes $non - deterministic\ choice$. It means that when two or more actions in an action system are ready to execute (their guards evaluate to true), then the action to be executed is chosen in a non-deterministic manner.

Since actions in an action system can be executed in parallel, action systems can be used to model concurrent programs. $Refinement$ [1], $Atomicity\ refinement$ [2] and $Verification$ [3] are three different approaches to developing concurrent programs. This is a report of the results of working out an example in each of the three approaches and studying the similarities and differences.

## 4.2 Refinement approach

In the refinement approach [1], data refinement method is applied for refinement of action systems. Data refinement is a technique using which one can change the state space in a refinement step. In this approach an action system is refined by introducing *stuttering actions*. The stuttering actions in the refined action system are those actions that do not correspond to any actions in the original action system.

**Data refinement of action systems(permitting stuttering) 1** *Let $S$ be an action system on the program variables $x$ and $S'$ be an action system on the program variables $x'$ given as follows:*

$$\begin{aligned} S &= \textbf{var } x := x_0; \textbf{ do } A \textbf{ od} \\ S' &= \textbf{var } x' := x'_0; \textbf{ do } A' [\!] H' \textbf{ od} \end{aligned}$$

*where $A$ denotes the actions in the action system $S$ and $A'$ denotes the actions in the action system $S'$.*

*Here $S$ is the original action system and $S'$ is the proposed data refined action system. In the action system $S'$, the actions $H'$ are the stuttering actions. The abstraction relation $R(x, x', z)$ is a relation between the local variables of $S$ and $S'$ and the global variables $z$.*

*The action system $S$ is data refined (with stuttering actions) by $S'$ using $R$ if the following rules hold:*

- *Initialization : The abstraction relation is established by the initialization of the local variables and under any initial values of the global variables $z$.*

  $$R(x_0, x'_0, z),$$

- *Main action : This rule states that the action $A'$ is data refined by the action $A$ using the abstraction relation $R$.*

  $$A \sqsubseteq_R A',$$

  *In other words, the guard of the actions $A'$ in the refined action system, implies the guard of the corresponding actions $A$ in the original action system i.e. the guard is strengthened. Also, in the refined action system, the effect of the action $A'$ on the global variables $z$ is the same as the effect of the corresponding action $A$ in the original action system.*

- *Exit conditions : This rule states that under the abstraction relation $R$, if in the original action system $S$ the action $A$ is enabled, then in the refined action system $S'$ either the main action $A'$ or the stuttering action $H'$ is enabled.*

  $$R \wedge gA \Rightarrow gA' \vee gH',$$

- *Auxiliary actions : This rule states that the stuttering actions act as skip statements on the global variables $z$ in the original action system. In other words, the stuttering actions do not correspond to any actions in the original action system.*

  $$skip \sqsubseteq_R H',$$

- *Internal convergence : This rule states that the stuttering actions on their own should terminate, otherwise it will lead to internal divergence in the refined action system.*

  $$R \;=\; wp(do\ H'\ od, true),$$

  *The fact that an action $M$ terminates, is written as*

  $$p = wp(M, true),$$

  *where $p$ is the weakest precondition for the action $M$ to terminate. Hence $R = wp(do\ H'\ od, true)$ denotes that the stuttering actions on their own terminate.*

- *Non-interference : This rule is used when the action system $S$ is in parallel composition with another action system whose actions are denoted by $B$. It ensures that the interleaved execution of the actions $B$ do not interfere with the abstraction relation.*

  $$R \wedge wp(B, true) \Rightarrow wp(B, R),$$

  *In other words, while the abstraction relation $R$ holds, if the actions $B$ terminate, then the actions $B$ also preserve the abstraction relation $R$.*

*When all of these above rules hold, then the action system $S$ is said to be data refined (with stuttering) by the action system $S'$, written as*

$$S \sqsubseteq_R S'.$$

The correctness being preserved by such refinement steps is total correctness.

## Refinement approach - Example

$MS_0$ is the action system that is to be refined using Peterson's mutual exclusion algorithm. [1]

$$
\begin{aligned}
MS_0 \;=\;& \textbf{var}\ y.i \;:\; Int,\ cr.i \;:\; Bool,\ i \in 0,1;\\
& (cr.i := false,\ i \in 0,1);\\
& \textbf{do}\\
& (\llbracket\ cr.i \rightarrow y.i := w + i + 1; w := y.i; cr.i := false;\ i \in 0,1) \quad [CS.i]\\
& (\llbracket\ \neg cr.i \rightarrow N.i;\ i \in 0,1) \qquad\qquad\qquad\qquad\qquad\qquad [NS.i]\\
& \textbf{od}
\end{aligned}
$$

The naming convention for the actions are as follows: CS.i : Critical Section, NS.i : Non-critical Section

In this action system, $w$ is a global variable that is updated by both the actions $CS.0$ and $CS.1$. When the actions $CS.0$ and $CS.1$ execute ideally the sequence of updates should be $y.0 := w + 1; w := y.0; y.1 := w + 2; w := y.1;$

However, when the actions $CS.0$ and $CS.1$ execute in parallel the following sequence of updates is possible : $y.0 := w + 1; y.1 := w + 2; w := y.1; w := y.0;$ Here the value of $w$ as computed by $CS.1$ is overwritten by the value computed by $CS.0$.

This is an undesirable result that can be prevented by introducing mutual exclusion. Peterson's mutual exclusion algorithm is applied to refine this action system so that such update sequences do not occur in the refined action system.The action $NS.i$ does not access $w$ or $y.i$ for $i = 0, 1$.
Rewriting the action system $MS_0$, we have :

$$
\begin{aligned}
MS_0 \;=\;& \textbf{var}\ cr.i \;:\; Bool,\ i \in 0,1;\\
& (cr.i := false,\ i \in 0,1);\\
& (CS_0 \llbracket NS_0)
\end{aligned}
$$

where

$$
\begin{aligned}
CS_0 \;=\;& \textbf{var}\ y.i \;:\; Int,\ i \in 0,1;\\
& \textbf{do}\\
& (\llbracket\ cr.i \rightarrow y.i := w + i + 1; w := y.i; cr.i := false;\ i \in 0,1) \qquad [CS.i]\\
& \textbf{od}
\end{aligned}
$$

and

$$
\begin{aligned}
NS_0 \;=\;& \\
& \textbf{do}\\
& (\llbracket\ \neg cr.i \rightarrow N.i;\ i \in 0,1) \qquad [NS.i]\\
& \textbf{od}
\end{aligned}
$$

Now as a refinement of $CS_0$, we have

$$
\begin{aligned}
CS_1 \;=\;& \textbf{var}\ b.i\ :\ Bool,\ pc.i, y.i\ :\ Int,\ i \in 0,1;\ t : 0...1; \\
& (b.i, pc.i := false, 0,\ i \in 0,1); \\
& (t := 0 \wedge t := 1); \\
& \textbf{do} \\
& ([\!] \ cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1;\ i \in 0,1) \quad [BS.i] \\
& ([\!] \ pc.i = 1 \rightarrow t := i; pc.i := 2;\ i \in 0,1) \qquad [TS.i] \\
& ([\!] \ pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1) \rightarrow y.i := w + i + 1; \\
& \qquad\qquad w := y.i; cr.i := false; pc.i := 3;\ i \in 0,1) \quad [CS'.i] \\
& ([\!] \ pc.i = 3 \rightarrow pc.i := 0; b.i := false;\ i \in 0,1) \qquad [BR.i] \\
& \textbf{od}
\end{aligned}
$$

The naming convention for the actions are as follows: BS.i : Section for setting variable b, TS.i : Section for setting variable t, BR.i : Section for restoring variable b
The abstraction relation is the invariant here. It is defined as follows:

| pc.i | b.i | cr.i | t |
|---|---|---|---|
| 0 | F | F,T | 0,1 |
| 1 | T | T | 0,1 |
| 2 | T | T | 0,1 |
| 3 | T | F,T | 0,1 |

Now we show the correctness of this refinement step by observing that the six rules of refinement outlined in the earlier section hold under this refinement.

- *Initialization* : As can be seen from the example, the initialization establishes the invariant.

- *Main action* : As observed from the abstraction relation $R$, when $pc.i = 2$ then $cr.i = T$, so the guard of $CS'.i$ implies the guard of $CS.i$. Also the effect of $CS'.i$ is the same as that of $CS.i$ on global variables. The invariant (abstraction relation) is also preserved. Therefore the action $CS.i$ is data refined by the action $CS'.i$

- *Exit conditions* : Assume that $cr.0 = T$. Then if $pc.0 = 0$ then action $BS.0$ is enabled. If $pc.0 = 1$ then action $TS.0$ is enabled. If $pc.0 = 3$ then $BR.0$ is enabled. If $\neg b.1 \vee t = 1$ holds then $CS'.0$ is enabled. Otherwise assume $b.1 \wedge t = 0$ holds. Then from the invariant, $pc.1 \neq 0$. If $pc.1 = 1$ then $TS.1$ is enabled else if $pc.1 = 3$ then $BR.1$ is enabled else if $pc.1 = 2$ then $CS'.1$ is enabled as $t = 0$.

  So in every case when a action from $MS_0$ is enabled, then either a main action or stuttering action from $MS_1$ is enabled. In $MS_1$, $BS.i$, $TS.i$ and $BR.i$ are the stuttering actions.

- *Auxiliary actions* : The actions $BS.i$, $TS.i$ and $BR.i$ all effect only the new variables and preserve the invariant. Hence all these actions refine a skip statement in the original action system.

- *Internal convergence* : Execution of the auxiliary actions alone terminates in a state where $pc.i$ is set to 2. So such an execution does not diverge.
- *Non-interference* : We need to show that the execution of $CS_0$ in the context of $NS_0$ preserves the invariant. The action $NS.i$ can only be enabled when $cr.i = F$, i.e. when $pc.i = 0$ or $pc.i = 3$. As $pc.i$, $b.i$ and $t$ are local to $CS.i$ and are not changed by $NS.i$, so the invariant is preserved if and when NS.i does terminate.

Since all of the above rules hold, $CS_1$ is a data refinement of $CS_0$.

Finally, we refine the action $CS'.i$ into the actions $CAS.i$, $CBS.i$ and $CCS.i$

$$
\begin{array}{lll}
CS_2 & = & \textbf{var}\ b.i\ :\ Bool,\ pc.i, y.i\ :\ Int,\ i \in 0,1; t : 0...1; \\
 & & (b.i, pc.i := false, 0,\ i \in 0,1); \\
 & & (t := 0 \wedge t := 1); \\
 & & \textbf{do} \\
 & & (\llbracket\ cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1;\ i \in 0,1) \qquad [BS'.i] \\
 & & (\llbracket\ pc.i = 1 \rightarrow t := i; pc.i := 2;\ i \in 0,1) \qquad [TS'.i] \\
 & & (\llbracket\ pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1) \rightarrow pc.i := 3;\ i \in 0,1) \quad [CAS.i] \\
 & & (\llbracket\ pc.i = 3 \rightarrow y.i := w + i + 1; pc.i := 4;\ i \in 0,1) \qquad [CBS.i] \\
 & & (\llbracket\ pc.i = 4 \rightarrow w := y.i; cr.i := false; pc.i := 5;\ i \in 0,1) \quad [CCS.i] \\
 & & (\llbracket\ pc.i = 5 \rightarrow pc.i := 0; b.i := false;\ i \in 0,1) \qquad [BR'.i] \\
 & & \textbf{od}
\end{array}
$$

Now we show the correctness of this refinement step by observing that the six rules of refinement outlined in the earlier section hold under this refinement.

In addition to the abstraction relation from the previous step, the following relation between $pc.i$ in $CS_1$ and $pc.i$ in $CS_2$ (denoted by $pc'.i$ in order to distinguish between the two)is defined as follows:

| $pc.i$ | $pc'.i$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 2 | 4 |
| 3 | 5 |

The invariant of the resulting action system is as follows:

$$
\begin{array}{lll}
Inv.i & = & pc'.i = 1 \vee pc'.i = 2 \Rightarrow b.i\ \wedge \\
 & & pc'.i = 3 \Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc'.j = 1)\ \wedge \\
 & & pc'.i = 4 \vee pc'.i = 5 \Rightarrow b.i \wedge (\neg b.j \vee t = j \vee pc'.j = 1) \\
 & & \qquad \wedge y.i = w + i + 1
\end{array}
$$

Each action in the refined action system should preserve this invariant $Inv.0 \wedge Inv.1$

Consider the invariant $Inv.0$ first. $NS.0$ cannot change $Inv.0$ because it cannot change the only global variable $w$ in $Inv.0$ From the abstraction relation and from the relation between $pc.i$ and $pc'.i$ we observe that action $CAS.0$ establishes the invariant, and actions $CBS.0$ and $CCS.0$ preserve the invariant. Also the actions $BS'.0$, $TS'.0$ and $BR'.0$ preserve the invariant. From the other process, actions $BS'.1$ and $TS'.1$ also preserve the invariant. Action $BR'.1$ establishes the disjunction in $Inv.0$ by setting $b.1$ to false. Actions $CAS.1$, $CBS.1$ and $CCS.1$ can only be enabled when $pc'.0 \neq 3, 4, 5$ and $pc'.1 = 0, 1$. In all these cases these actions preserve the invariant $Inv.0$. Similarly, each action can be shown to preserve $Inv.1$ Thus, each action in the action system preserves the invariant $Inv.0 \wedge Inv.1$

- *Initialization* : The initialization establishes the abstraction relation and the invariant.
- *Main action* : Taking into account the abstraction relation $R$ and the relation between $pc.i$ and $pc'.i$, the effect of the main actions $BS'.i$, $TS'.i$, and $BR'.i$ for $i = 0, 1$ on the global variables is unchanged from that of the original actions. Since $y.i = w + i + 1$ holds before execution of $CCS.i$, so $CCS.i$ also satisfies this rule. All these actions preserve the invariant too. Again, taking into account the abstraction relation $R$ and the relation between $pc.i$ and $pc'.i$, the guards of $BS'.i$, $TS'.i$, and $BR'.i$ imply the guards of $BS.i$, $TS.i$, and $BR.i$ respectively.
- *Exit conditions* : We only have to show that if $pc.i = 2 \wedge (\neg b.(1 - i) \vee t = 1 - i)$ then one of $CAS.i$, $CBS.i$ or $CCS.i$ is enabled in the refined action system. From the relation between $pc.i$ and $pc'.i$ we conclude that when $pc.i = 2$ then $pc'.i = 2, 3, 4$ and hence one of $CAS.i$, $CBS.i$, $CCS.i$ is enabled.
- *Auxiliary actions* : The actions $CAS.i$ and $CBS.i$ do not change any of the global variables of $CS_2$ and they preserve the invariant. Hence $CAS.i$ and $CBS.i$ refine skip actions.
- *Internal convergence* : Execution of the auxiliary actions alone does terminate.
- *Non-interference* : The only global variable in the invariant and in the relation between $pc.i$ and $pc'.i$ is $w$. From the initial problem statement, $NS.i$ does not access $w$. Hence the rule of non-interference is satisfied.

Since all of the above rules hold, $CS_2$ is a data refinement of $CS_1$.
Therefore, the action system

$$\begin{aligned} MS_1 \;=\; & \textbf{var}\ cr.i \;:\; Bool,\ i \in 0, 1; \\ & (cr.i := false,\ i \in 0, 1); \\ & (CS_2 [\!] \; NS_0) \end{aligned}$$

is a correct refinement of the action system $MS_0$.

With an example of this complexity, there can be many more refinement steps, each successively refining the action system. Here we have shown two important refinement steps which successfully demonstrate the technique of the refinement approach.

## 4.3 Atomicity Refinement Approach

In the atomicity refinement approach [2], the action system is refined by splitting up a main action into several constituent actions. Also, the grain of interleaving of actions in the action system is increased by executing the new constituent actions in parallel with the rest of the main actions in the action system while preserving the overall behavior of the original action system.
Let $S$ be an action system as follows:

$$S \quad = \quad \textbf{var } y; y := y_0; \textbf{ do } A [\!] B_1 [\!] ... [\!] B_n \textbf{ od}$$

Here, $A$ is a very big action and it can be split up into a number of smaller actions by refining it as follows:

$$A \quad = \quad gA \rightarrow P$$

where

$$P \quad = \quad \textbf{var } x; x := x_0; \textbf{ do } A_1 [\!] ... [\!] A_m \textbf{ od}$$

Finally, the original action system $S$ is refined as follows :

$$S \quad = \quad \textbf{var } x, y; y := y_0; \textbf{ do } A_0 [\!] A_1 [\!] ... [\!] A_m [\!] B_1 [\!] ... [\!] B_n \textbf{ od}$$

where

$$A_0 \quad = \quad gA_0 \rightarrow x := x_0$$

In order that the refined action system preserves the correctness of the original action system, certain conditions are to be satisfied. These conditions are outlined in the Atomicity Refinement Theorem.

Before defining the rules/conditions for atomicity refinement, let us consider some related concepts: Let $A$ and $B$ be two actions where $gA$ is the guard of action $A$ and $sA$ is the body of action $A$.

- *Enabledness*
  1. $A$ cannot enable $B$ if $\neg gB \sqsubseteq wp(A, \neg gB)$
  2. $A$ cannot disable $B$ if $gB \sqsubseteq wp(A, gB)$
- *Commutativity* $A$ commutes (right) with $B$, written as $A; B \sqsubseteq B; A$ if
  1. $B$ cannot enable $A$
  2. $A$ cannot disable $B$ and
  3. $\{gA \wedge gB\};\ sA; sB \sqsubseteq sB; sA$

If $A$ can disable $B$, then it leads to the sequence of actions $A; B$ being disabled. No action or action sequence should be able to refine such a disabled action sequence. Therefore, $A$ should not be able to disable $B$.

If originally $A$ is disabled, then the sequence of actions $A; B$ is disabled too. However, if $B$ can enable $A$, then the sequence of actions $B; A$ on the right-hand-side is enabled. No action or action sequence should be able to refine the disabled action sequence $A; B$. Therefore, $B$ should not be able to enable $A$

- Exclusion
  1. $A$ is excluded by $B$ if $gA \wedge gB = false$,
  2. $A$ cannot be followed by $B$ if $wp(A, \neg gB) = true$ (So $A$ establishes as postcondition that guard of $B$ is false)

## Atomicity Refinement Theorem

Let $S$ be an action system such that

$$S \;=\; I; \textbf{do } A[\!][\!]L[\!][\!]R[\!][\!]E \textbf{ od}$$

where

$$A \;=\; A0; \textbf{do } A1 \textbf{ od}$$

Here $I$ is the initialization of the action system $S$. $L$, $R$, $E$ are the outer actions and $A0$, $A1$ are the inner actions (belonging to $A$ - the inner loop). $A0$ is the initialization of the inner loop actions.
Let $S'$ be an action system defined as :

$$S' \;=\; I; \textbf{do } A0[\!][\!]A1[\!][\!]L[\!][\!]R[\!][\!]E \textbf{ od}$$

In $S'$, the actions of both the nested loops of $S$ are flattened out so that all these actions can be executed in an interleaved manner - in other words the atomicity of actions are refined in $S'$.

For $S'$ to be an atomicity refinement of $S$, the action system $S'$ should not allow execution sequences that are not possible in the action system $S$. Therefore there are rules defined to guide a proper atomicity refinement of an action system.

**Atomicity refinement theorem 1** *The action system $S'$ is an atomicity refinement of the action system $S$, written as $S \sqsubseteq S'$, if the following conditions hold :*

- *Enabledness*
  1. *an outer action cannot enable the inner action A1,*
     *This condition prevents the inner loop from being started in the middle bypassing the initialization A0.*

2. *an outer action cannot disable the inner action A1,*

   *This condition prevents the inner loop from being prematurely terminated by an interleaved outer action.*

3. *the initialization does not enable the inner action A1.*

   *This condition prevents the execution from starting with a jump into the inner loop bypassing the initialization A0.*

- *Exclusion*

1. *the inner actions A0 and A1 exclude each other,*

   *This condition prevents a new execution of the inner loop from being started before the previous loop execution has terminated.*

2. *the outer action E (excluded action) is excluded by the inner action A1.*

   *This condition prevents E from being enabled while the inner loop is being executed.*

- *Commutativity*

1. *L commutes with both inner actions A0 and A1,*

   *This condition states that in the execution of S', the order of actions can be changed so that L (the Left mover) moves left over an inner action A0, A1.*

2. *L commutes with R,*

   *This condition states that in the execution of S', the order of actions can be changed so that L (the Left mover) moves left over the action R (the Right mover).*

3. *A1 commutes with R,*

   *This condition states that in the execution of S', the order of actions can be changed so that R (the Right mover) moves right over an inner action A1.*

   *The effect of the above commutativity conditions is that it is possible to collect all the inner actions into a contiguous sequence of inner loop actions A1, with an inner initialization action A0.*

4. *do R od always terminates. This condition states that the sequence of R (right mover actions) should terminate, thus preventing internal divergence from being introduced in the atomicity refinement step.*

## Atomicity Refinement approach - Example

$MS_0$ is the action system that is to be refined using Peterson's mutual exclusion algorithm. [1]

$$
\begin{array}{lll}
MS_0 & = & \textbf{var}\ y.i\ :\ Int,\ cr.i\ :\ Bool,\ i \in 0,1; \\
 & & (cr.i := false,\ i \in 0,1); \\
 & & \textbf{do} \\
 & & (\llbracket\ cr.i \rightarrow y.i := w + i + 1; w := y.i; cr.i := false;\ i \in 0,1) \quad [CS.i] \\
 & & (\llbracket\ \neg cr.i \rightarrow N.i;\ i \in 0,1) \qquad\qquad\qquad\qquad\qquad\qquad [NS.i] \\
 & & \textbf{od}
\end{array}
$$

The naming convention for the actions are as follows: CS.i : Critical Section, NS.i : Non-critical Section

In this action system, $w$ is a global variable that is updated by both the actions $CS.0$ and $CS.1$ When the actions $CS.0$ and $CS.1$ execute ideally the sequence of updates should be $y.0 := w + 1; w := y.0; y.1 := w + 2; w := y.1;$

However, when the actions $CS.0$ and $CS.1$ execute in parallel the following sequence of updates is possible : $y.0 := w + 1; y.1 := w + 2; w := y.1; w := y.0;$ Here the value of $w$ as computed by $CS.1$ is overwritten by the value computed by $CS.0$.

This is an undesirable result that can be prevented by introducing mutual exclusion. Peterson's mutual exclusion algorithm is applied to refine this action system so that such update sequences do not occur in the refined action system. The action $NS.i$ does not access $w$ or $y.i$ for $i = 0, 1$.
Rewriting the action system $MS_0$ :

$$
\begin{array}{lll}
MS_0 & = & \textbf{var}\ cr.i\ :\ Bool,\ i \in 0,1; \\
 & & (cr.i := false,\ i \in 0,1); \\
 & & (CS_0 \llbracket NS_0)
\end{array}
$$

where

$$
\begin{array}{lll}
CS_0 & = & \textbf{var}\ y.i\ :\ Int,\ i \in 0,1; \\
 & & \textbf{do} \\
 & & (\llbracket\ cr.i \rightarrow y.i := w + i + 1; w := y.i; cr.i := false;\ i \in 0,1) \qquad [CS.i] \\
 & & \textbf{od}
\end{array}
$$

and

$$
\begin{array}{lll}
NS_0 & = & \\
 & & \textbf{do} \\
 & & (\llbracket\ \neg cr.i \rightarrow N.i;\ i \in 0,1) \qquad [NS.i] \\
 & & \textbf{od}
\end{array}
$$

Now as a refinement of the $CS_0$, we introduce the concept of mutual exclusion in an inner action system $ME.i$ (where $ME.i = ME0.i; doME1.iod$). By atomicity refinement $ME.i$ is flattened out into $ME0.i$ and $ME1.i$ in the refined action system $CS_1$.

$$
\begin{aligned}
CS_1 \;=\;& \textbf{var}\ b.i\ :\ Bool,\ pc.i, c.i, y.i\ :\ Int,\ i \in 0,1; t : 0...1; \\
& (b.i, pc.i, c.i := false, 0, 0,\ i \in 0,1); \\
& (t := 0 \wedge t := 1); \\
& \textbf{do} \\
& ([\!] \ cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1;\ i \in 0,1) \quad [ME0.i] \\
& ([\!] \ pc.i = 1 \rightarrow t := i; pc.i := 2;\ i \in 0,1) \qquad\qquad [ME1.i] \\
& ([\!] \ pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1) \rightarrow y.i := w + i + 1; w := y.i; \\
& \qquad\qquad cr.i := false; pc.i := 3;\ i \in 0,1) \qquad\qquad [CS'.i] \\
& ([\!] \ pc.i = 3 \rightarrow pc.i := 0; b.i := false;\ i \in 0,1) \qquad [BR.i] \\
& \textbf{od}
\end{aligned}
$$

The naming convention for the actions are as follows: ME0.i : first part of Mutual Exclusion section, ME1.i : second part of Mutual Exclusion section, BR.i : Section for restoring variable b
Now we show the correctness of the atomicity refinement step as follows:

- Enabledness

  1. an outer action $CS'.i$, $BR.i$, $NS.i$ cannot enable the inner action $ME1.i$,
     Since $NS.i$ does not have access to $pc.i$, it does not enable $ME1.i$.
     We can show that $BR.i$ does nor enable $ME1.i$ if $(\neg gME1.i) \sqsubseteq wp(BR.i, \neg gME1.i)$

$$
\begin{aligned}
& (\neg gME1.i) \sqsubseteq wp(BR.i, \neg gME1.i) \\
\equiv\ & pc.i \neq 1 \Rightarrow wp(pc.i = 3 \rightarrow pc.i = 0; b.i = false, pc.i \neq 1) \\
\equiv\ & pc.i \neq 1 \Rightarrow (pc.i = 3 \Rightarrow wp(pc.i = 0; b.i = false, pc.i \neq 1)) \\
\equiv\ & pc.i \neq 1 \Rightarrow (pc.i = 3 \Rightarrow wp(pc.i = 0, pc.i \neq 1)) \\
\equiv\ & pc.i \neq 1 \Rightarrow (pc.i = 3 \Rightarrow 0 \neq 1) \\
\equiv\ & pc.i \neq 1 \Rightarrow (pc.i = 3 \Rightarrow true) \\
\equiv\ & true
\end{aligned}
$$

Hence $BR.i$ does not enable $ME1.i$.

Similarly, it can be shown that $CS'.i$ does not enable the inner action $ME1.i$. Therefore the enabledness condition is satisfied.

2. an outer action $CS'.i$, $BR.i$, $NS.i$ cannot disable the inner action $ME1.i$,

   Since $NS.i$ does not have access to $pc.i$, it does not disable $ME1.i$.

   We can show that $BR.i$ does nor disable $ME1.i$ if $(gME1.i) \sqsubseteq wp(BR.i, gME1.i)$

$$
\begin{aligned}
& (gME1.i) \sqsubseteq wp(BR.i, gME1.i) \\
\equiv\ & pc.i = 1 \Rightarrow wp(pc.i = 3 \rightarrow pc.i = 0; b.i = false, pc.i = 1) \\
\equiv\ & pc.i = 1 \Rightarrow (pc.i = 3 \Rightarrow wp(pc.i = 0; b.i = false, pc.i = 1)) \\
\equiv\ & pc.i = 1 \Rightarrow (pc.i = 3 \Rightarrow wp(pc.i = 0, pc.i = 1)) \\
\equiv\ & pc.i = 1 \Rightarrow (pc.i = 3 \Rightarrow 0 = 1) \\
\equiv\ & pc.i = 1 \Rightarrow (pc.i = 3 \Rightarrow false) \\
\equiv\ & (pc.i = 1 \wedge pc.i = 3) \Rightarrow false \\
\equiv\ & false \Rightarrow false \\
\equiv\ & true
\end{aligned}
$$

   Hence $BR.i$ does not disable $ME1.i$.

   Similarly, it can be shown that $CS'.i$ does not disable the inner action $ME1.i$. Therefore the enabledness condition is satisfied.

3. the initialization does not enable the inner action $ME1.i$.

   The initialization sets $pc.i = 0$. So it clearly does not enable the inner action $ME1.i$. (It can be shown formally as in the above two conditions).

- Exclusion

1. the inner actions $ME0.i$ and $ME1.i$ exclude each other,

$$
\begin{aligned}
gME0.i &\equiv cr.i \wedge pc.i = 0 \\
gME1.i &\equiv pc.i = 1
\end{aligned}
$$

So, gME0.i $\wedge$ gME1.i $\equiv$ false

Hence the exclusion condition holds.

2. the outer action $E$ (excluded action) is excluded by the inner action $A1$.

   This condition prevents $E$ from being enabled while the inner loop is being executed.

   In this case $NS.i$ is the excluded action, as while the inner loop ($ME0.i$, $ME1.i$) is being executed, then $NS.i$ cannot be enabled. $NS.i$ is enabled when $cr.i = false$. Both $ME0.i$ and $ME1.i$ cannot enable $NS.i$ as they cannot have a pre or postcondition where $cr.i = false$. Hence the outer action $NS.i$ is excluded by the inner action $ME1.i$.

- Commutativity

1. There are no left mover actions in this action system.
2. There are no left mover actions in this action system.
3. $ME1.i$ commutes with $CS'.i$ and $BR.i$ (the right movers),

   We have already shown in 1(a) that $CS'.i$ and $BR.i$ cannot enable $ME1.i$. Also $ME1.i$ cannot disable $CS'.i$ as $ME1.i$ actually sets $pc.i = 2$. Depending on the values of $b.(1-i)$ and $t$, the value $pc.i = 2$ may enable the action $CS'.i$.

   At $ME1.i$ the value of $pc.i$ is 1. So $BR.i$ is already disabled. Hence $BR.i$ cannot be disabled by $ME1.i$.

   To check that $\{gME1.i \wedge gCS'.i\}; sME1.i; sCS'.i \sqsubseteq sCS'.i; sME1.i$

   $$\begin{aligned} &\equiv \{pc.i = 1 \wedge pc.i = 2 \wedge (\neg b(1-i) \vee t = 1-i)\}; sME1.i; sCS'.i \\ &\quad \sqsubseteq sCS'.i; sME1.i \\ \\ &\equiv \{false\}; sME1.i; sCS'.i \sqsubseteq sCS'.i; sME1.i \end{aligned}$$

   So $ME1.i$ commutes with $CS'.i$. Similarly it can be shown that $ME1.i$ commutes with $BR.i$.

4. This condition states that the sequence of right mover actions should terminate, thus preventing internal divergence from being introduced in the atomicity refinement step.

   When the right mover actions ($CS'.i$ and $BR.i$) are executed they eventually terminate in a state with $pc.i = 0$ and $cr.i = false$. Hence the actions in the ith process cannot execute all over again (till $cr.i$ is equal to false again).

   So internal divergence is not introduced in the atomicity refinement step by the right mover actions.

Since all of the above conditions hold, $CS_1$ is a proper atomicity refinement of $CS_0$.

Finally, we refine the action $CS'.i$ into the actions $CAS.i$, $CBS.i$ and $CCS.i$.

$$
\begin{array}{lll}
CS_2 & = & \textbf{var}\ b.i\ :\ Bool,\ pc.i, y.i\ :\ Int,\ i \in 0,1; t : 0...1; \\
 & & (b.i, pc.i := false, 0,\ i \in 0,1); \\
 & & (t := 0 \wedge t := 1); \\
 & & \textbf{do} \\
 & & (\llbracket\ cr.i \wedge pc.i = 0 \rightarrow b.i := true; pc.i := 1;\ i \in 0,1) \qquad [ME0.i] \\
 & & (\llbracket\ pc.i = 1 \rightarrow t := i; pc.i := 2;\ i \in 0,1) \qquad [ME1.i] \\
 & & (\llbracket\ pc.i = 2 \wedge (\neg b.(1-i) \vee t = 1) \rightarrow pc.i := 3;\ i \in 0,1) \quad [CAS.i] \\
 & & (\llbracket\ pc.i = 3 \rightarrow y.i := w + i + 1; pc.i := 4;\ i \in 0,1) \qquad [CBS.i] \\
 & & (\llbracket\ pc.i = 4 \rightarrow w := y.i; cr.i := false; pc.i := 5;\ i \in 0,1) \quad [CCS.i] \\
 & & (\llbracket\ pc.i = 5 \rightarrow pc.i := 0; b.i := false;\ i \in 0,1) \qquad [BR.i] \\
 & & \textbf{od}
\end{array}
$$

Now we show the correctness of this refinement step by observing that the conditions outlined for atomicity refinement also hold under this refinement step.

- Enabledness

  1. an outer actions $CAS.i$, $CBS.i$, $CCS.i$, $BR.i$ cannot enable the inner action $ME1.i$.

     Since each of the actions $CAS.i$, $CBS.i$, $CCS.i$ do not change $pc.i$ in such a way that the inner action $ME1.i$ is enabled, so each of these actions does not enable the inner action $ME1.i$. It can also be proved formally using the condition for enabledness (as done in 1(a)in the previous refinement step). Again working along the lines of 1(a) in the previos refinement step, it can be proved that $BR.i$ cannot enable $ME1.i$. Therefore this enabledness condition is satisfied.

  2. an outer actions $CAS.i$, $CBS.i$, $CCS.i$ cannot disable the inner action $ME1.i$.

     Using the condition for disabledness, it can be formally proved (as done in 1(b) in the previous refinement step) that each of the actions $CAS.i$, $CBS.i$, $CCS.i$ and $BR.i$ do not disable the inner action $ME1.i$. Therefore this enabledness condition is satisfied.

  3. the initialization does not enable the inner action $ME1.i$.

     The initialization has not changed with this atomicity refinement step. Hence this condition has not changed with this atomicity refinement step.

- Exclusion

  1. the inner actions $ME0.i$ and $ME1.i$ exclude each other.

     The inner actions $ME0.i$ and $ME1.i$ have not changed with this atomicity refinement step. Hence the exclusion condition still holds.

2. The excluded action is excluded by the inner action $A1$.
   Again there is no change in this condition with this atomicity refinement step.

- Commutativity

1. There are no left mover actions in this action system.
2. There are no left mover actions in this action system.
3. $ME1.i$ commutes with $CAS.i$, $CBS.i$, $CCS.i$ and $BR.i$ (the right movers),
   We have already shown in 1(a) of this refinement step that $CAS.i$, $CBS.i$ and $CCS.i$ and $BR.i$ cannot enable $ME1.i$. Also $ME1.i$ cannot disable $CAS.i$, $CBS.i$ and $CCS.i$ as $ME1.i$ actually sets $pc.i = 2$. Depending on the values of $b.(1-i)$ and $t$, the value $pc.i = 2$ may enable the action $CAS.i$, and eventually enable the actions $CBS.i$ and $CCS.i$.
   At $ME1.i$ the value of $pc.i$ is 1. So $BR.i$ is already disabled. Hence $BR.i$ cannot be disabled by $ME1.i$.
   To check that $\{gME1.i \wedge gCAS.i\}; sME1.i; sCAS.i \sqsubseteq sCAS.i; sME1.i$

$$\begin{aligned} &\equiv \{pc.i = 1 \wedge pc.i = 2 \wedge (\neg b(1-i) \vee t = 1-i)\}; sME1.i; sCAS.i \\ &\quad \sqsubseteq sCAS.i; sME1.i \\ &\equiv \{false\}; sME1.i; sCAS.i \sqsubseteq sCAS.i; sME1.i \end{aligned}$$

   So $ME1.i$ commutes with $CAS.i$. Similarly it can be shown that $ME1.i$ commutes with $CBS.i$, $CCS.i$ and $BR.i$.
4. This condition states that the sequence of right mover actions should terminate, thus preventing internal divergence from being introduced in the atomicity refinement step.
   When the right mover actions ($CAS.i$, $CBS.i$, $CCS.i$ and $BR.i$) are executed they eventually terminate in a state with $pc.i = 0$ and $cr.i = false$. Hence the actions in the ith process cannot execute all over again (till $cr.i$ is equal to false again).
   So no internal divergence is introduced in the atomicity refinement step by the right mover actions.

Since all of the above conditions hold, $CS_2$ is a proper atomicity refinement of $CS_1$. Therefore, the action system

$$\begin{aligned} MS_1 \;=\; & \textbf{var}\ cr.i \,:\, Bool,\ i \in 0,1; \\ & (cr.i := false,\ i \in 0,1); \\ & (CS_2 [\!] NS_0) \end{aligned}$$

is a correct refinement of the action system $MS_0$.

With an example of this complexity, there can be many more atomicity refinement steps, each successively refining the action system. Here we have shown two important atomicity refinement steps which successfully demonstrate the technique of the atomicity refinement approach.

In the atomicity refinement approach, actions that refer to common variables cannot be executed in parallel. So with this approach, in a final refinement step, such shared (common) variables are removed by further splitting up one of the actions and introducing new variables to replace the common variables.

## 4.4 Verification approach

In the study of the verification approach, parallel programs with shared variables are taken into consideration. In this approach correctness of a concurrent program is proved by using assertions.
The following are two important rules in this approach.

**Parallelism with Shared Variables 1** *This rule states that,*

$$\frac{\textit{The standard proof outlines } \{pi\}\ Si^*\ \{qi\},\ i \in \{1,,n\} \textit{ are interference free}}{\{\bigwedge_{i=1}^{n}\ pi\}\ [S1 \| ... \| Sn]\ \{\bigwedge_{i=1}^{n}\ qi\}}$$

**Interference Freedom : Partial Correctness 1** *This rule states that,*

1. *Let $S$ be a component program. Consider a standard proof outline $\{p\}S^*\{q\}$ for partial correctness and a statement $A$ with the precondition $pre(A)$. We say that $A$ does not interfere with $\{p\}S^*\{q\}$ if for all assertions $r$ in $\{p\}S^*\{q\}$ the correctness formula*

   $\{r \wedge pre(A)\}A\{r\}$

   *holds in the sense of partial correctness.*

2. *Let $[S1 \| ... \| Sn]$ be a parallel program. Standard proof outlines $\{pi\}Si^*\{qi\}, i \in \{1..n\}$, for partial correctness are called interference free if no normal assignment or atomic region of a program $Si$ interferes with the proof outline $\{pj\}Sj^*\{qj\}$, of another program $Sj$ where $i \neq j$.*

The above rule is used for establishing partial correctness. However, this approach can be used to establish either total or partial correctness.

### Verification approach - Example

The task is to compute the global minimum of a function $f$ in a given interval. If the interval is specified by an array of values $A$, then the task is to compute the minimum value of the function $f$ over the values in the given array (of length $N$).

The task is subdivided into two components that can execute in parallel : $S_o$ and $S_e$. $S_o$ finds the global minimum value of the function using the values at odd numbered locations

in the array. $S_e$ finds the global minimum value of the function using the values at even numbered locations in the array.
$MS_0$ denotes the parallel program that performs the above task.

$$MS_0 \;=\; h := \infty; i := 0; j := 1; [S_o \| S_e]$$

where

$$\begin{aligned} S_e \;=\; & \textbf{do} \\ & (i < N) \rightarrow \textbf{if } (f(A[i]) < h) \textbf{ then } h := f(A[i]);\ i := i + 2; \\ & \textbf{od} \end{aligned}$$

and

$$\begin{aligned} S_o \;=\; & \textbf{do} \\ & (j < N) \rightarrow \textbf{if } (f(A[j]) < h) \textbf{ then } h := f(A[j]);\ j := j + 2; \\ & \textbf{od} \end{aligned}$$

In this example, $h$ is the shared variable between $S_o$ and $S_e$ $p1$ is the invariant for $S_e$ and $p2$ is the invariant for $S_o$. p1 and p2 are specified as follows:

$$p1 \equiv (0 \leq i \leq N) \wedge \forall k.((0 \leq k < i) \wedge even(k) \mid f(A[k]) \geq h)$$
$$p2 \equiv (1 \leq j \leq N) \wedge \forall l.((1 \leq l < j) \wedge odd(l) \mid f(A[l]) \geq h)$$

The assertions are :

$$a1 \equiv i \geq 0 \Rightarrow \forall k.((0 \leq k < i) \wedge even(k) \mid f(A[k]) \geq h)$$
$$a2 \equiv j \geq 1 \Rightarrow \forall l.((1 \leq l < j) \wedge odd(l) \mid f(A[l]) \geq h)$$

The standard proof outline for $S_e$ is as follows:

$\{p1\}$
**do**
$\{a1 \wedge a2\}$
$(i < N) \rightarrow$ **if** $(f(A[i]) < h)$ **then** $h := f(A[i]);\ i := i + 2;$
**od**
$\{p1 \wedge i \geq N\}$

and the standard proof outline for $S_o$ is as follows:

$\{p2\}$
**do**
$\{a1 \wedge a2\}$
$(j < N) \rightarrow$ **if** $(f(A[j]) < h)$ **then** $h := f(A[j]);\ j := j + 2;$
**od**
$\{p2 \wedge j \geq N\}$

Now, using rule for interference freedom, in order to show that the component $S_o$ does not interfere with the component $S_e$, it has to be shown that the following holds:

$\{assertion\ of\ S_e \wedge pre(S_o)\}\ body\ of\ S_o\ \{assertion\ of\ S_e\}$

However, $pre(S_o) \Rightarrow assertion\ of\ S_o$. So,

$\{assertion\ of\ S_e \wedge assertion\ of\ S_o\}\ body\ of\ S_o\ \{assertion\ of\ S_e\}$

In other words,

$\{a1 \wedge a2\}$ **if** $(f(A[j]) < h)$ **then** $h := f(A[j]); j := j + 2; \{a1 \wedge a2\}$

holds.

Now,

$$wp(\textbf{if } (f(A[j]) < h) \textbf{ then } h := f(A[j]);\ j := j + 2, a1 \wedge a2)$$

$$\equiv\ wp(\textbf{if } (f(A[j]) < h) \textbf{ then } h := f(A[j]), a1 \wedge a2[j \setminus j + 2])$$

$$(let\ a2[j \setminus j + 2] = a2')$$

$$\equiv\ wp(\textbf{if } (f(A[j]) < h) \textbf{ then } h := f(A[j]), a1 \wedge a2')$$

$$\equiv\ ((f(A[j]) < h) \wedge (a1 \wedge a2')[h \setminus f(A[j])]) \vee ((f(A[j]) \geq h) \wedge a1 \wedge a2')$$

$$(on\ simplification)$$

$$\equiv\ a1 \wedge a2$$

Therefore,

$\{assertion\ of\ S_e \wedge assertion\ of\ pre(S_o)\}\ body\ of\ S_o\ \{assertion\ of\ S_e\}$

holds.
Hence, the component $S_o$ does not interfere with the component $S_e$

Similarly it can be shown that the component $S_e$ does not interfere with the component $S_o$

Now applying the rule of parallelism with shared variables
$\{p1 \wedge p2\}[S_e \| S_o]\{p1 \wedge p2 \wedge i \geq N \wedge j \geq N\}$

Therefore, the following holds:

$\{N \geq 1\}h := \infty; i := 0; j := 1; [S_o \| S_e]\{p1 \wedge p2 \wedge i \geq N \wedge j \geq N\}$

Hence, using the verification approach the partial correctness of the system has been established.

## 4.5 Conclusion

All the three approaches studied here are based on the concept of parallel execution of atomic actions. In the Refinement approach example, mutual exclusion is introduced in the refinement step as stuttering actions. Then the rules of Data refinement (with stuttering actions) are applied to establish the total correctness of the refinement steps. In the Atomicity refinement approach example, mutual exclusion is introduced as inner actions. Then the conditions outlined in the Atomicity refinement theorem are checked in order to ascertain that the refinement steps preserve total correctness of the action systems. In this approach emphasis is on increasing the level of granularity of interleaving of the actions. For the Verification approach, the example used in the previous two approaches turned out to be too complex. So a simpler example of finding the global minimum of a function is used for the Verification approach. In this approach the given concurrent program is annotated with assertions, preconditions and postconditions. Then using the Interference Freedom rule for partial correctness, the partial correctness of the concurrent program is established. With the Verification approach, when required, the Interference Freedom rule for total correctness can be applied to check for total correctness of a concurrent program.

# Bibliography

[1] R. J. R Back. *Refinement of Parallel and Reactive Programs.* In Manfred Broy (Ed.), *Program Design Calculi, Series F: Computer and System Sciences, Vol 118, NATO ASI Series*, Springer-Verlag, pp 73-92, 1993

[2] R. J. R. Back. *Atomicity Refinement in a Refinement Calculus Framework.* Abo Akademi, *Reports on Computer Science & Mathematics, Ser. A. No 141,* 1993

[3] K. R. Apt, E-R Olderog. *Verification of Sequential and Concurrent Programs,* Springer-Verlag, 1997

# Chapter 5

# John Xu: Survey of Static Analysis Techniques and Tools

## 5.1 Introduction

Static analysis is a process that seeks to discover properties of software without code execution. Manual inspection is the simplest form of static analysis by definition. Its goal is to confirm that good coding practices have been used and the requirements have been complied. However, these manual tasks are often tedious and lack systematic process. Manual inspection is not adequate to give enough information about a program in particular; even it can still be useful in formalized quality assurance inspection procedures. In general, the term *static analysis* refers to an automated process.

Static analysis originates in the field of compiler optimization. Some classical examples are liveness analysis for register allocation and flow analysis for expression elimination. Many earlier research work covered such analysis techniques in compiler design [1, 33]. Static analysis is used in many areas of software development, such as automated documentation [35], program understanding [37, 20], testing [30, 31], reverse engineering [16], and recently automated model checking and program verification [3, 4].

This article surveys a subset of static analysis techniques and available tools. It covers techniques from traditional methods to recent topics in automated program verification, and tools from sophisticated systems used in safety-critical computing area, to some simpler analyzers. Static analysis is a very broad topic and it is impossible to cover every issue in one paper. In fact the methods and articles chosen here may not necessarily be the most important ones. Furthermore, this survey is not meant to be complete and does not provide comprehensive introduction to each topic presented. It is meant to give samples to an overall picture of the subject and a hint of underlying trends.

## 5.2 Static Analysis Techniques

Static analyses are often characterized by their behavior with respect to certain language features [17]. A method can process a combination of the following characteristics:

Procedural aspect—indicates whether operations are within a single subroutine (*intraprocedural analysis*) or a set of subroutines (*interprocedural analysis*);

Context sensitivity—indicates whether analysis is performed on each separate call to a subroutine (*context-sensitive*) or on each subroutine once, by approximating all inputs and outputs to that subroutine (*context-insensitive*);

Flow sensitivity—indicates whether analysis considers the order of statement execution (*flow-sensitive*) or assumes that statements can be executed in any order (*flow-insensitive*);

Path sensitivity—indicates whether analysis involves analyzing each path separately (*path-sensitive*) or merges the results from separate paths together (*path-insensitive*).

### Symbolic Execution and Abstract Interpretation

Symbolic execution [21] is a static analysis technique of simulating program execution by given symbolic inputs that approximates actual values. Program output is expressed in expressions using these symbols. At conditional branches, different inputs may cause the program to follow different execution paths. The abstract program state is split ("forked") into two identical copies, one of which assumes that the condition is true and the other assumes that the condition is false [17]. The remainder of the program is executed for each of these program states. The number of resulting program states may be exponential to the number of conditional branches.

Abstract interpretation [9] works like symbolic execution. It uses a symbolic representation of the values of program variables to perform static analysis. It is only an approximation of the semantics to a source program. Abstract interpretation differs from symbolic execution in the way of handling multiple program states. Abstract interpretation computes program states at every program point simultaneously, and performs program state merging operations at all control flow joins. This eliminates the danger of having very large number of program states at termination.

Abstract interpretation is a powerful analysis technique employed in automated model checking and program verification. A common framework in such an approach is shown in Figure 5.1. First a translator translates source code into an intermediate language understood by the analyzer. The semantics of the language is transformed to an abstract model keeping the necessary information. Analysis is then performed on this model by checking assertions. Abstract interpretation allows analysis to deduce more abstract semantic information. An initial abstract model may be too coarse for checking a property and often a more refined model is needed. This process could be repeated until an answer is obtained.

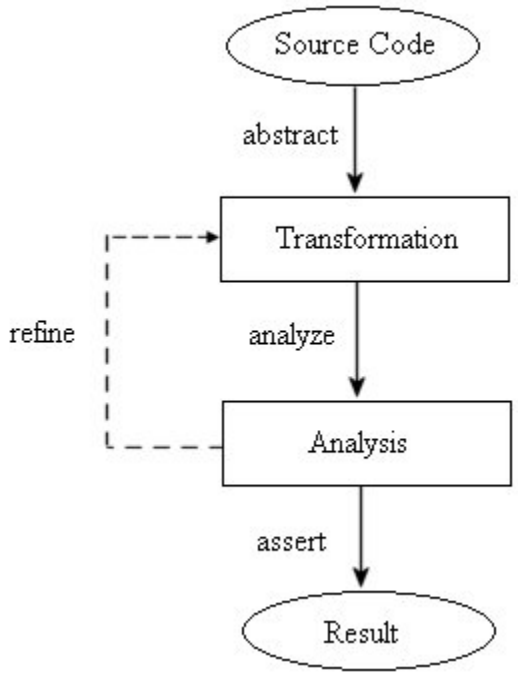



Figure 5.1: Static Analysis Process

A formal definition of abstraction interpretation using rigorous mathematical model is described in the article by Cousot [9]. This article shows that program properties obtained by an abstract interpretation are consistent with those obtained by a more refined interpretation. It further proved that abstract interpretation can be a safe approximation of programs. This property ensures that analysis on abstract model is meaningful.

The abstraction level of a language-level static analysis is determined by the abstraction level of the language. For example, a static analysis for an assembly language can derive information relative to machine registers and memory states, but can not deduce the types of parameters passed to a function, because this information is not available [19].

## Control Flow Graph

A directed graph is used to describe the control flow information of a program. It is a basis for performing many flow analyses. In a control flow graph, the nodes represent the basic blocks of a program and the edges represent the control flow paths. A basic block is a linear sequence of instructions having one entry point and one exit point. A path is a sequence of nodes obtained by successive applications of successor function.

Figure 5.2 shows the examples of control flow graphs for two pseudo programs. For an ALGOL-like program, a flow graph is constructed by translating the program into an intermediate language which does not contain nested statements. Certain sequences of instructions are grouped together as basic blocks to form a graph. The information on states is

available usually at compilation time. Codifying the flow relationships in the program may be in connectivity matrices, in predecessor-successor tables, or in dominance lists [2].

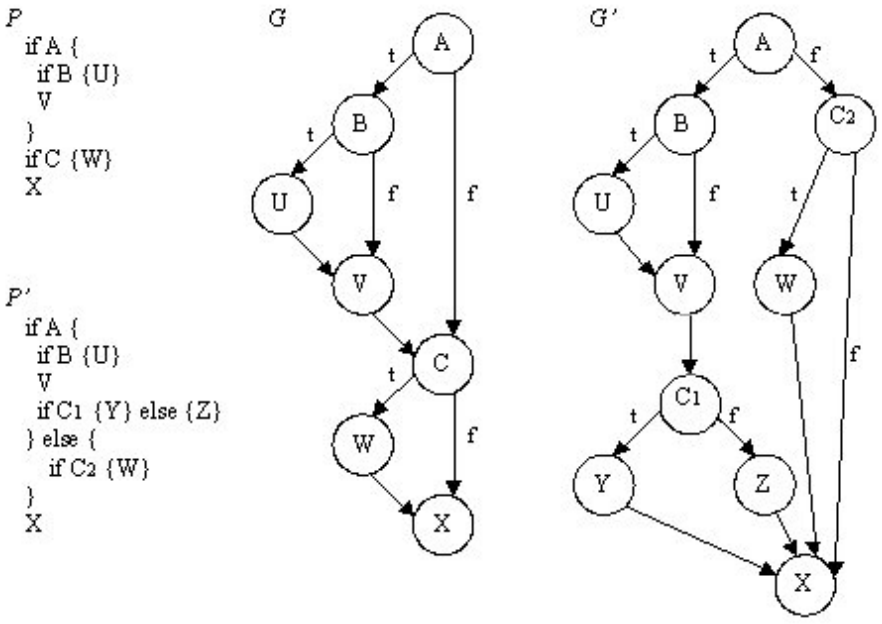



Figure 5.2: Control Flow Graphs

To obtain basic control flow graph is not straight-forward with Scheme-like languages. The problem with Lisp is that there is no static control flow graph at compile time. Consider the following fragment of Scheme code:

```
(let ((f (foe 7 g k))
(h (aref a7 i j)))
(if (< i j) (h 30) (f h)))
```

The control flow of the "if" depends on values of $f$ and $h$, but these values are only available at compile time or by performing data flow analysis. However to perform data flow analysis, it needs a control flow graph. A solution to this dilemma is to develop an intermediate representation for the Lisp programs, which is suitable for doing flow analysis. The solution is presented in [34].

## Flow Analysis

Control flow analysis is used to ensure a piece of code is executed in the right sequence, well structured and contains no syntactically unreachable code. One technique to perform control flow analysis was given by Allen [2] using intervals. This article gives a procedure for partitioning a graph into intervals. The interval construct described in this paper has many

properties which facilitate global analysis and which are of particular interest in optimization. The partial ordering relationships between nodes in an interval provide a natural processing order. Partitioning a graph into a hierarchy of intervals enables propagating information rapidly through the graph. The dominance relationships in a graph are easily discovered and nests of strongly connected regions can be detected.

The objective of data flow analysis is to show that no execution paths in the software exist that would access uninitialized variables. It uses the result of control flow analysis in conjunction with read/write access to variables. Data flow analysis can be a complex activity, as global variables can be accessed from anywhere. It can also detect anomalies such as multiple writes without intervening reads. Many variations of data flow analysis techniques exist including those were used in earlier compiler design [1, 33].

Information flow analysis identifies how execution of a unit of code creates dependencies between the inputs and outputs, which are then verified against the dependencies in the specification. This analysis is useful for some critical output to be traced back to the inputs. Information flow analysis may be augmented in some tools by using annotations. These stylized comments contain assumptions about functions, variables, parameters, and types. They enable an analysis to proceed more efficiently by giving more information relevant to a particular block of code.

## Pointer Analysis and 3-valued Logic

Use of pointers and dynamically-allocated storage is a major obstacle to the goal of addressing software reliability by means of static analysis [32]. In particular, the effects of assignments through pointer variables and pointer-valued fields make it hard to determine the aliasing relationships among different pointer expressions in a program. For example, dereferencing NULL-valued pointers and accessing previously deallocated storage are two common programming mistakes. Analysis tools for finding bugs and detecting security vulnerabilities need answers to questions about pointer variables, their contents, and the structure of the heap—which refers to the collection of nodes in, and allocated from, the free-storage pool.

Flow-insensitive points-to analysis uses a very simple abstraction of heap-allocated storage: all nodes allocated at site $s$ are folded together into a single summary node $n$. Such an approach has rather severe consequences for precision. If allocation site $s$ is in a loop, or in a function that is called more than once, then $s$ can allocate multiple nodes with different addresses. A points-to fact "$p$ points to $n$" means that program variable $p$ may point to one of the nodes that $n$ represents. Consequently, most of the literature on points-to analysis leads to almost no useful information about the structure of the heap overly pessimistic assessments of the program's behavior [32].

This report [32] proposes a parametric framework for program analysis to address these issues. The key aspect is the use of 2-valued and 3-valued logical structures to represent concrete and abstract stores respectively. In 3-valued logical structures, a third truth value, denoted by 1/2, is introduced to denote uncertainty. Canonical Abstraction of 2-valued logical structures related class of 3-valued logical structures is created over the same vocabulary.

Formulas are also used to specify how the store is affected by the execution of the different kinds of statements in the programming language. A prototype implementation that implements this approach has been created, called TVLA [24].

## Boolean Programs and Model Checking

One recent challenge to static analysis is automated model checking and program verification. Ball and Rajamani [5] presented a model called boolean programs, which is expressive and amenable to model checking. They also presented a model checking algorithm for boolean programs using context-free-language reachability. This algorithm allows procedure calls with unbounded recursion, exploits locality of variable scopes, and gives short error traces. They gave a process for incrementally refining boolean program with respect to a particular reachability query. This model checker rules out infeasible paths by introducing boolean variables in a refined model. The process is illustrated in the following example.

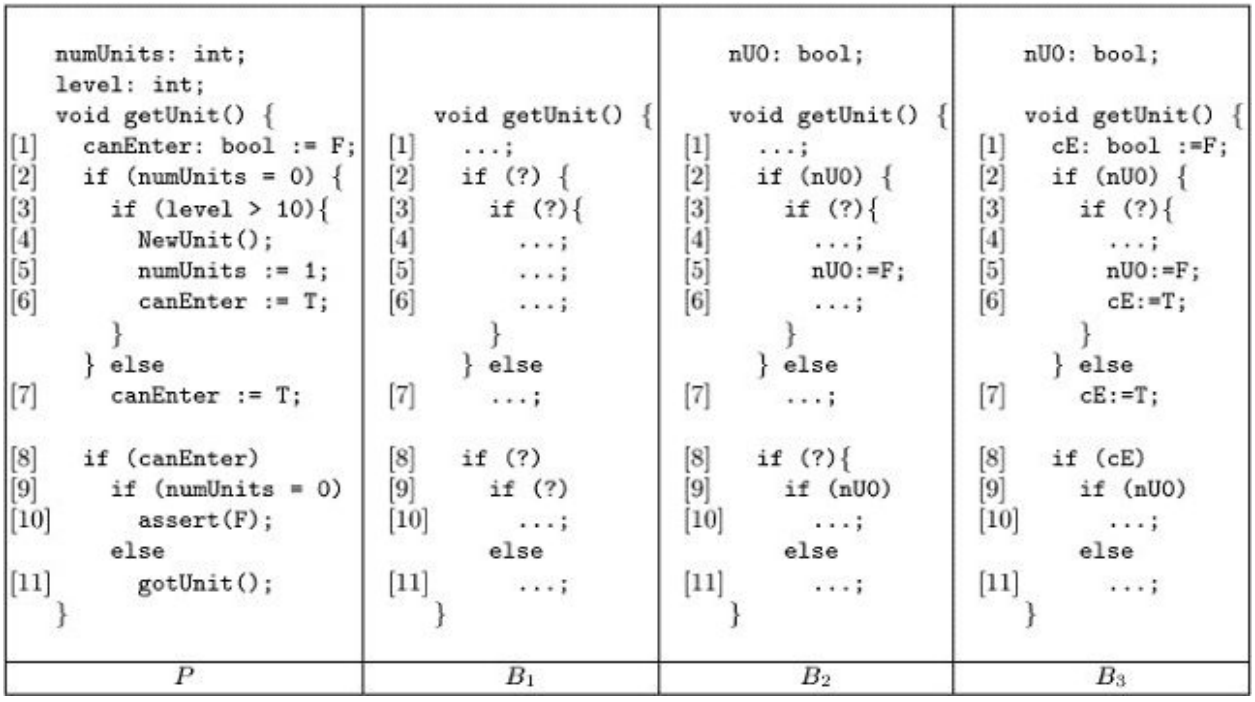



Figure 5.3: Abstraction Refinement with Boolean Programs

In Figure 5.3, the source program $P$ contains annotation to simulate a temporal property, and *B1*, *B2* and *B3* are boolean programs that abstract $P$ with an increasing level of precision. The direct problem is to find if line 10 is reachable in $P$, and this problem is translated to whether a sequential program obeys a temporal property. It can also be reduced to the problem of invariant checking. One of the motivating problems is to show that device drivers for operating systems obey certain temporal properties. The process is as follows:

- Generate "skeletal" Boolean Program *B1* from *P*, retaining control flow structure. In general it is undecidable to check if a program can reach some statement and also impractical to analyze large set of possible states. Abstract interpretation is created to approximate semantics of the program;

- Path [1, 2, 7–10] is possible in *B1* but infeasible in *P*. Refine *B1* to eliminate this path. Construct abstraction refinement *B2* by using the condition ($numUnits = 0$) and updating all statements affecting this condition;

- Path [1–3, 8–10] is possible in *B2* but infeasible in *P*. Again refine *B2* to eliminate this path. Construct abstraction refinement *B3* by using the condition ($canEnter = \text{true}$) and updating all statements affecting this condition;

- *B3* concludes that line 10 is not reachable in *P*.

Ball and Rajamani gave a new language to define the source program as well as boolean programs, and they later proved that model checking of boolean programs is sound [5]. In SLAM project, Ball and Rajamani put the theory into practice [7]. They developed a static analyzer called SLAM that employs the boolean programs technique. The SLAM toolkit is used to check whether or not a program obeys "API usage rules". The tool itself is sound as compliance with the theory.

The SLAM toolkit statically analyzes a C program to determine whether or not it violates some temporal safety properties. These properties are encoded in a language called SLIC (Specification Language for Interface Checking) [6]. The toolkit has two unique aspects: it does not require programmer to annotate source program (invariants are inferred); and it minimizes noise (false error messages) through a process known as "counterexample-driven refinement". The SLAM toolkit was successfully applied to Windows XP device drivers. It both validates the behavior and finds defects in their usage of kernel APIs.

## 5.3 Static Analysis Tools

The earliest static analysis tool was developed more than two decades ago. Program PFORT was designed in 1979 to check FORTRAN code for the presence of non-standard code [18]. More static analysis tools have been developed since and today there exist many different types. They vary from simple tools—these produce basic cross reference of variables, to very complex tools—these provide various analysis techniques and automatic program verification [36]. Many existing tools can describe flow structure of a program, classify uses made of data, give basic relationships between inputs and outputs, and give transfer functions through a section of code. Also most compilers provide basic static analysis facilities to assist compiler operations.

Many commercial tools use a common analysis phase after converting source into a standard form to support multiple languages. However this may reduce their effectiveness on

aspects that are specific to just one language. For example, there are several products available for the analysis of C code. But the weak type checking and lack of dynamic checking in C imply that such tools may not be useful for more strongly typed languages such as Pascal, Ada and Modula-2 [38].

## MALPAS-Malvern Program Analysis Suite

MALPAS was developed by Royal Signals and Radar Establishment Malvern based on the research of graph theory. It is a comprehensive static analyzer and contains several configurations to address different properties of a program. This tool is also capable of performing deep analysis including formal proof. User can choose various analyzers from the tool set:

> Control Flow Analyzer—examines program structure, identifying features such as entry and exit points, loops, branches and unreachable code. It gives a summary to any undesirable constructs and prepares items for the later analyzers enabling items to be handled regardless of the construction of original source;
>
> Data Use Analyzer—classifies data uses, revealing errors such as the use of uninitialized data, data written but never used, writing to procedure parameters specified as inputs etc.;
>
> Information Flow Analyzer—gives dependencies of outputs on inputs and on branches, revealing any expected dependencies, unused variables and redundant statements. It can also be used to assist in the design of dynamic testing;
>
> Path Assessor—counts the number of paths through each section of a program;
>
> Semantic Analyzer—reveals relationships between inputs and outputs in algebraic terms, giving the exact functionality of a program. It can also be used to give abstract information about a design or specification;
>
> Compliance Analyzer—compares the actual transfer function of a program with a formal specification, revealing the differences. This is used in correctness prove of a program for compliance of its specification.

MALPAS supports multiple programming languages including Ada, C, Pascal, FORTRAN, JSP, EPOS(S) and OBJ. A reader translates the source code into a special Intermediate Language (IL). The IL model is an internal directed graph with associated algebras for use by the analyzers. It also stores some statistical information which is sufficient to enable various metrics to be calculated. There is no pointer handling in IL, and to analyze C code it would first have to purge the use of pointers, which is a potentially formidable task [15]. As the requirements are informally expressed, manual analysis of the MALPAS results is required with some subjective judgments [38].

## SPARK Development Tools

Widely used in the UK safety systems, SPARK is an annotated sub-language of Ada which is unambiguous and suitable for rigorous static analysis. It was first formalized by Bernard Carré and Trevor Jennings of Southampton University in 1988 [8]. The development system was designed to aid the development of critical code in Ada 83. The main tool is SPARK Examiner, which enforces a small subset of Ada to make static analysis much easier. Examiner only works for SPARK set of Ada 83. A more general purpose analyzer for Ada is described in the article [10] which works for any Ada 83 or 95 programs.

The tool requires that some annotations are added to the source code. These annotations provide weak specifications of the functionality of each subprogram, and allow important properties to be checked by simple linear analysis. These properties include the absence of side-effects in functions and aliasing between parameters of a subprogram. Both aliasing and side-effects may cause Ada programs to be erroneous without being detected by a compiler. Their absence is also a prerequisite for the validity of the later stages of analysis performed by PARK Examiner [38].

The analysis is an automatic consequence of the subset and hence access to an unassigned variable is impossible. A conventional Ada compiler can be used for code generation. Control flow analysis is not needed as it is subsumed into the SPARK grammar. Data and information have simple recursive formulations to enable them to be performed as the language is parsed. Data and information flow requirements have been expressed in program design rules. The most important property of a SPARK program is exception-free which means it can not dynamically breaking certain language rules. An existing tool can be used to verify these properties [25].

SPARK Examiner performs standard static analyses on data and information flow and generates verification conditions. The result is fully automated partial correctness proof of code. The partial correctness states that in Hoare triple $\{P\}\ S\ \{Q\}$, if $S$ starts in a state satisfying $P$ and terminates, then $Q$ is satisfied upon termination. However no termination proof is provided. The following example shows a small SPARK program for swapping two variables, and how Examiner detects the errors [26]:

```
Procedure Exchange(X, Y: in out Float)
--# derives X from Y;
--# Y from X;
--# post X = Y' and Y = X';
is T : Float;
begin T := X; X := Y; Y := X;
end Exchange;

SPARK Examiner output is:
!!! (1) Flow error. Ineffective statement.
!!! (2) Importation of initial value of X ineffective.
!!! (3) Variable T is neither referenced nor exported.
```

```
!!! (4) Imported value of X not used in derivation of Y.
??? (5) Imported value of Y may be used in derivation of Y.
```

This example shows the detection of the unused variable $T$ and misuse of input variable $X$ by Examiner. SPARK Examiner is very much a program development tool rather than validation after development, and cannot be applied to arbitrary Ada code. However, the integration of the automatic forms of static analysis into the development process is claimed both to increase their benefit and reduce their cost [38].

## PREfix/PREfast

Within Microsoft, in addition to finding common programming errors, static analysis tools are used as early indicators of pre-release defect density [27]. Fault-proneness is defined as the probability of the presence of faults in the software [11]. One area in such research is to find the relationship between software quality metrics and fault-proneness.

There are a number of techniques used for the analysis of software quality. For example, multiple linear regression analysis was used to model the dependence of quality on software metrics. These techniques use structural and complexity metrics of the source code to act as predictors of defect density. One difficulty is that the multicollinearity among the metrics possibly leads to inflated variance in the estimation of reliability [27].

The PREfix tool works by symbolic executing selected paths through a program and look for a multitude of common low-level programming errors, including NULL pointer dereferences, the use of uninitialized memory, double freeing of resources, etc. Errors are automatically entered into a defect database to be fixed by programmers. The PREfix analysis is expensive and requires effective deployments of target system.

The PREfast tool is a fast version of the PREfix tool, aiming at inexpensive desktop deployment. Certain PREfast analyses are based on pattern matching to find simple programming mistakes. Other analyses are based on local data flow analyses to find uninitialized use of variables, NULL pointer dereferences, etc. Errors found by PREfast are mainly feedback to developers and may not be recorded. Figure 5.4 shows a model of PREfast and PREfix in software development process.

An experiment [27] was carried out using 199 components of Windows Server 2003 to find the correlation between the density of errors found by the tools and the pre-release defect density. The result showed that static analysis tools can be effectively used as early indicator and to predict pre-release defect density. They can also be used to discriminate between components of high and low quality. These results help inform decisions on testing, code inspections, design rework etc.

## ESC and lint-like Static Checkers

Extended Static Checker (ESC) is a static analyzer intended to catch errors at compile time. Common errors caught by ESC include array index bound errors, NULL dereferences, and deadlocks in multi-threaded programs. The tool is like a type-checker or like the C tool lint.

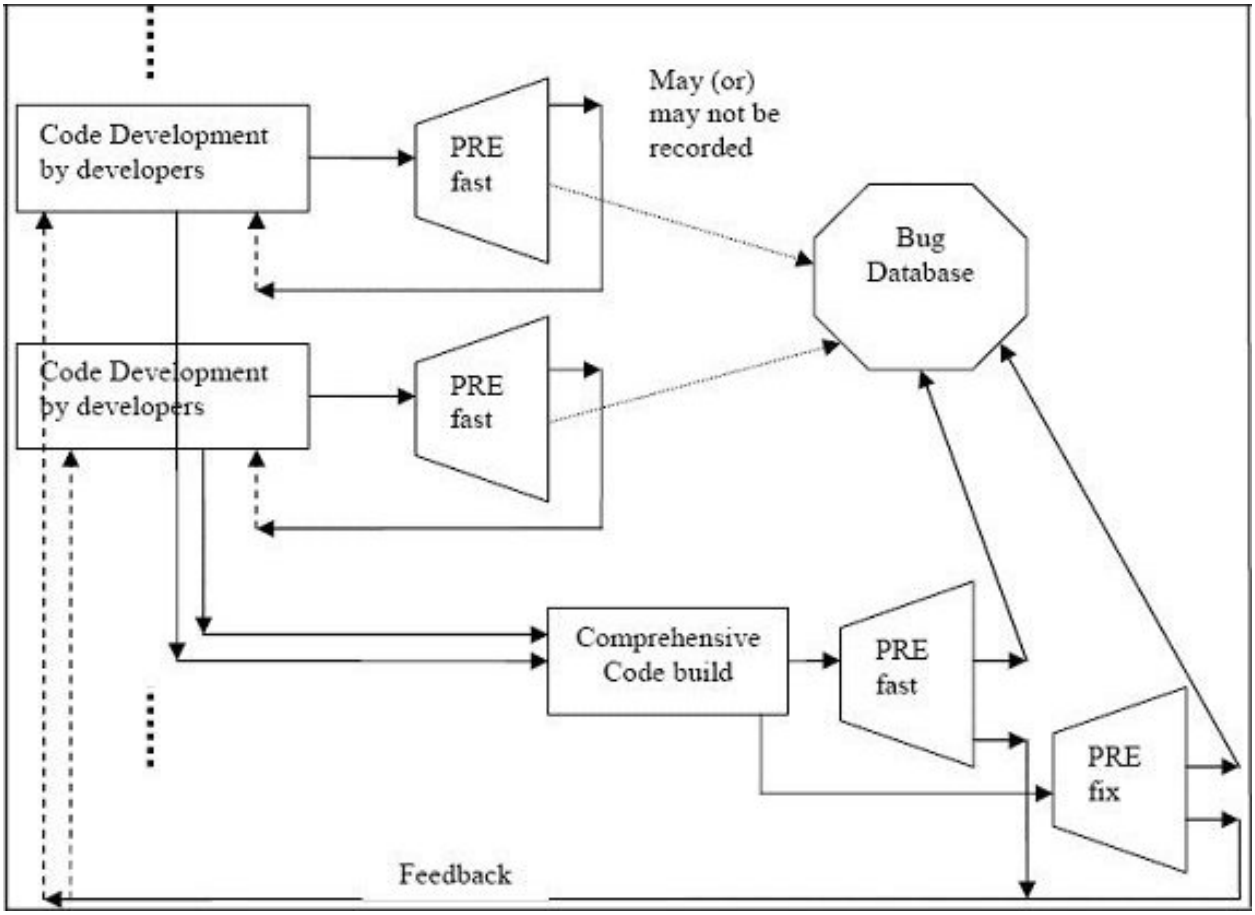



Figure 5.4: Software Development Model Using PREfix/PREfast

The output is intended to be interpreted by the author of a program being checked. The first version of ESC is for Modula-3 and the second version is for Java [12, 13].

This checker is implemented using the techniques of program verification. First, the source program is annotated with specifications. The annotated program is presented to a "verification condition generator", which produces logical formulas that are provable. These formulas are then presented to an automatic theorem-prover to validate correctness. ESC/Java uses annotation language called Java Modeling Language (JML) [22, 23]. The goals of ESC/Java and JML are different: JML is intended to allow full specification of programs, whereas ESC/Java is intended only for light-weight specification. Adding annotations is a manual process and requires programmer to be familiar with the rules.

ESC is different than program verification, because it does not prove that a program does what it is supposed to do. In fact it only intends to find certain types of errors. ESC is more feasible than full-scale program verification. Unlike SLAM which is sound but not complete, ESC is neither sound nor complete. This means that ESC could give false error messages. In fact many lint-like analyzers are like ESC; even unsound, they are still useful

in finding common bugs in programs.

## 5.4 Industrial Perspectives on Static Analysis

There are many static analysis tools available, whether light-weight or heavy-weight, commercial or non-commercial, and there likely exist one or more static analyzers for any programming language. Static analysis can be used as a retrospective analysis tool, however it is far better and more effective to use it at all stages of software development life cycle. Evidence exists to show that, if used correctly through the life cycle, static analysis can result in reduced costs, because errors are detected and removed at the earliest possible point [36].

Industry is increasingly using static analysis for quality assurance purposes, especially in safety critical systems. There are two major safety-critical software standards giving support for static analysis: the UK Interim Defence Standard 00-55 [28] and the international civil avionics standard [14]. The UK 00-55 puts forward the use of static analysis in conjunction with formal methods in critical software. It also restricts the use of assembly languages, interrupts, dynamic storage allocation and other techniques that could not be statically analyzed.

From the insight of the SLAM project, we can see a few trends in static analysis industry. Automated conformance checking will remain a challenge area in static analysis and become more and more important. Soundness of a tool itself is crucial to eliminate false error messages, which represent a major problem in many of the existing static analyzers [29]. A tool that can reduce additional efforts of programmer for use is proven to be valuable, for example automatic annotation insertion. Also a scalable analyzer will enable us to check larger and more complicated systems.

# Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, 1970.

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *TACAS'02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 158–172, London, UK, 2002. Springer-Verlag.

[5] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report 2000-14, Microsoft Research, February 2000.

[6] Thomas Ball and Sriram K. Rajamani. SLIC: A specification language for interface checking. Technical Report 2001-21, Microsoft Research, 2001.

[7] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.

[8] B. A. Carré and T. J. Jennings. SPARK - the SPADE Ada kernel. Technical report, University of Southampton, March 1988.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[10] Krzysztof Czarnecki, Michael Himsolt, Ernst Richter, Falk Vieweg, and Alfred Rosskopf. DataFAN: A practical approach to data flow analysis for Ada 95. In *Ada-Europe*, pages 231–244, 2002.

[11] Giovanni Denaro, Sandro Morasca, and Mauro Pezze;. Deriving models of software fault-proneness. In *SEKE'02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 361–368, New York, NY, USA, 2002. ACM Press.

[12] D. Detlefs, K. Leino, G. Nelson, and J. Saxe. *Extended Static Checking, Compaq SRC Research Report 159*, 1998.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *PLDI'02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[14] European Organization for Civil Aviation Electronics. *Requirements and Technical Concepts for Aviation document RTCA SC167/Do-178B, European Organization for Civil Aviation Electronics EUROCAE document D-14B.*

[15] A. German. Software static code analysis lessons learned. *CrossTalk, The Journal of Defense Software Engineering*, Nov 2003.

[16] Rajeev Gopal and Stephan R. Schach. Using automatic program decomposition techniques in software maintenance tools. In *In Proceedings of ICSM'89, International Conference on Software Maintenance*, pages 132–141. IEEE, ieeecsp, 1989.

[17] Douglas Gregor. *High-level Static Analysis for Generic Libraries.* PhD thesis, Rensselaer Polytechnic Institute, May 2004.

[18] A. D. Hall and B. G. Ryder. PFORT verifier. In *QCPE 11, 374*. Bell Laboratories, Murray Hill, New Jersey, 1979.

[19] V. Hirvisalo. *Using Static Program Analysis to Compile Fast Cache Simulators*. PhD thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Otaniemi, Finland, March 2004.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 35–46, New York, NY, USA, 1988. ACM Press.

[21] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[22] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Iowa State University, Department of Computer Science, 1999.

[23] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106, 2000.

[24] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *SAS'00: Proceedings of the 7th International Symposium on Static Analysis*, pages 280–301, London, UK, 2000. Springer-Verlag.

[25] Program Validation Ltd. *Generation of Path Functions and Verification Conditions for SPARK Programs, Edition 1.lb*, January 1992.

[26] Course Note, Lecture 9: Verification with Assertions: The SPARK Ada System. http://www.cs.bham.ac.uk/~mzk/courses/SafetyCrit/, March 2005.

[27] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE'05: Proceedings of the 27th international conference on Software engineering*, pages 580–586, New York, NY, USA, 2005. ACM Press.

[28] Ministry of Defence. *Interim Defence Standard 00-55: The procurement of safety critical software in defence equipment*. Ministry of Defence, Directorate of Standardization, April 1991.

[29] Hideto Ogasawara, Minoru Aizawa, and Atsushi Yamada. Experiences with program static analysis. In *IEEE METRICS*, page 109, 1998.

[30] Sandra Rapps and Elaine J. Weyuker. Data flow analysis techniques for test data selection. In *ICSE'82: Proceedings of the 6th international conference on Software engineering*, pages 272–278, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[31] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *ISSTA'94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 138–153, New York, NY, USA, 1994. ACM Press.

[32] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL'99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–118, New York, NY, USA, 1999. ACM Press.

[33] V. Seshadri, S. Weber, D. B. Wortman, C. P. Yu, and I. Small. Semantic analysis in a concurrent compiler. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 233–240, New York, NY, USA, 1988. ACM Press.

[34] O. Shivers. Control flow analysis in Scheme. In *PLDI'88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 164–174, New York, NY, USA, 1988. ACM Press.

[35] H. Sneed. SOFTDOC - A system for automated software static analysis and documentation. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 173–177, New York, NY, USA, 1981. ACM Press.

[36] J. T. Webb and D. Mannering. MALPAS - verification of a safety critical system. *SARSS'87: Achieving Safety and Reliability with Computer Systems*, page 44, 1987.

[37] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[38] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, March 1995.

# Chapter 6

# Ning Zhou: Stepwise Refinement of Object-Oriented Models

Traditional refinement technique based on the structured programming is a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. In the area of object-oriented programming, object-oriented models may contain elements like general associations (which may have attributes themselves) that cannot be directly mapped to a programming language. The work presented here is a survey of techniques of general refinement steps that allow object-oriented models to be transformed, possibly in various ways, towards implementations.

## 6.1 Introduction

In this paper the concept of refinement is presented in the specific context of UML class diagrams. The current research in this area implies that it is very beneficial and is supported by the scientific community [2]. UML refinement is beneficial for the following reasons: (1) it is based on the UML which is a fully supported and widely used modeling language for software design [5]; (2) it can be extrapolated onto various implementation domains (e.g., B-method, Java, code generation tools (Rational, System Architect) [9, 1], etc.); and (3) it is sufficiently abstract that implementation details are removed thereby making it clearer to communicate modeling concepts with particular emphasis on refinement. This is inline with the manner in which other abstract design strategies are based, such as the B-method, and the Z-specification.

This paper is divided into five sections. The first section presents the motivation for this study. The second section discusses the theoretical framework of the UML refinement technique. The third section compares object-oriented refinement approaches. The fourth section discusses a Scrabble game example. The last section presents the conclusion and future direction of research in the area of UML refinement.

## 6.2 Motivation for the Study

The purpose of this paper is to use *Model-driven Architecture* (MDA) approach to help structure the specification of systems with *platform-specific models* (PSMs) for generating implemented Java code from *platform-independent models* (PIMs). Abstraction facilitates the ability to look at major issues of a design or a model by omitting details of realization. Refinement creates a link between specification and implementation. With it, the code for a component can be conformed to the interface expected by its clients; each business goal can be related to each specific feature of the design and ultimately to each line of code.

## 6.3 Theoretical Framework of the UML Refinement Technique

There has been a significant amount of research and development in the area of refinement[4]. Much of these types of refinement methods deal with abstract machines and specification of non-deterministic programs, concurrent or control software, etc. For example, the following program presents a stepwise refinement of the square-root algorithm using the B-Method[7].

**MACHINE** ***SquareRoot***
**OPERATIONS**
$sqrt \longleftarrow SquareRoot(xx) \equiv$
**PRE** $xx \in \mathbf{N}$
**THEN**
**ANY** $yy$
**WHERE** $yy \in \mathbf{N} \wedge square(yy) \leq xx \wedge xx < square(yy+1)$
**THEN** $sqrt := yy$
**END**
**END**
**DEFINITIONS**
$square(x) \equiv x \times x$
**END**

**REFINEMENT** ***SquareRootR***
**REFINES** ***SquareRoot***
**OPERATIONS**
**SquareRoot**
$sqrt \longleftarrow SquareRoot(xx) \equiv$
**ANY** $yy, zz$
**WHERE** $yy \in \mathbf{N} \wedge zz \in \mathbf{N} \wedge sqinv(xx, yy, zz) \wedge zz = yy + 1$
**THEN**
$sqrt := yy$
**END**

**DEFINITIONS**
$squre(x) \equiv x \times x$
$sqinv(x, y, z) \equiv y < z \wedge square(y) \leq x \wedge x < square(z)$
**END**

**IMPLEMENTATION *SquareRootRI***
**REFINES *SquareRootR***
**IMPORTS *SquareRootUtils***
**OPERATIONS**
**SquareRoot**
$sqrt \longleftarrow SquareRoot(xx) \equiv$
**VAR** $yy, zz$
**IN** $yy := 0;$
$zz := (xx + 1) \div 2 + 1;$
**WHERE** $yy + 1 \neq zz$
**DO** $yy, zz \longleftarrow$ *ChooseNewApprox(xx,yy,zz)* $\equiv$
**INVARIANT** $yy \in \mathbf{N} \wedge zz \in \mathbf{N} \wedge sqinv(xx, yy, zz)$
**VARIANT** $zz - yy$
**END**;
$sqrt := yy$
**END**
**DEFINITIONS**
$squre(x) \equiv x \times x$
$sqinv(x, y, z) \equiv y < z \wedge square(y) \leq x \wedge x < square(z)$
**END**

The implementation of the *SquareRoot* machine calls another function *ChooseNewApprox(xx,yy,zz)* which is implemented by an imported machine *SquareRootUtils*. When a newly invented machine imported for the purpose of enabling implementation, the new machine must refined and implemented. This layered process construct stepwise refinement of abstract machine in B method.

The goal of the research presented in this paper is to use the concepts developed by N. Wirth [10] "Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decisions", combined with the idea of layered refinement with B method and apply them to the UML. Specifically this paper addresses how refinement can be applied to UML class diagrams.

## 6.4 Comparison of Object-Oriented Refinement Approaches

This section take the example of [3], with different analysis methods and objective, offer a comparison of object-oriented refinement approaches.

### Use Case Refinement

- Refining by Action Decomposition

  A composite use case can be decomposed to several component use cases, this kind of relationship reflect a refinement hidden in UML use cases. Figure 6.1 shows a Use Case Refinement Under a Decomposition.

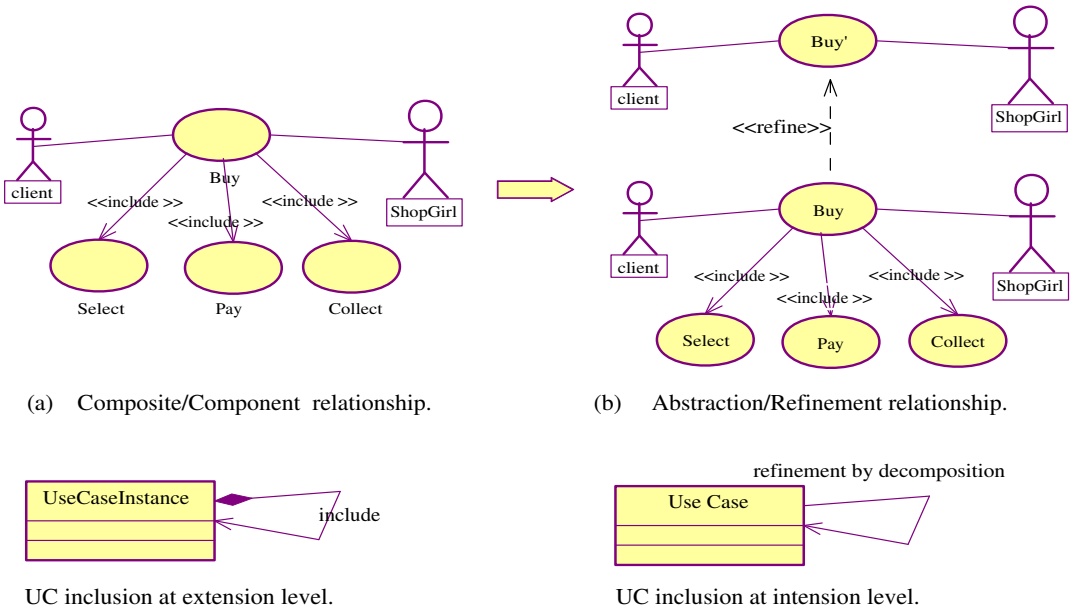



Figure 6.1: Use Case refining by Action Decomposition

Note that this refining is in the instance level. For example, the use case instance <Ana-buys-a-dress> includes: <Ana-selects-a-dress>,<Ana-pays-for-the-dress> and <Ana-collects-her-dress>.

- Refining by Specialization

  An inherited use case can be refined by adding Object-Constraint Language to it according to the specialization relationship between parent and kid. Figure 6.2 is a example for this refinement. The refinement keep the use case "Pay" the same but add an precondition and a postcondition for the "PayByCreditCard" use case by copying OCL of "Pay" and replacing resources and availability which are two formula in specialization.

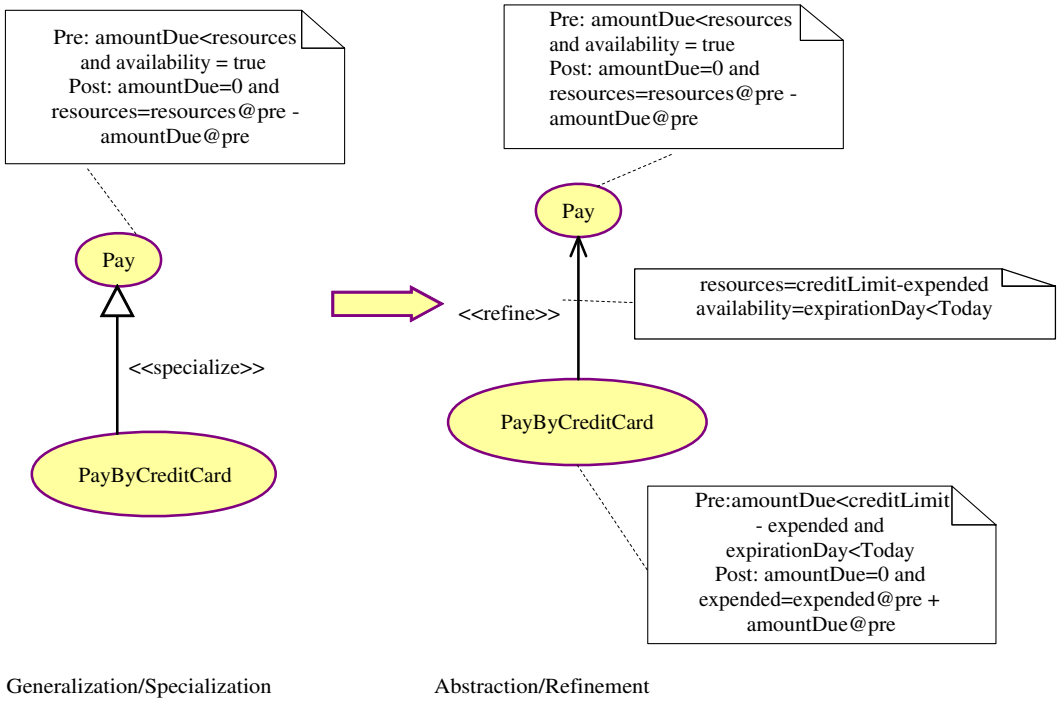



Figure 6.2: Use Case Refining Under a Specialization

**Class Refinement** Class refinement is a mapping from refined class to the original class as shown in Figure 6.3

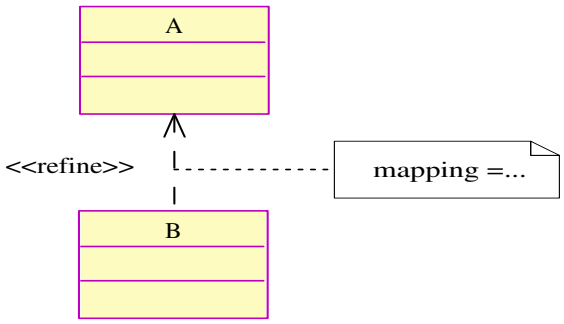



Figure 6.3: Class Refinement

- Attribute Refinement

  Attribute refinement is to add one or several new attribute or to replace an attribute by one or more attribute. Figure 6.4 shows that a segment class can be refined by replacing its' attribute length by the initial and final x-coordinate of the segment. Accordingly, the invariant length is greater than zero can be transferred to xinitial minus xfinal is greater than zero.

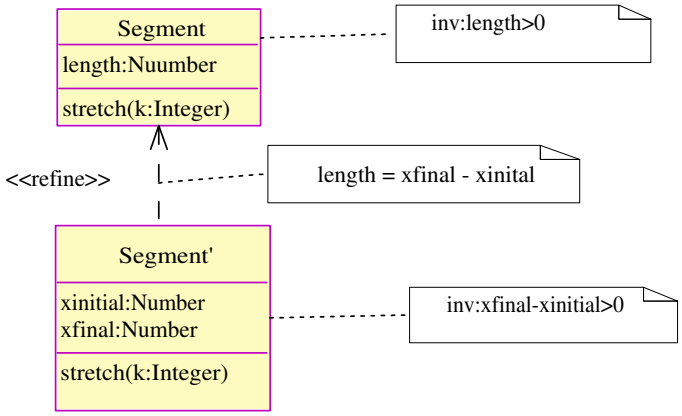



Figure 6.4: Attribute Refinement

- Operation Refinement

  Operation refinement is to add one or several new operation or to replace an attribute by one or more operation. Figure 6.5 shows that the operation stretch k can be replaced by moving initial and final x-coordinate of the segment k/2.

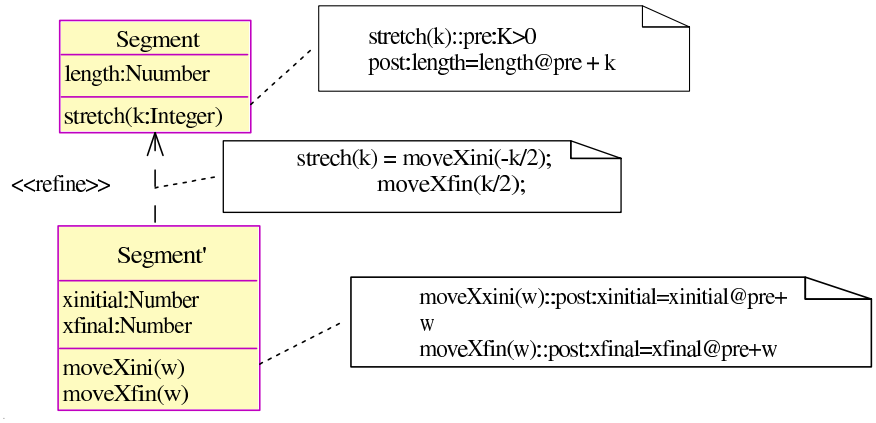



Figure 6.5: Operation Refinement

### Association Refinement

- Refining by Constraining Properties

  An approach to add some constraints to the properties of association is shown in Figure 6.6 which set account for each client to be ordered.

- Refining by Specialization

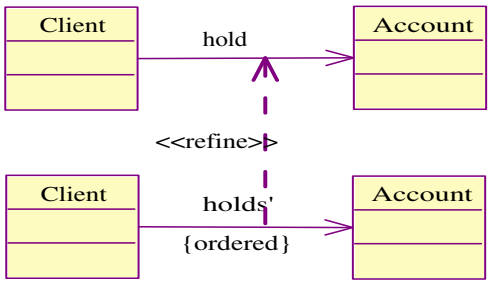



Figure 6.6: Association Refinement by Constraining Properties

In Figure 6.7, client holds account; young client is inherited from client and can only hold savings account; elder client can hold both savings and checking account. This inheritance relationship reflect a refinement between a specific client and a client.

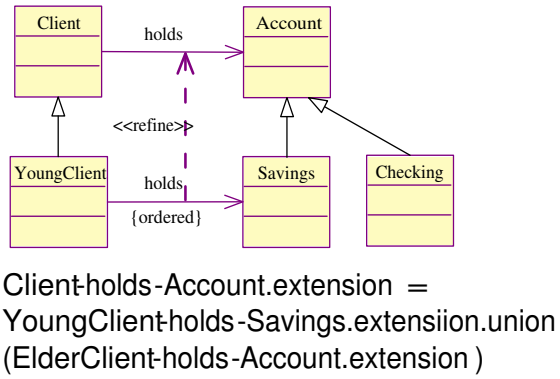



Figure 6.7: Refining by Specialization

- Refining by Link Decomposition

  This Refinement is not used in UML since UML does not support multiple associations. In Figure 6.8, the association work with between Client and Bank can be decomposed to two associations (holdsAccount and hasCredit).

**Refining by Model Transformation** Model transformation improves the structure of a model by providing techniques of refining from analysis model towards platform independent design. The following are some examples of common model transformation.

- Removal of Many-many Association

  It replaces one many-many association by two many-one associations as shown in Figure 6.9.

- Replace Inheritance by Association

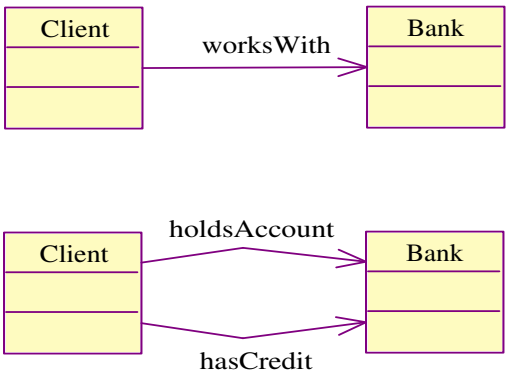



Figure 6.8: Refining by Link Decomposition

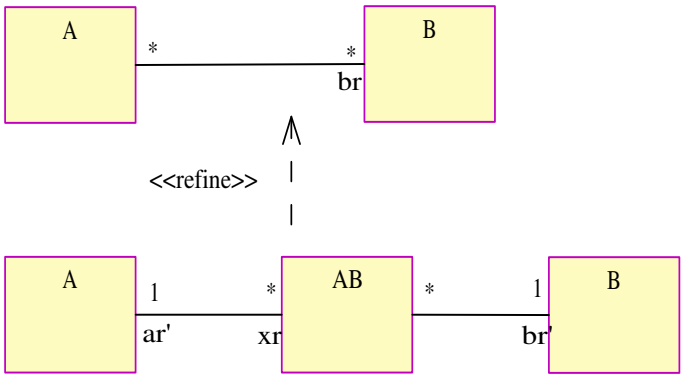



Figure 6.9: Removing a many-many association

It is useful in removing multiple inheritance when transforming a PIM to a PSM, a PSM for a programming language such as Java do not support multiple inheritance. Figure 6.10 shows this transformation.

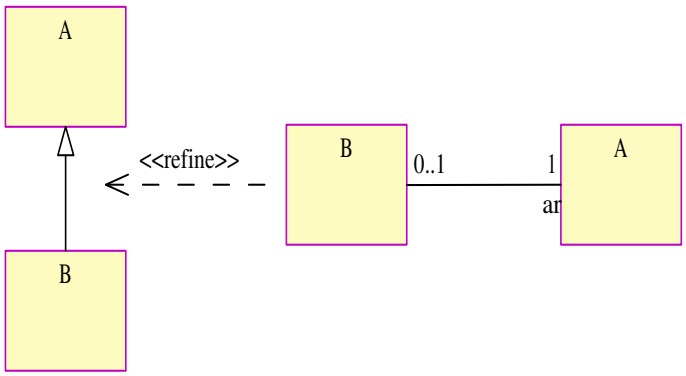



Figure 6.10: Replacing an Inheritance by Association

- Introducing a qualified association

  Suppose a, b are instances of classes A and B separately, x is the identity attribute of object b, after replacing many-many association by a single qualified association with the qualifier x, a and x relates to b by association *A_B*' which is shown in Figure 6.11.

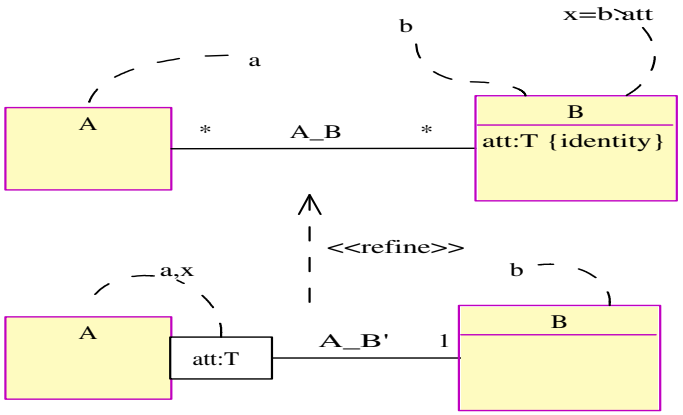



Figure 6.11: Introducing a Qualified Association

- Introducing a super class

  Figure 6.12 shows the situation when two or more classes have common attributes or operations, a super class can be introduced to include the common attributes and the declaration of the common operations.

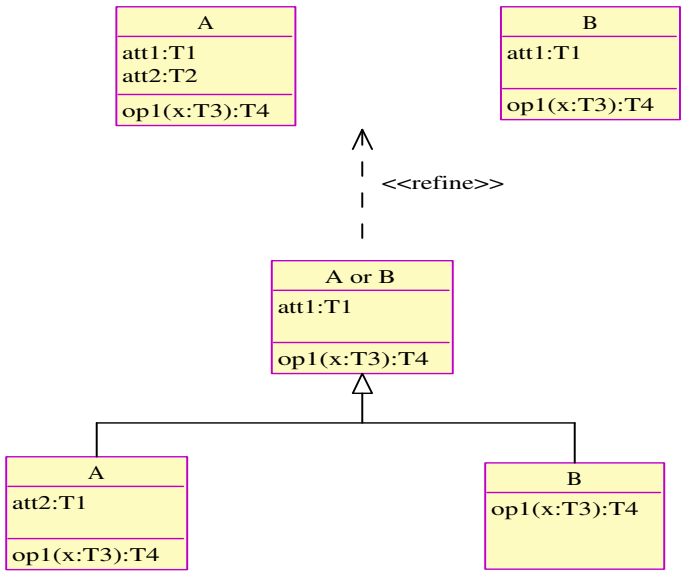



Figure 6.12: Introducing a Super Class

## 6.5 An Refinement of The Scrabble Game Model

With the concept of stepwise refinement and UML artifacts in mind, how to construct a system such as the Scrabble game? As work has been previously done in this area by Kevin Lano [6], this paper summarizes and extends his work, presents the advantages and disadvantages and provides directions for future research.

Firstly we should go over all the information provided by the system, from documentation or people who play the game before. Next, we take the description, perform a systematic requirements analysis by extracting the use case scenario(s). Based on the use cases, we find out the entities of the system and properties of the entities to make a class diagram. We perform several stepwise refinements for this basic model and finally get our Java platform-specific model for Scrabble game.

**Build an Initial Use Case for Scrabble Game** We can use the rule book of the game or the experience of playing it to determine there are only a few actions at the very beginning of creating a platform-independent model:

1. **Add player** Add a player to the system, up to four. The player could be a computer player or a human player.

2. **Start game** When a human player press "start game" button, the system pick up a letter randomly for each player, the player who has the lowest alphabetic letter get the first turn to play the game.

3. **Select letters** The system distribute seven letters from the bag to the turn player's rack randomly. The player chooses letters remaining in the rack and then put them on

the board when it is his turn to play the game.

4. **Generate move** A computer turn player generate a word from the dictionary.

5. **Make move** The system check the validity of the move according to the playing rule, the move is made permanent onto the board and the score is calculated and updated if the move is valid.

6. **End game** The game will be finished when either the bag is empty, one of the players' rack is empty or one player pass twice. The score will be calculated for each player and the winner player will be point out.

Figure 6.13 shows a picture of the initial use case diagram.

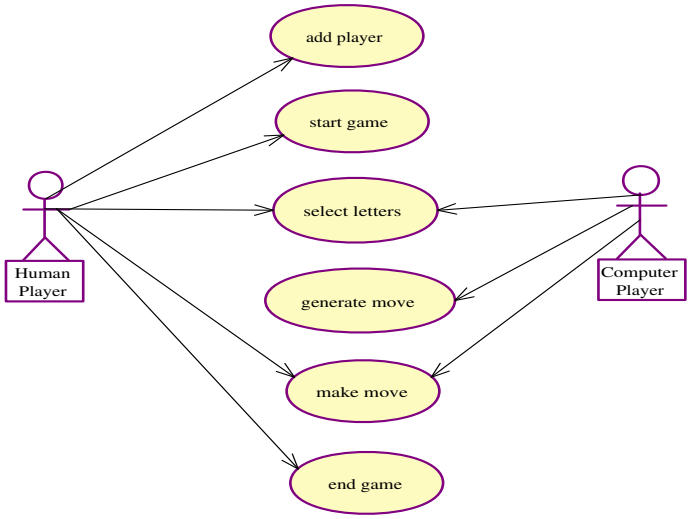



Figure 6.13: Initial Use Case Diagram for Scrabble Game

**Build an Initial Class Diagram for Scrabble Game** For the data model, we need to capture the entities for our class diagram. The simpler and efficient way to do this is to list all the key nouns of the system. Here is the list of all candidates for entities:

player, game, score-keeper, turn, score, score sheet, tile, bag, blank, rack, board, word,Rack, board, word, square, line, dictionary,double letter square, center square, row,column, premium square, triple word square, double word square.

An entity is an inclusion of objects which have some common properties and can perform some operations. According to it, we differentiate the attribute from entities. The "turn" and "score" should belong to other classes as attributes.

Now is the time to construct an abstract class diagram which consists of those basic classes and their relationships:

1. **Board.** Board consists of 15 by 15 squares, so it is related to the entity Square with multiplicity 225 at the Square end

2. **Square.** Any one square has a location on the board, the attributes x and y represent its' coordinate on the board. A square may be occupied by a tile or empty, so there should be a Boolean function isOccupied() to check its' status. A square can be an "Ordinary Square", "DoubleLetter Square", "DoubleWord Square", "TripleLetter Square" or "TripleWord Square". Each kind of square has different score calculating rules. The operation getTileScore() returns the score of each occupied square.

3. **Game.** Each game has one Board, one Bag, two to four Players and one Current player which are classes associated with Game. Notice that the association with role name currentPlayer at Player end is the subset of the association with role name players at Player end. Only one of the players can be the current player, which is expressed by multiplicity of 0..1 at Game end.

4. **Bag.** Each bag contains up to 100 tiles which is represented by multiplicity 0..100 at Tile end. Whether a bag is empty can be checked by a query operation isEmpty(). A bag needs an operation to give any set of 7 tiles to the rack.

5. **Tile.** Each tile has a symbol and a score as attribute. A tile may be in a Bag, a Square or a Rack. So the multiplicity at these three ends are 0..1.

6. **Rack.** A rack needs operations to add up to seven tiles from the Bag or to remove any tile by the current player

7. **Dictionary.** The system needs a dictionary to check the validity of the generated word.

Figure 6.14 shows a picture of initial scrabble class diagram.

**A Class Refinement Based On Specification** If we look at the rules and requirements hidden in the system more thoroughly, not just by the physical characteristic of board game, we will find some points need to be improved to clarify the specification of the system.

1. We should give each player a name, so that the players' score can be stored and listed on system GUI.

2. One of the requirements specified is that the history of the move should be kept, so we need to add an new class Move associated with game. Each move corresponds to one or more letter moves, therefore, an association to the class LetterMove is built.

3. The multiplicity at the association end should be checked since it is a easy point to make mistakes. Cases where an association end should be ordered can be identified: in our refined class diagram the association ends with role names history and players need to be ordered(note that ordered with curly bracket is a built-in object constraint language).

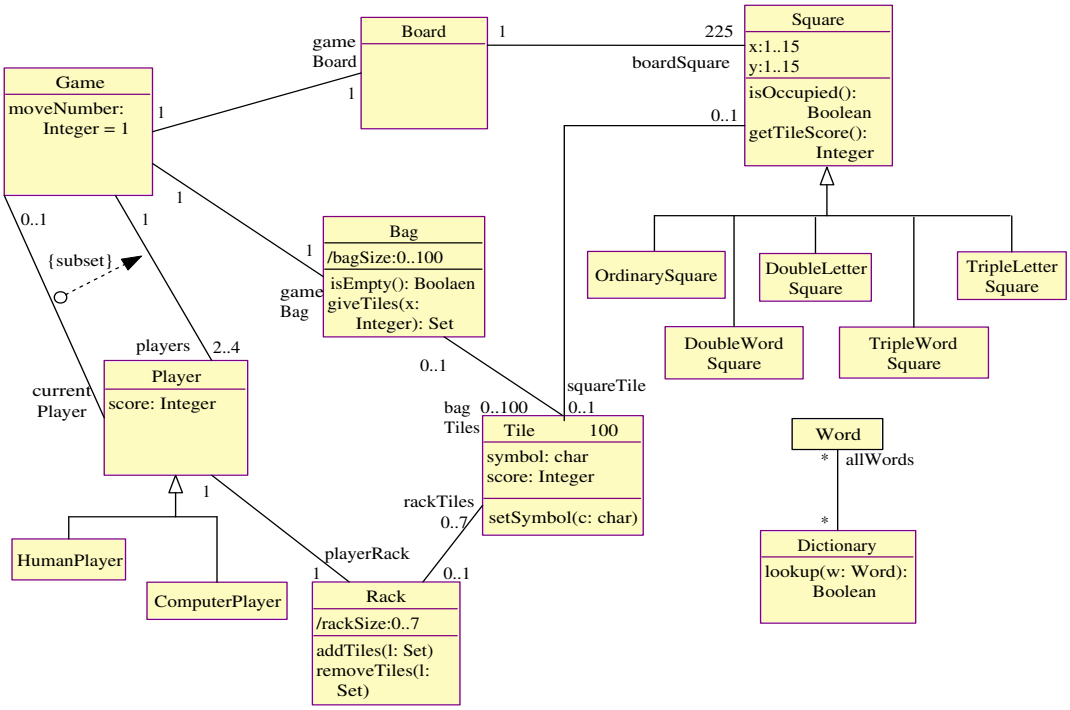



Figure 6.14: Initial Scrabble Class Diagram

4. Since Players role has a natural ordering, the order each player take the turn, we can optimize the currentPlayer association by adding an integer index turn as an attribute to the Game Class.

5. Attribute types can be made more precise. For example the score of a Tile must be between 0 and 10, this would be over-specified, since it prevents other non-English version of Scrabble game using tiles with score greater than 10.

6. The linear list of the boardSquares with multiplicity of 225 can be rationalized to double array structure which represented by two quantifiers between 1 and 15 added at the opposite end of association. By doing this the physical layout of the system is obvious.

7. Class or attribute names should also be reviewed in order to see if there is any improvement for the better understanding of the system. For example, the name Tile is a technical name correspond to physical characteristics of the class, it can be replaced by letter to emphasize the use of the tile.

After these considerations, the refined class diagram Figure 6.15 is formed.

**A Use Case Refinement by Action Decomposition** In a Scrabble Game, verifying a human player move and generating a computer move corresponding to validMove operation

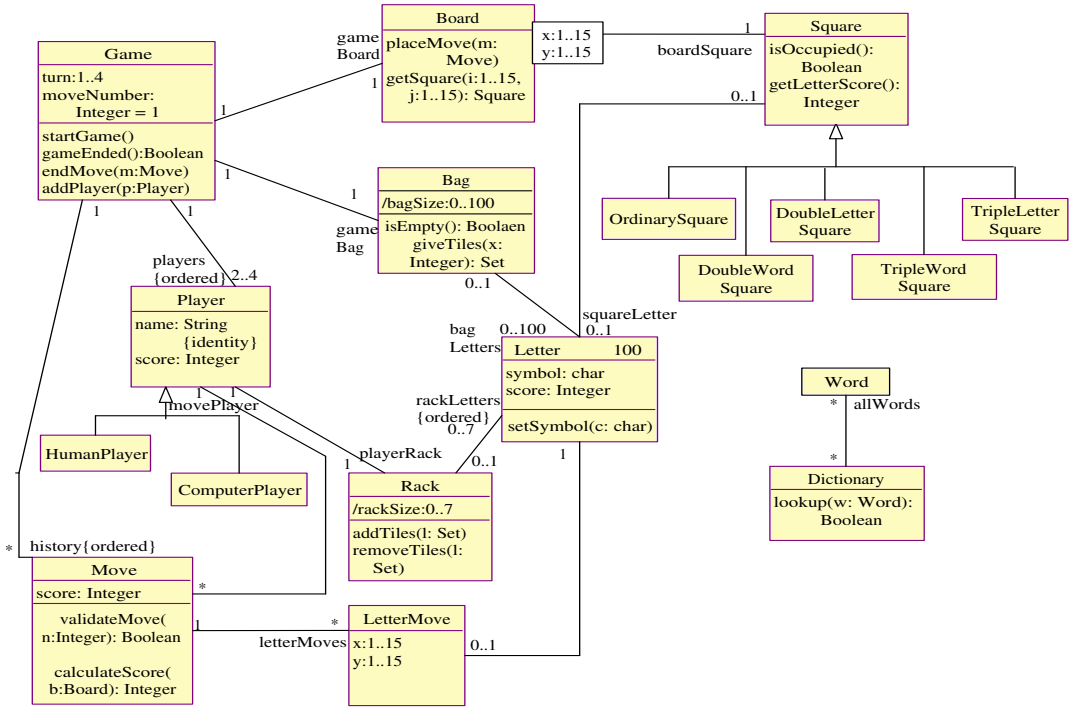



Figure 6.15: Refined Scrabble Class Diagram

in Move class both involve looking up a word in the dictionary. Calculating a bonus score could be an extension of calculate score use case. We refine the Scrabble Game use cases with the technique of action decomposition by using relationship of inclusion and extension between use cases.

Figure 6.16 shows a picture of refined use case diagram.

The operation validateMove in class Move is used to examine in the dictionary if the current move is valid. Regarding the refined Use Case there should be a new association between classes move and word.

**A Refinement By Adding Constraints with OCL** The Scrabble class diagram is not refined enough now to provide all the relevant features of a specification. There is a need for OCL to fill this gap. OCL is typically useful:

- Describe invariants of classes – the properties of attributes and role names.
- Describe pre and post condition of operations of classes.
- Constrains the relationship between association ends.

Below is the refinement realized by adding constraints to the class diagram.

1. We add an invariant "endx=startx or endy=starty" to the class Word to state that a word on the board would be either horizontal or vertical.

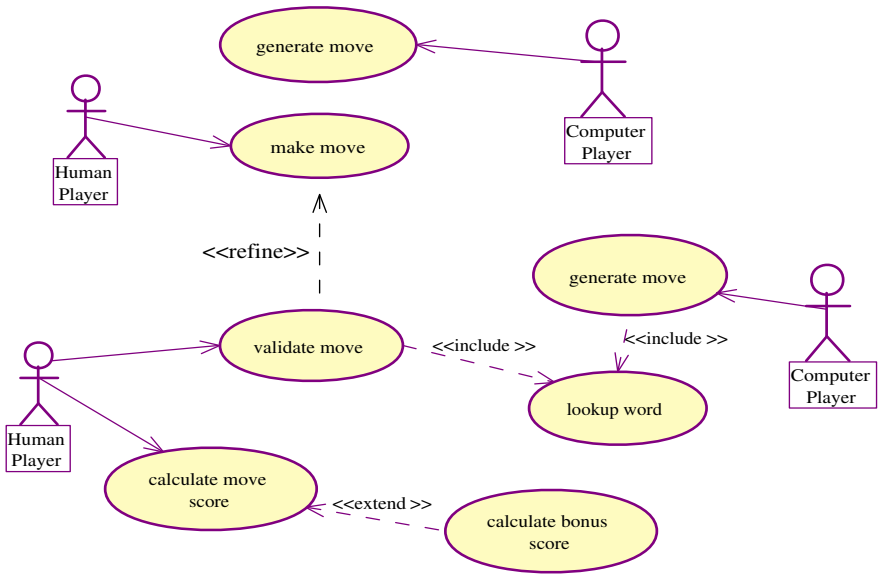



Figure 6.16: Use Case Refinement By Action Decomposition

2. The precondition "score=0" of setSymbol(c:char) operation in class Letter says that only the blank letter with the score equal to 0 can be used as any letter.

3. The post conditions "squareLetter = {} → result=false" and "squareLetter = {} → result=true" on isOccupied(): Boolean of Square define the value of this query operation.

4. The built-in constraints {subset} added between two association ends wordsFormed and allWords express that all words formed at a valid move is in the dictionary. The constraint {readOnly} beside allWords means that all the words we look up in a dictionary cannot be modified.

5. We add another constraints {disjoint} between two association ends bagLetters and rackLetters to indicate that a letter would either be in a Rack or in a Bag, but can not be in both at the same time.

After adding these constraints, we construct our refined Scrabble class diagram in Figure 6.17 by constraints.

**Further Considering the Behavior Notations** Navigation expression s in OCL enable assertion be made on compositions of associations and attributes. Quantifier expression: $P \rightarrow forAll(s)$ express the meaning "P is true for every element of the collection s". These complex constraints and behavior notations are used to obtain further refined class diagrams.

The navigation expression C1: "LetterMoves.x.size = 1 or letterMoves.y.size = 1" as an invariant of Move says that the set of letters placed in one move must either have the same x coordinate or have the same y coordinate.

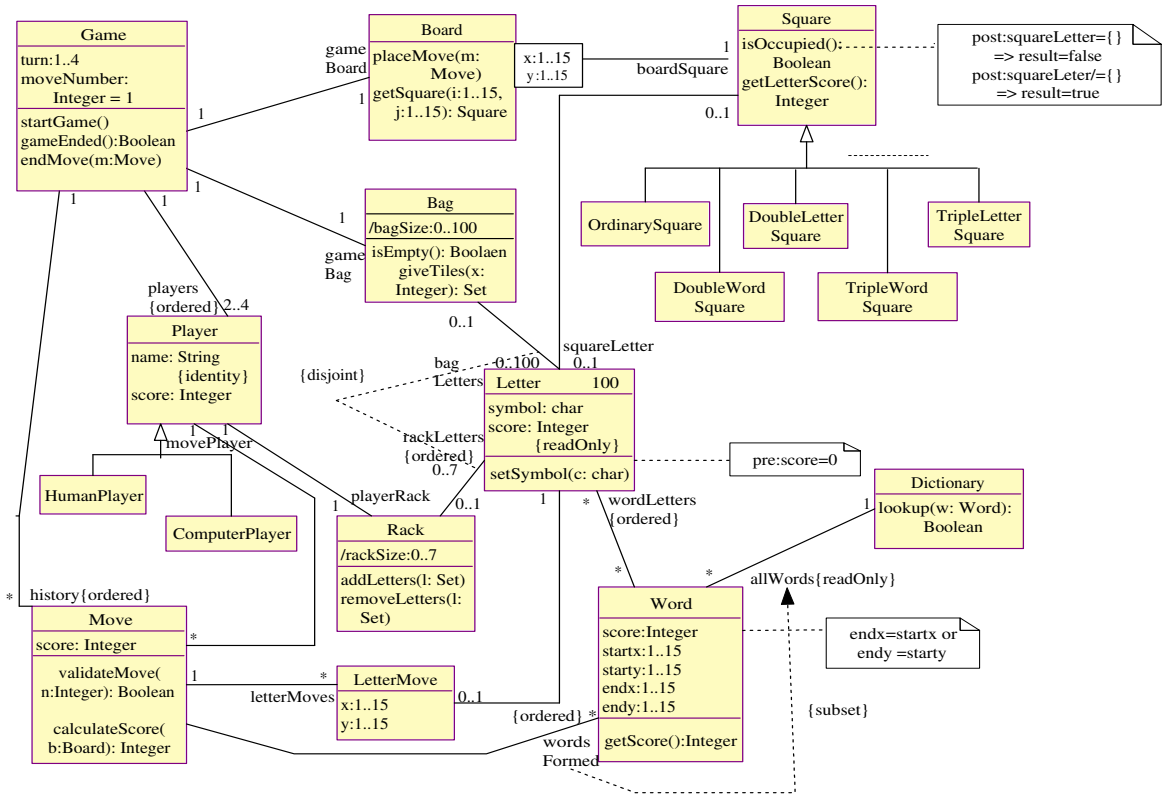



Figure 6.17: Refinement by Constraints of Scrabble Class Diagram

The invariant of Game $C2$ : "8 : $history$[1].$letterMoves.x$&8 : $history$[1].$letterMoves.y$" indicate that the first move must include the center square.

The invariant of Game C3: "gameBag.bagLetters $\rightarrow$ $forAll(score \leq 0)$ states that all the letters in the game bag have scores at most 10.

According to the validity condition for a move: "all letters played in a move must be co-linear", when adding an operation "addLetterMove(lm: LetterMove)" to Move, the pre and post condition of the operation should be:

addLetterMove(lm: LetterMoves)
// pre:
// lm.letter : m.movePlayer.playerRack.rackLetters
// post:
// if lm in line with letterMoves@pre
// letterMoves = letterMoves@pre $\vee\{lm\}$
// else
// letterMoves = letterMoves@pre

Only if "lm in line with letterMoves" be satisfied, letterMoves can be modified in order to satisfy the invariant C1.

"The player removes tiles from their rack and places them on the board" can be stated from the precondition of addLetterMove by the invariant of Move.

C4: letterMoves.letter:
movePlayer.playerRack.rackLetters.

After this further refinement, the Scrabble class diagram looks like Figure 6.18.

### Model Transformation from PIM to PSM

- In PSM class diagram the navigation direction of associations needs to be specified.
- The visibility of features (eg. public, protected or private) needs to be made explicit as Table 6.1 if they are not already defined.

| Notation | Meaning |
|---|---|
| - | private: accessible in the defining class only |
| # | protected: accessible in defining class and in its subclasses |
| + | public: accessible in all classes |

Table 6.1: Visibility annotations in UML

- For Java PSM, attribute types can be transformed from the PIM types as Table 6.2.

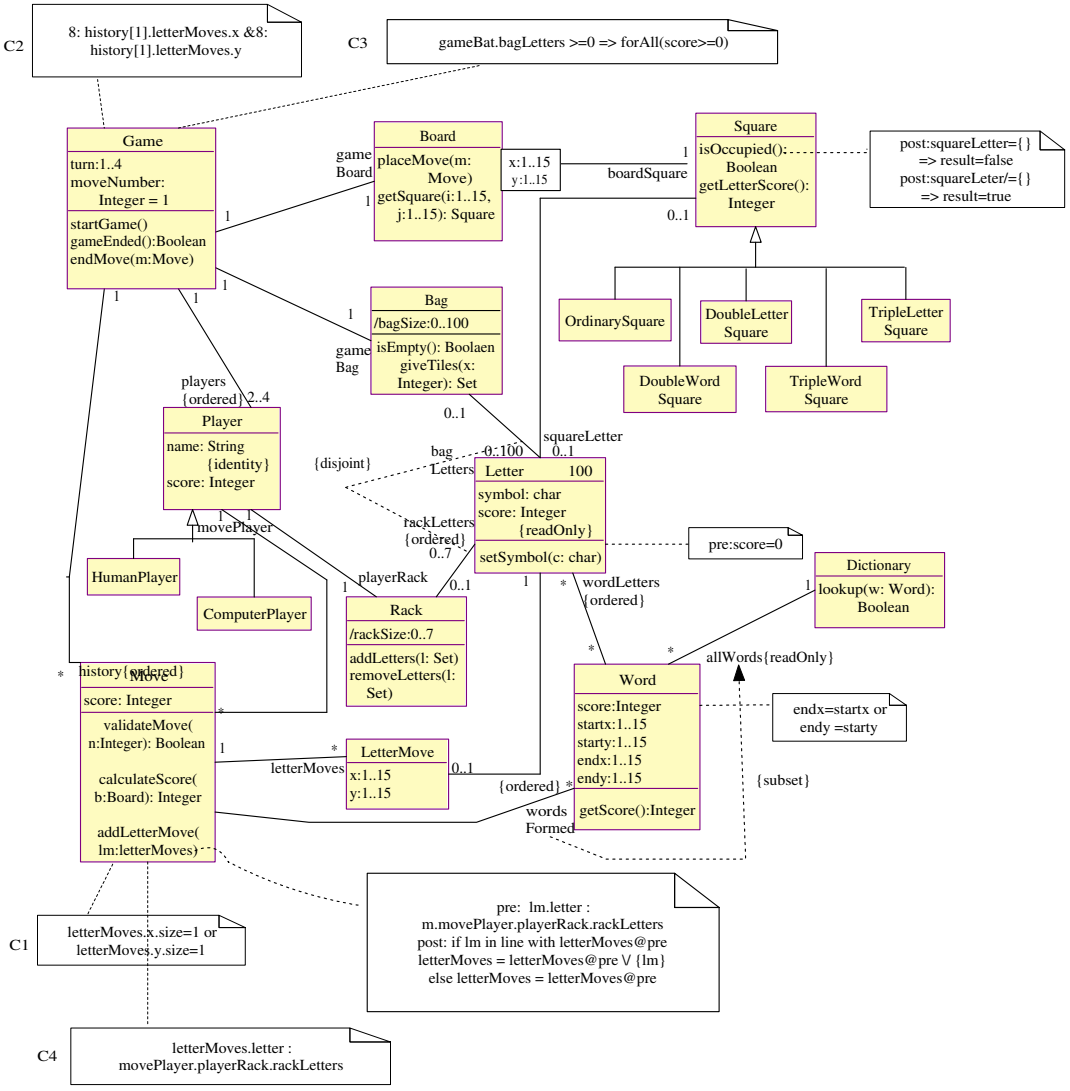



Figure 6.18: Further Refinement of Scrabble Class Diagram

| PIM type | Java PSM type |
|---|---|
| integer | int |
| Real | double |
| Boolean | boolean |
| user-defined enumerated type<br>eg. Direction {vertical, horizontal} | construction<br>public static final int vertical =0<br>public static final int horizontal =1 |

Table 6.2: Mapping UML Types to Java

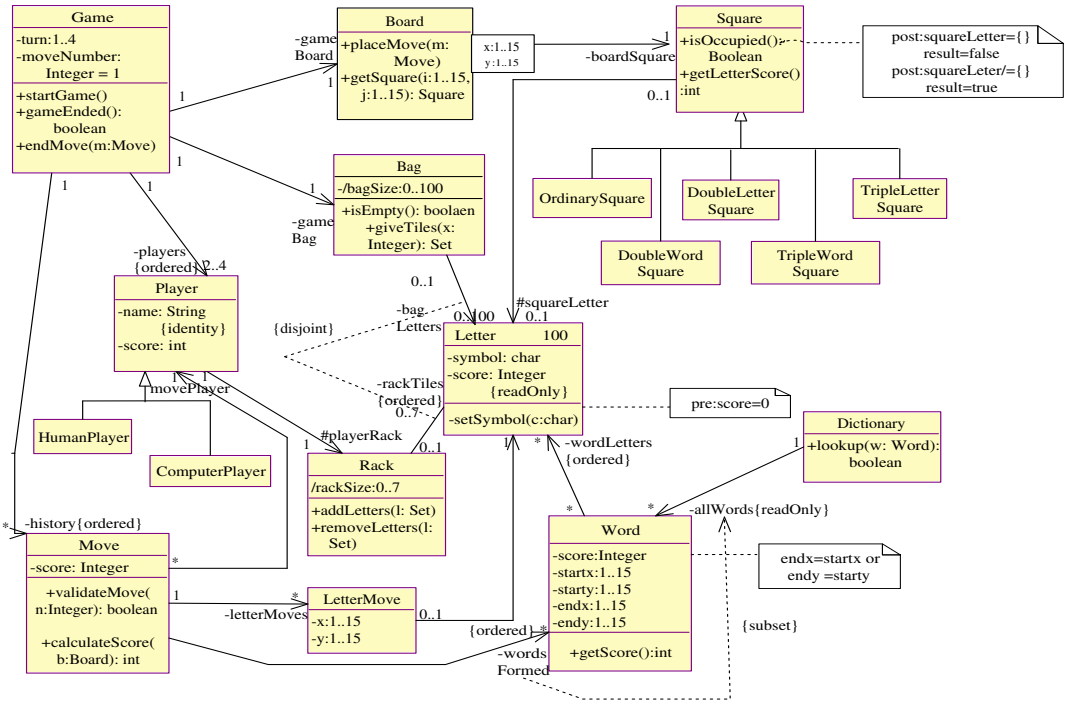



Figure 6.19: Scrabble Java PSM

Regarding rules of transformation, we get our PSM Scrabble class diagram in Figure 6.19.

Note that the role name playerRack and squareLetter have protected types since these two role names are attributes belonging to the super classes, so these features may be referenced directly from their subclasses.

**Producing Executable Code** After the platform-specific model for Java has been formed, an implementation for executable code can be produced. Table 6.3 maps PSM elements to Java Program Construct

Most of the elements in a class diagram can be matched directly to Java construct except for different names, for example the readOnly attribute are replaced by final attribute in Java. The last three items are much more difficult to be translated.

The identity attribute requires that no two object of the same class have the same attributes. When adding a new value to an identity attribute we should examine if the value has already exist, if so, the java constructor can not add the value. Same happens on the modification operation.

An association can be matched to an attributes of a class. For an association direct from A to B, the role name at B end of the association can be matched to an attribute of the class A. If the multiplicity at B end of the association is one then the corresponding attribute has type B. other multiplicity represent the attribute has the type of collection of type B.

For the collection type of B, cases are divided based on the multiplicity and ordering,

| Class Diagram Element | Java Program Construct |
|---|---|
| class | class |
| abstract class | abstract class |
| interface | interface |
| guarded class | synchronised class |
| leaf class | final class |
| utility class | static class |
| inheritance | extends |
| interface inheritance | implements |
| static attribute | static (class) attribute |
| non-static attribute | instance attribute |
| readOnly attribute | final attribute |
| abstract operation | abstract method |
| static operation | static method |
| non-static operation | instance method |
| guarded operation | synchronised method |
| identity attribute | no direct translation |
| association | attribute of class/collection type |
| constraint | no direct translation |

Table 6.3: Mapping UML elements to Java

reference to Table 6.4, suppose association has role name br at B end, we conclude that an ordered br with multiplicity from 0 to N correspond to Java array of size N, an ordered br with multiplicity * correspond to Java List, an unordered br with multiplicity * correspond to Java Set, a single br with multiplicity 1 and a qualifier at the opposite end correspond to Java Map.

| Association from class A to B | Declaration in Java class for A |
|---|---|
| {ordered} br 0..N — B | Java array B[] br = new B[N]; |
| {ordered} br * — B | Java List List br = new ArrayList(); //of B |
| br * — B | Java Set Set br = new Set(); //of B |
| x:T — br 1 — B | Java Map Map br = new HashMap();// T-->B |

Table 6.4: Correspondence of associations and program data

According to these rules, the Java definition of Scrabble game is formed.

```
public class Game {
private Board gameBoard;
private Player[] players = new Player[4];
private List history = new ArrayList();
private Bag gameBag;
```

```
    private int turn;
    private int moveNumber = 1;
    public void startGame()
    public Boolean gameEnded()
    public void endMove(Move m)
}
public class Board {
    private Square[][] boardSquare = new Square[15][15];
    public void placeMove(Move m)
    public Square getSquare(int i, int j)
}
public abstract class Square {
    private Letter squareLetter;
    public Boolean isOccupied()
    public int getLetterScore()
}
```

Transforming constraints of class diagram to executable code is the most difficult part in producing executable code in Java.

Because constraints mainly describe the preconditions and postconditions of operations, invariants of classes and properties of associations such as {ordered}, {subset}, the properties expressed by these constraints should be realized by algorithms using proper data structure. Normally the realization can not be totally automated.

Based on the defined attribute types and algorithms, Java code can be synthesized according to the code generation strategies and a proper data structure should be selected, which is out of the scope of this paper.

## 6.6 Conclusion

This paper only scratches the surface of stepwise object-oriented refinement by comparing different refinement approaches and combining them at Scrabble game development process. Research has been done in the area of object-oriented refinement with formal method [8]. The idea is to extend object-oriented refinement calculus to support a variety of object-oriented programming styles with modular reasoning. The reason why the informal method of object-oriented refinement is still popular now is because it has a good comprehensibility which makes it easy to communicate about development ideas between software developers and programmers, or even between customers and software developers.

## Bibliography

[1] *http://www.popkin.com/products/system_architect.htm.*

[2] *http://www.sepher.nl/why_uml.html*.

[3] G. Perez C. Pons and R-D Kutsche. Revealing undercover refinement in uml modeling. Sep. 2004.

[4] F. Erasmy and E. Sekerinski. Stepwise refinement of control software - a case study using raise. *Springer-Verlag*, pages 547–566, 1994.

[5] I. Jacobson G. Booch, J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[6] K. Lano. *Design for change: advanced system design with Java, UML and MDA*. Butterworth-Heinemann, 2005.

[7] E. Sekerinski and K. Sere. *Program Development by Refinement - Case Studies Using the B Method*. 1998.

[8] M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, April 1992.

[9] M. Boggs W. Boggs. *Mastering UML with Rational Rose*. Butterworth-Heinemann, 2002.

[10] N. Wirth. Program development by stepwise refinement. *Association for Computing Machinery*, 14(4), April 1971.

# Chapter 7

# Zhuo Zheng: Comparison of Modularization Techniques

In this paper, we consider three different techniques of modularization: traditional modular structure, object-oriented structure and aspect-oriented structure respectively, by taking a small-size program — binary search tree algorithm as an example. We make a fair comparison among these approaches, and comment on the advantages and drawbacks with respect to each approach. The analysis of the sample code serves as a case study, from which the modularization based on these approaches is demonstrated.

## 7.1 Introduction

The modularization of large complex software system has long been advocated, since it facilitates the software development and maintenance to a tremendous extent. By definition, the main task of the modularization is to decompose a large piece of system into some smaller and easy-to-manipulate units, thereby greatly improving the flexibility and comprehensibility of the overall structure of the original system. In the Parnas' seminal paper [6], "On the Criteria To Be Used in Decomposing Systems into Modules" that was published in Communications of the ACM in 1972, some clear guidelines on how to split up the software system into modules were stated. In particular, the author argued that software modularization adhere to the principles of changeability, independent work as well as comprehensibility, among others. The criteria serve as a cornerstone for the future development of the modularization techniques. However, one question arises at this point: what are the modularization techniques for which those criteria could be applied? Alternatively, we may ask ourselves if these criteria take the same effect when we modularize different systems in completely different scenarios. In this paper, we intend to address the above question(s) by demonstrating three distinct modularization techniques. We also take a concrete small-size example (binary tree search algorithm) to illustrate our points. We analyze different features of these techniques by implementing the example in different ways.

## 7.2 Modularization Techniques

### Procedural Programming Modularization

When it comes to the procedural (structural) programming, we can not neglect C programming language. However, even though C was designed to deal with large-scale system programming, it had come into being before the subject of the software modularization matured. Thus creating and modularizing in C is time-consuming and more often than not requires sophisticated knowledge of the language itself and the compiler. As far as general procedural programming is concerned, functions are the main and lowest level of modularity, which suffer from some practical defects: they can not be separately compiled unless they are in different files, and they request the access of variables and global data types.

The earliest modules were separate source code files containing functions (with returned data) or procedures (without returned data), which could be compiled separately into object code, then the linker would merge the object code files. Later programming languages allowed various extensions on existing modules, such as permission to be nested, i.e., modules declared inside other modules, to run simultaneously, or multiple instances of the same basic module to be made, etc. Even though there are different levels of modularity in C language, an individual module is an incomplete program, which can contain the following elements:

- Type-constant declarations
- Global variables
- Functions

Therefore, we can be able to group the modules into several categories:

- Declaration only modules – contains only type-constant declarations;
- Global variables only modules – contains only global variables;
- Abstract data type modules – contains type-constant declarations and functions;

The precise way of modularizing the system is dependent upon the situation where the system is implemented. In general, by the combination of the above fundamental modules, we would be able to achieve the modularization in C language in a proper manner. However, there is an important restriction on modules in C we should bear in mind that we can not permit nested modules, like Ada, Modula-2 do. This is probably due to the fact that we cannot have nested global spaces, since there is only one global space.

### Object-Oriented Modularization

Modularization has many designations in the object-oriented (OO) world although it usually corresponds to system partitioning in groups (clusters) of classes [4]. The concept of

class is essential in the object-oriented programming as a paradigm in computing that has been developed over many years till now. A class acts as an abstraction of data, and data abstraction is a principle of separating data from the code that manipulates it, and allows the modification of data structures as well as primitive functions that directly manipulate it. These methods give a common interface to other program that needs to access or modify the data such that the calling code does not have to know how the data is organized. The information is hidden from the outside world in this way. Therefore, a module is used to implement an abstraction of data, and data can be encapsulated in a module which allows access to the data only through a defined interface, thus preventing undesirable or unknown modification of the data. The change of the data structure or implementation detail is limited to the module that directly handles the abstract data type. No other modules in the system need to be affected. We gain a direct benefit by injecting the concept of OO into our implementation: the resulting program may have good scalability and maintainability.

In the C++ programming language, namespaces support modularization by providing a mechanism for expressing logical grouping [8]. In Java, packages are an important modularization mechanism, which may contain any combination of interfaces (defining types) and implementations (classes) [3]. In Smalltalk development environment, such as Envy, there is modularization support also through the use of packages. During the runtime, those packages are loaded in a specific order, starting by the kernel one. In the Delphi language, an extension of Pascal for object-oriented programming, modules are called units [1]. Eiffel language empathizes a modularization abstraction, the cluster, which is the basis of Meyer's cluster model [5]. In OMT, the modularization unit is called subsystem [7].

## Aspect-Oriented Modularization

Modularization features offered by programming languages are normally limited to what we call the tyranny of the dominant decomposition: they allow separation of only certain kinds of concerns (e.g. data in object-oriented approaches). Other important kinds of concerns cut across the dominant modules, and cannot be encapsulated effectively [9]. Aspect-oriented programming has introduced new design perspectives that permit the superimpositions of different abstraction models on top of one another [10]. Aspect-oriented software development is an emerging area that is aimed at modularizing design concerns that cut across parts or all of the system of interest. Some typical examples of these concerns include synchronization, debugging, logging, memory management, security etc. The modularization of such concerns allows for higher-degrees of reuse throughout the software system and eases the maintainability of the system [10].

One option to realize the aspect-oriented modularization is through AspectJ language, which is an extension of the Java programming language. It provides clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols.

## 7.3 Comparison

### Procedural Programming VS Object-Oriented

From what has been discussed so far, we know that procedural programming and object-oriented programming are two different modularization strategies. The former structure of the program is comprised of modules that represent the subprograms as functions or procedures. Therefore, the modularization involves the identification, within the overall project, of distinct subtasks and isolating them in algorithms. The latter modularization is engaged in modelling the real world with abstract objects as classes, which encapsulate the object's state and behavior. However, they do have the similar goal: separating the software system into modules in such a way that independent development can be promoted and the "highly cohesive, loosely coupled " modularization principle be achieved.

To make our argument more solidly grounded, let's take an example implemented in these two different strategies. Suppose that we would like to perform a series of operations (insert a node, inorder traversal, preorder traversal, postorder traversal, delete a node, search a node, realize a threaded tree etc) defined on a binary search tree. In order to keep our example concise yet still convincing, we just take a fragment of code that could be used to exemplify the points in the best way. The essence of the program is to use the recursive function calls to realize the insertion and traversal functionalities. To go into the details of this program is beyond the scope of this paper, what we would do instead is to state the points we consider relevant to our topic.

```
Binary Search Tree Algorithm
-------------------------------------------------------------
struct btreenode {
struct btreenode *leftchild ;

int data ;

struct btreenode *rightchild ;} ;

-----------------------
main( ) {

/*Initialization */
      ...

/* Data Input */
      ...

insert(btnode, num);
```

```
printinorder(btnode);

printpostorder(btnode);

...

}
-----------------------
/* inserts a new node in a binary search tree */

insert ( struct btreenode **btnode, int number ) {


  if ( *btnode == NULL )
     {
       *btnode = malloc ( sizeof ( struct btreenode ) );
       ( *btnode ) ->leftchild = NULL ;
       ( *btnode ) -> data =number ;
       ( *btnode ) -> rightchild = NULL ;
       return ;
     }
  else
     {
       if ( num < ( *btnode ) -> data )
           insert ( &( ( *btnode ) -> leftchild ), number ) ;
       else
           insert (&( ( *btnode ) -> rightchild ), number ) ;
     }
  return ;
}

printinorder ( struct btreenode *btnode ) {
   if ( btnode!= NULL ) {
       inorder ( btnode -> leftchild ) ;
       printf ( "%d ", btnode-> data ) ;
       inorder ( btnode -> rightchild ) ;
}
   else
       return ;
}
...
...
```

Figure 1 Procedural Programming Implementation

Just to take a quick look at this code, we can get a rough idea that in the procedural programming context, the main() function calls a sequence of subroutines such as insert(), printinorder(), printpreorder(), printpostorder(), delete() etc to realize what the algorithm is expected to. Each subroutine accepts some parameter, performs the computation, and then either returns or not returns a certain result.

As for the object-oriented programming implemented in Java, everything can be "wrapped up" in a class (Note: we intentionally use the class name "IntegerSet" instead of "Binary-Tree" for the reason that will be explained later). The encapsulation of the implementation details is one important difference between the procedural programming and the object-oriented programming. It does not matter what the class below is named, i.e., it can be arbitrarily entitled, and not necessarily "IntegerSet", even though we are not really implementing an integer set inside the body of our code. It is our understanding that only when we place the two programming strategies under the same context can we be able to come up a fair comparison.

```
-----------------------------------------------------------------
public class IntegerSet {

private btreenode root = null;

private static class btreenode {
    btreenode leftchild;
    int data;
    btreenode rightchild;


    btreenode(int newData) {
      leftchild = null;
      rightchild = null;
      data = newData;
    }
  }

private btreenode insert(btreenode btnode, int data) {
    if (btnode==null) {
      btnode = new btreenode(data);
    }
    else {
      if (data <= btnode.data) {
        btnode.leftchild = insert(btnode.leftchild, data);
```

```
      }
      else {
        btnode.rightchild = insert(btnode.rightchild, data);
      }
    }

    return(btnode);
  }

private void printpostorder(btreenode btnode) {
  if (btnode == null)
     return;
  printpostorder(btnode.leftchild);
  printpostorder(btnode.rightchild);
  System.out.print(btnode.data + "  ");
  }
  ...
  ...

public void insert (int data){
  root = insert(root, data);
}

public void printpostorder() {
 printpostorder(root);
 System.out.println();
}
  ...
  ...
}
```

Figure 2 Object-oriented Programming Implementation

As it is clear from the code that the most obvious difference between the two representations is that procedural programming deals with the algorithms whereas the object-oriented programming deals with the abstract objects (classes). It should be noted that the class architecture is a higher level view than the algorithm in that it illustrates the relationships between classes and the encapsulation of state and behavior within each class, while the algorithmic view does not manifest such characteristics: all algorithms are only recipes of what to do, not the properties of objects. In our example, we deliberately use a class name "IntegerSet" in disguise of the real "detail" of the BinaryTree implementation. We can see that the IntegerSet class has a subclass btreenode as its attribute (data), and this attribute is encapsulated with the IntegerSet class. Apparently, the processing of any operations within the IntegerSet class are completely independent of other classes in the whole program (say if

we have another module which defines a UserInterface class). Each class is highly cohesive, i.e. contains everything necessary to describe objects of that type. The fact that the data and operations are encapsulated makes it easy to modify the modules than in the procedural programming. We can think of a scenario that if the UserInterface class has been replaced by another class which presents a totally distinct interface, the IntegerSet can still be kept intact without any change, as long as the new interface interacts with this class in the same way.

We recognize that those concrete operations such as insert(), printpostorder(), printpostorder(), etc, are similar in the OO context to that in the procedural programming, except that we have a pair of procedures for each of these operations, one "private" type, one "public" type. The purpose of doing so is to hide the real detail of our implementation of the binary tree algorithm. Anyone standing outside the class does not know what is going on inside the class. Furthermore, in OO environment, these operations all belong to the class IntegerSet, i.e. they are the behaviors of that class. In order to insert a new node to the binary tree, we just need to instantiate the class with an object and send a message to the object to specify its values. On the other hand, in the procedural programming, the operations are not associated with any class, but the program itself, whereas in the OO, the operations as insert(), printpostorder() etc do NOT have an input argument that belongs to the btreenode class! This reflects the fact that these operations do not need supply data because the data (a node that belongs to the btreenode class) is automatically associated with the operations as all data and operations are encapsulated together within the class. Whenever we want to call these methods, we just need to call the public procedure with the private counterpart hidden, i.e., to use a module, we only need to know a small and well-specified amount of information about how to access the working code. This is also the idea that object-oriented programming encourages.

## Aspect-Oriented VS Object-Oriented

Sometimes, when we maintain or upgrade a software system, or try to understand its behavior in some way, our concern is not necessarily focused on a particular module or even a set of modules, but with an "aspect" of the entire application. However, the desired aspect may spread or distribute all over the system, making it extremely difficult to identify or modify, especially for the large-scale system. In many cases, when the modifications are requested to be made "on site", this may not even possible since the source code is not available. While modularization offers its advantage of making big problems tractable, it has to tackle such problem like some issues need to be managed over multiple modules. In addition to the general concerns, there are many that are specific to a particular application. These cross-module concerns are normally called "program aspects".

Aspect-oriented programming is intended to address these concerns by identifying aspects and coding "cutpoints" for each aspect. In an explanatory article by Ramnivas Ladded, we have found the following statement: Object-oriented techniques for implementing such cross-cutting concerns result in systems that are invasive to implement, tough to understand, and

difficult to evolve. The new aspect-oriented programming methodology facilitates modularization of crosscutting concerns. Besides, we are now seeing that many requirements do not decompose neatly into behavior centered on a single locus. Object technology has difficulty localizing concerns involving global constraints and pandemic behaviors and applying domain-specific knowledge [2].

Similarly, we will see that what the aspect-oriented program with the binary tree algorithm is going to be like:

```
--------------------------------
pointcut login():
       call (void IntegerSet.insert(int)) ||
       call (void IntegerSet.delete(int)) ||
       call (void IntegerSet.printpostorder(int)) ||
       call (void IntegerSet.printinorder())||
       ...;
/* In this step, we just pick out the join points of the program
*/

--------------------------------
before() : login() {
      System.out.println("Access the binary tree");
}

after() :  login() {
      System.out.println("Operation has been done");
}

/* In this step, we implement crosscutting behavior */

--------------------------------
aspect logging {
       outputStream logStream = system.err;

       before(): login() {
         logStream.println("Access the binary tree");
       }
}

/* In this step, we provide a means to express the concern in one
place */
--------------------------------
```

Note: we have just used a simplified example. Those concerns could be crosscutting, and fragmented over many classes.

Personally, my critique on aspect-oriented programming is that it sometimes makes the program less easy to comprehend. We are likely to grasp the whole picture of certain things only after we have followed the pattern of learning from the beginning to the end, i.e. it is better that every part is self-contained and stays where it should be. It is undeniable that when we have moved the "concerns" into one place, it is easier for the program developers to maintain the whole system or deal with a particular issue related to that concern, but it is not necessarily so for the readers to understand the program in the fastest manner.

## 7.4 Conclusions

We have presented three different ways of modularizing the program. The methodology has been illustrated in a way that these techniques can be extended to any kind of large-scale software system. We have also made a fair comparison of these approaches based on the specific binary tree algorithm implemented in these methods.

In a nutshell, the modularization of procedural programming can be realized via its procedures or functions. The object-oriented programming modularizes by modelling the system and representing modules with classes. In the aspect-oriented scenario, we are more concerned with those crosscutting aspects of the program, which we would like to separate them and aggregate them into a new module. In my opinion, the difference of procedural and object-oriented modularization is more of structural disparity, i.e. the way the modules are represented, whereas for aspect-oriented, it emerges in response to the needs of dealing with special issues in a system, therefore, the purpose of modularization is to strengthen this target somehow. Since aspect-oriented programming has not matured yet, it deserves further in-depth research.

# Bibliography

[1] M. Cantu. *Mastering Delphi 2 for Windows 95/NT: Sybex*, 1996.

[2] T. Elrad, R.E. Filman, and A. Bader. Aspect-Oriented Programming, *Communications of the ACM*, 44, No. 10, 2001

[3] J. Gosling and F. Yellin. *The Java Application Programming Interface*, Massachusetts USA: Addison-Wesley Publishing Company, 1997.

[4] M. Goulao. A Merit Factor Driven Approach to the Modularization of Object-Oriented Systems, *L'Objet*, 7, No.4, 2001.

[5] B. Meyer. *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1997.

[6] D.L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15, No.12, pp:1053-1058, 1972.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen. *Object-Oriented Modelling and Design*, Englewood Cliffs, NJ, EVA: Prentice Hall, 1991.

[8] B. Stroustrup. *The C++ Programming Language*, 3rd ed. Reading. Massachusetts USA: Addison-Wesley Publishing Company, 1997.

[9] Workshops on Aspect-Oriented Programming at ECOOP'97,'98 and '99 and ICSE'98.

[10] C. Zhang and H.-A. Jacobsen. A Prism for Research in Software Modularization Through Aspect Mining. *Technical Communication*, University of Toronto, 2003.

# Chapter 8

# Ning Liu: A Survey of Verification of Floating-Point Arithmetic

Formal verification of software is used in many places, especially critical situations, to ensure the correctness and robustness of the system. However, in the presence of floating-point numbers, this approach becomes very difficult due to the difference between finite-precision floating-point numbers and mathematical real numbers. In this project, I perform a survey on various techniques used in verifying programs involving float-point numbers. In particular, a case study of proving the correctness of square root operation is conducted, and a comparison of informal testing/proving and formal method is performed.

## 8.1 Introduction

While verification of integer programs can be done using pre/postconditions and guarded commands, verifying programs involving floating-point numbers is less obvious because of the nature of inaccuracy and nondeterminism of floating-point numbers. In this survey, I first give a brief introduction to floating-point arithmetic and some of its main properties. Then several difficulties of verifying floating-point numbers are discussed. In the third part of the paper, we look at how to use a mix of various mathematical tools to verify the correctness of iterative floating-point square root algorithm. Some recent approaches in result verification of numerical software is also shown. Finally, we will compare the technique of formal verification with testing in case of floating-point numbers, and give a conclusion on how to ensure the correctness of a program containing floating-point numbers.

### Floating-point Number System

First of all, let's look at what floating-point numbers are. A floating-point number system $F \subset R$ is a subset of the real numbers whose elements are represented as a concatenation of a sign bit, and an $M$-bit exponent field, and an $N$-bit significant field. Mathematically,

$f = \sigma \cdot s \cdot \beta^e$, where $\sigma = \pm 1$; $s$, the mantissa, is an integer satisfying $0 \leq s \leq \beta^N - 1$; and the exponent $e$ is ranging from $-\beta^{M-1} + 2$ to $\beta^{M-1} - 1$. Different formats of floating-point numbers are supported in various machines, among which most commonly used are IEEE-754 Standard single precision ($\beta = 2, M = 8, N = 24$) and double precision ($\beta = 2, M = 12, N = 54$).

Following from its definition, it is easy to see that a floating-point number system is a finite set of unevenly-spaced points along the real axis. Using this finite set of numbers to represent infinite many real numbers unpreventablely leads to cases of inaccuracy. Therefore, the concept of rounding is introduced to deal with the situation where the result of computation falls into the gap between two consecutive floating-point numbers. The IEEE-754 Standard enacts four kinds of rounding modes, namely round to nearest, to positive and negative infinity, and towards zero.

Because of the limit of precision, floating-point arithmetic is not absolutely accurate in nature. The IEEE-754 Standard requires that the result of a floating-point operation be calculated as if in infinite precision, and then rounded to one of the two nearest floating-point numbers of the specified precision that surround the exact result. This requirement specifies that to verify a program is correct, one needs to show the computed result and exact result, before rounding, lie within an interval such that through rounding they end up with the same floating-point number.

Several important concepts and properties of floating-point numbers also need to be mentioned before we look at the techniques of verification. Two commonly used measurements in evaluating the accuracy of computing are absolute and relative errors. In floating-point system, absolute error is usually expressed in terms of *unit of last place* or *ulp*. It is defined as the smallest number $\epsilon$ in the system such that $1 + \epsilon > 1$, and has a value of $\beta^{-N+1}$. Due to nonuniform distribution of floating-point number in the axis, it is difficult to specify a requirement using absolute errors. Instead, relative error can eliminate the problems caused by different exponents. So most algorithms try to limit the relative errors generated by the computation.

## Difficulties of Verifying Floating-Point Numbers

To verify a software, first we need a complete and precise specification of it. However, in many cases, for example the random number generator, to specify the exact behaviors of numerical programs is impossible. Secondly, the performance of programs varies on specific sets of data. The following example is due to [7]:

```
fb=f(b);
while abs(a - b) > tol
   mid = (a + b)/2.0;
   fmid = f(mid);
```

```
if fb*fmid > 0
   b = mid; fb = fmid;
else
   a = mid;
```

The above program computes a zero of function $f(x)$ using the fact that the sign of function value changes at the point of the root of $f(x)$. The program terminates and returns a good approximation of a root of $f(x)$ except when the tolerance is chosen to be even smaller than the distance between two consecutive floating-point numbers surround the exact solution, in which case the "mid" point calculated in the program will be rounded back to one of the two representable points, so the loop condition is always satisfied and the program never terminates. There is another issue when conventional mathematical rules may break. For example:

```
eps = 1.0;
t = 0;
while (1.0 + eps) > 1.0
   eps = eps/2.0;
   t = t + 1;
```

In the real number system, this loop will never terminate since *eps* is always positive. But in case of floating-point, when $1.0 + eps$ is too close to 1.0, it will be rounded down and the loop condition is broken.

## 8.2 Verifying Floating-Point Square Root Algorithm

After introducing the general concepts and properties of floating-point arithmetic, we now look at the method of proving the correctness of a floating-point square root algorithm. As one of the basic operations in computing, most of machines today implement square root in hardware. Recently research has been done to develop software-based implementation of operations such as square root and division. Several advantages can be achieved over hardware approach, such as ability of pipelining, changeability, and reduced size of chipset. As a result, to ensure that the algorithm always produces correct results under IEEE requirement becomes one of the major tasks. In this survey, we study the techniques used by Marius Cornea-Hasegan [1] in proving of square root.

The algorithm chosen for calculating the square root is a Newton-type iteration on the function $f(x) = \frac{1}{x^2} - a$, where the next approximation point is obtained as $x_{i+1} = (x_i - \frac{f(x_i)}{f'(x_i)})(1 + e_i)$. To show that after a sufficient number of iterations, $x_i$ becomes the IEEE-correct result of square root of $a$, Cornea-Hasegan first establishes a criterion to determine "difficult cases of rounding" in the square root operation; and then an "exclusion zone" around the floating-point number surrounding the exact result is chosen to cover all "difficult cases"; next, prove

that all computed results which fall into the exception zone are IEEE correct except difficult cases; finally, explicitly verify the correctness for the special points determined in step 2.

Difficult cases for rounding correspond to different rounding modes. Here we choose rounding-to-nearest($rn$) as an example, and other situation can be derived in a similar way. The difficult cases for $rn$ are defined as number a's such that $\sqrt{a}$ is different from, but very close to, a midpoint between two consecutive floating-point numbers. The following lemma helps to determine difficult cases:

**Lemma 1** *(a) Let $a \in F$, $a = s \cdot 2^e$. It can be rewritten as*

$$y = \begin{cases} A \cdot 2^{e-2N+2} & : \quad e = 2k \\ A \cdot 2^{e-2N+1} & : \quad e = 2k+1 \end{cases}$$

*Where $A \in [2^{2N-2}, 2^{2N-1})$, $A \equiv 0 \bmod 2^{N-1}$ for $e = 2k$, and $A \in [2^{2N-1}, 2^{2N})$, $A \equiv 0 \bmod 2^N$ for $e = 2k+1$.*

*(b) $\sqrt{a}$ is representable in F, if and only if $\sqrt{A} \in [2^{2N-2}, 2^{2N-1}) \cap Z$ for $e = 2k$, or $\sqrt{A} \in [2^{2N-1}, 2^{2N}) \cap Z$ for $e = 2k+1$.*

*(c) $\sqrt{a}$ is a midpoint of two consecutive floating-point numbers in F if and only if $\sqrt{A}$ is an integer$+\frac{1}{2}$, and $\sqrt{A} \in [2^{N-1}, 2^n)$.*

*(d) If $a \in F_N$ and $\sqrt{a} \notin F_N$, then $\sqrt{a} \notin F_{N+1}$ ($F_N$ represents the set of floating point numbers that can be represented using N digits of significant).*

Because of the scope of this survey, the proof of the lemma is not included (so does the other theorems). By the definition of difficult case above, if $\sqrt{a} \notin F_N$, then its distance to a midpoint of two floating-point numbers can be transformed to the distance between $\sqrt{A}$ and an N-bit integer plus $\frac{1}{2}$ according to ***Lemma1(c)***. To find out this, we need to solve the equation:

$$(2^{N-1} + k + \frac{1}{2})^2 = A + \frac{1}{4} + \sigma$$

for values of $\sigma \in Z$ and $0 \leq k \leq 2^{N-1} - 1$, $k \in Z$. Two cases arise here: if the exponent $e$ of $a$ is even, the previous equation can be rewritten as

$$(2^{N-1} + k + \frac{1}{2})^2 = 2^{2N-2} + m \cdot 2^{N-1} + \frac{1}{4} + \sigma$$

if the exponent $e$ of $a$ is odd, the previous equation can be rewritten as

$$(2^{N-1} + k + \frac{1}{2})^2 = 2^{2N-1} + m \cdot 2^N + \frac{1}{4} + \sigma$$

for some integer $m$, $0 \leq m \leq 2^{N-1}-1$. Solving the diophantine equations, we can find difficult values of $A$ for each possible value of $\sigma$. For example, if we choose $\sigma = 0$, $A = 2^{2N-2} + 2^{N-1}$

is at a distance of $\frac{1}{4}$ from $(2^{N-1}+\frac{1}{2})^2$, which gives a difficult case of rounding to nearest.

Having seen how to determine the difficult cases of rounding, we move on to two main theorems that are used to prove the IEEE correctness of square root.

**Theorem 1** *Let $a \in F_N$, $a > 0$, $a = \sigma \cdot s \cdot 2^e$. If $\sqrt{a}$ is not representable in $F_N$, and $A$ is determined by Lemma 1, then for any integer $K \in [2^{N-1}, 2^N)$, and for any $f \in F_N$:*
*(a) The distance $w_{\sqrt{A}}$ between $\sqrt{A}$ and $K$ satisfies*

$$w_{\sqrt{A}} = |\sqrt{A} - K| > \frac{1}{2^{N+1}}$$

*(b) The distance$w_{\sqrt{a}}$ between $\sqrt{a}$ and $f$ satisfies*

$$w_{\sqrt{a}} = |\sqrt{a} - f| > 2^{\frac{e}{2}-2N}, e = 2k$$

*or*

$$w_{\sqrt{a}} = |\sqrt{a} - f| > 2^{\frac{e}{2}-2N-\frac{1}{2}}, e = 2k+1$$

Theorem 1 basically states that if $\sqrt{a}$ is not representable as a floating-point number using $N$ significant bits, then there is an exclusion zone of known minimum width around any floating point number, within which $\sqrt{a}$ can't exist. The minimum distance between $\sqrt{a}$ and $f$, or equivalently $\sqrt{A}$ and $F$, is determined by the distance between $A$ and $F^2$ instead.

**Theorem 2** *Let $a \in F_N$, $a > 0$, $a = \sigma \cdot s \cdot 2^e$. If $\sqrt{a}$ is not representable in $F_N$, and $A$ is determined by Lemma 1, then for any midpoint $m$ of two consecutive floating-point numbers in $F_N$, and any midpoint $M$ between two consecutive integers in $[2^{N-1}, 2^N)$:*
*(a) The distance $w_{\sqrt{A}'}$ between $\sqrt{A}$ and $M$ satisfies*

$$w_{\sqrt{A}'} = |\sqrt{A} - M| > \frac{1}{2^{N+3}}$$

*(b) The distance$w_{\sqrt{a}'}$ between $\sqrt{a}'$ and $m$ satisfies*

$$w_{\sqrt{a}'} = |\sqrt{a} - m| > 2^{\frac{e}{2}-2N-2}, e = 2k$$

*or*

$$w_{\sqrt{a}'} = |\sqrt{a} - m| > 2^{\frac{e}{2}-2N-\frac{5}{2}}, e = 2k+1$$

Similar to Theorem 1, Theorem 2 specifies that if $\sqrt{a}$ is not representable with $N$-bit significant, exclusion zones exist around any midpoint of two consecutive floating-point numbers, whose width can be determined by the distance between $A$ and $M^2$.

In order to prove the IEEE correctness of the square root algorithm, we need to show the computed result and exact result round to the same floating-point number. This can be further reduced to show that the computed result $\sqrt{a}_f$ before rounding is closer to the exact result $\sqrt{a}$ than half of the minimum width of any exclusion zone determined by Theorem 1 and Theorem 2. In practice, the inequalities to be verified are

$$|\sqrt{a}_f - \sqrt{a}| \leq (w_{\sqrt{a}})_{min}$$

$$|\sqrt{a}_f - \sqrt{a}| \leq (w_{\sqrt{a}'})_{min}$$

In his paper, Cornea-Hasegan summarizes following sequence of verification steps:

**Step 1** *Evaluate the relative error $\epsilon$ of the final result before rounding*

$$\sqrt{a}_f = \sqrt{a} \cdot (1 + \epsilon)$$

*where $|\epsilon| \leq \frac{1}{2} ulp$*

**Step 2** *Evaluate $|\sqrt{a}_f - \sqrt{a}| = |\sqrt{a} \cdot \epsilon| \leq \sqrt{a} \cdot \frac{1}{2} ulp$ and determine the minimum widths $w_{\sqrt{a}}$ and $w_{\sqrt{a}'}$.*

**Step 3** *For Theorem 1 and 2, determine sufficient number of difficult cases for rounding, to allow augmenting the widths of the exclusion zones to the values determined in Step 2. Except possibly difficult cases, the iterative algorithm for calculating $\sqrt{a}$ is IEEE correct.*

**Step 4** *Verify the result for difficult rounding cases directly. Once this is done, the whole algorithm is proven to be IEEE correct.*

## 8.3 Static Analysis-Based Validation of Floating-Point Computation

The method of proving IEEE correctness of iterative square root algorithm described in the previous section is not suitable for general floating-point algorithms. One reason is that the complexity of verification could simply make it infeasible to be done. Another consideration is that unlike basic operations such as addition, multiplication, square root, which are required to be accurate to the last possible digit, most, probably all, applications do not need such accuracy because issues like lack of stable algorithms or inaccurate measurement of input data can easily introduce some unavoidable errors. Instead, we want to verify that the result of computation does not grow too far away from the exact result and the degree of accuracy depends specific requirements.

One of the recent researches on this area is done by Sylvie Putot, together with Eric Goubault, and Matthieu Martel [2] in France. They introduce a static analysis-based validation method for floating-point computation. This approach is based on studying the

propagation of rounding errors during the intermediate steps and aims to identify the operations responsible for main losses of accuracy rather than compute the estimation of real result for given inputs.

Program analysis is based on the semantics. One possible semantics, called concrete semantics, is to "decompose the error between the results of the same computation achieved respectively with floating-point and real numbers in a sum of error terms corresponding to the elementary operations of this computation" [2]. For example, in a system of floating-point numbers with four digits of significant and base ten, two intermediate results $a = 621.3 + 0.055_{e_{l1}}$ and $b = 1.287 + 0.00055_{e_{l12}}$ are not computed exactly with error $0.055_{e_{l1}}$ and $0.00055_{e_{l12}}$ respectively. Considering the product of $a$ and $b$, the exact result is $a \times b = 799.6131$. However, due to the limit of significant, the result has to be round to 799.6, assuming round to nearest mode is used. A rounding error $a_f \times b_f - c_f = 0.131$ is committed. If we take the original error of $a$ and $b$ into account and consider first order error analysis only, we get

$$a_f \times b_f = c_f + 0.070785_{e_{l1}} + 0.341715_{e_{l2}} + 0.0131_{e_{l3}} + e_{high}$$

The analysis is quite obvious in this simple example, but would become much more difficult and tedious to establish for more complex programs.

Although the concrete semantics reflects the natural of computation, it can not be used for the analyzer because the errors are real numbers which are not always representable in floating-point system. Instead an abstract semantics is derived in a way that over-estimation of values and errors are computed using intervals. These intervals allow to consider set of inputs and make it a implementable version of concrete semantics. Considering the above example of multiplication again, the result will be written as

$$\begin{aligned} a_f \times b_f &= [c_f, c_f] + [0.070785, 0.070785]_{e_{l1}} + [0.341715, 0.341715]_{e_{l2}} + \\ &\quad [0.0131, 0.0131]_{e_{l3}} + e_{high} \end{aligned}$$

If we consider one step further that input $a$ is computed using intervals, for example $a' = [610, 630] + [0.055, 0.055]_{e_{l1}}$, and the result of multiplication would be

$$\begin{aligned} a'_f \times b_f &= [785.1, 810.8] + [0.070785, 0.070785]_{e_{l1}} + [0.3355, 0.3465]_{e_{l2}} + \\ &\quad [-0.05, 0.05]_{e_{l3}} + e_{high} \end{aligned}$$

Indeed, the results obtained using concrete semantics fall into each of the corresponding intervals. In general, the author use abstract semantics by interpreting operations over error series with interval coefficients, and outward rounding and possibly more precision for the propagation of existing errors.

A prototype implementation of the abstract interpretation is done in [3] with a graphic interface, showing lines of codes being analyzed, list of variables, a graph representation of error series related to selected variables, and some control variables. According to the author, this

program can be used for validation of critical embedded systems that are numerically simple but have fairly large sizes (up to 100000 lines of code). Future work needs to be done to handle numerically more complex programs.

## 8.4 Conclusion

So far we have discussed techniques for verifying floating-point numbers and seen two examples of application. However, as stated before, there are many limitations and difficulties to expand the techniques used in the examples to general cases of floating-point program because of complexity and other issues. Yet another observation we can make is that the verifications are all done focusing on computation errors. That is to make sure the result from each intermediate step is bounded. However, algorithmic error is seldom detected because it is very hard to keep a view of the whole picture and look into each step at the same time. Unfortunately, some numerical problems are very sensitive to the data. For many cases, even each step is verified to produce a result within a tight bound, the final output could still be very inaccurate. So, unlike integer programs, intermediate correctness doesn't imply total correctness.

On the other hand, specially chosen test sets can be used in combination with theoretical proofs to verify the correctness. These test sets can be artificially constructed to expose the potential problems for a specific floating-operation [5], or can be a collection of "standard" problems that are used as benchmarks in one application domain.

Another issue is that since floating-point number system is implemented differently in various programming languages and hardware platforms, it is essential that the verification and/or testing are performed in the case of changing environment. Examples can been found in [6].

As a conclusion, in this survey we gave a general background introduction of floating-point arithmetic, proposed some difficulties that arise in verifying floating-point programs. We then looked at two specific examples to show how the IEEE correctness of floating-point operation can be proved. Finally, some issues that need special attention are discussed. In general, verifying floating-point numbers is a extremely hard, sometimes even infeasible. Today, only basic floating-point operations such as addition, multiplication, square root and some numerically simple programs can be verified. No algorithm or tools available for more complex numerical softwares. In particular, the verification process usually involves complicated mathematical theorem proving and knowledge of the application domain.

# Bibliography

[1] Marius Cornea-Hasegan. *Proving the IEEE Correctness of Iterative Floating-Point Square Root, Divide, and Remainder Algorithms*. Intel Technology Journal, 1998

[2] Sylvie Putot, Eric Goubault, Matthieu Martel. *Static Analysis-based Validation of Floating-Point Computations*. Lecture Notes in Computer Science, Springer, Vol.2991/2004 206-225

[3] E. Goubault, M. Martel, and S. Puto. *Asserting the Precision of Floating-Point Computations: a Simple Abstrac Interpreter*. Lecture Notes in Computer Science, Springer, Vol.2305/2002 209-212

[4] S. Qiao. *CAS 703 Coursenotes*. McMaster University, 2004

[5] W. Kahan. *A Test for Correctness Rounded SQRT*. Stanford University, 1996

[6] W.Kahan, J.D. Darcy *How Java's Floating-Point Hurts Everyone Everywhere*. ACM Workshop on Java for High-Performance Network Computing, 1998

# Chapter 9

# Nima Dezhkam: Survey of Techniques for Reverse Engineering, Architecture and Design Recovery

In this paper, an overview of software reverse engineering and its taxonomy is presented. As major activities in reverse engineering, architecture and design recovery which are a means to increase the understandability, changeability, and maintainability of (legacy) software systems are discussed. Also, some different approaches to architecture and design recovery based on different techniques used, namely data mining, clustering, and scenario-based are presented. [1]

## 9.1 Introduction

Legacy software systems are mission critical systems that typically have been used for 10-15 years. These systems are considered to be problematic for several reasons such as difficulty in maintenance, improvement, expansion and integration with other software systems. These are due to lack of understanding of system because of imprecise documentation or design specification and non accessibility of the original designers and developers [1]. Changing nature of the environment in which most of legacy software systems work, leads to changes in requirements of the system from time to time. Since replacing these systems are usually very expensive, maintenance and advancing these systems in order to fit the changes are the chosen solution most of the times. "Reverse Engineering" is the common technique used for the process of maintaining, enhancing, or migrating legacy software systems. The origins of the term "reverse engineering" is in the analysis of hardware – where discovering designs from finished products is a common practice. In a landmark paper on the topic, M.G. Rekoff defines reverse engineering as "the process of developing a set of specifications for a complex

[1]The scenario-based design recovery part of this paper is taken from an ongoing research mainly done by myself which is a part of a paper co-written with Kamran Sartipi and Hossein Safyallah and submitted for WCRE 2005 conference.

hardware system by an orderly examination of specimens of that system" [2]. However, while in hardware usually the objective of reverse engineering is to duplicate the system, software reverse engineering's objective is mostly gaining a sufficient design-level understanding of the system to help maintenance activities and/or facilitate enhancements [2]. Software reverse engineering is typically performed using some tools, and different techniques and approaches are available in this regard. In the rest of this paper, we use the general term of "reverse engineering" to refer to "software reverse engineering".
In the area of reverse engineering, "software architecture and design recovery" are considered as major activities and their names are sometimes used interchangeably. It constitutes a major part of software maintenance phase and is applied with some specific goals, such as increasing understandability and changeability.

## 9.2 Reverse Engineering and Architecture recovery

Software architecture describes the building elements of a system, their interactions, patterns that guide their interactions, and constraints on these patterns. Software architecture recovery can be defined as extracting the description of the components of the system and the inter-relation between these components from a low-level software representation such as source code [3]. It is a key activity in supporting maintenance tasks such as re-engineering or re-structuring. In the following section, the taxonomy of reverse engineering introduced in [2] is explained.

### Taxonomy of software reverse engineering

There are some different terms that are used in the reverse engineering area. Figure 9.1 shows these terms and their relations to each other and to three major phases – namely requirements, design, and implementation – in life-cycle model of a software system. The model can be of type of the traditional waterfall, spiral or any other model. Also, the software system can be a single program or code fragment, or a complex set of programs and data files. Below, a brief description of each of the terms related to reverse engineering arena is presented.

- **Forward engineering.** Forward engineering is the traditional process of moving to real implementation of a software system at source-code level from higher-level abstractions and logical design. It is like a sequence of activities starting from the requirements of the system until the physical implementation.

- **Reverse engineering.** "Reverse engineering in and on itself does not involve changing the subject system. It is a process of examination, not change or replication." as Chikofsky mentions in [2]. In a reverse engineering process software system is analyzed in order to identify the components of the system and their relations, and to generate a

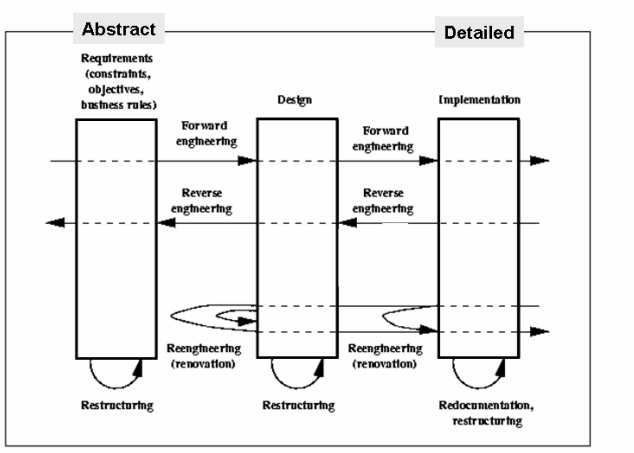



Figure 9.1: Taxonomy of software reverse engineering [2].

representation of system in a higher level of abstraction. Reverse engineering is present between different stages of life cycle, starting from the implementation source-code, design documents creation, and generating the original requirements of the system on which the whole development is based.

- **Re-documentation.** Re-documentation is generally called to creation, re-creation, or revision of a presentation in same level of abstraction which is equivalent to it semantically. Some common tools that are used for re-documentation are diagram generating tools (that reflect control or data structure of the program), or cross-reference listing generators. Here, the key goal is to make the connections between different components of the system more clear and recognizable.

- **Re-structuring.** Re-structuring is changing the structure of a software system while keeping the functionality of the system (external behavior) unchanged. It can be in the form of a transformation in the appearance of the code to a better-structured format, for example transformation from spaghetti code to a structured code form; or reshaping data models, for example "data normalization" is a restructuring in data model in database design which improves the model logically. As another example, transforming the system requirements from unformatted informal sentences to a tabular or diagrammatic representation improves the understandability and changeability of the requirements.

- **Re-engineering.** Re-engineering is changing or transforming the system or some parts of it to a new form and then implementing the new form. Generally, re-engineering can be divided to a reverse-engineering phase that gives a more abstract level of the system, and then a forward engineering or re-structuring phase to implement the new form of the system. Such changes might be needed because of the changes that occur

in the requirements of the system due to the changes in the working environment of the system. Another reason can be finding some requirements that are not met or well-supported by the system functionalities [2].

### Architecture and design recovery

Architecture and design recovery can be assumed as a subset of reverse engineering, which indeed is the dominant activity in this field. The purpose of architecture and design recovery is to extract some sound high-level and abstract knowledge from the system, using the system itself, domain knowledge, reasoning, and other external information. The abstraction makes the results of such recovery different from results if a direct study of the system. As Ted Biggerstaff mentions in [4] "Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains ... Design recovery must reproduce all the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code." Architecture recovery and design recovery are considered to be identical by some authors, and in some other they are considered to be different form the aspect that architecture recovery is more focused on the structure of the system components which has a static nature, while design recovery brings some high-level design decision descriptions and logical concerns other than static structure. For now, since they are identical in many aspects, we assume them to be identical unless one of them is explicitly mentioned.

## 9.3 Different Techniques for Reverse Engineering

We can consider four different views for software architecture which are structure, behavior, environment, and domain-specific views [5]. These views can be considered as the result of separation of concerns on a design in order to classify the knowledge about the design of the system into more understandable and manageable categories [6]. There exist different approaches and techniques for reverse engineering and architecture and design recovery in particular. Different approaches try to extract the architecture-related information from system using different technique like pattern-matching, clustering, data mining, and so forth. Since, performing these techniques are not feasible by hand only, each approach uses corresponding tool(s) based on the techniques it uses. These tools typically provide an interactive environment for the user to perform different phases of the reverse engineering process and provide necessary feedback.
Because of the non-deterministic nature of this area of software engineering, there is no perfect technique available (till now) and all the techniques are based on sort of approximation. Also, new approaches are introduced time by time which are sometimes based on new techniques, and sometimes based on a mixture of previous techniques with some refinements to give a more satisfying result. In the rest of this section some major techniques in architecture

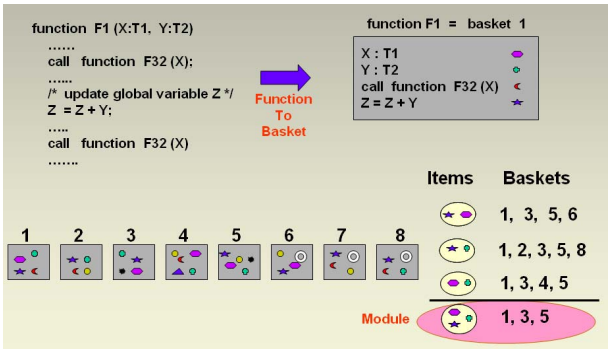



Figure 9.2: Software architecture recovery using data mining technique.

and design recovery are discussed.

## Architecture and design recovery using data mining techniques

Data mining which is also called Knowledge Discovery in Databases (KDD), refers to a collection of algorithms for discovering and extracting interesting and non-trivial relationships from data in large databases [11]. As discussed in [8] and [6] in architecture and design recovery using data mining a high-level conceptual model of the system which is provided by the user is tested against the real existing relation between modules of the system. Here, the tool decomposes the system into interacting modules. Also, there are some clustering techniques available that modularize the system based on the module interactions and partitioning methods [9]. In this part the architectural design recovery technique based on data mining proposed in [6] is discussed in more detail.

The high-level conceptual model, which is also called the architectural plan, is represented by some description language to present the components of the system and their relations. Here, the description is in form of a query which is applied upon a database containing information about the components of the system extracted from the source-code. This information is a higher-level representation of the source code in terms of level of abstraction which is represented using Abstract Syntax Trees (ASTs) and entity relationship tuples which can be achieved by parsing the source code. Here, the entities can be *functions, types*, and *variables* and the relationships are in the form of *use type, use variable*, and *call function*. In this way, we have the ability to abstract away the subject systems syntactical and implementation variations. Then, data mining techniques and a search algorithm (a modified version of branch and bound search) are used to perform the matching process be-

tween the elements of the query and the information in the database, so that the "candidate" elements in the query become instantiated to result in a concrete architecture as defined in [10]. The formalism used to present the descriptions as queries is called Architecture Query Language (AQL) [6]. By using AQL as a means to represent our conceptual model, we can define several candidate modules for the system, for each of which we can define a *main seed* (core entity of the module) any desired number of functions, types, and variables, and a number of outgoing or incoming relations with other modules. After the matching was successfully done, a ranking of the alternative results based on the associations found by data mining techniques is shown to the user. In the case of no feasible solution, the process is back-tracked in order to refine the query.

To further clarify the role of data-mining in this approach, the term *frequent itemsets* should be defined.

An itemset is a set of elements and a k-frequent itemset is an itemset elements of which are contained in a group of size k of supporting *transactions* or *baskets*. Elements of an itemset are of the same type of the entities (function, type, variable) and the transactions/baskets are considered to be functions. Frequent itemsets are generated using data mining techniques, for example *Apriori* algorithm [7].

Given the set of frequent itemsets and a main seed for a module we can generate a collection of entities which can be considered as candidate to be in that module. To do so, we group all the entities that co-exist with the main seed of the module in any single frequent itemset. These groups which are called *domains* are the basis to form the modules of the system.

The other issue is the relative associations between the entities that form the modules. This association or *closeness* is calculated based on the shared features and the common relations between two entities. The criteria for computing such an association are to minimize the coupling of the modules (minimal association among entities of different modules) and maximize the cohesion of each module (maximal association among entities of a module) and to augment modules as much as possible (minimal number of modules) [6]. These association values are computed in a pre-process phase and comes up with a preferred domain for each candidate main-seed based on the criteria mentioned above. The matching process uses these suggested modules in its search to instantiate the candidate variables in the query modules. Therefore, for each core entity (main seed) a module is built, and data mining provides a domain of highly-associated values for the variables of that domain.

Figure 9.2 shows an example that illustrates the role of data mining in forming modules. As was seen in this example function F1 is transformed into a corresponding basket form and the useful information (entities and relations) in it is stored in the basket. The same transformation is done on seven other functions to result the eight baskets as you see in the example. From these baskets, frequent itemsets are extracted, which are shown under the "Items" column, and also the baskets in which these items are present is shown under the "Basket" column. From these itemsets and baskets a suggested module is generated.

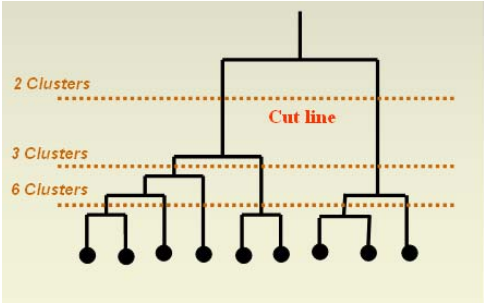



Figure 9.3: Software architecture recovery using hierarchial clustering technique.

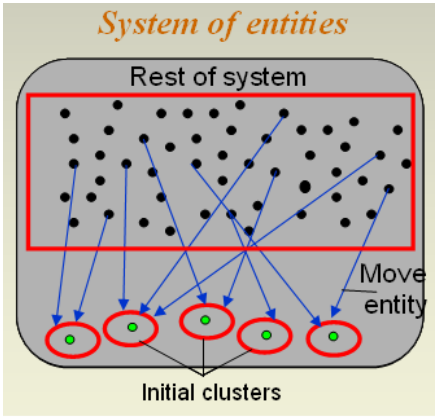
## Architecture and design recovery using clustering techniques

A clustering algorithm is a process that organizes a set of objects into various classes, in a way that objects within the same class share certain characteristics or similarities [12]. Clustering is found to be a systematic and effective approach to help the designer to restructure or re-engineer an architecture for improvement as well as for software architecture recovery [13]. From articles on software clustering we realize that there is a huge research potential in software clustering field [14], and also we conclude that clustering methods may be a very proper starting point for the re-modularization of software [15]. There is no single way of doing clustering on a software system code and different approaches exist in this regard. In this writing three different clustering techniques, namely *hierarchial, partitioning,* and *incremental* are presented.

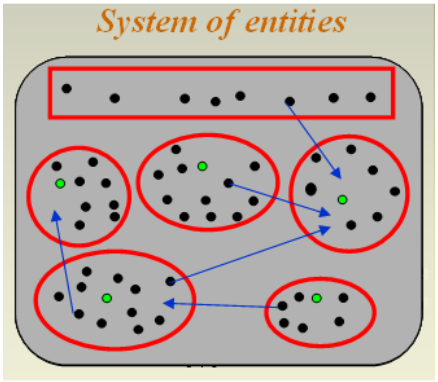
- **Hierarchial clustering.** Hierarchial clustering starts at its lowest level with singleton clusters which are single entities of the system. Then, at each level the existing clusters are merged together based on a similarity metric. If we borrow the term "main seed" from Section 3.1, we can define the similarity metric as the association value of the main seeds of two different clusters. Another rational candidate for similarity metric is the average similarity between the entities of two clusters. Since the entities of the system are functions, types, and variables, the association between them can be defined in the same way as Section 3.1 based on the *uses, calls, etc* relations. So, by applying hierarchial clustering to a set of entities (singleton clusters) a tree structure is generated with the entities as its leaves and the whole system as root. By moving up from the leaves of the tree toward the root, at each level there are a number of clusters, which decreases as we move up. This types of clustering gives a hierarchial view of the system (as expected). In another sense, if you start from the root you can dive into each cluster to view the sub clusters of it and so on till you get to the single entities. Figure 9.3 illustrates a depict of hierarchial clustering. As you see if you cut the tree at any level, you get a number of clusters and the nearer the cut to the leaves, the more the number of clusters. Each cluster represents a set of system entities and as we move upper in the tree, the clusters become more general-purpose.

- **Clustering by Partitioning.** In clustering by partitioning technique the system is divided into two general parts; one is a set of clusters and the the other is the rest of the system. Initially, the set of clusters are some selected entities which form singleton clusters, and the rest of the system is the set of other entities. Then, entities are relocated from the rest of the system to clusters, or vice versa, or between clusters themselves. The criterium for relocating entities can be computed by a value function, which is usually computed using a greedy algorithm. A good candidate for such an algorithm is relocating based on highest average "closeness" in clusters. In other words, a relocation is done if it causes an increase in the total average closeness of the entities of the relocation source and target clusters. In this way, each relocation will increase the total closeness for all clusters. The "closeness" itself can be defined as the average similarity among all the entities in a cluster, or the average similarity between the entities of the cluster and the main seed of the cluster. The partitioning process will stop when there is no more relocation done, which means the average closeness of clusters can not increase anymore. Figure 9.4 illustrates a snapshot of the partitioning process. Figure 9.4.a illustrates the initial configuration of the system, with singleton clusters and rest of the system; Figure 9.4.b shows an intermediate step of the process where entities are relocated between clusters themselves and also rest of the system and clusters.

- **Incremental clustering.** Incremental or iterative clustering is a method for clustering that recovers clusters based on average similarity values among entities. At each iteration a search algorithm collects entities in a cluster based on average similarity value, which is calculated in a similar way in previous sections resulting in "one" more cluster at each iteration. The domain of each cluster is dependent to the already recovered clusters. This in a sense means that the earlier a cluster is recovered the more coherent it is (it has higher average similarity among its entities than later recovered clusters) and so it is fair to let it restrict the domain of future clusters. In this method, clusters can have overlaps with each other which is tried to be minimized in the selection of new clusters. For each cluster a core or main seed is considered, which is the entity with highest similarity to other entities of the cluster. The incremental clustering process stops when the newly recovered cluster has high overlaps with existing clusters. Finally, the entities that have not been grouped in any cluster are distributed in the existing clusters based on a criterium such as the similarity to main seeds of the clusters or the average similarity between them and entities of clusters. Figure 9.5 shows a snapshot of incremental clustering process. As you see four clusters S1,..., S4 are recovered and there are still some entities that do not exist in any cluster. These entities may form other clusters themselves, or in case of high overlap may be distributed in the existing clusters.




(a) Initial step of clustering by partitioning. At this step initial clusters and the rest of the system are formed.




(b) An intermediate step of clustering by partitioning. Entities are relocated among both clusters and rest of the system.

Figure 9.4: Software architecture recovery using clustering by partitioning technique.

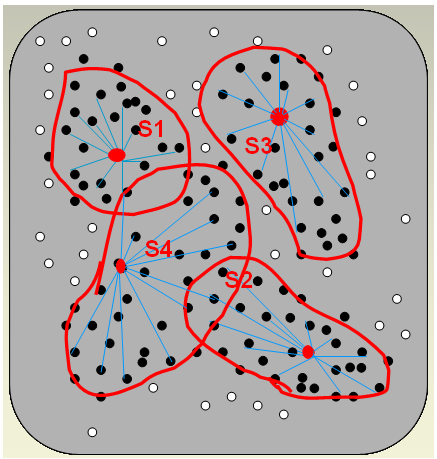



Figure 9.5: Software architecture recovery using incremental clustering technique.

## Scenario-based design recovery

In this section, we discuss the steps for recovery of the software system design from task scenarios. The proposed approach defines and gradually enhances a set of evidence-driven task scenarios and uses a schema to transform these scenarios into the view-based software design documents. We adopt Zachman's framework [16] consisting of a collection of perspectives and views and set our objective to transform the information from the task scenarios into the data and function views of the owner's perspective that is defined in this framework. Then, by following Zachman's guidelines we use the owner's perspective to generate the design diagrams at the designer's perspective and developer's perspective. The owner's perspective provides a comprehensive knowledge about the software system to be designed, from the owner's view-point, i.e., all the entities and the relationships are real in the corresponding business. The owner's perspective consists of entity-relationship diagram that can be understandable by the systems owner who in most cases is not a software engineer and can communicate through relevant terms and relations in his/her business. The steps of the approach are discussed in following sections.

## Scenario schema

Task scenarios are represented in a variety of formal and informal notations [17, 18]. Moreover, the way they are described varies from text statements, static graphics, or dynamic animation (by user interaction). In this paper, we adopt an informal text-based scenario representation which are checked against a domain model that we call scenario schema. The proposed scenario generation process involves the steps for scenario completeness and consistency checking. The class diagram representation of the proposed scenario schema is presented in Figure 9.6 and is intended to cover the potential task scenarios in different application areas. We applied the proposed scenario schema on three application areas, including a software analysis tool "Alborz toolkit"[19], a fast-food restaurant system, and an Automatic Banking Machine (ABM system). In the remainder of this section the entities (as classes) in the scenario schema are introduced using a restaurant system example.

- **Goal:** each goal represents a main reason for which the subject system is used. Based on evidences such as available documents (e.g., manual, requirements, design), expert user, and application domain knowledge, a set of goals is derived. Each scenario must be related to and hence satisfy one or more goals within the goal set. The goals in a restaurant system include: ordering food, preparing food, assembling food, handling inventory, managing restaurant, etc.

- **Actor:** an actor can be a human or a system that interacts with the subject system to perform the scenario. Examples of actors are, order taker, food assembly station, food preparation unit, inventory.

- **Action:** an action is a kind of operation that takes place within a scenario and can be of type input, output, or internal operation. Examples includes taking order (input),

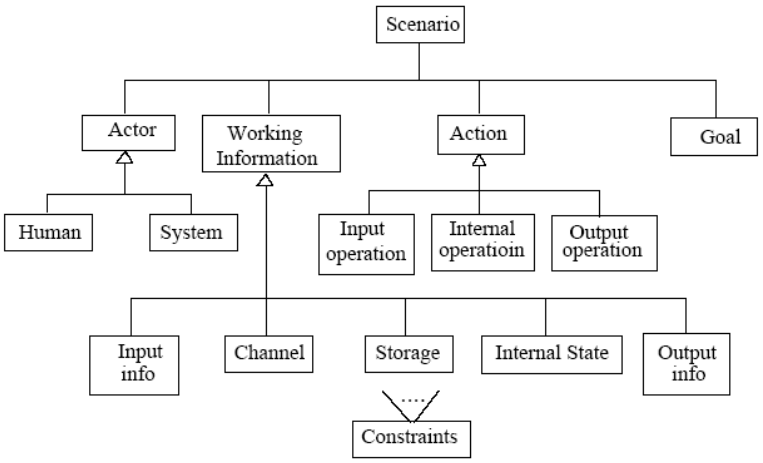



Figure 9.6: Proposed schema for scenarios.

delivering food (output), moving order from order taker station to food preparation station (internal).

- **Working Information:** which refers to the kind of information that is communicated, operated on, or stored in the system, via the execution of the scenario. The information is either electronic or non-electronic and can be of one of the following types (for electronic and non-electronic respectively): "input info" such as *order item* or *raw material*; "output" such as *total amount due of an order*or *assembled food*; "channel" such as *computer network* or *food item chute*; "storage" such as *computer memory* or *inventory shelves*; "internal state" such as *unpaid orders* or *food item in kitchen*. The examples of "input info" in a reverse engineering tool can be an GXL or XML file and for an Automatic Banking Machine (ABM) can be money bills or bank card.
  The proposed scenario schema in Figure 9.6 also includes a constraint class with a relation to every entity in the schema. This class contains information about the possible constraints corresponding to non-functional qualities that may be associated with each class of the schema. The scenario schema plays a key role in the proposed framework. It is used to verify the structure of the scenario set; controls the level of details of the information they contain, and checks the completeness of the ingredients of the system; and finally it is the major source to feed information required for the design documents of the different views of the Zachman's framework at the owner's perspective.

## Scenario to design document transformation

The steps for generating the task scenarios and transforming the scenarios into design documents are discussed below.

1. **Scenario generation.**'This activity is performed by human (using his/her domain knowledge) and involves discovering and generating different scenarios by the aid of evidences such as systems user manual, tutorials, wizards, help utility, human knowledge, or any other valid and useful documentation. Each suggested scenario must pass the completeness check (below) to be considered as a valid scenario.

2. **Completeness test.** A completeness check is applied on each scenario to see if it contains enough information to fill the major entities of schema, i.e., subclasses of the classes *Goal, Actor, Action* and *Working Information*. One of the three cases can happen: i) if the scenario passes the completeness check then it is added to the set of valid task scenarios; ii) if the scenario requires additional information to be complete, the information is added to the scenario and the completeness check is repeated; otherwise, iii) the scenario fails the completeness test, meaning the scenario is inherently unacceptable and it will be rejected. In the latter case we return to Step 1 to get another scenario.

3. **Decomposition test.** For each scenario in the set of the valid scenarios, we investigate to determine whether it is decomposable to finer scenarios or not. In other words, we should be able to realize whether the level of details of the scenario is appropriate or not. This is determined by checking the scenario against the major classes in the schema to check whether it can instantiate the subclasses of the major classes. This decomposition step requires familiarity with the Zachman's framework (specially the owner's perspective), the domain knowledge, and awareness of the corresponding business rules. This step is repeated until we derive the scenarios with appropriate details, and then we proceed to Step 4 for each of them. This set of refined scenarios become our new set of valid scenarios.

4. **Mapping.** At this step each valid scenario (with proper granularity level) is mapped onto the scenario schema to instantiate the subclasses of the major classes, and since it is in a right granularity level it corresponds to exactly one subclass of each major class.

5. **Approval.** The appropriateness of the whole extracted scenario set must be approved by the system owner who is expert in the relevant business and the application area. This set of scenarios can be grouped according to the specified goals in the framework, and hence the owner can determine whether all his expectations have been fulfilled or still more scenarios need to be defined.

6. **Design document generation.** Finally, we generate the subject systems design documents at the owners perspective that is proposed by the Zachman's framework. At this stage, the owner can still find incomplete relations in the design document which triggers new scenarios generation. A few iterations of the above steps will ensure

capturing of a comprehensive requirements extracted from the task scenarios, which in turn allow us to produce the design of the subject system.

## Restaurant system example

In this part, we discuss an example from the restaurant system to show the treatment of the task scenarios in our proposed approach.

**Textual description of a scenario** (Step 1)
*"Computing and reporting the total amount due of the order and sending the order for preparation to preparation station."*

**Processing the scenario** (Steps 2 to 6)
At Step 2 the completeness check is performed on the scenario and it turns out that the scenario can instantiate the subclasses of Goal, Action and Working Information classes in the scenario schema but it lacks information about the Actor class. Since the scenario failed the completeness check we perform a few corrections on the scenario and the corrected version would be:

*"System computes and reports the total amount due of an order and sends the order for preparation to the preparation station."*

After this correction we repeat the completeness check and this time it passes the test, and we proceed to the next step. In Step 3 we verify if the scenario has appropriate level of detail using the domain knowledge and business rules, and if applicable we decompose the scenario. By inspection, we realize that this scenario corresponds to more than one instances for the Internal Operation (a subclass of Action). It contains both computation of the total amount due and sending the order to the preparation station. So, we conclude that this scenario is decomposable to two finer scenarios that are in an appropriate level. After the decomposition we generate two scenarios as follows:

1. *"System computes and report the total amount due of an order."*
2. *"System sends the order for the preparation to preparation unit."*

Now, for each of these two scenarios we perform Step 4 which involves the mapping of the scenarios to the scenario schema. The result from above two scenarios is as follows:

**Scenario #1:**
$Goal = Taking\ order\ \&\ Handling\ payment$
$Actor|_{System} = Order\text{-}taking\ station$
$Action|_{Internal} = Financial\ computation$
$Action|_{Output} = Reporting\ amount$

$Input = Order$
$Channel = Order\ taking\ input\ interface$
$Output = Order\ taking\ output\ interface$

**Scenario #2:**
$Goal = Preparing\ order$
$Actor|_{System} = Order\text{-}taking\ station$
$Action|_{Internal} = Information\ transformation$
$Input = Order$
$Channel = Order\ taking\ to\ preparation\ station\ link$
$Internal\ state = Unprepared\ order$

At Step 5 the owner should decide if the set of extracted scenarios are enough for covering all the requirements of the system. If so, we can proceed to the next step and produce the design documents at the owners perspective. If not, we produce additional scenarios to cover the deficiencies and go back to Step 1. In our example we assume the owner is convinced that these two scenarios are enough to cover all the requirements of the system. So we move to Step 6 to produce design documents at the owners perspective according to Zachman's framework.
We can use the values assigned to Actor, Input, Storage, Internal state, and Output classes in the scenario schema to infer the entities that exist in Data view of the Zachman's framework, such as *order* and *order-taking station*. Similarly, the values assigned to the Action subclasses can be used to infer the different activities of system in the function view of the framework, such as *calculation of the amount due* and *sending an order to food preparation station from order-taking station*. Once we produced the complete design document of the system at the owners level, we can follow the Zachman's framework to extract the design of the system down to the implementation level.

## 9.4 Conclusions

In this paper, we studied reverse engineering and architecture and design recovery as major activities in this area which are widely used in legacy software systems maintenance phase. Also, three different techniques of data mining, clustering, and scenario-based for architecture and design recovery were discussed. As was seen, the goal of reverse engineering is to increase the understanding of software systems by producing design level facts from the source code, domain knowledge, and other available information about the system. This facilitates future enhancements and changes on the software systems. The research area for reverse engineering is very wide and there is still much to do in order ro enhance existing methods, or invent new ones. Design recovery, for instance, is one of such areas. It can be considered as a separate activity from architecture recovery, exchanging feedback with which can result in more accurate facts to facilitate the understanding of software systems.

# Bibliography

[1] Bisbal, J., Lawless, D., Wu, B. & Grimson, J. (1999)."Legacy Information System Migration: A Brief Review of Problems, Solutions and Research Issues", *IEEE Software*, 16, 103-111.

[2] Elliot J. Chikofsky , James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy", *IEEE Software*, v.7 n.1, p.13-17, January 1990.

[3] Hausi A. Muller Dept. of Computer Science University of Victoria, Canada, Jens H. Jahnke Dept. of Computer Science University of Victoria, Canada, B. Smith. "Reverse Engineering: A Roadmap", *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 47-60.

[4] T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse", *Computer*, July 1989, pp. 36-49.

[5] *"*Architectural Design Recovery using Data Mining Techniques", Web site, URL = http://se.math.uwaterloo.ca/ksartipi/papers/techrep. ps.

[6] Kamran Sartipi and Kostas Kontogiannis and F. Mavaddat. "Architectural Design Recovery using Data Mining Techniques", *IEEE Computer Society Press*, 2000, pp. 129-139.

[7] R. Agrawal and R. Srikant. "Fast algorithm for mining association rules", *In Proceedings of the 20th International Conference on Very Large Databases*, Santiago, Chile, 1994.

[8] Murphy, Notkin, and Sullivan. "Software Reflexion Models: Bridging the Gap Between Design and Implementation", *IEEE Transactions on Software Engineering*, April 2001.

[9] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. "Using automatic clustering to produce high-level system organizations of source code", *In Proceedings of IWPC98*, pages 4553, Ischia, Italy, 1998.

[10] R. C. Holt. "Structural manipulations of software architecture using Tarski relational algebra", *In WCRE: Working Conference on Reverse Engineering*, Honolulu, Hawaii, October 1998.

[11] U. M. Fayyad. "Advances in knowledge discovery and data mining", *MIT Press*, Menlo Park, Calif., 1996.

[12] R. Castro, M. Coates and R. Nowak, "Likelihood Based Hierarchical Clustering", *IEEE Transactions in Signal Processing*, August 2004.

[13] X. Xu, C.-H. Lung, M. Zaman, A. Srinivasan, "Program Restructure through Clustering Technique", *Proc. of the 4th IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM) in conjunction with IEEE Int'l Conf on Software Maintenance*, Chicago, IL, Sept 2004, pp. 75-84.

[14] Tzerpos, V. and Holt, R. C. (1998). "Software Botryology Automatic Clustering of Software Systems", *Proc. of the 20th Annual Intl Conf. of the IEEE*, 3, pp. 811-818.

[15] Wiggerts, T. A. (1997). "Using Clustering Algorithms in Legacy Systems Modularization", *Proc. of the 4th Working conf. on Reverse Eng.*, pp. 33-43.

[16] J. A. Zachman. "A framework for information systems architecture", *IBM Systems Journal*, 26(3): pp. 276292, 1987.

[17] J. Desharnais, R. Khedri, and A. Mili. "Representation, validation and integration of scenarios using tabular expressions", *Journal of Formal Methods in Software Development. Special issue on tabular expressions*, To appear, 2002.

[18] J. Ralyte. "Reusing scenario based approaches in requirement engineering methods: Crews method base", *In REP99*, pp. 305309, 1999.

[19] K. Sartipi. "Alborz: A query-based tool for software architecture recovery", *In Proceedings of the IEEE International Workshop on Program Comprehension (IWPC01)*, pp. 115116, Toronto, Canada, May 2001.

[20] K.Sartipi, H.Safyallah, and N.Dezhkam. "A Multi-view Architectural Reconstruction Environment to Enhance an Evolving Software System", *Submitted for WCRE 2005 conference*, 2005.

# Chapter 10

# Shu Wang: File Comparison Techniques

The Unix **diff** and **diff3** commands are central to configuration tools and useful as general tools for programmers. The original versions of **diff** work on a line-by-line basis for comparing files. This works well on some formats of documents which are line based, but does not work well on several others such as sentence or markup based documents and block-structured programming code. In this report we first give an introduction to the **diff** command and its variants then investigate the algorithms behinds **diff** (**diff3**). Finally we discuss the drawback of its line-by-line comparison scheme and how it can be improved using techniques such as syntactic diff.

## 10.1 Introduction

In computing, there are many situations that we need to know how two files differ. Such as in Software Configuration Management, we want to manage the changes made by two programmers to one file of source code. Also in Bioinformatics we may want to compare two DNA sequence and align them in the optimal way. There are also a lot of tools exist which perform file comparisons, such as Unix **diff**.

### Diff

**Diff** is a standard Unix command that outputs the difference between two text files. It was first developed in early 1970s in Bell lab by Douglas McIlroy [1]. The output of **diff** is an **edit script** that can be used as an input to **ed** program in Unix. The output (usually also called 'diff') consist of three types of operations - ***a*** stands for added, ***d*** for deleted and ***c*** for changed. Lines common to both files are generally not outputted. Figure 10.1 illustrates a sample input texts followed by the output produced by **diff**.

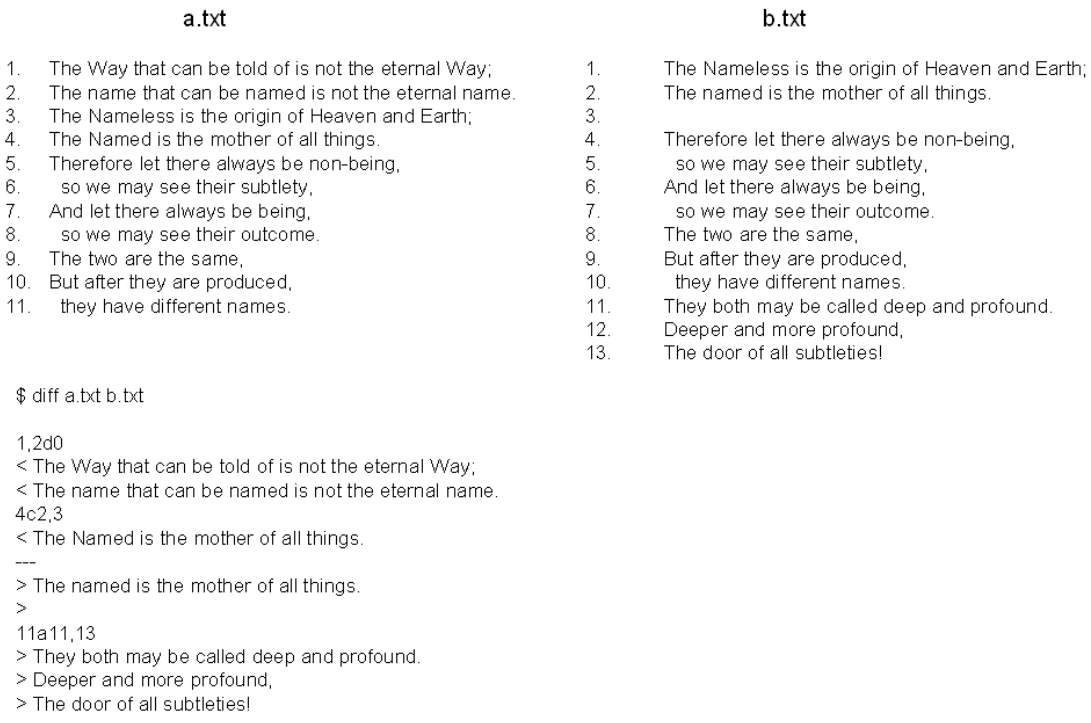



Figure 10.1: A sample input and output of **diff** [2]

## Variants of diff

In the last three decades the **diff** command was widely used as a standard file comparison tool. Many variants of **diff** also emerged in which features such as white space compression, case folding, comparison of the whole directory, character by character and side by side comparison are added etc. One of these variants, called **diff3**, demonstrate the differences among three files. And many latest variants of **diff** such as **WinDiff** and **Active File Compare** provides users with a graphical interface.

Figure 10.2 shows the interface of the **Active File Compare** program from [7]. This program demonstrates the difference between two files in a side-by-side style, which is quite popular in latest graphical file comparison programs. The two files are aligned (or synchronized) by the program using extra blank lines. Lines which the program considers different are highlighted and marked. Insertions, deletions and substitutions are shown using special symbols at the left of each line. The details of difference are shown in the down-left corner in characters.

**Active File Compare** and **diff** as general file comparison tools output differences without consideration of the syntax and semantics of the source code. For example, in Figure 10.2


Figure 10.2: An interface of **Active File Compare** Program

**Active File Compare** output 6 lines of difference while in practice especially in Change Management we may prefer to narrow down the difference to just 1 (line 589). Because other differences are usually considered syntactically or semantically insignificant. In the following sections we'll perform taxonomy of the **diff** program and discuss how to solve the above problem.

## 10.2 The algorithms behind diff

First of all let us have a look at the algorithms lie behind the **diff** program. In order to do this we need first to give some basic definitions.

### Definition

A file can be seen as a **string** of characters. The basic element of file comparison will naturally be character. However, in the early versions of **diff** in 1970s, the basic comparison

element was a line. The **diff** will first hash each line of a file into one basic element and then perform the string comparison based on these hashed elements. The reason why the line is chosen as the basic element is probably due to the following facts: first, comparison based on line reduces the number of basic elements and thus increases the speed, this is especially important in the 1970s when computers were much slower than they were today; second, **diff** was originally designed for comparing line-based program code. So evolved from the original **diff**, modern variants of **diff** generally follows this tradition but many of them support character-by-character comparison as well. In summary the problem of computing the difference between two files can be reduced to computing the difference between two strings of basic elements.

The meaning of "difference" as stated previously, is the minimum operations to transform one string into another. In many references it was often substituted by "distance". We give the formal definition of distance as follows.

**Definition 1** *Hamming distance is $d_H$ is the minimum number of substitutions needed to transform strings $x$ to $y$ of the same length.*

**Definition 2** *Levenshtein distance $d_L$ is the minimum number of deletions and insertions needed to transform strings $x$ to $y$ of the same length.*

**Definition 3** *Edit distance $d_E$ is the minimum number of deletions, insertions and substitutions needed to transform string $x$ to $y$.*

Note that **difference** and **distance** are not exactly the same, one refers to the operations and one refers to the number. But it is usually the case that if you calculate one you can easily calculate the other. That is, in order to compute difference you'll have to compute the distance and vice versa. So many references don't distinguish this two. We will discuss it in details in the next section. Here is an example showing distance $x$ and $y$ using different definitions:

**Example 1**

Let $x$ = CGACG $y$ = GTCGA
Differences:

| | | |
|---|---|---|
| CGACG | CGA–CG– | CGACG– |
| GTCGA | –G–TCGA | –GTCGA |
| $d_H = 5$ | $d_L = 4$ | $d_E = 3$ |

Note that we can also use alignment to represent the set of operations. In the above example, the results are showed in the form of **optimal alignment**. The "–" symbol in the first line represent an insertion, in the second line it represent a deletion and if two different characters are aligned it represents a substitution. If two identical characters are aligned then it represents a match.

**LCS** stands for "the longest common subsequence" of two strings. For example, *cold* = LCS (*scrolled*, *could*). The problem of computing optimal alignment is closely related to the problem of computing **LCS**. That is, if we make no distinction between every substitution, i.e. "A" substituted by "a" is the same to "A" substituted by "b", then the problem of computing the optimum alignment can be reduced to computing the LCS.

The **diff** program use **edit distance** to calculate the difference between files. Note that the Hamming distance can only apply to two strings with the same length, while Levenshtein distance and edit distance can both be used to calculate the distance of two strings of different length. And also there is a straightforward relationship between results computed from Levenshtein distance and that from edit distance. If we replace a pair of consecutive "insert" and "delete" operations into one "substitute" operation, then we transform the result computed from Levenshtein distance into result computed from edit distance, and vice versa. For instance, in the previous example, taking the result computed form Levenshtein distance, if we replace the "delete A" operation in the third position and the "insert T" operation in the fourth position with a single "substitute A with T" operation, then we get a result exactly the same to the result computed from edit distance.

There is a connection between LCS and Levenshtein distance as well. If the previous example, taking the result computed from Levenshtein distance, if we eliminate those positions involved in a "insert" or "delete" operation, then we will get a common subsequence. Let $i$ be the number of insertions and $d$ be the number of deletions, $|c|$ be the length of common subsequence, and let $m, n$ be the length of string $x, y$ respectively, then the following equation holds:

$$i + d + 2 * |c| = m + n \tag{10.1}$$

In Example 1, the result computed from Levenshtein Distance contains 2 insertions and 2 deletions, $m = n = 5$, $|c| = 3$, so $2 + 2 + 2 * 3 = 5 + 5$. From equation 1 it proves that if $i + d$ is minimum then $|c|$ is maximum, thus it is the longest common sequences of $x$ and $y$.

In summary, we conclude that if we can compute one of the following, we can easily compute all the others. And that is why in many references sometimes one of them is used to mention the other without very strict distinctions.

- Diff
- Difference between two strings
- Edit distance
- Levenshtein distance
- Longest Common Subsequences
- Minimum edit operations
- Optimal alignment

## The basic dynamic programming algorithm

In this section we discuss several algorithms behinds Unix **diff** command. We start with the basic Dynamic Programming algorithm, which is the underlining algorithm of all the others.

The basic dynamic programming approach computes minimum edit distance of string $x$ and $y$. First we create a $m \times n$ matrix $T$ such that every entry in $T$ represent the minimum edit distance of $x[0, 1, ..., i]$ to $y[0, 1, ..., j]$.

Then we compute the matrix column by column using the following formula:

$$T[i,j] = \min \begin{cases} T[i-1,j]+1 \\ T[i,j-1]+1 \\ T[i-1,j-1]+W[i,j]) \end{cases} \tag{10.2}$$

$$W[i,j] = \begin{cases} 0 & \text{if } x[i] = y[j] \\ 1 & \text{otherwise} \end{cases} \tag{10.3}$$

Each entry in $T$ can be computed from three adjacent entries as Figure 10.3 shows.

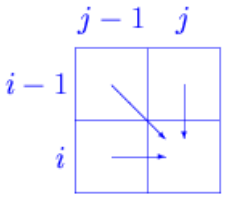



Figure 10.3: The computation of $T[i, j]$

The complete algorithm is described as follows [4]. Note that in real programming language, an array with an index of -1 is generally not legal, here it is used for demonstration purpose, and the problem can be avoid by increasing all the index in the array by one.

Algorithm: Edit_Distance$(x, y)$
1. $m :=$ the length of the sequence $x$.
2. $n :=$ the length of the sequence $y$.
3. Initialization. $T[i, -1] := i$ for $i = 0..m$. $T[-1, j] := j$ for $j = 0..n$.
4. for $i := 0$ to $m$ do
5. for $j := 0$ to $n$ do
6. $T[i, j] := \min(T[i, j-1] + 1, T[i-1, j] + 1, T[i-1, j-1] + W[i, j])$
7. where $T[i, j] = 0$ if $x[i] = y[j]$ and $T[i, j] = 1$ otherwise.
8. od
9. od
10. return $(T[m, n])$

Figure 10.4 is a sample computation of edit distance between string $x = ACGA$ and string $y = ATGCTA$. After computing the whole matrix we can "trace back" to recover the minimum edit operations as demonstrated by the shaded squares. The result is not necessary unique.

| $T$ | $j$ | $-1$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|
| $i$ | | $y[j]$ | A | T | G | C | T | A |
| $-1$ | $x[i]$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | A | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | C | 2 | 1 | 1 | 2 | 2 | 3 | 4 |
| 2 | G | 3 | 2 | 2 | 1 | 2 | 3 | 4 |
| 3 | A | 4 | 3 | 3 | 2 | 2 | 3 | 3 |

$$\begin{pmatrix} A & - & - & C & G & A \\ A & T & G & C & T & A \end{pmatrix} \quad \begin{pmatrix} A & C & G & - & - & A \\ A & T & G & C & T & A \end{pmatrix}$$

Figure 10.4: A sample computation of basic dynamic programming [5]

The time and space complexity of the basic dynamic programming algorithm is $O(mn)$, while the space complexity can be reduced to $O(m)$ using the technique introduced by Hirschberg [6], because in order to compute every entry we need only to refer to the three adjacent entries, thus at every moment during the computation we need only to keep record of current row and the previous row.

## Hunt-Szymanski

Hunt-Szymanski is the algorithm behind the original **diff** program [1]. It is based on the computation of Longest Common Subsequences of strings $x$ and $y$. The basic procedures are as follows:

- Compute an array of lists called MatchList.
- Compute an array $j$ such that $j[i, h]$ gives the length of the shortest prefix of $y$ that has an LCS of length $h$ with $x[1..i]$.
- Trace back from the last row to retrieve the longest common subsequence.

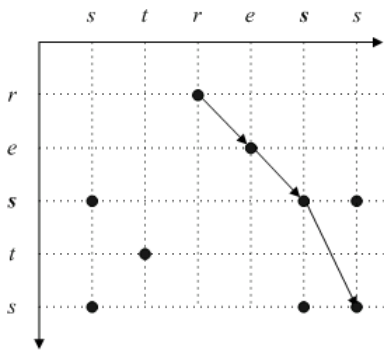



Figure 10.5: An example of Hunt-Szymanski algorithm [3]

The details of algorithm can be found in [3]. The average time complexity of Hunt-Szymanski algorithm is $O(n\log n)$, $n = \max\{m, n\}$, while the worst case time complexity is $O(mn)$.

## Ukkonen-Myers

The Ukkonen-Myers algorithm, is used by recent version of GNU-Unix **diff** [2]. It is also based on the basic Dynamic Programming algorithm discussed previously, but adopts a "Trial-and-Verify" approach. It uses the same edit distance matrix in the basic dynamic-programming approach while tries to avoid unnecessary comparisons as much as possible. The main procedure is as follows.

Without loss of genericity we assume $m < n$. Let diagonal $j$ be the diagonal with a upper-left vertex at position $T[0, j]$. First we take the diagonal 0 and diagonal $(n - m)$ as the boundaries, as the two bold segments in Figure 10.6 show. Then we calculate the value of $T[m, n]$ by filling only the entries within the boundaries. That is, we start from position $T[0, 0]$ and try to "reach" $T[m, n]$ using routes only within the two boundaries. At the end of the calculation we get a value in position $T[m, n]$, let's call it $T[m, n][0]$. Of course this value is not necessarily optimal, because there can exist some other routes that has a smaller value but not within the two boundaries. However, it is proved in [3] that if $T[m, n][0] = 0$ then it must be optimal. In the case of $T[m, n][0] \neq 0$, we then expand the boundaries by using diagonal $0 - D$ and $n - m + D$, the value of $D$ will start from 1 and be doubled in every expansion. Which means, $D = 1, 2, 4, ...m$. Every time we expand the boundaries we will get a new value $T[m, n][D]$. It is demonstrated in [3] that if $T[m, n][D] > D$, then it means the trial failed, we doubled the value of $D$ and repeat until $T[m, n][D] \leq D$ or $D \geq m$ which terminate the calculation. Then we backtrack as in basic dynamic programming to

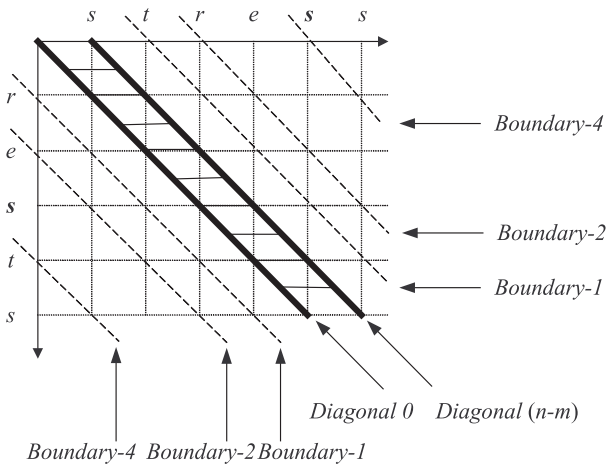



Figure 10.6: Ukkonen-Myers algorithm

retrieve the minimum edit operations.

The time complexity of Ukkonen-Myers algorithm is $O(nD)$ [3], where $n$ is the greater length of $x$ and $y$, and $D$ is the edit distance between $x$ and $y$. We can see that in the worse case, $D$ can be as great as $m$, so this algorithm did not improve the worse case time complexity of $O(mn)$ achieved by the basic dynamic programming approach. However when the edit distance of two strings is small, in other words $D$ is small, the Ukkonen-Myers algorithm becomes the algorithm of choice.

## 10.3 Advanced file comparison techniques

In the previous section we have already investigated the algorithms behind Unix **diff** command. We also mentioned in Section 1 that **diff** sometimes produce unsatisfactory comparisons because they cannot accurately pinpoint the differences and because they sometimes produce irrelevant differences [4]. This is especially true when the files to be compared are source codes.

One reason causing the above problem is that the original **diff** was designed in the early 1970s and the comparison was performed on a line-by-line basis. This scheme doesn't work quite well on latest block-structured program codes. In order to overcome this drawback several efforts are made to improve the result produced by **diff**. In the following we discuss Syntactic Diff and Semantic Diff and finally we propose another way to improve **diff** using comparison by sentences.

## Syntactic Diff

In [4] a new way called **Syntactic Diff** to compute the difference between two programs has been proposed. This new method differs from the original **diff** in that the basic element in comparison is not lines but tokens, and irrelevant details are filtered out by a parser [4]. The basic procedure is as follows:

- Parsing. The source code of the program is first parsed into a parse tree by a parser. The leaves in the tree represent tokens and non-leaf nodes represent substructures such as expressions just as an ordinary parser in compiler.
- Tree Comparison. The two parse tree of the program are compared using dynamic programming scheme.
- Synchronization. The two files are synchronized (aligned) by a 'pretty-printer' to increase the readability.

The key point here is the dynamic programming tree-matching algorithms. The basic dynamic programming approach which we discussed in the previous section aligns two one-dimensional strings, while here dynamic programming is used to align the nodes in the first level of the two trees [4]. The edit distance of insertions and deletions are a little bit different than the basic dynamic programming approach and the substitutions are determined by the edit distance of their subtrees. The subtree edit distance is again calculated using dynamic programming so this tree matching algorithm is a recursive algorithm. The algorithm can be formally described as follows [4]:

Algorithm: Tree_Comparison(Tree $A$, Tree $B$)
1. $m :=$ the number of first-level subtrees of $A$.
2. $n :=$ the number of first-level subtrees of $B$.
3. Initialization. $T[i,-1] := i$ for $i = 0..m$. $T[-1,j] := j$ for $j = 0..n$.
4. for $i := 0$ to $m$ do
5. for $j := 0$ to $n$ do
6. $T[i,j] := \min(T[i,j-1] + U[i], T[i-1,j] + V[j],$
$T[i-1,j-1] + W[i,j])$
7. where $T[i,j]$=Tree_Comparison($A_i$,$B_j$), $A_i$ and $B_j$ are the $i_{th}$ and $j_{th}$ first-level subtrees of $A$ and $B$, respectively. $U[i]$ and $V[j]$ are the number of nodes in the $i_{th}$ and $j_{th}$ first-level subtrees of $A$ and $B$.
8. od
9. od
10. return ($T[m,n]$+Edit_Distance(root(A), root(B)))

Consider the following example. Imaging we have two source code files as the Figure 10.7 shows,

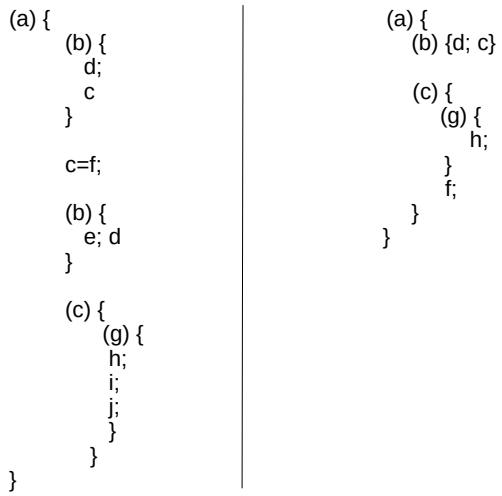



Figure 10.7: Two source code before alignment [4]

In the first step we parse the two source codes into two trees as in Figure 10.8. In order to compare (or align) tree $A$ and $B$ which has root node of $N1$ and $N15$ respectively, first we construct a matrix $T$ such that the column and row represent the first level subtree of $A$ and $B$. In this case, the rows will be N2, N3, N4, N5 and the column will be N16, N17. To calculate each entry we consider three options – deletion, insertion and substitution, and choose the minimum cost from these three options just like the basic dynamic programming approach. But in order to calculate the substitution cost we have to construct another matrix and perform the same dynamic programming procedure for their subtrees. So in order to calculate the cost of substituting N2 for N16, we construct another matrix and using N6, N7 as rows, and N18, N19 as columns. In this case the edit distance will be 0, consequently the substitution cost of substituting N2 for N16 is 0. The cost of substituting N4 for N16 however, according to this scheme will be greater than 0. So substituting N2 for N16 will have a smaller edit distance value than substituting N4 for N16. Similarly for all other subtrees. In this example finally N2 is aligned with N16 and N5 with N17, yielding the minimum edit distance.

After calculating the the minimum edit distance, the diff and optimal alignment can be calculated using the backtrack approach described in [6]. The alignment will look like that in Figure 10.9. Note that the conventional **diff** will report that almost every line is different for a input like in Figure 10.7. While we see that syntactic diff will maximize the similarity syntactically, which is what user usually preferred.

An implementation of Syntactic Diff called **CDiff** was mentioned in reference [4]. However, it hasn't become very popular probably due to the following facts:

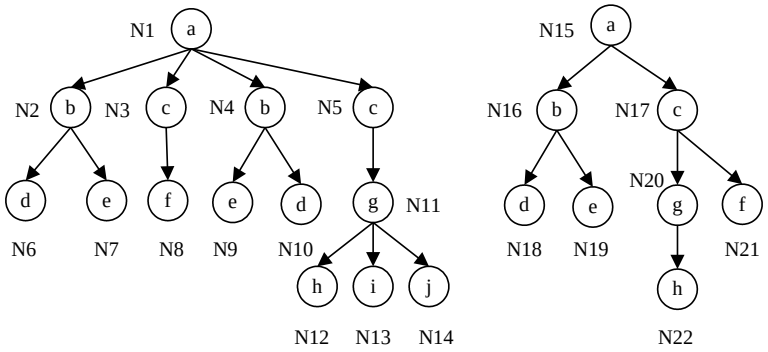



Figure 10.8: Comparison of two trees [4]

```
(a) {                                  (a) {
      (b) {                                (b) {
         d;                                   d;
         c                                    c
      }                                    }

      c = f;

      (b) {
        e; d
      }

      (c) {                                      (c) {
         (g) {                                       (g) {
           h;                                          h;
           i;
           j;
         }                                           }
      }                                          }
}                                      }
```

Figure 10.9: Two source code after aligned by Syntactic Diff [4]

- The input files must have a rigid syntactic structure, in other words, both files must be syntactically correct, otherwise the tool won't work.

- Its recursive dynamic-programming probably will not have a very good performance on large length codes.

- Although theoretically the Syntactic Diff can be applied to any programming language that is defined by a context-free grammar [4], an implementation such as **CDiff** can only take source code written in one programming language, or even worse, one specific version. A possible solution to this problem is to use the parser from the compiler directly instead of building parser of its own.

## Semantic Diff

In [8] it described a new way called **Semantic Diff** to compute difference between two procedures. Instead of tools we discussed so far that based on comparison of strings or trees, Semantic Diff expresses its results in terms of the observable input-output behavior of the procedure [8].

Suppose we are given a C procedure such as follows [8]:

```
void add (int x){
 if(x!=HI){
   TOT = TOT + x;
 }
 else TOT = TOT + DEF;
}
```

The Semantic Diff first produce a set of binary relationships over the set of variables accessed by the procedure: its arguments, results and any global variables. In the above example following pairs are produced:

(TOT,TOT),(TOT,$x$),(TOT,DEF),(TOT,HI)
($x$,$x$),(DEF,DEF),(HI,HI)

This means that the value of TOT after depends on the value of TOT, $x$, DEF, HI before, and the values of $x$, DEF and HI depends on their value before [8].

```
void add (int x){
 if(x=HI){
   TOT = TOT + DEF;
 }
 else TOT = TOT + x;
}
```

If the above change is made to the code, then relationship pairs will become

(TOT,TOT),(TOT,DEF),(TOT,HI)
($x$,HI),(DEF,DEF),(HI,HI)

And the program will output:

*new version removes dependencies*: $x$ *on* $x$, TOT *on* $x$
*new version adds dependencies*: $x$ on HI.

We can see that the Semantic Diff produces an approximate binary relationship between the input and output variables, so that meaning-preserving transformations (such as renaming local variables) will be correctly determined to have no visible effect [8].

### Comparison by sentences

It is also possible to compare two files by sentences. At the first step we can "parse" the text file into sentences, hash each sentence into a basic element, and then calculate the edit distance just like the basic dynamic programming approach. However, one potential difficulty is that sometimes it is difficult to break text into sentences. For example, period can be seen as one of the symbols that ends a sentence but may have other uses that causes confusion. For example, "Ph.D." and "Figure.1" are generally considered words other than sentences. So the key point of compare by sentences will lie on natural language processing, that is, how to break articles into sentences properly.

## 10.4 Summary

In this report we gave an overview of Unix **diff** program and other similar tools, and we investigated the algorithms and rationales behind **diff** in details. Almost all these algorithms are based on dynamic programming approach while vary in some ways to enhance (average)performance. Finally we discussed some advanced file comparison techniques such as Syntactic Diff and discuss its advantages and drawbacks.

# Bibliography

[1] Hunt, James W. and McIlroy, M. Douglas, **An Algorithm for Differential File Comparison**, 41, *Computing Science Technical Report*, Bell Laboratories, June 1976.

[2] **Comparing and Merging Files**, *GNU Diffutils User Manul*, http://www.gnu.org/software/diffutils/manual/html_mono/diff.html

[3] Bill Smyth, ***Computing Patterns in Strings***, Pearson Addison-Wesley (2003) 423 pp.

[4] Wuu Yang, **Identifying Syntactic Differences Between Two Programs**, *Software — Practice & Experience* VOL 21(7) (1991) 739–755.

[5] M. Chrochemore, ***Text Searching and Processing***, course slides, Kings College London, 2003. http://igm.univ-mlv.fr/ mac/ENS/DEA-IFA.html

[6] D. S. Hirschberg, **A linear space algorithm for computing maximal common subsequences**, *Comm* . ACM, 18, (6), 341-343 (1975).

[7] **Active File Compare**, verison 1.8 beta, Copyright 2001-2005, Formula Software, Inc. http://www.formulasoft.com

[8] D.Jackson, D.A Ladd **Semantic Diff: A Tool for Summarizing the Effects of Modifications**, *Proceedings of the International Conference on Software Maintenance*, 1994.

[9] J.E.Grass, **Cdiff: A syntax Directed Diff for C++ programs**, *Proc. USENIX C++ Conference*, Protland, OR, pp.181-193, 1992.

# Chapter 11

# Huarong Chen: Survey of Empirical Studies on Testing

## 11.1 Introduction

The software testing is the process that takes the software system as a whole (including the documentation and other such objects) and run a series of testing suits to ensure the functionality and correctness of the software.

Software testing may be partitioned into several testing phrases or types from different points of view. From the hierarchical approach perspective, software testing includes unit testing, integration testing, regression testing and system acceptance testing; From the testing type perspective, software testing consists of white-box testing and black box testing; from the testing methodology perspective, software testing contains state-based testing , transaction flow testing, exception testing, control-flow testing, and data-flow testing [6].

In this survey we focus on the empirical evaluation of white-box testing, regression testing and state-based class testing. For each testing methodology, the different testing techniques are applied. The objective of the survey is to see how effective techniques are at detecting faults, and what the cost-effectiveness trade-offs are.

## 11.2 Regression Testing

Regression testing is the process of retesting the modified software, hoping to find errors caused by changes, and provide confidence that modifications are correct. Usually developers create an initial test suite, and then reuse it for regression testing. It is an expensive process since rerunning all test cases in the test suite may require an unacceptable amount of time, especially near the end of the development cycle. Researchers have proposed five regression testing techniques as an alternative way to reduce costs by selecting and running only a

subset of the test cases in the existing test suite. In the next section the five regression testing techniques are introduced. The purpose of the experiments is to try to answer following questions:

- How do techniques differ in terms of their ability to reduce regression testing cost?
- How do techniques differ in terms of their ability to detect faults?
- What trade-offs exist between the test suite size reduction and faults detection?
- What is the testing technique most cost-effective?

## Techniques

**Minimization Techniques:** Select minimal sets of tests cases from the test suite for the program that yield coverage of modified or affected portions of the program.

**Dataflow Techniques:** Select test cases that exercise data intersections that have been affected by the modification.

**Safe Technique:** Select every test case in the test suite that, exercised at least one statement that has been deleted from the program, or that, exercise at least one statement that is new or modified in the new version of the program.

**Random Techniques:** Randomly select a predetermined number of test cases from the test suite. It may be good choice when there is no enough time to use the retest-all approach, but not other test selection tool is available.

**Retest-All Technique:** Simply reuses all existing test cases to test the modified program.

## Experiment Programs

The experiment uses nine C programs with a number of modified versions and test suites for each program are described in Table 11.1 [1].

## Experiment Measurements

### Modelling Cost

The cost is measured in terms of test suite size reduction, as T'/T. In which T' denotes the cost of regression test selection that consists of two parts, the cost of analysis required to select test cases and the cost of executing and validating the selected test cases; T denotes the total cost of executing the whole test suite.

| Program Name | Functions # | Lines of Code | Number of Versions | Average Test cases |
|---|---|---|---|---|
| replace | 21 | 516 | 32 | 398 |
| print_tokens | 18 | 402 | 7 | 318 |
| print_tokens2 | 19 | 483 | 10 | 389 |
| schedule2 | 16 | 297 | 10 | 234 |
| schedule | 18 | 299 | 9 | 225 |
| totinfo | 7 | 346 | 23 | 199 |
| tcas | 9 | 346 | 23 | 199 |
| space | 136 | 6218 | 33 | 4361 |
| player | 766 | 49316 | 5 | 154 |

Table 11.1: Subject programs for Regression Testing

**Effectiveness of Modelling Faults Detection**

The measurement of faults detection is to classify the result of testing into three outcomes. First, no test case in T reveals fault, thus, no test case in T' reveals fault, which implies that the test suite is inadequate; Second, some test cases in both T and T' reveal faults, which indicates the test selection technique does not reduce fault detection; Last, some test cases in T reveal faults, but no test case in T' reveal faults, which means the test selection technique compromises faults detection.

## Experiment Result

Table 11.2 depicts the ability of each technique to reduce test suite size [1]. Table 11.3 summarizes the effectiveness of faults detection for the five testing techniques [1]. In which the random techniques extract a constant percentage of the whole test cases with 25%, 50%, and 75

| Program Name | Safe | Dataflow | Minimization | Retest-All | Rand25 | Rand50 | Rand75 |
|---|---|---|---|---|---|---|---|
| replace | 0-100 | 0-100 | 2 | 100 | 75 | 50 | 25 |
| print_tokens | 0-100 | 0-100 | 2 | 100 | 75 | 50 | 25 |
| print_tokens2 | 5-40 | 0-40 | 2 | 100 | 75 | 50 | 25 |
| schedule2 | 65-100 | 65-100 | 2 | 100 | 75 | 50 | 25 |
| schedule | 45-100 | 45-100 | 2 | 100 | 75 | 50 | 25 |
| totinfo | 20-100 | 25-100 | 2 | 100 | 75 | 50 | 25 |
| tcas | 5-95 | 25-95 | 2-5 | 100 | 75 | 50 | 25 |
| space | 0-100 | N/A | 2 | 100 | 75 | 50 | 25 |
| player | 10 | N/A | 2 | 100 | 75 | 50 | 25 |

Table 11.2: Test Suite Size after selection (%)

| Program Name | Safe | Dataflow | Minimization | Retest-All | Rand25 | Rand50 | Rand75 |
|---|---|---|---|---|---|---|---|
| replace | 100 | 100 | 0-100 | 100 | 85-100 | 65-100 | 45-100 |
| print_tokens | 100 | 100 | 0-65 | 100 | 90-100 | 70-100 | 50-100 |
| print_tokens2 | 100 | 100 | 0-100 | 100 | 95-100 | 80-100 | 60-100 |
| schedule2 | 100 | 100 | 0-100 | 100 | 90-100 | 60-100 | 45-100 |
| schedule | 100 | 100 | 0-20 | 100 | 85-100 | 65-100 | 40-100 |
| totinfo | 100 | 100 | 0-85 | 100 | 80-100 | 65-100 | 40-100 |
| tcas | 100 | 100 | 0-100 | 100 | 80-100 | 60-100 | 40-100 |
| space | 100 | N/A | 5-40 | 100 | 85-100 | 70-100 | 50-100 |
| player | 100 | N/A | 85-100 | 100 | 85-100 | 80-100 | 45-100 |

Table 11.3: Fault Detection Effectiveness After Selection (%)

### Result Analysis

The safe technique always has 100% effectiveness, but its reduced test suite size vary widely from 0% to 100%. Dataflow shows very similar performance as safe, however, it fails to select some fault detecting test case. For example, several undetected faults occur in the schedule, schedule2, and print_tokens2 program.

The effectiveness of the random technique increases with the test suite size, but the rate of increase diminish as size increas, it is very effective in general. For example, it achieves 88% faults detection rate for random25. On the other hand, the minimization technique chooses very few test cases, while its effectiveness varies widely from 0% to 100%.

## 11.3 State-Based Testing

Statechart models are widely used to design object-oriented software systems, in particular to specify the behavior of objects. The state-based testing strategy is capable of detecting many different faults resulting from an incorrect implementation of the statechart model. Here two kinds of state-based testing techniques are studied, the round-trip testing technique and the category testing technique. The purpose of the experiments is to address following questions.

- What is the cost-effectiveness of the round-trip testing technique?
- How effectiveness can be improved by combining different techniques, such as the category partition testing technique?

## Techniques

**Round-Trip Technique:** It has two alternative forms, the first one is only to test statements in which the guard condition is true on the transition tree, the tree is constructed by breath-first or depth-first traversal of the statechart model. It is also called the weaker form of round-trip, denoted as RT; the other one is to test the RT at all possibilities of the disjunct coverage of guard condition, denoted as DC.

**Category Partition Technique:** It combines the round-trip testing technique and the black-box testing technique, denoted as CP.

## Experiment Programs

Four subject classes are used for experiments: *OrdSet* was programmed in C++, the *NameSet* and *Cache* were two subclasses inherited from the *Name* class, all three of them were programmed in Java. More characteristics of the subject classes are briefly described in Table 11.4, in which, LOC stands for the lines of the code.

| Classes | LOC | method # | States # | Transactions # | Paths # | Covered Disjunction # |
|---|---|---|---|---|---|---|
| OrdSet | 450 | 45 | 5 | 24 | 26 | N/A |
| Name | 462 | 25 | 5 | 32 | 28 | 19 |
| NameSet | 145 | 11 | 2 | 11 | 9 | 4 |
| Cache | 495 | 19 | 4 | 36 | 37 | 15 |

Table 11.4: Four Subject Classes

## Experiment Measurements

### Mutant Analysis

Mutant analysis is used to investigate the effectiveness of the round-trip testing and category partition testing on the given set of subject classes. In each experiment, the mutant versions of classes are produced by including faults seeded, the test cases are run on the mutant versions to see whether they are able to capture mutants. It is regarded as failure when uncover faults are found.

### Oracle Strategy

Oracle Strategy is used to detect failures while executing test cases. There are two kinds of approaches for that, one is to check abstract state, in which assertions are instrumented in the code so that state transition is verified by checking the expected state invariant of whole statechart; the other one is to check concrete state, in which the exact resulting state of the objects are checked while test cases are run.

## Experiment Result

In the experiments testers were divided into 10 teams with the slight variability in the number of people (3 or 4). Table 11.5 and Table 11.6 shows the effectiveness for each team by using RT and DC testing strategies respectively, the effectiveness of all subject classes were calculated by the concrete state oracle strategy [2].

Table 11.7 and Table 11.8 show the cost-effectiveness data of the class *Name* and class *NameSet* by using RT, DC, and CP respectively, the oracle strategy is applied for faults detection in both classes [2]. In addition, tables provide the overall performance of experiments, the following formulas are for three types of performance rate.

$$R_1 = \frac{\text{increased number of mutants killed}}{\text{increased number of test cases executed}}$$

$$R_2 = \frac{\text{increased number of mutants killed}}{\text{increased number of execution time executed}}$$

$$R_3 = \frac{\text{increased number of mutants killed}}{\text{increased number of LOC developed}}$$

| Classes | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | Average | Min | Max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OrdSet | 83% | 93% | 79% | 88% | N/A | N/A | N/A | N/A | N/ A | 86% | 79% | 93% |
| Name | 70% | 79% | 73% | 67% | 74% | 65% | 69% | 73% | 91% | 74% | 65% | 91% |
| NameSet | 79% | 79% | 75% | 75% | 75% | N/A | N/A | N/A | N/A | 77% | 75% | 79% |
| Cache | 76% | 76% | 60% | 60% | 60% | N/A | N/A | N/A | N/A | 66% | 60% | 76% |

Table 11.5: Effectiveness of Each Team Using RT

| Classes | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | Average | Min | Max |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OrdSet | 83% | 93% | 79% | 88% | N/A | N/A | N/A | N/A | N/ A | 86% | 79% | 93% |
| Name | 86% | 91% | 91% | 86% | 93% | 89% | 89% | 91% | 93% | 90% | 86% | 93% |
| NameSet | 79% | 79% | 79% | 79% | 75% | N/A | N/A | N/A | N/A | 78% | 75% | 79% |
| Cache | 76% | 76% | 76% | 64% | 64% | N/A | N/A | N/A | N/A | 71% | 64% | 76% |

Table 11.6: Effectiveness of Each Team Using DC

## Result Analysis

For class *OrdSet*, the effectiveness of RT and DC are identical since there were no disjunctive guard condition in its statechart. The average effectiveness of RT ranges from 66 percent to 86 percent percent, whereas effectiveness of DC ranges from 71 percent to 91, which shows that DC is slightly more efficient for faults detection than RT, the result is based on the

| | Abstract State Oracle | | | Concrete State Oracle | | |
|---|---|---|---|---|---|---|
| | RT | DC | CP | RT | DC | CP |
| Mutants killed (average) | 46 | 13 | 8 | 60 | 12 | 3 |
| Test cases | 28 | 19 | 30 | 28 | 19 | 30 |
| Driver LOC (average) | 605 | 459 | 510 | 125 | 558 | 600 |
| Execution time (avg. sec) | 5.07 | 2.53 | 5.42 | 5.10 | 8.66 | 14.11 |
| $R_1$ | 1.65 | 0.68 | 0.28 | 2.13 | 0.70 | 0.10 |
| $R_2$ | 9.11 | 1.68 | 1.56 | 11.69 | 3.74 | 0.57 |
| $R_3$ | 76.3 | 28.3 | 16.5 | 84.9 | 23.9 | 5.2 |

Table 11.7: Cost-Effectiveness Data for *Name*

| | Abstract State Oracle | | | Concrete State Oracle | | |
|---|---|---|---|---|---|---|
| | RT | DC | CP | RT | DC | CP |
| Mutants killed (average) | 17 | 0.2 | 3 | 18 | 0.4 | 2 |
| Test cases | 9 | 4 | 7 | 9 | 4 | 7 |
| Driver LOC (average) | 284 | 155 | 246 | 1003 | 485 | 456 |
| Execution time (avg. sec) | 1.66 | 0.72 | 1.15 | 1.67 | 0.74 | 1.53 |
| $R_1$ | 1.91 | 0.05 | 0.46 | 2.04 | 0.10 | 0.29 |
| $R_2$ | 10.37 | 0.28 | 2.11 | 11.02 | 0.54 | 1.30 |
| $R_3$ | 60.6 | 1.3 | 13 | 18.3 | 0.8 | 4.4 |

Table 11.8: Cost-Effectiveness Data for Class *NameSet*

abstract state oracle strategy.

When testing the class *Name* by using the concrete state oracle strategy, the ratios of RT and DC increase, the ratio of CP decreases. For instance, $R_1$ of RT increases from 1.65 to 2.13, $R_1$ of DC increase from 0.68 to 0.70, $R_1$ of CP decrease from 0.28 to 0.10. It means that, when using concrete state oracles, RT and DC are more effective for killing mutants, when using abstract state oracles, CP becomes more effective.

For Class *NameSet*, both ratios of RT and DC decrease. For example, $R_1$ decreases from 1.91 to 0.05 when using abstract state oracles, and from 2.04 to 0.10 when using concrete state oracles. This is because that the *NameSet* statechart has less disjunctive guard conditions than *Name*'s.

## 11.4 White-box Testing

Unlike black-box testing based on requirements and functionality of the software application, white-box testing is based on knowledge of the internal logic of the application code, such as the coverage of code statements, branches, paths, and conditions. All-edges testing and all-uses data flow testing are two commonly used techniques for white-box testing, which are described details in the next section. The experiments intent to address the following two questions.

- Which testing technique is more effective at faults detection and at what condition?
- What is the relationship between the percentage of definition-use associations covered by a test set and the set's effectiveness?

### Techniques

**All-Edges Testing:** Also known as branch testing, it demands that every edge in the program's flow graph should be executed by at least one test case. The all-edges testing is a relatively weak criterion since it is often easy to derive a test set which covers all the edges of the program without exposing a bug [3] [4] [5].

**All-Uses Data Flow Testing:** It requires the test data to exercise paths from points at which variables are defined to points at which their values are subsequently used, namely, the all-uses flow testing demands that the test data cover every definition-use association (DUA) in the program [3] [4] [5].

## Experiment Programs

The selected nine C programs are used for experiments, the information of programs are briefly described in Table 11.9 [5].

| Program Name | Edges # | DUAs # | Executed Edges | Executed DUAs | Failure Rate |
|---|---|---|---|---|---|
| detm | 78 | 298 | 74 | 103 | 0.001 |
| find1 | 34 | 114 | 34 | 93 | 0.066 |
| find2 | 34 | 114 | 34 | 93 | 0.018 |
| matinv1 | 78 | 298 | 74 | 106 | 0.001 |
| matinv2 | 30 | 81 | 30 | 62 | 0.001 |
| strmtch1 | 13 | 49 | 13 | 49 | 0.032 |
| strmtch2 | 14 | 56 | 14 | 54 | 0.062 |
| textfmt | 21 | 50 | 21 | 42 | 0.052 |
| transpose | 44 | 97 | 42 | 88 | 0.023 |

Table 11.9: Subject Programs for White-box Testing

## Experiment Result

To estimate the effectiveness of the two testing techniques in general, the statistics hypothesis testing method is used. The confidence interval gives an indication of how much a testing technique is better than the other, the default confidence level is with $\alpha = 0.01$, namely, 99% confidence.

Table 11.10 summarizes the results of comparisons of effectiveness of all-edges testing technique and all-uses testing technique [5]. Table 11.11 shows the effectiveness of two testing techniques by grouping the test cases into the different size [5], in which $N_e$, $N_u$ denote the total numbers of test cases for all-edges testing and all-uses testing respectively; $p'_e$ indicates the sample proportion of all-edges testing sets that expose and error, $p'_u$ is for the sample proportion of all-uses testing sets; $p_e$ and $p_u$ stands for the popular proportion of all-edges and all-uses testing techniques in general.

## Result Analysis

All-uses testing has better performance than all-edges testing in the find1 program, but not true for the find2 program. For matinv1 program, all-uses testing appears to be guaranteed to detect faults, while for matinv2 program, all-uses testing performs poorly. In the four subject programs, determinant, mative1, testfmt, and strmtch1, all-uses testing technique guarantees the faults detection.

| Program Name | $N_e$ | $P'_e$ | $N_u$ | $P'_u$ | $p_e < p_u$ | Confidence Interval |
|---|---|---|---|---|---|---|
| detm | 169 | 0.141 | 7 | 1.000 | yes | [0.00,0.08] vs [0.52,1.00] |
| find1 | 1678 | 0.557 | 775 | 0.667 | yes | [0.06,0.16] |
| find2 | 3178 | 0.252 | 43 | 0.256 | no | |
| matinv1 | 3410 | 0.023 | 76 | 1.000 | yes | [0.02,0.03] vs [0.94,1.00] |
| matinv2 | 4789 | 0.001 | 4406 | 0.001 | no | |
| strmtch1 | 1584 | 0.361 | 238 | 1.000 | yes | [0.33,0.39] vs [0.98,1.00] |
| strmtch2 | 1669 | 0.535 | 169 | 0.615 | no | |
| textfmt | 1125 | 0.520 | 12 | 1.000 | yes | [0.48,0.56] vs [0.68,1.00] |
| transpose | 1294 | 0.447 | 13 | 0.462 | no | |

Table 11.10: Effectiveness of all-edges and all-uses

As a result, all-uses testing is more effective than all-edges testing at 99% confidence in the five of nine subject programs. Moreover, the probability P of error detection is 1.0 at high values of coverage, and it decreases rapidly as the coverage decreases.

| Program Name | Size | $N_e$ | $P'_e$ | $N_u$ | $P'_u$ | $p_e < p_u$ |
|---|---|---|---|---|---|---|
| detm | 1-6 | 5 | 0.000 | 0 | N/A | |
| | 7-12 | 5 | 0.000 | 0 | N/A | |
| | 13-18 | 10 | 0.000 | 0 | N/A | |
| | 19-24 | 25 | 0.040 | 1 | 1.000 | |
| | 25-30 | 73 | 0.000 | 0 | N/A | |
| | >30 | 51 | 0.118 | 6 | 1.000 | yes |
| find1 | 1-5 | 211 | 0.299 | 1 | 0.000 | |
| | 6-10 | 470 | 0.440 | 97 | 0.474 | no |
| | 11-15 | 497 | 0.616 | 285 | 0.653 | no |
| | 16-20 | 500 | 0.718 | 392 | 0.727 | no |
| find2 | 1-5 | 205 | 0.088 | 0 | N/A | |
| | 6-10 | 482 | 0.618 | 0 | N/A | |
| | 11-15 | 496 | 0.228 | 4 | 0.000 | |
| | 16-20 | 1999 | 0.296 | 39 | 0. 282 | no |
| matinv1 | 1-6 | 205 | 0.000 | 0 | N/A | |
| | 7-12 | 484 | 0.015 | 6 | 1.000 | yes |
| | 13-18 | 550 | 0.013 | 7 | 1.000 | yes |
| | 19-24 | 579 | 0.040 | 1 | 1.000 | yes |
| | 25-30 | 73 | 0.021 | 12 | 1.000 | yes |
| | 31-35 | 498 | 0.032 | 16 | 1.000 | yes |
| | 36-40 | 499 | 0.042 | 21 | 1.000 | yes |
| matinv2 | 1-5 | 4090 | 0.001 | 3714 | 0.001 | no |
| | 6-10 | 699 | 0.001 | 692 | 0.001 | |
| strmtch1 | 1-5 | 194 | 0.155 | 0 | N/A | |
| | 6-10 | 416 | 0.298 | 77 | 1.000 | yes |
| | 11-15 | 484 | 0.388 | 161 | 1.000 | yes |
| | 16-20 | 490 | 0.469 | 0 | N/A | |
| strmtch2 | 1-5 | 238 | 0.366 | 1 | 1.000 | |
| | 6-10 | 447 | 0.438 | 16 | 0.438 | no |
| | 11-15 | 486 | 0.591 | 49 | 0.592 | no |
| | 16-20 | 498 | 0.649 | 103 | 0.650 | no |
| textfmt | 1-5 | 99 | 0.354 | 0 | N/A | |
| | 6-10 | 258 | 0.399 | 0 | N/A | |
| | 11-15 | 348 | 0.511 | 2 | 1.000 | |
| | 16-20 | 420 | 0.640 | 10 | 1.000 | yes |
| transpose | 10-20 | 359 | 0.306 | 1 | 0.000 | |
| | 22-32 | 935 | 0.0.501 | 12 | 0.5000 | no |

Table 11.11: Effectiveness comparison of All-edges and all-uses by size

## 11.5 Conclusion

For regression testing, minimization selection technique produces the smallest and the least effective test cases. The random selection technique is effective as minimization at the half of time, half of time it is not. The safe and dataflow techniques have nearly equivalent average behavior in the testing cost and effectiveness of faults detection. Because dataflow technique requires at least as much analysis as the safe techniques, we may conclude that dataflow testing technique is very useful in regression task.

For state-based class testing, round-trip path testing technique is not likely to be sufficient to catch most of faults presented in the code. For example, only one of conditions is evaluated while the weaker form of round-trip path testing is chosen for the disjunctive guard conditions. The category partition technique can be used to test methods independently since it combined round trip testing with black-box testing methodology, also it is able to detect a large percentage of potential faults while its cost significantly increases. Furthermore, when adopting oracle strategy for failure rate detection, to check state invariant assertions are not effective as to check the concrete state.

For white-box testing, although all-uses testing technique does not always perform significantly better than all-edges testing technique, it does in most cases. For example, all-uses testing is extremely effective than all-edges testing for the five of nine subject programs, it could guarantee faults detection efficiently.

## Bibliography

[1] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pages 184-208, 2001.

[2] Lionel C. Briand, Massimiliano Di Penta, and Yvan Labiche, Assessing and Improving State-Based Class Testing: A Series of Experiments, *IEEE Transactions on Software Engineering*, pages 770-793, November, 2004.

[3] Phyllis G. Frankl, and Oleg Iakounenko. Further Empirical Studies of Test Effectiveness, *Foundations of Software Engineeringb*, pages 153-162, 1998.

[4] Phyllis G. Frankl, and Stewart N. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing, *IEEE Transactions on Software Engineering*, pages 774 -787, 1993.

[5] Phyllis G. Frankl, and Stewart N. Weiss. An Experimental Comparison of the Effectiveness of the All- uses and All-edges Adequacy Criteria, *International Symposium on Software Testing and Analysis*, pages 154-164, 1991.

[6] Shil Siegel, and Robert J. Muller. Object Oriented Software Testing, *Wiley Computer Publishing*, 1996.

# Chapter 12

# Wen Yu: Survey of Studies on User Interface Design

This paper reports on the survey of study on user interface design. The highest level of guidance for user interface designers is theory or model. Theories provide abstract model for the user interface design. After building the model, we need some design knowledge, such as principles and guidelines, to guide us in the following design. However, there are some problems have been reported when we apply principles and guidelines without knowing what the context or problem is. Patterns explicitly are related to a context and are problem centered. The solution in patterns is proven. In addition, pattern language provides common language for the design team member to communicate with each other.

## 12.1 Introduction

User interface design plays an important part in many disciplines. Since computers have been used in so many areas of our everyday life, user interface design for computer system plays a vital part in the system's efficiency. User's requirements have changed from *being able to do a task on a computer* to *being able to do a task easily by using the computer*. This paper will give a survey on improvement of the usability of user interface design.

## 12.2 Theories, Principles, and Guidelines

Traditionally, there are three level of guidance for designers of user interfaces [1]:

- high-level theories or models,
- middle-level principles,
- specific and practical guidelines.

The theories or models offer a framework or language to discuss issues that are application independent, whereas the middle-level principles are useful in weighing more specific design alternatives. The practical guidelines provide helpful reminders of rules uncovered by previous designers.

## High-level Theories

A theory should be understandable, produce similar conclusions for all who use it. In this subsection, we will look at some examples of high-level theories (models).

### Conceptual, Semantic, Syntactic, and Lexical Model

This four-level approach model was proposed by Foley and van Dam in the late 1970's. [2]

1. The *conceptual* level is the user's mental model of the interactive system.
2. The *semantic* level describes the meanings conveyed by the user's command input and by the computer's output.
3. The *syntactic* level defines how the units are assembled into a complete sentence.
4. The *lexical* deals with device dependencies and with the precise mechanisms.

This approach uses top-down strategy. It matches the software architecture and is easy to understand.

### GOMS Model

Card, Moran, and Newell proposed this model in 1980. GOMS stands for *goals, operators, methods, and selection rules.* [3]

Users formulate goals and subgoals. Then, they achieve them by using methods. The operators are cognitive acts whose execution is necessary to change any aspect of the user's mental states. The selection rules are the control structures for choosing among the several methods.

This model is very popular in 1980's, and there are lot of variations for different task types.

### Seven Stages of Action

Norman developed this model in 1988. [4]

The seven stages are:

1. Forming the goal;
2. Forming the intention;

3. Specifying the action;
4. Executing the action;
5. Perceiving the system state;
6. Interpreting the system state;
7. Evaluating the outcome.

Placing his stages in the context of cycles of action and evaluation, Norman distinguishes his model from other models.

### MODEST

Larry Birnbaum, Ray Bareiss, Tom Hinrichs, and Christopher Johns proposed this interesting model in 1997. [8] MODEST stands for *Model-based Design Employing Standardized Tasks*. It is a prototype tool that uses explicit, standardized task models from libraries of standardized, reusable tasks to drive the interface design process.

The design of an interface using MODEST comprises three primary phases:

1. Modeling the current application task by selecting and parameterizing task models from a standardized library;
2. Identifying specific interface actions (gestures) and multimedia resources to represent those sub-tasks and conceptual entities;
3. Graphically arranging and sizing the interface objects on the screen.

Birnbaum and his colleagues believe that by using their models, it is possible to capture some of the design expertise and to amortize much of the labor required for building effective user interfaces.

This idea is somehow similar to the pattern approach, which we introduce later.

### Object Model

The Object Model defined in [5] is as follows:

1. An *object* is a building block for simulation models.
2. Objects may be *composite*.
3. Part of the state of every object is its *class*, which determines its behavior in the sense that two object of the same class will respond differently to messages only if their states differ.

4. A class may have subclass (or derived classes); the class is called a superclass (or base class) relative to its subclasses, which *inherit* from it.

From above definition, we can see that Object Model of user interface design contains the three main points of object-oriented model in software design, which is encapsulation, polymorphism, and inheritance. This model leads us to design object-oriented user interfaces, which are very popular today.

The basic components of an object-oriented user interface include all of the features of graphical user interfaces. However, they are different. The main differences lie in the models underlying the interface. Graphical user interfaces are application oriented, and as their name suggest, object-oriented user interfaces are object oriented. The differences are listed in [6].

## Principles and Guidelines

Some researchers refer to the principles and guidelines being the same level. In 1986, Smith and Mosier have compiled 944 guidelines in a 478 page report.

The purpose of this guidelines is to capture design knowledge into small rules, which can then be used in construction of new user interfaces. It has often been reported that it has a number of problems when being used [10]. For example, it is often too simplistic or too abstract; it can be difficult of select; it can be difficult to interpret; it can be conflicting, etc.

One of the reasons for these problems is the fact that the guidelines suggest a general absolute validity but in fact, they can only be applied in a specific *context* to solve a specific *problem*. The context and problem is crucial for knowing which guidelines to use and why. Only having the guidelines, it is often difficult to see what the problem is and why the guideline is like it is.

There is plenty of good literature out there on the guidelines. By studying these guidelines, we all know that we should direct manipulation, immediate feedback, protection from accidental mistakes, etc. However, it is hard to remember all these guidelines. And if you are a novice designer, it is sometimes difficult make the trade-offs among these guidelines when they come into conflict. They often have to figure out the best solution by guessing.

One excellent way to verify your guesses is to test your design with potential users, and lots has been written on usability testing and other field methods. Users' feedback can help us with exploring different design possibilities, refining and building the chosen design. However, if we make good design choices right at the beginning, it will cost us far less.

How can the community help inexperienced designers move away from clumsy designs and labor-intensive process without spending years learning it all the hard way?

To begin with, we could start building a user interface pattern language. As the *Design Patterns* book does to software design community, such a language would aid individual interface designers in their day-to-day work. It also could help the whole industry develop better tools and paradigms.

## 12.3 History of Pattern Languages

### Patterns in Architecture Design

In the 1970's, architect Christopher Alexander shook the architectural world with his landmark book *The Timeless Way of Building*. In this book, he explains how a hierarchical collection of architectural design patterns can be identified to make future buildings and urban environments more usable and pleasing for their inhabitants. The 253 patterns that he and his colleagues defined, published in his second book *A Pattern Language*, range from large-scale issues, via smaller-scale patterns down to patterns for the design of buildings. They are not abstract, nor specific. Instead, they are somewhere in between. A pattern describes possible good solutions to a common design problem within a certain context, by describing the invariant qualities of all those solutions.

Since the quality of a well-designed building is hard to put into words, the patterns themselves that make up that building are remarkable simple and easy to understand by laymen. It is less known that Alexander's goal in publishing this pattern language was to allow not architects, but the inhabitants themselves to design their environments. This is very similar to the ideas of user-centered design, which aim to involve end users in all stages of the software development cycle.

Each of Alexander's patterns has following components:

- *name*: identifies the pattern with a meaningful word;
- *ranking*: indicates the validity of the pattern;
- *picture*: gives a easily understood example of the pattern applied;
- *context*: explains which larger patterns it helps to implement;
- *problem* : summarizes the competing *forces*, or design trade offs, and gives the background information;
- *solution*: generalizes the examples into a clear, but generic set of instructions that can be applied in varying situations
- *diagram*: describes this solution and its constituents graphically;

- *references*: point the reader to smaller patterns that can be used to implement this pattern.

### Patterns in Software Design

Around 1987, parts of the software engineering community began to embrace the patterns concept. It is interesting that the first software pattern experiment, reported by Beck et al. at the OOPSLA conference on object-orientation, actually dealt with user interface design. In this experiment, the domain experts without prior Smalltalk experience successfully designed their own Smalltalk user interfaces after introduced basic Smalltalk user interface concepts using pattern language.

Gamma et al., known as *Gang of Four*, published their influential collection of patterns for object-oriented software design in the book *Design Patterns: Elements of Reusable Object-Oriented Software* in 1995. The essential parts of their patterns are *name*, *context*, *problem*, *solution*, *examples*, *diagrams*, and *cross-references*, which are not very different from Alexander's pattern language. However, people do not live *in* their software applications, the idea of end users designing their own product (software in this case) has not been taken over. Instead, software design patterns are become useful common language for communication among software developers, and useful tools for novice designers to shorten their learning time.

### Patterns in User Interface Design

Using patterns in user interface design was mentioned by Norman and Draper in 1986, earlier than most people expect. Apple's *Macintosh Human Interface Guidelines* quotes Alexander's books as seminal in the fields of environmental design in 1992, and the Utrecht School of Arts uses patterns from 1994. Only recently, a first workshop dedicated to pattern language for interaction design took place within the HCI community. The patterns reported by this workshop were necessarily not strictly design patterns, but rather activity patterns describing observed behavior, without judging whether these represented *good* or *bad* solutions. Subsequent workshops at UPA'99, INTREACT'99, and at CHI 2000 have confirmed the growing interest in pattern languages within the HCI community.

## 12.4 User Interface Design Patterns and Pattern Languages

A pattern is a proven solution to a recurring design problem. It pays special attention to the context in which it is applicable, to the competing *forces* it needs to balance, and to positive and negative consequences of its application. It references high-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after

the current one to further refine the solution. This hierarchy structures are comprehensive collection of patterns into a *pattern language*. [9]

As [11] indicates, patterns would help individuals build better interfaces by:

- Capturing the collective wisdom of other designers.
- Giving us a common language.
- Helping to keep one focused on essential values.
- Expressing design invariants.

A good pattern language also benefit the HCI design community:

- It would be a new vocabulary.
- It would enable us to draw on expertise in related fields.
- It may serve as a solid practical foundation on which to build new user interface tools or concepts.

Although interest in patterns for user interface design has existed for some years, patterns are still not widely available, let alone pattern collections. In this section, we will introduce four influencing pattern collections and the pattern languages being defined.

## Tidwell Collection

In [11], Tidwell has collected about 60 patterns. This is the first collection of user interface design patterns, and it covers a substantial field of user interface design issues.

The language Tidwell defined includes a *context* of use, a *problem* the designer needs to solve, a set of *forces* pushing the designer in different directions, and a *primary rule* on how those forces might be resolved to best solve the problem. *Examples* are also provided, both good and bad.

## Brighton Collection

The collection that Usability Group of the University of Brighton proposed in [12] includes a dozen patterns. This collection is not so structured and used a narrative form filled with examples of *bad* design as introduction to the pattern.

## Amsterdam Collection

Welie offers 27 patterns in his collection in [13]. These patterns are grouped according to the ergonomic principle. A categorization he used includes Modes, Selection, Guidance/Feedback, Navigation, Presentation, and Physical Interaction.

Welie suggests that a pattern for user interface design should be focused on solutions that improve the usability of the system in use [10]. The main elements of his pattern basically come from ones in the book *Design Patterns*. However, he suggests that these fields should come up with right *view*. The fields and views he suggested are as following:

- *Problem*. In contrast to SE patterns, problems in UID patterns should not be focused on constructional problems designers are facing. Problems in UID patterns should be usability problems of the system in use.
- *Context*. The context is also focused on the user. What are the characteristics of the context of use, including the tasks, users and environment for which the pattern can be applied?
- *Solution*. A solution must be described very concretely and must not impose new problems. However, a solution describes only the core of the solution and other patterns might be needed to solve sub-problems. Other patterns relevant to the solution should be referenced to.
- *Examples*. The example should show how the pattern has been used successfully in a system. It is preferred to use examples for real-life systems so that the validity of the pattern is enforced. If a writer cannot find any real-life example, the pattern is either not a good pattern or rarely applied.

He believes that the development of a pattern language is the highest goal in pattern research. It should be concluded after good patterns are developed.

## PLML

Borchers pays more attention to the pattern languages than to pattern collection. The patterns in his book *A Pattern Approach to Interaction Design* focus on *Interactive Exhibits*. In his paper [9], he gives a formal hypertext model of a pattern language, which is also called PLML (The Pattern Language Markup Language).

The formal syntactic definition is as following:

- A pattern language is a directed acyclic graph (DAG) $\mathbf{PL}=(\mathbf{P},\mathbf{R})$ with nodes $\mathbf{P} = \{P_1 \dots P_n\}$ and edges $\mathbf{R} = \{R_1 \dots R_n\}$.
- Each node $P \in \mathbf{P}$ is called a *Pattern*.

- For $P,Q \in \mathbf{P}$ : *P references* $Q \iff \exists R = (P,Q) \in \mathbf{R}$.
- The set of edges leaving a node $P \in \mathbf{P}$ is called its *references*. The set of edges entering it is called its *context*.
- The set of edges leaving a node $P \in \mathbf{P}$ is itself a set $P = \{n, r, i, p, f_1 \dots f_i, e_1 \dots e_j, s, d\}$ of a name $n$, ranking $r$, illustration $i$, problem $p$ with forces $f_1 \dots f_i$, examples $e_1 \dots e_j$, the solution $s$, and diagram $d$.

The syntactic definition is augmented with the following semantics:

Each **pattern** of a language captures a recurring design problem, and suggests a proven solution to it. The language consists of a set of such patterns for a specific design domain, such as urban architecture.

Each pattern has a **context** represented by edges pointing to it from higher-level patterns. They sketch the design situations in which it can be used. Similarly, its **references** show what lower-level patterns can be applied after it has been used. This relationship creates a *hierarchy* within the pattern language. It leads the designer from patterns addressing large-scale design issues, to patterns about small design details, and helps him locate related patterns quickly.

The **name** of a pattern helps to refer to its central idea quickly, and build a vocabulary for communication within a team or design community. The **ranking** shows how universally valid the pattern author believes this pattern is. It helps readers to distinguish early pattern ideas from truly timeless patterns that have been confirmed on countless occasions.

The opening **illustration** gives readers a quick idea of a typical example situation for the pattern, even if they are not professionals. Media choice depends on the domain of the language: Architecture can be represented by photos of buildings and locations; HCI may prefer screen shots, video sequences of an interaction, audio recordings for a voice controlled menu, etc.

The **problem** states what the major issue is that the pattern addresses. The **forces** further elaborate the problem statement. They are aspects of the design that need to be optimized. They usually come in pairs contradicting each other.

The **examples** section is the largest of each pattern. It shows existing situations in which the problem at hand can be (or has been) encountered, and how it has been solved in those situations.

The **solution** generalizes from the examples a proven way to balance the forces at hand optimally for the given design context. It is not simply prescriptive, but generic so that it

can generate a solution when it is applied to concrete problem situations of the form specified by the context.

The **diagram** supports the solution by summarizing its main idea in a graphical way, omitting any unnecessary details. For experts, the diagram is quicker to grasp than the opening illustration. Media choice again depends on the domain: a graphical sketch for architecture, pseudo-code or UML diagram for software engineering, a storyboard sketch for HCI, a score fragment for music, etc.

Each part of a pattern, and its connections to other patterns, are usually presented as several paragraphs in the pattern description. Other media, such as images, animations, audio recordings, etc., are used to augment the pattern description as described above.

In this paper, he gives an example *Designing Interactive Music Exhibits* to illustrate the usage of his pattern language.

## 12.5 Conclusions

Since the first PC's appeared, more changes have occurred in the computing technology. So does the interface between end users and the system. From command line, menu to graphical object-oriented user interfaces, the evolution of the user interface shows how dramatic the change of the apparent of the user interface. However, from designer point of view, the change is actually the model underlining each interface.

The Object Model is based in human cognition. The issue of objects is not about user interfaces, or object-oriented programming, or anything to do with computer. It is about how people perceive and act upon the world. [5]

Middle-level principles and guidelines have since long been used to capture design knowledge and help designers in using that knowledge when designing user interfaces. However, as listed in [10], there are problems when they are applied. Patterns represent proven design knowledge. They focus on the context and problem, and tell the designer *when*, *how* and *why* the solution can be applied. Therefore, patterns are more powerful.

To design high usability user interfaces, user interface designers need to cooperate with experts from different disciplines, for example, application domain experts, programmers, psychologists, graphic and media designers, etc. However,the team members lack a common terminology to exchange ideas, opinions, and values. A pattern language can overcome this problem since it is domain-independent. It can provide the common language within the design team.

The design patterns and pattern languages in user interface design does not enter their

mature stage. They need to be proven and improved by widespread use.

# Bibliography

[1] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (Second Edition). Addison-Wesley Publishing Company, 1992.5.

[2] Foley,James D., van Dam, Andries, Feiner, Steven K., and Hughes, John F.. *Computer Graphics: principles and practice* (Second Edition). Addison-Wesley, Reading, MA, 1990.

[3] Card, Stuart, Moran, Thomas P, and Newell, Allen. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[4] Norman, Donald A.. *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[5] Dave Collins. *Designing Object-Oriented User Interfaces*. Benjamin Publishing Company, Inc., 1994.

[6] Theo Mandel. *The Elements of User Interface Design*. John Wiley & Sons, Inc., 1997.

[7] Lon Barfield. *The User Interface: Concepts & Design*. Addison-Wesley Publishing Company, 1997.

[8] Birnbaum et al.. Interface design based on standardized task models. *International Conference on Intelligent User Interfaces*, ACM Press, New York, 65–72, 1997.

[9] Jan O. Borchers. A Pattern Approach to Interaction Design. *Symposium on Designing Interactive Systems*, ACM Press, New York, 369–378, 2000.

[10] M. van Welie, G.C. van der Veer, A. Eliens. Patterns as Tools for User Interface Design. *International Workshop on Tools for Working with Guidelines*, Springer-Verlag, London, 313–324, 2000.

[11] Jenifer Tidwell. *COMMON GROUND: A Pattern Language for Human-Computer Interface Design*. http://www.mit.edu/~jtidwell/common_ground_onefile.html.

[12] Usability Group of the University of Brighto. *Design Patterns for HCI*. http://www.it.bton.ac.uk/cil/usability/patterns/.

[13] Martijn van Welie. *GUI Design patterns*. http://www.welie.com/patterns/gui/index.html.

# Chapter 13

# Gabriel Indik: Literate Programming Editor

Introduced in the early 80's, Donald Knuth's Literate Programming approach to writing computer programs combines programming and formatting languages to produce well structured easy-to-understand software. To implement this technique a set of tools known as WEB has been developed to automatically obtain human readable documentation and computer source code from a single unified program description. In this paper a critique of Knuth's approach will be developed where both its advantages and disadvantages are analyzed. A set of changes is then proposed and used to develop a new Literate Programming tool that enhances this technique.

## 13.1 Introduction

The introduction of structured programming methodology in the early 70's significantly improved the software development process, leading to more reliable and easy to understand programs. However, the results obtained by the use of such methodology was not always not entirely satisfactory, further improvements were required to achieve high quality programs that could be easily written as well as read. Donald E. Knuth tries to fill this gap by introducing in 1985 a new technique called "Literate Programming." First published in The Computer Journal - May 1985 [4], Literate Programming changes the traditional perspective of software construction by setting the focus not on how to instruct a computer what to do, but instead, writing programs in a way that other people can understand what we want the computer to do. Thus, programs are not considered merely computer code, but works of literature, hence the name "Literate Programming." The idea behind this technique resides in the fact that formatting and programming languages combined are much more powerful than either single language by itself, so using them together enables programmers to produce rich documents that properly describe the design decisions taken at each step of a program development together with the actual source code. Automated tools can then by applied to obtain user friendly documentation for other programmers to read, and

computer source code for compilers to generate the executable version of the system. There are however drawbacks in the Literate Programming approach, such as overhead cause by the simultaneous use of different languages and additional tools, difficulty to actively debug the code being written, and extensive reading/writing required to perform structural changes to the program being developed to mention some. In this paper I describe in further detail the Literate Programming technique and analyze its advantages and disadvantages. I'll adopt a critical perspective of the tool, detailing key aspects that could be improved. Based on this critique, I then propose a set of changes that would lead to a better more refined version of the tool. Finally, all proposed changes are taken into consideration for the development and implementation of a new software tool called "Visual Literate Programming Editor." A practical example of the use of the new tool is then presented together with a set of conclusions.

## 13.2 The Literate Programming Approach

Literate Programming is a phrase coined by Donald Knuth to describe the approach of developing computer programs from the perspective of a report or prose. This is in contrast to the normal approach of focusing on the code. Knuth's insight is to think of the program as a message from its author to its readers. While typical programs are organized for the convenience of their compilers, literate programs are designed for their human readers. At some point, of course, the program must be executed by a computer. Knuth's system allows the programmer to think at a high level, and has the computer do the dirty work of translating the literate description into an executable program. It was Knuth's intention to provide a new programming perspective by which the programmer could typeset his or her work in book or article form, so that each choice of implementation, each algorithm, was clearly explained and justified. The resulting work "of literature" would then stand as the quintessential definition of a solution for the problem it addressed. The first literate programming system (WEB) to implement this concept was published by Knuth in 1981 for his TeX typesetting system. The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TeX for formatting, and a language like Pascal or C for programming. Neither type of language can provide the best documentation by itself; but by properly combining them, we obtain a system that is much more powerful than either language separately. The structure of a software program may be thought of as a "web" that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbors. TeX provides typographic tools to explain the local structure of such parts while the programming languages make it possible to specify the algorithms formally and unambiguously. Thus, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into an executable software system that matches the documentation. A more detailed introduction of Knuths approach can be found in [2]

## The WEB System

The first software tool to implement the literate programming approach was called "WEB". Developed by Donald Knuth in 1981, it was initially used to document the development of the TeX typesetting system. WEB programs are coded in modules (also called sections). A part (major module) begins a new train of thought or logical grouping for a program. All modules must contain at least one of three parts: documentation, definitions, and program code. Ideally, each part of a module should be no more than 25 lines. Modules should be subdivided until their functionality is easily comprehensible. In the documentation portion of a WEB program, the programmer should provide a literate description of the code being written and may refer to pieces from the code portion of the current module (i.e. variables, function names, reserved words, etc). Definitions are abbreviations for code to make the code more comprehensible. This will provide short cuts for the programmer, since the abbreviation can be substituted instead of rewriting the code in full each time it is needed. Finally, the code portion consists of the actual source code of the program.

**WEB Structure and Tools** In WEB, a user writes a program that serves as the source language for two different system routines as showed in Fig. 13.1. One line of processing is called weaving the web; it produces a document that describes the program clearly and that facilitates program maintenance. The other line of processing is called tangling the web; it produces a machine-executable program. The program and its documentation are both generated from the same source, so they are consistent with each other. Besides providing a documentation tool, WEB enhances the programming language by providing the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small sections and their local interrelationships. The tangle program is so named because it takes a given web and moves the sections from their web structure into the order required by the compiler. More information about the web system can be found in [3].

**Weave Output** The Weave routine produces a document which appears more like a work of literature than the standard source file listing. Weave enhances the typeset out in many ways:

1. The code will be like an annotated pseudo-code (both top-down and bottom-up code).
2. Reserved words are shown in boldface and variable names are shown in italics.
3. Weave will automatically line up and indent all statements (i.e. begin/end, if-then-else, loop constructs) regardless of the format of the source.
4. Calls to named modules will contain the module number where it was defined. Named modules will contain the statement 'This code is used in section X' where X is the module number(s) that contain the call(s) to that specific named module.
5. Boolean and relational operators will be replaced with their corresponding mathematical symbols.

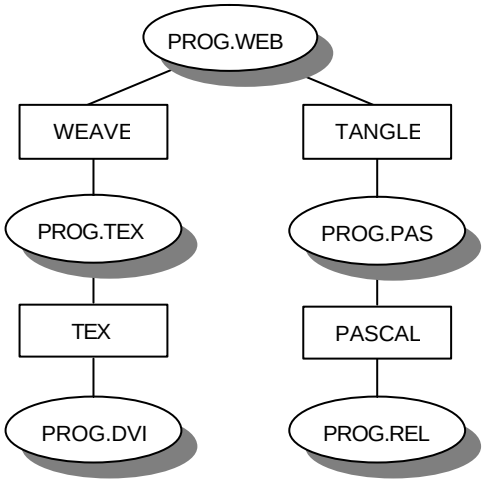



Figure 13.1: WEB structure

6. It contains a table of contents which lists each title of each part and chapter, followed by the actual sections, and ending with an index and a list of sections names.

7. Weave automatically numbers modules sequentially in the order they appear in the WEB source file. Numbering the modules makes referencing code quicker and easier.

8. Index entries include variable names, functions, keywords used in documentation.

9. Weave will automatically number the pages and produce a page header which will contain the program title.

A more detailed description of the the Weave routine enhancements can be found in [1].

**Tangle Output** Literate programming encourages programmers to write in a logical order; however, compilers can not always handle code that is "logical" to humans. To make the code readable by the compiler, Tangle will replace the module calls with the actual module definitions no matter where they appear in the program. "Since the compilation order of the code no longer dictates how the program is designed and presented the resulting program is much more comprehensible and thus will be more maintainable for the future."

## Advantages of the Literate Programming approach

Some of the advantages obtained from the use of the Literate Programming approach include flexible order of elaboration, factoring, readability, maintainability. In other words, this

technique not only provides an efficient way to combine code and documentation, but it actually enhances the quality of programs. Next each of this aspects will be explained in further detail.

**Flexible Order of Elaboration** This enables the author to divide the source program into chunks and write the chunks in any order, independent of the order required by the compiler. In principle, a programmer can choose the order best suited to explaining what he or she is doing. More subtly, this discipline encourages the author of a literate program to take the time to consider each fragment of the program in its proper sphere. The reordering is especially useful for encapsulating tasks such as input validation, error checking, and printing output fit for humans, all tasks that tend to obscure "real work" when left inline.

**Factoring** With file based languages it is quite common to see a single function definition of 80 lines or more. This is normally because the function is required to hold the full text of the algorithm it is implementing. Whilst it is possible to break the algorithm down further the overhead associated with defining the relevant functions outways its usefulness. To improve this situation Knuth introduced a decomposition facility into his meta-language. This allows one to break up the definition into its constituent parts without the need for defining new functions. Thus we are able to decompose the algorithm, discussing both it and its implementation in a more natural way, with the various parts being defined and discussed in their natural place. A tool is provided to collect the various parts together and reconstitute them into the correct order.

**Readability** By allowing users to use a more natural literary style of writing to describe the application, programmers are free to discuss the design decisions and constraints that have led to certain intricacies in the implementation. Presenting this discussion in book form allows programmers to break it up into discrete sections. Knuth believes that creating a program should be viewed as creating a work of art. The result will automatically be more readably as the author's intentions will be laid out in much more detail. The reader will have seen the development of the software through its description. Such descriptions should be of interest to any programmer.

**Maintainability** Better factoring will lead to more well thought out development. The literary style of presentation allows programmers to not only lay out the software better, but to discuss the algorithms and their intricacies in detail. When an alteration is required it should be fairly obvious which part or section of the book will need to be altered. As the system has been fully described we can read the intention of the original author. We are then required to re-write the relevant section of the book to reflect the desired alteration.

**Quality** Given that programmers have to elaborate a prose explaining each step of the program development, the quality of the program is improved. With better factoring and

documentation it is inevitable that other programmers will be able to understand the program better. Along with this understanding comes improved maintainability. If the software is easer to maintain, is better documented and better structured in the first place this has to lead to better quality software.

## 13.3 Critique of Literate Programming

We have introduced Knuth's Literate Programming approach, detailed the tools that support it and some of the advantages that can be obtained from its use. In particular, we have seen how it improves software quality, together with readability and maintainability. However, Literate Programming did not become a mainstream technique in software development. The reason for this resides in the fact that it is not possible to write literate programs quickly, and as we all know, the software industry is an impatient one. Knuth addressed the time cost issue of using the Literate Programming approach in the section "Economic Issues" of his Computer Journal article. His claim is that writing literate programs takes no longer than writing "illiterate" programs. The argument used to support this claim is that since the Weave and Tangle routines consume relatively small CPU time, then there is no overhead created by the tools. And since the programmer is forced to clarify his thoughts during the program development, then the code being written is less likely to have errors, reducing the debugging time. While these facts turn out to be true, they are not sufficient to guarantee that there is no overhead. As we will see in the next section, there are several other issues that contribute to a significant overhead.

### Time Overhead

Using the Literate Programming approach requires additional time in comparison to writing "illiterate" programs. The reason for this resides in the fact that learning this new technique, its philosophy and tools takes time. Also, writing programs for others takes longer than writing them for oneself. The simultaneous use of different languages makes the development more complex and time consuming, especially considering the combination of such languages introduce a new set of errors the developer will have to handle. Finally, we see that the debugging process also becomes more time consuming. Next, we will detail each of these aspects and explain how is it that they produce a time overhead.

**Learning the technique, its philosophy and tools** In order to develop literate programs, a software developer should first learn this new approach, install and configure the set of tools that support it, and most important of all, learn how to properly write literate programs. All these tasks take time, especially considering that one should become familiar not only with the programming language in which the software is going to be built, but also with the WEB and formatting languages required to use the Literate Programming approach. It could take a long time for a programmer who has never used this technique to mater it.

**Writing for others takes longer than writing for oneself** Literate Programming forces programmer to develop software using a completely different perspective, where the developer should first make his or her thoughts clear to others before writing the actual code. For someone who has never used this approach before, writing literate programs can be very time consuming. Also we have to consider the case where people are not comfortable with the idea of exposing the way they think out programs, in which case the time required to properly adopt this new technique would most probably be even longer.

**Use of different languages concurrently** When developing software using a traditional approach, a programmer has to think out programs in terms of the programming language. Doing this requires time, but once the developer masters the programming language, this task becomes easier. In Literate Programming on the other hand, a developer has to simultaneously work with thee different languages: the programming language in which the program is going to be built, WEB's own tag based language, and a formatting language (such as TeX). Dealing with three different languages can be not only difficult, but time consuming also. This issue is explained in further detail in [5].

**New types of errors introduced by the technique** In addition to programming errors, two new types of errors are introduced when using the Literate Programming approach: WEB structural errors and formatting errors. Web structural errors are those caused by the incorrect use of the WEB's own language required to define the structure of a program. Since both Weave and Tangle routines use such structure as an input, this kind of error can then be propagated into programming and formatting language errors. Unfortunately, neither tool provides feedback on syntactic errors. Finally, formatting errors are those cause by the incorrect use of the formatting language. Again, this errors could propagate into other types of errors (programming and structural) when executing the Weave and Tangle routines.

**Clear source code unavailable** In the Literate Programming approach, in order to ensure that all changes to the program are made to the WEB file only, the output of the Tangle routine has been specifically made as hard to read for humans as possible. While this might sound like a logical thing to do, in practice we see that in certain circumstances it is not only preferable, but mandatory to have a clear easy-to-understand source-code-only version of the program. Take as an example the use of automated tools for extended static checking, for which program source code is required. Also, it is sometimes easier to spot certain semantic errors directly from the source code than from the Weave file.

**Debugging takes longer** In the Literate Programming approach, the only way to obtain the executable version of a program is to first run the Tangle routine over the WEB file, and then compiling the output obtained. If there are programming errors in the program, these will not appear until the source code has been compiled, and in order to correct such errors, the developer should go back to the weave file, make the changes, then run the Tangle

routine again, obtain the source code and compile one more time the program. Because of this, the debugging process gets more complex and takes more time.

## 13.4 Aspects to be Improved

We have seen some of the shortcomings of the Literate Programming approach, and how they create an overhead. Next, we will propose a set of changes in order to improve this technique reducing the time required to write a literate program.

### Simultaneous use of different languages

In the Literate Programming approach, three languages are required to write a program: the WEB language, a formatting language such as TeX, and a programming language such as Pascal or C. These are combined into a single file that serves as the input to the Weave and Tangle routines. The output of the Weave routine is then processed to obtain a rich text format document while the Tangle output will be compiled to obtain the executable version of the program. The key is to observe that ultimate product that we want to obtain is not a WEB file or the Weave routine output, but rather the rich text format document obtained at the end. Thus, the reason why the programmer has to learn to use a formatting language resides in the fact that this one is needed to produce such rich text format output.

An alternative to this schema could be that where instead of combining three different languages into a single file to then process it by different routines, a developer writes a program directly using rich text format and have automated format-based parsers do the work of structuring and extracting the source code. It is common to see this kind of technique applied to modern word processors, where the user predefines a specific format for titles and then has the program generate a table of contents based on it. By using this schema, we eliminate the need of learning and simultaneously using different languages. We also get rid of the errors introduced by the use of such languages.

### Program structure specified in WEB language

Knuth's idea of decomposing a program into sections and interrelate them as a web is a very powerful innovating concept. Programmers can develop software using a top down approach, writing different parts of the program at different moments and have at all times a clear overview of the entire project. There is however a shortcoming in the way this concept has been implemented in WEB: all program sections have to be written sequentially into one single file. The problem with this approach is that extensive skimming is required when reading the program. This problem becomes more evident when dealing with large size projects. An alternative to this schema would be to utilize a visual user interface capable of showing the program sections in a tree-like structure, and displaying the contents of each section according which leaf of the tree has been selected, similar to the document explorer of a visual based operating system. This way we make it easier to read and locate any of

the program sections. Other visual oriented tools to explore literate programs can be found in [6].

### Tangle output unreadable

As mentioned in the previous section, the output of the Tangle routine has been purposely made as hard to read as possible. The reason for this is that Knuth believes that this way programmers will have no choice but to make modifications in the WEB file only. This way, code and documentation are kept and updated together. The problem with this approach is that for those situations where having clear source code is desirable, this wont be available. Such situations include, for example, the use of external automated tools for extended static checking. An alternative to this approach is to have the source code generator tool produce source code that is easy to read, and leave the decision of whether going back to the WEB file or not to the programmer.

### Time consuming program debugging

As introduced earlier, in the Literate Programming approach, the only way to obtain the executable version of a program is to first run the Tangle routine over the WEB file, and then compiling the output obtained. If there are programming errors in the program, these will not appear until the source code has been compiled, and in order to correct such errors, the developer should go back to the weave file, make the changes, then run the Tangle routine again, obtain the source code and compile one more time the program. An alternative to this schema would consist in integrating the text editor being used to develop the Literate Program with a compiler/debugger application. This way, the use of external routines is eliminated and debugging time is reduced.

## 13.5 Development of a new Literate Programming tool

A critique of the Literate Programming approach has been introduced together with a set of proposed changes in order to improve and enhance this technique. These changes have been then taken into consideration for the development of a new tool called "Literate Programming Editor." This tool allows programmers to develop programs using the Literate Programming approach, and by incorporating the set of proposals discussed in this paper, it makes literate program writing an easier and less time consuming. The software development model selected for the construction of this tool is the "Evolutionary Model." The reason for this selection resides in the fact that there isn't a clear outline or definition of the software to be constructed but instead a proposed set of changes. The evolutionary model [7] is based on the idea of developing an initial implementation and refining it through many versions until an adequate system has been developed as shown in Fig. 13.2.

Rather than have separate specification, development and validation activities, these are carried out concurrently with rapid feedback across these activities. The programming lan-

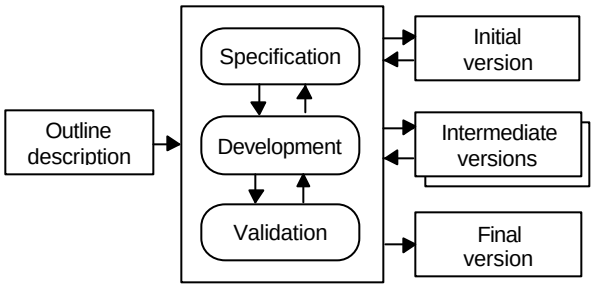



Figure 13.2: Evolutionary model

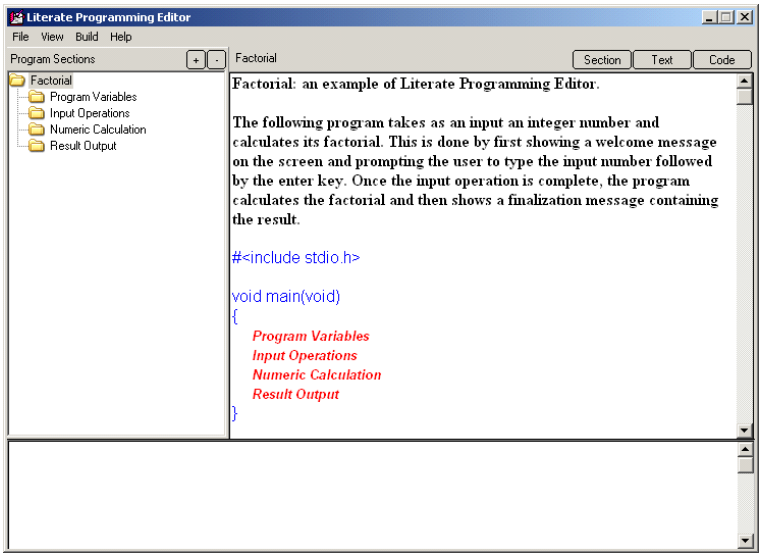
guage selected for this development is OpenScript. The reason for this is that this language supports native rich text format (RTF) while at the same time it offers exceptional string handling, making it an ideal tool for this project. The programming language selected to be supported by the new Literate Programming Editor is C, and the integrated compiler is Borland C++ Compiler.

## 13.6 Factorial: an example of Literate Programming Editor

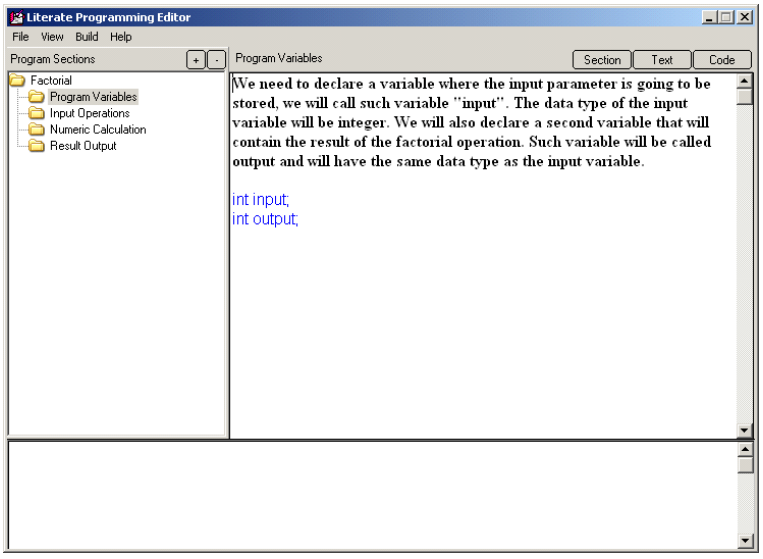
To show how the Literate Programming Editor tool works, an example is introduced. This example consists of a small simple program called "Factorial." The purpose of this program is to iteratively calculate factorial numbers. When started, the program should display a welcome message on screen and prompt the user to type a number on the key board. Next, the program should calculate the factorial of the number entered and display the result on the screen.

### Factorial program sections

The graphical user interface of the Literate Programming Editor is divided into two main areas: program structure on the left, which displays the different sections of the program in a tree-view fashion, and the contents area on the right, which contains the literate prose together with the program source code and section names. In this example we see that that five sections have been defined: "Factorial": top level description of the program, "Program Variables": section where the variables required are defined, "Input Operations": set of statements required to perform the data input, "Numeric calculation": section where the input data will be used to calculate the factorial number, and "Result Output": section where the result of the calculation is displayed on the screen. Note that the literate prose, source code and section names on the contents screen area appear with different styles and colors. This is done by selecting each of these types of elements and clicking on the corresponding

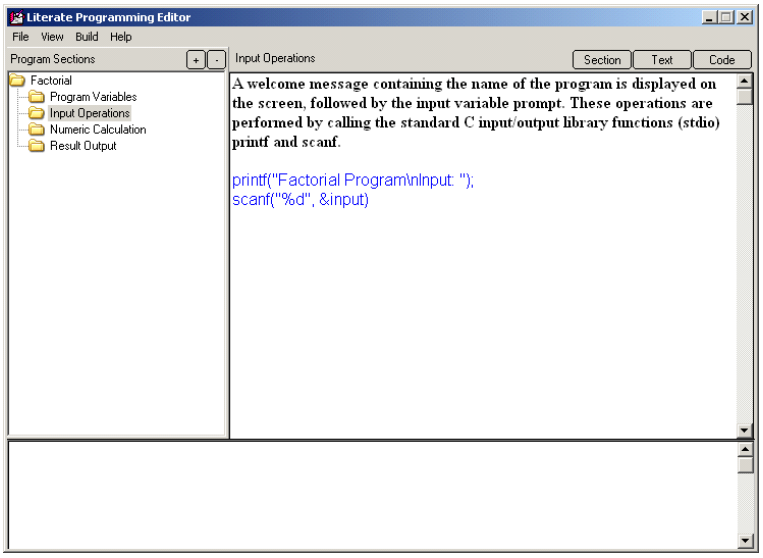



button located in the top-right corner of the screen. Once a font style and color has been assigned to a text fragment, the Literate Programming Editor can differentiate between these three kinds of elements and process them properly for program structuring, index creation, code generation and debugging.

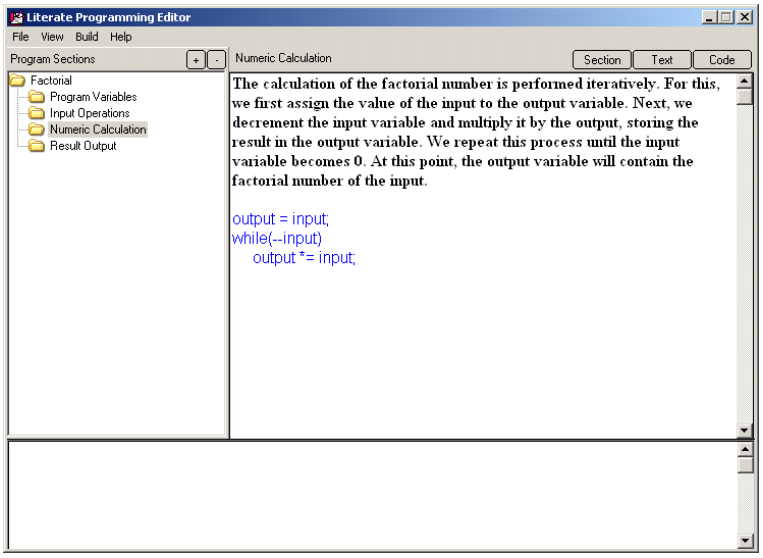



**Program Variables section** As it can be observed, the contents area displays only the section highlighted in the program structure tree. This eliminates the need to skim through the entire program code and description to locate a specific section. Once again, a predefined

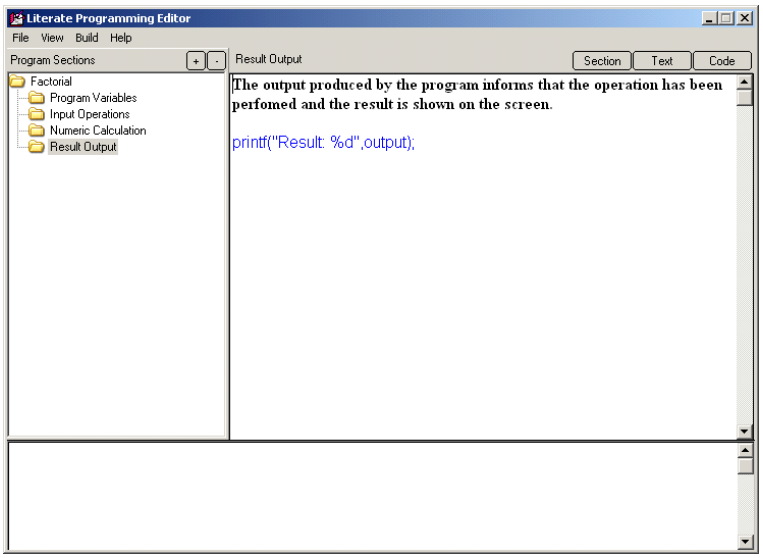
text style and color has been assigned to the text by using the button commands located in the upper-left corner of the screen.




**Input Operations section** Each time a section needs to be added (or eliminated), It suffices to use the "+" (add) and "−" (remove) commands located in the upper-left corner of the structure screen area of the program. For this example, the "+" (add) command has been used four times to add each of the sections that are included into the factorial program.



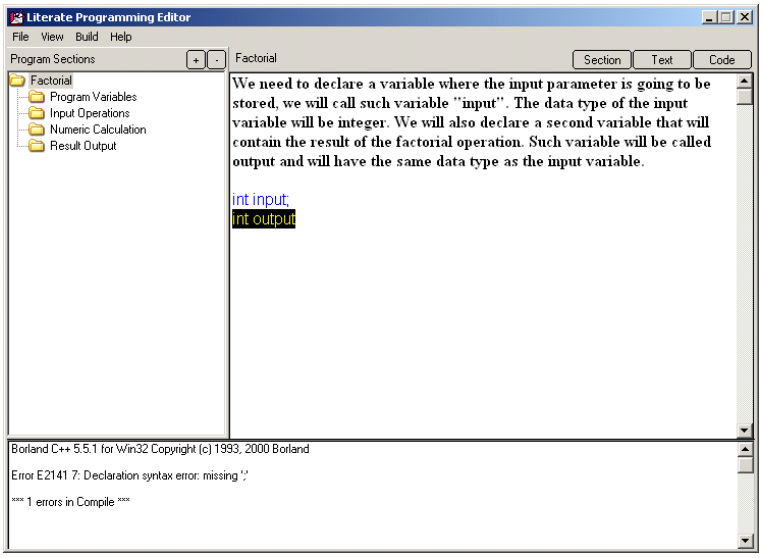
**Numeric Calculation section** The purpose of this example is to show how a small Literate Program can be written easily and quickly. For this reason, the factorial number calculation is performed iteratively in order to keep the example short and easy to understand.




**Result Output** Once the calculation of the factorial number has been performed, the last section of the program displays the result on the screen.

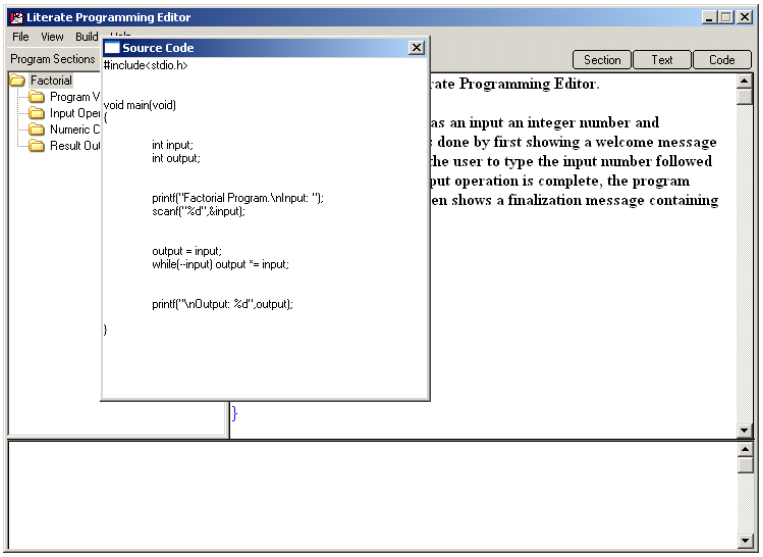
## Debugging

As previously mentioned, one of the key aspects to improve from the traditional Literate Programming approach in order to eliminate time overhead is its debugging process. In Literate Programming Editor, the compiler has been integrated into the Literate editor tool, so there is no need to switch to different programs for debugging. Also, error references are not only made directly to the code in the Literate description, but automatically shown on screen. Take the following example. Here, an attempt to compile the program has been made and a syntax error has been found. Under these circumstances, the Literate Programming Editor will automatically show on the contents screen area the section where the error is located, highlight the statement that is producing the error, and give a textual description on what the error is. In this particular example, we see that after the declaration of one of the variables (output), is missing its end-of-statement semicolon (recall the language selected for this version of the tool is C, and in C language semicolon is mandatory after every single statement).



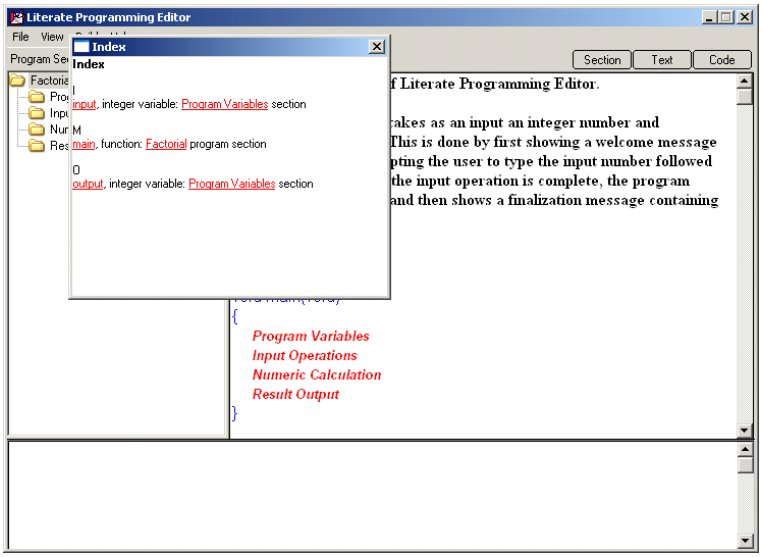

## Source code generation

Another key element to improve from Knuth's original Literate Programming approach is the automatic code generation output. In the Literate Programming editor, the source code the tool outputs is easy to understand, making it possible to apply other automated tools to it, such as extended static checking tools.



### Index creation

The Literate Programming Editor automatically generates an index containing all the elements of the program. This feature is also available in WEB, however, in order to generate the index in WEB it is necessary to execute the Weave routine and then process the TeX output to obtain it. In the Literate Programming Editor on the other hand, the index is available at all times, making it easier to use during the development. All identifiers defined




in the program are highlighted together with the program sections in which they are located.

## 13.7 An Insight into the Literate Programming Editor development

It is beyond the scope of this paper to detail all the different processes that were involved in the implementation of the new tool. However, as its author I believe it is worth introducing some of the key challenges that had to be overcame during its development, in particular rich text format parsing and compiler integration.

### Rich text format parsing

As introduced in Section 5, the programming language selected for the implementation of the Literate Programming Editor tool is OpenScript. The reason for this is that this language supports native rich text format (RTF) while at the same time it offers exceptional string handling, making it an ideal tool for this project. In OpenScript, each character that composes the text entered by the user in the contents screen area is an object. These objects (text characters) are of course ordered, this is, they have an associated integer value

indicating their order in the text. At the same time, each of these objects have font style and color attributes. In order to automatically process the program the user entered, the tool must be able to identify each element in the text and determine whether it is literate prose, source code or a program section name. By using the set of character attributes in OpenScript, a parser that identifies text depending on its format can be written as follows.

```
i = 1
  while i < charCount(text)
    while the fontStyle of the
    character i of the text
    is the fontStyle of character
    i + 1 of the text
            put the character i of
            the text after the buffer
            increment i
  end
  process(buffer, fontStyle of the
  character i - 1 of the text)
  clear buffer
end
```

As it can be seen, this parser traverses the characters that composes the text, storing in a buffer the set of characters read so far, and when the font style changes, it calls a processing routine using the buffer contents. This routine can then call auxiliary functions depending on the font style parameter, this is, if the font style is that defined for source code, then a routine to handle a source code token is called, giving the contents of the buffer as the token. The reader may be surprised to know that the parser segment just introduced is not pseudo code, but actual OpenScript statements. Thus, the parser used in the implementation of the Literate Programming Editor tool is an extended version of the one introduced in this paper.

## Compiler integration

In order to be able to debug the program code within the Literate Programming Editor, a C/C++ compiler has been incorporated into the tool. As we have seen, the errors found in compilation are directly referred to the Literate program description. To achieve this, whenever the user selects the compile option, the Literate Programming Editor tool automatically generates the program source code using the rich text format parsing technique introduced in the previous section. It adds special comments at the end of each line indicating where exactly each statement is located in the Literate description. Following the Factorial example, the output code would look as follows.

```
#include<stdio.h> //Factorial line 11
```

```
void main(void) //Factorial line 12
{ //Factorial line 13
int input; //Program Variables, line 7
int output //Program Variables, line 8
printf("Factorial Program.\nInput: ");
  //Input Operations, line 5
scanf("%d",&input);
  //Input Operations, line 6
output = input;
  //Numeric calculation, line 6
while(--input) output *= input;
  //Numeric calculation, line 7
printf("\nOutput: %d",output);
  //Result Output, line 3
}//Factorial line 14
```

The compiler then processes the source code file and outputs the compilation result into another file. These last file will either be empty, in which case compilation was successful, or it will contain a list of errors with their descriptions and locations. In the semicolon missing error introduced in the debugging example, this file would look as follows.

```
Error E2141 line 5:
  Declaration syntax error: missing ";".
```

Note that the error is being referenced to the source code file that was generated for compilation. For this, the Literate Programming Editor tool parses this error description file and automatically sets the references using the source code file previously generated in order to be able to show the errors within the editing environment. This is, not that the error that the compiler output is line 5, and line 5 of the souce code file contains a comment at the end: "Program Variables, line 8". This is all the program needs to be able to perform debugging. A key element that was taken into consideration for the development of this part of the tool, is that all these operations and intermediate files are hidden to the user, giving the impression the Literate Program is being compiled.

## 13.8 A Look into the Future

We have introduced Knuth's Literate Programming approach to writing structured programs, reviewed its tools (WEB), its advantages and disadvantages. A critique of the technique has been developed and a set of changes to it has been proposed. These changes were then used to implement a new Literate Programming tool to enhance the Literate Programming approach. Now, many of the changes proposed are not radical changes to the approach itself, but rather improvements over the way the set of tools that implement it work. The reader may wonder why is it that Donald Knuth didn't think about such set of improvements at the time he

introduced Literate Programming. The reason for this resides in the fact that this technique was introduced in the early 80's. In other words, Knuth developed this technique within the limitations of the tools available at that time. Still, Knuth actually did think about many of the changes that have been introduced in this paper, and stated them in the "Retrospect and Prospect" section of [4]. He envisioned tools that would integrate editors, compilers into graphical development environments to assist programmers construct software. One can interpret the work presented in this paper as a step forward in the development of this technique, a step Knuth already envisioned at the time he introduced this technique. As a last reflection, the reader may wonder what new improvements might be proposed over this new tool in the following years.

# Bibliography

[1] Bart Childs. An introduction to the web style of literate programming. Technical report, Texas A&M University, College Station, 1992. ftp.cs.tamu.edu/pub/tex-web/web/docs.

[2] Bart Childs. Literate programming, a practitioner's view. *TUGboat*, 13(3):261–268, 1992.

[3] D. E. Knuth. The web system of structured documentation. Technical Report CS980, Stanford University, 1983.

[4] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[5] Norman Ramsey. *Literate Programming Simplified*. IEEE Computer Society, 1994.

[6] Thomas Setz. Experience with literate programming or towards qualified programming. Technical Report No. TI-15/97, Technische Hochshule Darmstadt, 1997.

[7] Ian Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1995.

# Chapter 14

# Michael Kucera and Reza Sherafat: Empirical Analysis of the Use of Exception Handling

## 14.1 Preliminaries

The support for exception handling in modern object oriented languages such as Java is very sophisticated. There are many different ways in which exception handling can be used, and because of this, diverse opinions on the proper use of exceptions have arisen [2, 4, 10, 3]. One of the largest areas of debate centers on the use of checked exceptions. The creators of Java believed that checked exceptions lead to good programming practices and overall more robust code. Some groups disagree with this and feel that checked exceptions have the opposite effect, making programs less maintainable and leading to situations that hurt robustness such as empty catch blocks. For these reasons the designers of C# made a controversial decision to omit checked exceptions from their language.

Furthermore there are many different and sometimes conflicting possible usage patterns for exceptions. For example, it is often argued that exceptions should only be used to indicate errors, and that it is bad programming style to use exceptions as part of normal program flow. This is a debatable view but the question we are most concerned with is how exceptions are actually used in practice.

In order to understand many of the views it is important to study the usage patterns of exception handling, and to measure how exceptions are used in real world projects. In order to accomplish this, we first set out several hypotheses for which we wish to be able to gather statistics. Then we will select some open source projects, based on criteria we choose, and analyze the sources in order to gather useful metrics regarding the use of exception handling.

We chose Java as the language of study because of its modern exception handling architecture and also for the abundance of freely available open source projects. Furthermore we wish to shed some light on the real consequences of the design decision to include checked exceptions in the Java language and whether checked exceptions actually lead to the bad

situations that the proponents of C# describe. For this reason we choose many open source projects, some of which were developed by professional programmers that should be disciplined enough to write what many would consider to be good code.

We begin by setting out several hypotheses regarding the use of exception handling. Then we create analysis techniques that will gather information regarding these hypotheses. We wish to automate the process of gathering usage statistics as much as possible so that we can analyze large amounts of code.

This paper is organized as follows: in Section 2 we provide an introduction to the exception handling architecture of Java. Section 3 describes the checked exception debate and the issues involved. Section 4 outlines each hypothesis in detail. Section 5 describes the analysis techniques that were used. Gathered data is presented and conclusions are drawn in Section 6. Source code for the automatic analysis tools that were developed for this study is given in the appendices.

## 14.2 Exception Handling in Java

An *exception* is an event that causes a program to jump out of its normal flow of execution [9, 5]. Control flow is then passed to an appropriate block of code known as an *exception handler*. In Java, exceptions are represented as objects and usually contain information about the event that created the exception. Creating an exception and passing it to the run-time system is known as *throwing an exception*. The run-time system then attempts to find an appropriate exception handler. This is known as *catching an exception*.

The search starts within the currently executing method and proceeds up the call stack until a handler is found. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler. If an appropriate exception handler cannot be found then the program is terminated and a message is usually displayed.

Exception handling is supported by many programming languages including Eiffel, OCaml, Ruby and several others [11, 8]. However, the exception handling mechanism in Java is more sophisticated than in the languages just mentioned. Since exceptions are objects they are subject to the same typing and inheritance rules as the rest of the Java language. Hierarchies of custom exception types can be created and subtyping can be used in the process of choosing an appropriate exception handler. Exception objects can be made to carry a great deal of information if necessary. This is in contrast to a language like Eiffel in which an exception carries no information whatsoever. Exceptions can be thrown by the system in response to undesirable events or then can be thrown manually in the program code. Exceptions can even be stored within other exceptions; this is known as *exception wrapping*.

One very interesting thing about the Java exception architecture is that it supports two broad categories of exception; runtime exceptions and checked exceptions. Runtime exceptions do not have any restrictions placed on them by the compiler whereas checked exceptions are subject to the *catch or specify* requirement [2]. This requirement states that if a program statement can throw a checked exception then one of two conditions must be

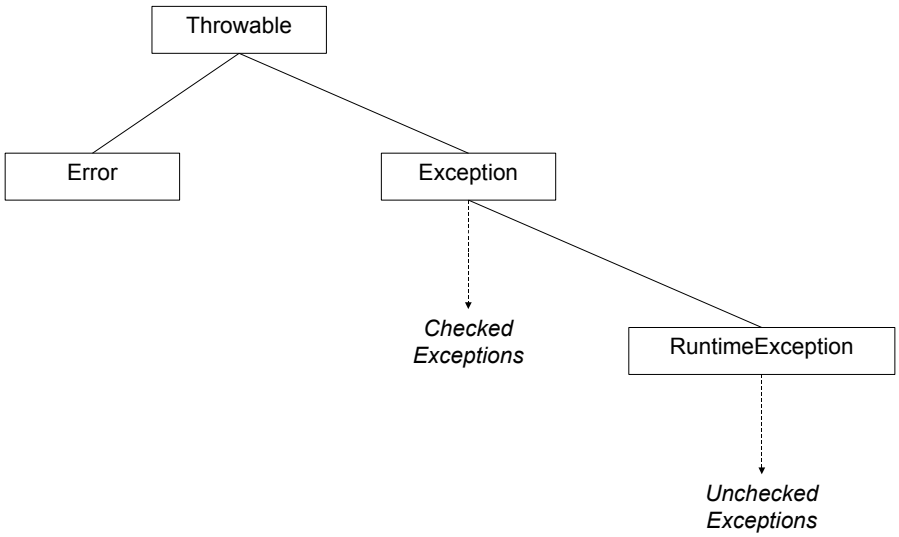



Figure 14.1: The Java core library exception hierarchy

satisfied; 1) the statement must be within a try/catch structure that is of a matching type or 2) the enclosing method must declare to throw that type of exception (or a supertype) with a throws clause.

The Java core library provides a simple class hierarchy that is the basis for all exceptions [5]. The parent of all exceptions and errors is the Throwable class. If a program contains a throw statement then the type of the expression in the statement must be Throwable. The direct descendents of Throwable are Error and Exception. Errors are meant to represent unrecoverable system errors that no program should attempt to handle, examples include OutOfMemoryError and StackOverflowError. The Exception class has a subclass named RuntimeException. Checked exceptions are descendants of Exception but not of RuntimeException, unchecked exceptions are descendants of RuntimeException.

When a custom program exception is being defined there is the choice to make it checked or unchecked. There are many schools of opinion regarding this choice.

## 14.3 Issues Regarding Java Exception Handling

Checked exceptions in Java are a controversial language feature spurring on much debate [2, 4, 10, 3]. They are usually described as promoting good program structure and robustness but many feel that in reality they have the opposite effect. For these reasons checked exceptions were left out of the C# programming language.

## The Conventional Wisdom Regarding Checked Exceptions

Sun Microsystems, the creator of Java technology, promotes checked exceptions as an essential language feature that leads to more robust and maintainable programs [2]. In the Java Tutorial it is recommended that the programmer deal exclusively with checked exceptions and that runtime exceptions should only be generated by the run-time system (the Java Virtual Machine). The idea being that the compiler enforcement of the *catch or specify* requirement ensures good programming practices, mainly because it forces client code to manage exceptions. The idea is that this should ensure that exceptions are always properly dealt with and that no exception goes unnoticed. Also, throws clauses on method definitions provide source level documentation of the possible exceptions thrown, thus better documentation is enforced.

## How Checked Exceptions Can Be Misused

In reality the *catch or specify* requirement may not necessarily enforce good programming practices. All it really enforces is that either a catch block exists or the enclosing method has a throws clause. It does not enforce that the catch block contains code appropriate to handle the exception or that the block contains any code at all.

The *catch or specify* requirement can become an annoyance to programmers during development as it must be satisfied in order for the program to compile. Many programmers resort to work-arounds such as;

- Wrapping large chunks of code in a generic try/catch that doesnt do anything to handle the exception.
- Often if a chunk of code contains many different types of checked exception then a single generic catch block of type Exception is provided as a catch-all. This is a bad practice because all exceptions, even ones indicating bugs, will be caught. This situation is known as exception swallowing and can have serious consequences in terms of robustness.
- Simply declaring the enclosing method as throws Exception, throws clauses like this then tend to proliferate through the program.

The risk then is that these "code smells" do not get cleaned up before the program goes into production, hurting robustness as opposed to promoting it.

Since the compiler enforces the *catch or specify* requirement, programmers may become dependant on this feature, assuming that if the compiler doesnt report an error then all possible exceptions are being dealt with appropriately. This is a serious risk because it is still possible that APIs being used throw unchecked exceptions, which will then go unnoticed.

Checked exceptions can lead to tangled code with a large amount of try/catch blocks spread through the program. This is especially a problem when using an API such as JDBC (Java Database Connectivity) where almost every method throws a checked exception.

Aspect oriented extensions to Java such as AspectJ can be used to deal with this problem [7]. The exception handling code can be localized within an aspect and then weaved into the appropriate places in the system. This leads to greater modularity and code reuse. AspectJ also has a very interesting feature called exception softening. This feature allows a checked exception to be selectively redefined as an unchecked exception, often with just one line of code. This might be an attractive feature for Java developers who do not like to be forced to use checked exceptions.

### Exception Chaining

Checked exceptions tend not to scale well to large multi-tiered projects [6, 10]. The layered nature of such systems in combination with heavy use of checked exceptions leads to a proliferation of try/catch/wrap/rethrow situations, also know as *exception chaining*.

For example, a web application might have a data access tier for the purposes of abstracting the manipulation of a relational database. Such a tier might use the JDBC API and as such must deal with the fact that almost every method in JDBC throws SQLException, a checked exception [1]. If the intent is to handle SQLException by throwing it up to higher tiers (perhaps for the purpose of displaying an error message to the user) then throwing SQLException directly makes the higher tiers dependant on JDBC. A better idea is to wrap the SQLException in a custom application exception and then throw that. This is an instance of the generally accepted practice of throwing an exception appropriate to the abstraction while hiding implementation details. For example; it makes more sense for the pop() method of a Stack class to throw a StackEmptyException rather than an ArrayIndexOutOfBoundsException.

Exception chaining is the practice of catching and rethrowing wrapped exceptions from tier to tier. All of the code from the point where the exception is thrown to the point where it is eventually handled is affected by the consequences of *atch or specify*, the biggest problem being the amount of code bloat involved. Generally exception chaining leads to a proliferation of throws clauses and catch blocks that just wrap the exception and rethrow it. Also it may be the case that several custom exceptions must be defined.

An arguably better approach is to use unchecked exceptions. This has the advantage that much less of the application code is affected by the fact that lower tiers throw exceptions. The addition of a new exception type only affects the points where the exception is thrown and where it is caught and handled. Thus this approach is more flexible, but stricter documentation practices may be necessary.

## 14.4 Outline of Hypotheses

We begin our empirical analysis by setting out several hypotheses regarding the use of exception handling.

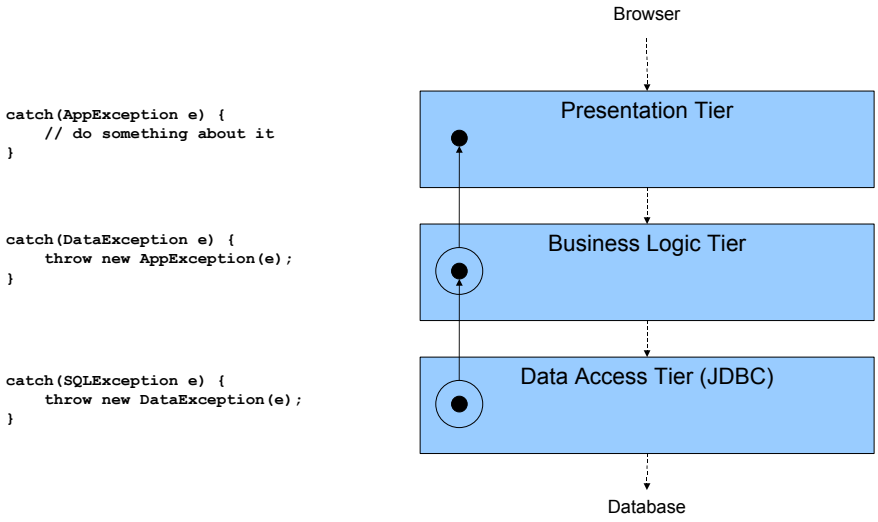



Figure 14.2: A multi-tiered project using exception chaining.

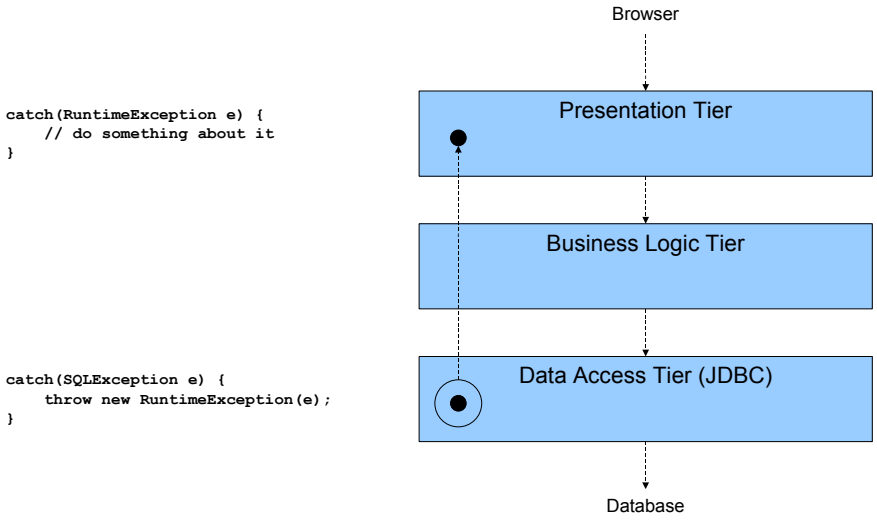



Figure 14.3: A multi-tiered project using runtime exceptions to avoid exception chaining.

## Are the Arguments Made Against Checked Exceptions Well Founded?

Do the following situations occur often in Java code; empty catch blocks, exception swallowing, methods declared as throwing Exception? These are some of the main symptoms of the use of checked exceptions combined with programmer laziness. It should be safe to assume that professional programmers have the discipline (or the requirement) to avoid these situations. It is interesting to see if this actually is the case in practice. We wish to measure these situations in order to see if the arguments against checked exceptions (outlined in the previous section) are well founded or not.

## Does Exception Chaining Occur In Large Systems?

Is exception chaining used in real systems, and if so to what extent? This is essentially a measurement of the amount of code bloat caused by heavy use of checked exceptions in a multi-tiered system.

## Are exceptions only used to indicate errors, signal a breach in the contract, or are they part of normal program flow?

Exceptions are used to indicate undesirable situations that occur during program execution. Some examples of common exceptions used for this purpose are; NullPointerExeption, FileNotFoundException and IOException. One line of argument says that exceptions should only ever be used to indicate bugs or serious errors, so called exceptional situations. It is also possible to use exceptions for other purposes. One very practical use for exceptions is to signal a breach in the contract of an interface. For example; a factorial method might throw IllegalArgumentException if called to compute the factorial of a negative number. This is essentially the same as the concept of precondition assertion checking found in a language such as Eiffel. It is also possible to use exceptions in order to control the flow of the application. When an exception is thrown control jumps to the appropriate catch block, therefore it is possible to use exceptions solely for the purpose of avoiding the use of an if/else statement. It would be easy to argue that this is an abuse of the exception handling system. Again, the real question is if exceptions are actually used for this purpose in real world projects.

## Are exception hierarchies used effectively?

In Java it is possible to create a class hierarchy of custom exceptions, having more general exceptions towards the top of the hierarchy and more specific ones towards the bottom. It can be very useful and powerful to design custom application exceptions in this way.

## Are multiple catch blocks used to catch subclasses of exceptions?

This is in line with the previous hypothesis regarding the use of exception hierarchies. If an exception hierarchy is used then there is the choice of using multiple catch blocks to catch specific types of exceptions.

## Are many custom exceptions defined or are the ones given in the core library used directly?

Do developers create a large number of custom exception classes for use in their applications or do they just use the exceptions that come with the Java core library. This often means throwing Exception or RuntimeException as opposed to custom exceptions.

## Do custom exceptions extend Exception or RuntimeException directly?

Sometimes developers wish to create an exception handling block that will catch and handle many types of exceptions in a general way. Java provides a facility for this that can be a convenient way to avoid duplicated code. However there is a risky practice of having the exception handler specified as type Exception, the parent of all exceptions. The problem is that this will catch all possible exceptions, even unexpected ones that may be indicative of serious errors or program bugs, this is know as exception swallowing. Bugs may go unnoticed, thus severely hurting program robustness.

There is a relatively simple pattern for avoiding exception swallowing in an application. The idea is to create a generic custom exception that will be the parent of all other custom exceptions defined in the application. For example this exception might be named ApplicationException.

```
public class ApplicationException extends Exception { ... }
public class BadPasswordException extends ApplicationException { ... }
```

Now catch-all exception handlers can be specified to catch ApplicationException instead of Exception. This way the catch block will catch any custom exception but exceptions that are part of the core library can still get through. Exceptions indicating serious bugs will not get swallowed. This of course means that the developers should avoid the throwing the core library exceptions directly in the program. So the question is, do developers actually use this simple practice of avoiding exception swallowing?

## Are exceptions used to reestablish the class invariant and to ensure liveness or are they used just for diagnosis?

Generally an exception handler should ensure that the class invariant is preserved or reestablished, essentially making sure that the object is still in a usable state. This avoids having

objects floating around that are in a bad state thus making the system unstable. If an object is in a state that violates its class invariant then the behavior of the object may be unpredictable. Exception handlers should also attempt to free resources that are no longer needed. For example, closing database connections or open files is a good idea. This helps to ensure the liveness of the system. The question is if exception handlers actually do this, or if exceptions are just passed around to indicate the occurrence of an exceptional event.

### Are exceptions handled locally or are they caught in the "main loop"?

Exceptions may sometimes be handled at or near the point where they are thrown. However, sometimes this approach may become impractical. Having several exception handlers that do similar things can lead to code duplication, therefore it may be a better idea to move the exception handler up the call stack. Also sometimes the only way to handle an exception is to display an error message to the user. For these reasons it may be practical to have a catch-all handler in the main loop of the program, handling many types of exception in a general way.

### Are exception messages used and do they contain meaningful information?

When an exception is instantiated and thrown there is the option to provide a message, perhaps giving a clue as to why the exception was thrown. For example, a factorial method might throw an exception on a negative value as input. The exception message could indicate what the offending value was. This can be a great help in tracking down bugs.

```
public static int factorial(int n) {
    if (n < 0)
        throw new IllegalArgumentException(
                  "factorial called with negative value: " + n);
    ...
}
```

## 14.5 Analysis

### Choice of open source projects.

The first step in analysis is to choose the open source projects which we will analyze. Generally we want projects that are popular and in widespread use. For this reason we chose the top twenty most active Java projects on SourceForge.net as of April 2005. These projects have a wide variety in terms of:

- Project size: From a few thousand lines of code to millions of lines.

- Type of project: Standalone applications, GUI applications, middleware, frameworks, application servers and more.

- Professional and non-professional projects: Some open source projects are created by programmers that donate their time for free and work from home when they can. Other projects are created by professional programmers that work for money, for example the JBoss application server. We wish to detect differences in exception handling use between these two categories of project.

## Automatic Analysis

Since we are dealing with open source projects that potentially contain thousands of lines of code, any attempt at manual analysis of the code would be much too time consuming to be practical. Therefore we use automated analysis tools, created by us, that automatically detect certain usage patterns of exceptions.

There are two approaches to automatic analysis of source code that we use. The first technique is to analyze the source text directly. This involves writing the analyzer as a script in the text processing language Perl. Usage patterns of exception handling will be detected using regular expressions. We have had some success using this technique but some problems did arise. When analyzing textual code directly the presence of white space and comments must be taken into consideration. Also, arbitrary levels of nesting, for example catch blocks within catch blocks, are difficult to detect reliably. Therefore, detecting even simple usage patterns requires rather large regular expressions, which can become difficult to write and debug.

A better approach to automatic analysis would be to parse the code to obtain an abstract syntax tree, then write some recursive procedures that analyze the structure of this tree. This is a more reliable approach because the parser removes white space and comments. Even so, there are still difficulties when complex rules define our usage patterns, such as variable scope and typing rules. Future work would involve writing a second analysis tool in Java using the jParse API which parses Java source and returns an abstract syntax tree.

## Data and Limitations

The amount of data on exception handling use that can be gathered in an automatic way is limited. For example, there is no reliable automatic way to detect that an exception handler reestablishes a class invariant. Furthermore, several of our hypotheses are difficult to detect automatically.

Therefore we concentrate only on the hypotheses that do lend themselves to automatic detection. Specifically the usage patterns for which metrics are gathered include:

1. Total number of catch blocks in the program. Useful for comparison to the total number of empty catch blocks.

2. Number of catch blocks that throw an exception. This indicates catch/rethrow situations.

3. Number of methods that have a throws clause. Gives an indication of the effect of checked exceptions on the interfaces in the program.

4. Number of catch blocks of type Exception. This is also potentially a symptom of checked exceptions.

5. Number of empty catch blocks. Another symptom of checked exceptions and/or programmer laziness.

6. Number of Java source files in the project. This gives an indication of the size of the project.

Some hypotheses are measurable in a semi-automatic way. For example, it is not possible to detect that exceptions are given meaningful messages when created. How would an automated tool be able to detect what should be considered meaningful and what should not. However, an automated tool can report on locations with program files in which exceptions are instantiated using a constructor method that takes a String as an argument. Researchers can then view these instances and determine if they actually do represent meaningful messages. For the purposes of this study we simply concentrate on metrics that can be gathered in a fully automatic way.

## List of open source projects

We have analyzed twenty open source projects based on the criteria previously listed. These are the twenty most active projects on Sourceforge as of April 2005, listed here in alphabetical order.

1. Art of Illusion
   Full featured 3D modeling, rendering, and animation studio.
   http://prdownloads.sourceforge.net/aoi/aoisrc20.zip?download

2. Azureus
   BitTorrent client.
   http://prdownloads.sourceforge.net/azureus/Azureus2.3.0.0_source.zip?download

3. FindBugs
   A static analysis tool for Java programs.
   http://prdownloads.sourceforge.net/findbugs/findbugs-0.8.8-source.zip?download

4. HSQL Database Engine
   HSQLDB is a small, lightweight relational database engine.
   http://prdownloads.sourceforge.net/hsqldb/hsqldb_1_8_0_RC8.zip

5. HTML Unit
   Unit testing framework for use when testing html based web sites.
   http://prdownloads.sourceforge.net/htmlunit/htmlunit-src-1.5.zip?download

6. iReport-Designer for JasperReports
   Visual reporting tool based on JasperReports.
   http://prdownloads.sourceforge.net/ireport/iReport-0.4.1-src.zip?download

7. iText, a JAVA-PDF library
   Generates documents in Portable Document Format (PDF) or HTML.
   http://prdownloads.sourceforge.net/itext/itext-src-1.3.0.tar.gz?download

8. JabRef
   JabRef is a graphical application for managing bibliographical databases.
   http://prdownloads.sourceforge.net/jabref/JabRef-1.7.1.src.tar.bz2?download

9. JasperReports
   Uses XML report templates to generate ready to print documents using data from customizable data sources.
   http://prdownloads.sourceforge.net/jasperreports/jasperreports-0.6.6-project.zip?download

10. JavaHMO TiVo HMO Serve
    Media server for the Home Media Option (HMO) from TiVo.
    http://prdownloads.sourceforge.net/javahmo/javaHMO2.4.src.zip?download

11. JBidwatcher
    An auction site (eBay, Yahoo, etc.) bidding, sniping, and tracking tool.
    http://prdownloads.sourceforge.net/jbidwatcher/jbidwatcher-0.9.7pre2.tar.gz?download

12. JBoss.org
    Standards-compliant, J2EE based application server.
    http://prdownloads.sourceforge.net/jboss/jboss-4.0.2RC1-src.tar.bz2?download

13. jBpm.org
    WorkFlow Management System.
    http://prdownloads.sourceforge.net/jbpm/jbpm-1.0.1-src.zip?download

14. Mantaray
    A fully distributed peer-to-peer serverless communication and messaging solution.
    http://prdownloads.sourceforge.net/mantaray/mantaray_1.7_src.tar.gz?download

15. MegaMek
    Turn-based sci-fi boardgame for two or more players.
    http://prdownloads.sourceforge.net/megamek/MegaMek-v0.29-stable-10.zip

16. OpenReports
    Complete web based reporting solution.
    http://prdownloads.sourceforge.net/oreports/openreports-0.9.0.zip?download

17. PMD
    Java source code analyzer.
    http://prdownloads.sourceforge.net/pmd/pmd-src-3.0.zip?download

18. RSSOwl
    RSS/RDF/Atom Newsreader
    http://prdownloads.sourceforge.net/rssowl/rssowl_1_1_src.tar.gz?download

19. Tyrant
    Graphical fantasy adventure game.
    http://prdownloads.sourceforge.net/tyrant/tyrant-src-0.333.zip?download

20. XPlanner
    Web-based project planning and tracking tool for agile development teams.
    http://prdownloads.sourceforge.net/xplanner/xplanner_0.6.2.tar.gz?download

## 14.6 Data And Conclusions

| Project | Analysis Criteria | | | | | |
|---|---|---|---|---|---|---|
| | Catch Blocks | Catch/ Rethrow | Throws Clauses | Catch *Exception* | Empty Catch Blocks | Files |
| Art of Illusion | 239 | 29 | 8 | 95 | 49 | 395 |
| Azureus | 1710 | 129 | 19 | 370 | 55 | 1913 |
| FindBugs | 450 | 91 | 71 | 44 | 22 | 687 |
| HSQL Database Engine | 777 | 189 | 134 | 331 | 193 | 286 |
| HTML Unit | 165 | 69 | 662 | 0 | 0 | 274 |
| iReport-Designer | 311 | 8 | 6 | 202 | 57 | 245 |
| iText | 495 | 231 | 3 | 188 | 33 | 399 |
| JabRef | 390 | 44 | 22 | 67 | 70 | 295 |
| JasperReports | 317 | 125 | 1 | 75 | 0 | 363 |
| JavaHMO | 438 | 20 | 19 | 248 | 46 | 166 |
| JBidwatcher | 175 | 13 | 4 | 35 | 9 | 118 |
| JBoss | 9517 | 3129 | 5951 | 3184 | 947 | 6216 |
| jBpm | 279 | 135 | 100 | 39 | 1 | 381 |
| Mantaray | 615 | 110 | 38 | 69 | 23 | 384 |
| MegaMek | 167 | 43 | 11 | 39 | 8 | 181 |
| OpenReports | 111 | 43 | 0 | 67 | 0 | 121 |
| PMD | 506 | 34 | 18 | 45 | 29 | 573 |
| RSSOwl | 160 | 13 | 0 | 3 | 0 | 180 |
| Tyrant | 62 | 3 | 75 | 32 | 1 | 178 |
| XPlanner | 327 | 163 | 502 | 133 | 4 | 382 |

### Conclusions and Future work

From the large amount of catch blocks in each project it is clear that exception handling is an essential part of Java applications. It is interesting to see that many of the projects have a large number of empty catch blocks while others have a very small number. Also there is a rather large number of catch/rethrow situations indicating that exception wrapping is in widespread use. A few of the projects have little or no throws clauses on method definitions, this is very interesting considering the *catch or specify* requirement. This suggests that these projects may be using unchecked exceptions extensively. We can also see from the large amounts of empty catch blocks and throws clauses of type Exception that unchecked exceptions may be having a detrimental impact on how applications are written.

In future work it would be of great benefit to gather many more categories of metrics than in this preliminary study. We have focused mainly on the hypotheses that are related to the checked exception debate, in future work it would be better to go beyond this and to study more hypotheses in detail.

Automated analysis of exception use is very limited. A better approach might be to have the programmers cooperate in the study by documenting exception use as the application is being developed. They could record instances of certain patterns as they are being written or when a component has been finalized. This might even have an impact on their awareness of exception use and lead to a more robust final product.

The work we have done so far has been a static analysis of exception use, but we believe it is possible to perform dynamic analysis as well. Aspect oriented programming would be an effective tool in this regard. It is possible to create pointcuts that capture creation and handling of exceptions, and then partner these pointcuts with advice that records the data. It might even be possible to detect such things as the reestablishment of a class invariant, by executing assertions after the completion of a catch block within a class.

This has been a [9] preliminary study on the use of exception handling. Many issues regarding different options have been explored. Hopefully we have shed some light on the real world use of exception handling.

## 14.7 Analysis Tool Source Code

```
use File::Find;

sub process_file1{
    if ( ! ($_ =~ /java/) ) {
print "$_ is not a java file.";
return;
    }
    if ( $_ =~ /Exception/ ){
print "EXCEPTION FILE\n";
    }

    open ( FILE, "< $_") or die "could not open";
    read ( FILE, $src, 1000000 );
    pattern1();
    pattern2();
    pattern3();
    pattern4();
    pattern5();
}

sub pattern1{
#EMPTY CATCH BLOCK
    @finds1 = $src =~ /catch[ \t\n]*\([^\)]+\)[ \n\t]*{[ \t\n]*}/g;
    foreach $f1 (@finds1){
print "\nFOUND AN EMPTY BLOCK IN: $_\n";
        print "$f1\n";
```

```
print "\n------\n";
    }
}

sub pattern2{
#CATCHING EXCEPTION
    @finds2 = $src =~ /catch[ \n\t]*\([ \n\t]*Exception[ \n\t]*[^\)]+\)/g;
    foreach $f2 (@finds2){
print "\nFOUND CATCH OF EXCEPTIONS IN: $_\n";
print "\n$f2";
print "\n------\n";
    }
}

sub pattern3{
#EXCETOPTION THROWING METHODS
    @finds3 = $src =~ /\bthrows[ \n\t]*Exception[ \n\t]*{/g;
    foreach $f3 (@finds3){
print "\nFOUND A METHOD THROWING Exception IN: $_";
print "\n$f3";
    }
}

sub pattern4{
#CATCH/RETHROW
    @finds4 = $src =~ /\bcatch([^}])+throw[^s]\b/g;
    foreach $f4 (@finds4){
print "FOUND CATCH/RETHROW IN: $_\n";
print "$f4\n";
    }
}

sub pattern5{
#CATCHES
    @finds5 = $src =~ /catch/g;
    foreach $f5 (@finds5){
print "FOUND A CATCH: $_\n";
print "$f5\n";
    }
}
find ( \&process_file1, '.');
```

# Bibliography

[1] Java 2 platform, standard edition, v 1.4.2, api specification: Package java.sql. http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html, 2003.

[2] Essential java classes: The catch or specify requirement. http://java.sun.com/docs/books/tutorial/essential/exceptions/catchOrDeclare.html, 2005.

[3] Bruce Eckel. Does java need checked exceptions? http://www.mindview.net/Etc/Discussions/CheckedExceptions, 2004.

[4] Brian Goetz. Ibm developerworks: Java theory and practice, the exceptions debate. http://www-106.ibm.com/developerworks/java/library/j-jtp05254.html, 2004.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Sun Microsystems, Inc., 2000.

[6] Rod Johnson. *Expert One-on-one J2EE Design and Development*. Wiley Publishing Inc., 2003.

[7] Ramnivas Laddad. *AspectJ in Action*. Manning Publications, 2003.

[8] Xavier Leroy. The objective caml system, release 3.08, documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/index.html, 2004.

[9] Allen Tucker and Robert Noonan. *Programming Languages: Principles and Paradigms*. McGraw Hill, 2002.

[10] Bill Venners with Bruce Eckel. The trouble with checked exceptions: A conversation with anders hejlsberg, part 2. http://www.artima.com/intv/handcuffsP.html, 2003.

[11] Dave Thomas with Chad Fowler and Andy Hunt. *Programming Ruby: The Pragmatic Programmer's Guide*. Addison Wesley Longman, Inc., 2001.

# Chapter 15

# Ed Sykes: Licensing of the Computing Professional

Licensing of the Computing Professional is becoming increasingly debated in the arena of Information Technology (IT). There are certain benefits of certification and licensing such as prestige, professional networking, job enhancement opportunities, and many others. An overview of the advantages, disadvantages of popular licensing organizations and certification programs is presented. The goal of this paper is to: (i) promote awareness of the various issues associated with each type of credential; and (ii) help the reader decide if these computing credentials are desirable for him/herself.

Despite the apparent advantages of certification and licensing there are surprisingly few computing professionals that pursue certification and even less for licensing. This paper attempts to address this question and provides rationale why this is occurring. It also presents a hypothetical projection of the future of certification and licensing for the Computing Professional.

## 15.1 Introduction

"Licensing ... Certification ... " – terms that will cross the minds of every Computing Professional many times during his/her career as it becomes increasingly popular in the IT sector. There appears to be some distinct benefits of certification and licensing such as prestige, professional networking, job enhancement opportunities, and many others. Furthermore, judging by the various advertisements and propaganda of some organizations, the vendors promote certification is absolutely mandatory in order for one to succeed in the IT marketplace. However, how is a computer specialist supposed to sift through all the hype and media to find a clear path through the IT licensing and certification maze?

This paper presents an objective overview of the advantages and disadvantages of popular computing licenses and certifications and includes eligibility requirements, fee structures, and membership restrictions for the various types of credential-issuing organizations. The goal of this first section of the paper is to present information that will aid the reader in determining

if these computing credentials are desirable for him/herself and aims to promote awareness of the various issues associated with each type of credential.

The second section of this paper examines the fact that although there are advantages to licensing and certification, there are relatively few computing professionals that pursue these credentials and even fewer are pursuing licenses.

The third section of this paper addresses this phenomenon and provides a rationale as to why this is occurring. A hypothetical projection of the future of certification and licensing for the Computing Professional is presented.

## 15.2 Licenses and Credentials

There are many advantages and disadvantages of computing licenses and certifications. The goal of this section is to present information that will aid the reader in determining if these computing credentials are desirable for him/herself and aims to promote awareness of the various issues associated with each type of credential.

The scope of the various licenses and certifications is primarily focused on those available to Canadians, however, for comparison reasons a select few licensing organizations from other countries are included.

### Licensing of the Computing Professional

In Ontario, there are only two relevant IT licensing associations that are supported by federal and provincial law and internationally recognized by IT professionals in other countries. The names of the two organizations are the *"Association of Professional Engineers"* (e.g., the Professional Engineers of Ontario (PEO), and the *"Canadian Information Processing Society"* (CIPS). Both have a similar purpose and perform the same kind of functions. The following list provides some specifics with regard to licensing associations.

- A regulating body for the specific practice of engineering, computing, or similar service;
- It fulfills the same role for Computing Professionals as the College of Physicians and Surgeons for doctors or the Law Society of Upper Canada for lawyers;
- Under provincial statutory law, the association is responsible for the licensing and discipline of its members and companies providing IT services;
- The association protects the public by ensuring all members are qualified computing professionals; and
- Individuals may not use the specific title unless they are licensed by the organization. (e.g., one cannot call themselves a professional engineer (P.Eng.), or use any similar title that may lead to the belief that they are qualified to practice professional engineering).

### Rights and Obligations of Earning and Holding a Computing License

One of the main benefits, as claimed by the licensing organizations, is that of maintaining and enhancing public perception [1]. In other words, those who have earned the license are thought to be more considerate and cognisant of public concerns than those who are not licensed. Each association feels that the public's view is critical and places a great deal of emphasis on this, both during the licensing process and for the duration of the individual's membership. In this way, the associations claim that this ensures that the public's safety and well-being is upheld to the highest degree.

The following list provides a summary of the main benefits of licensing for the computing professional from the viewpoint of the public. That is, the public perceives the licensed individual is:

- ***capable*** of providing services within the practice of the professional body (e.g., professional engineering, software engineering, systems design, etc.);
- that they are ***accountable*** to the regulatory body for the success of the work responsible for the *safety* of their work (for the public);
- ***will establish contacts*** and network with top-performing professionals to ensure their knowledge and skills are current;
- eligible to use ***legally protected titles***: e.g., "P.Eng., i.e., Professional Engineer", "I.S.P. – Information Systems Professional".

### Licensing Background

In order to appreciate the situation of licensing for the computing professional in today's environment it is necessary to review the history of the foundation of licensing for the computing industry, engineering.

The first law related to professional engineering in Ontario was created in 1922 and allowed for the creation of a voluntary association to oversee registration of engineers. The Act of 1922 was "open", meaning that membership in the association was not mandatory for practising engineers [1]. For example, in Ontario, regulation of engineering practice dates to 1937, when the Professional Engineers Act was amended and the engineering profession was "closed" to non-qualified individuals. In other words, licensure was made mandatory for anyone practising professional engineering. The provincial government determined that it would be in the public interest to restrict the practice of engineering to those who were qualified, and the right to practise was "closed" to non-engineers as a result of the failures of bridges and buildings, which had been designed by unskilled individuals [1].

Today, the *Professional Engineers Act* is very much intact and has been amended several times over the year, with the most recent occurring in 1999. This latest amendment was a Regulation made to provide additional details and guidance for implementation of the Act. The regulation prescribes the process to be followed when electing professional engineers to

the Professional Engineer's Council. With respect to professional practice, the Regulation prescribes a *Code of Ethics*, defines negligence and professional misconduct, addresses the requirement for professional engineers to report unsafe situations and unethical practices, and states that all professional engineers shall have a seal and describes its use [1].

In summary, a *Professional Engineer* is a legally protected title, only registered *Professional Engineers* are allowed to use the title and carry out the work of *Engineering.* These licensed individuals have the authority to "sign off" or "stamp" a design or a structure, thus taking ***legal responsibility*** for it. From the perspective of the Professional Engineering Association, the government has delegated authority to the federally situated Canadian Council of Professional Engineers which in turn has delegated responsibility to the provincial level. In Ontario, the PEO mandate is to *protect the public* interest, *safety* and *well-being* through licensing and regulation of the practice of professional engineering in this province. Each association is based on a provincial (or state in the USA) jurisdiction. For example, in Canada there are governing bodies in each of the 10 provinces and the two territories.

**Professional Engineer–Licensing Requirements**

Although times have changed, the requirements to become licensed Professional Engineer have remained remarkably close to their original mandate set out in 1922. The following list presents the requirements an individual must attain to be licensed with the "P.Eng." designation.

For applicants to earn the PEng. designation, the most common route is outlined below. The applicants must:

- be a citizen or permanent resident of Canada;
- have successfully completed a B.Eng from an accredited Canadian institution;
- have 4 years or more of engineering work experience;
- pass the Professional Practice Exam on engineering law and ethics; and
- pay the required licensing fee(s) (i.e., currently between $160 to $370 per year, depending on the province)

While this may be the most common and desirable route from the perspective of the PEO, there are, of course, exceptions. For instance, the association recognizes individuals who may have a degree but it is not from an accredited Canadian university, or the individual may have a B.Sc. degree instead of engineering, etc. The association has paths in place for those seeking certification but do not fit in the typical licensing path.

Regardless of the path to become licensed, the individual must satisfy the requirements above (or equivalent), agree to and sign a "Code of Ethics". This "Code of Ethics" is a basic guide to professional conduct and imposes duties on the practicing Professional engineer, with respect to:

- society;
- employers;
- clients;
- colleagues, including employees and subordinates;
- the engineering profession; and
- himself/herself.

The Professional Engineer's Code of Ethics states, "...it is the duty of a practitioner to the public, to the practitioner's employer, to the practitioner's clients, to other licensed engineers of the practitioner's profession, and to the practitioner to act at all times with,

- *fairness* and *loyalty* to the practitioner's associates, employers, clients, subordinates and employees,
- fidelity to *public needs*,
- *devotion* to high ideals of personal *honour* and professional *integrity*,
- *knowledge of developments* in the area of professional engineering relevant to any services that are undertaken, and
- *competence* in the performance of any professional engineering services that are undertake." [1]

Legislatively the Code of Ethics is legally binding on Professional Engineers [2]. There are currently 67,000 licensed Professional Engineers in Ontario and 160,000 across Canada [3].

The following section provides two sample questions from the engineering law and ethics exam. A P.Eng. candidate must pass the exam as part of the requirements of being licensed. The questions are selected from the ethics section of the exam.

**Professional Engineer–Sample Exam Questions**

> QUESTION 1
>
> Shortly after signing a consulting contract with the XYZ Company to oversee the construction of a new manufacturing plant, you receive a letter from Mr. Smith, P.Eng. In his letter Mr. Smith points out that he was contracted by the XYZ Company for the work that you are now doing. He goes on to state that, despite having been notified by the XYZ Company that his services are no longer required, he feels that he has been terminated improperly and has taken the position that until the matter is settled he is still engaged on this project.

Do you have any ethical obligation to Mr. Smith? Does the receipt of this letter and the knowledge of Mr. Smith's position with respect to the XYZ Company have any effect on your position vis a vis the XYZ Company?

QUESTION 2

You are a Professional Engineer in the mechanical department of a large general contracting firm. You have recently fallen heir to shares of stock in a company that manufactures air handling equipment. You hold this company's products in very high regard and often specify them on projects that your company is building. Now that you are a shareholder of this company can you ethically continue to specify its products? If you feel that you can, is there any action that you should take to do so ethically?

There are distinct advantages of becoming licensed. The following presents the main benefits.

**Professional Engineer – Benefits**

First and foremost, the main benefit is the title that may be appended to the licensed person's name. Depending on the location, there are variations to the title but they represent the same concept. For example, in Canada, it is primarily "*P.Eng.*", except in Quebec where it is "*Ing.*". In the U.S.A., it is "*P.E.* or *PE*". In the United Kingdom and Ireland it is "*CEng"* (representing *"Chartered Engineer"*). The second most evident benefit is a reflection on the responsibility borne by being licensed – that is, ownership of a stamp and seal, which means the bearer assumes legal responsibility for the design or structure for which they are used. Figure 1 presents a sample of a Professional Engineer's stamp.

In some disciplines of engineering licensure is mandatory for anyone practising professional engineering. This however, does not apply to the area of software engineering [4]. There are a number of other benefits such as personal networking with similar minded professionals, conferences, perceived increased job opportunities, insurance discounts, etc.

Other benefits of licensure is that the Professional Engineer who is in good standing with the association may keep the license for life. In other words, there is no expiration date on the license.

**Professional Engineers – Challenges in the Software Discipline**

Despite the evident advantages of becoming licensed as a Professional Engineer in a specific discipline like Software Engineering, there are some distinct challenges. The majority of the issues arise from the specific clause in the Act entitled 'enforcement'.

In Canada, the Professional Engineers Act makes it very clear that in order for an individual to practise as an engineer, one must be licensed. Enforcement is the legal authority the governing body has to prosecute those who practice engineering who do not have a valid license [1]. Enforcement also relates to granting the privilege of the use of the terms "*software engineer*" and "*software engineering*" to licensed individuals.

The problem in today's IT sector is that the 'enforcement' clause has become considerably weakened in the specific discipline of computing. The Professional Engineers of Ontario (PEO) feels that this is a problem because it *misleads the public*. The PEO feel that the


*Figure 1.* Sample Stamp of a Professional Engineer.

public clearly perceives computing a discipline of engineering and those licensed are capable of providing services and that they are accountable to their regulatory body. However, in today's ever changing IT market, many people who develop computer software refer to themselves as "software engineers" and their work as "software engineering", even though they have *never studied engineering* and are *not licensed* or regulated in any way. This perspective is supported by recent publications from the IEEE on the topic of software practitioners. They state "*[becoming licensed] depends. If the work you do falls within the definition of professional engineering, you must be licensed by PEO unless someone else who is a licensed engineer takes responsibility for your work.*". The challenge arises from the definition of what precisely is the discipline of "software engineering".

### Professional Engineers–Challenges in the Software Discipline: A Case Study

In 1997, PEO advised Microsoft Canada that use of its terms "Microsoft Certified Systems Engineer" and "Microsoft Certified Professional Systems Engineer" violate the Act.

On July 25, 2002, Microsoft Canada announced that they will continue to use the term "engineer" in their certifications.

Struggles and losses like these significantly hurt the mandate of the Professional Engineering Act in terms crippling the power the PEO and other governing bodies have in terms of enforcement on its membership. It further erodes the incentive for new engineers to become certified.

## Licensing of the Computing Professional–for non-Engineers

In Canada, there is an equivalent licensed title to the Professional Engineer title that can be earned for those computer practitioners who are not formally trained as engineers. The "Canadian Information Processing Society" (CIPS) is the second main association in Canada that governs the licensure of individuals in the computing industry. CIPS is fully supported

by Canadian law and the designation it awards is internationally recognized by IT professionals in many countries around the world [5].

### CIPS Background

The Canadian Information Processing Society is organized into four different bodies. Each is responsible for a different aspect of Member services: CIPS National; CIPS Sections; CIPS Provincial Bodies; and CIPS Special Interest Groups [5].

**CIPS National** is the governing and policy-making body of the Society. It is responsible for the administration of services to all Members, overall governance of CIPS Sections and Provincial Bodies, federal and international advocacy, formal agreements with external organizations, and media awareness.

**CIPS Sections** are found in many cities across Canada. They provide local programs, newsletters, social events and benefits. Section activities are often the main point of interest and participation for Members. This is equivalent to *Chapters* in the Professional Engineering society.

**CIPS Provincial Bodies** oversee the professional regulations within the provincial. The Provincial Bodies within CIPS are responsible for the pursuit of the legislation and regulation of the *"Information Systems Professional"* (I.S.P.) designation. Once regulation has occurred through legislation, the Provincial Bodies are also responsible for the administration of the I.S.P. within that province.

**CIPS Special Interest Groups** provide a forum for the various areas of specialization within the information technology field. Based at the National level, these groups are organizations who have been contracted with CIPS to provide Member processing services and support.

Currently CIPS has a membership of 8,000 individuals. It is the only Canadian organization that has a government recognized designation for IT professionals by awarding the *"Information Systems Professional"* designation.

### Licensing Requirements

In order for an individual to earn the *"Information Systems Professional"* license it requires the candidate to:

- have a 4 year B.Sc. plus 2 years work experience (or equivalent education and experience); and
- agree to a Code of Ethics.

### Benefits

The main benefit in earning the *"Information Systems Professional"* designation is that it shows that the individual takes responsibility for ensuring IT projects and products have

predictable levels of quality, reliability and support. Since the I.S.P. is essentially equivalent to the "P.Eng" designation, it carries a significant degree of credibility. Additionally, the I.S.P. is an internationally recognized designation, and is registered under the Canada Trademarks Act.

For the individual, the I.S.P. designation demonstrates:

- Credibility;
- Professional image; and
- Career development.

For the profession, the I.S.P. establishes and maintains the highest standards of:

- Practice
- Ethics
- Public protection.

The following list is the promoted benefits of the I.S.P. designation [6].

1. Rigorous designation criteria ensures I.S.P.-designated professionals will be superior contributors to an organization's bottom-line.
2. Customers are assured of high-quality information systems being used to develop and support products and services.
3. The perception of an organization is enhanced internationally through broad recognition of the I.S.P.
4. Three hundred hours of professional training completed every three years by each I.S.P., ensures an organization is applying the best of current practices.
5. Access to the educational resources and networking opportunities provided by CIPS, keeps contractors and staff current and informed.
6. More than 1,700-plus I.S.P. holders across Canada
7. National nature of the designation provides consistency of standards, easy transferability, and cost efficiencies.
8. As a government-registered professional designation for information systems professionals, the I.S.P. provides a unique competitive advantage for organizations when bidding internationally.
9. Effective self regulation of the profession provides confidence to customers and the general public while avoiding the burden of regulation.
10. Supporting staff to receive and maintain the I.S.P., reflects an organization's appreciation that its staff are professionals.

One main distinction between the I.S.P. and the P.Eng. license is the issue around mandatory upgrading of skills and knowledge. As may be seen from the list above, in order for an I.S.P member to continue their license, they must complete at least 300 hours of professional training every three years [6]. The CIPS feels this ensures that the best of current practices are being applied in IT projects. On the other hand, there is no such criterion that the PEO stipulates on it's P.Eng. members [3]. Recall, a Professional Engineer has a valid license for life providing s/he pays the annual membership fee to the local Professional Engineering association.

#### Disadvantages

While it may be considered an advantage that I.S.P. members need to complete 300 hours of professional training every three years, from the perspective of the member it may looked upon as a disadvantage.

Initially, I.S.P. holders are given 3 years to acquire their re-certification requirements. Thereafter, re-certification is required every year. The membership fee is comparable to the P.Eng. license, approximately $200 per year.

The main disadvantage of this designation is its size. Compared to the number of licensed Professional Engineers (160,000 across Canada), the number holding the I.S.P. designation dwarfs in relative terms – currently, there are 1,700 I.S.P. members in all of Canada.

## 15.3 Certification of the Computing Professional

While there are surprising few licensing associations available to computing specialists, there is an abundance of different certifications. There are over 400 different types of certifications available for the IT practitioner. Three different categories are presented below which represent the organizations that issue computer certifications.

1. professional associations or similarly managed organizations (non-profit), (e.g., IEEE Computer Society);

2. industry or product-related certifications, e.g., Oracle, Novell, Cisco; and

3. certifications granted by government agencies that train individuals for specific jobs and validate the student's competency (e.g., Sheridan's Database Administration Certificate).

### Benefits of Earning a Certification

From the perspective of the issuing organizations, computer certifications have clearly defined benefits. Whereas the emphasis of licenses is on professionalism and ethical conduct, certifications emphasize the following benefits.

The additional knowledge and skills that are developed by the individual allow him/herself to move into a new area or perform your current job more effectively [7, 8]. Many certification programs offer training that provide exposure to the latest software or equipment that might not be otherwise available [8]. A claim that certification organizations state is that through the certification process the computing professional increases his/her level of expertise. Another benefit of certification is *increased customer confidence* based on the evidence of qualifications [7]. This has the added benefit for the individual who may be interested in changing jobs or a different area of the IT sector. From an outsider, computing practitioners with certification may be more suitable for a specific task or project put out for bids. The last main benefit associated with certification is being in a like-minded society. The certified member can easily establish contacts and network with top-performing professionals. Other popular perks to certification are of a propaganda-oriented nature. Such perks may include complimentary jackets, discounts on conferences and software, and other logo-laden paraphernalia [7].

## Common Confusions Regarding Certification vs. Licensure

Since there can be ambiguity among the terms certification, licensure and accreditation, this section provides a clear explanation of these terms in the context of the computing practitioner.

**Certification:** Certification is an *occupational designation* issued by an organization that provides confirmation of an individual's qualifications in a specific profession or occupational specialty. All certifications are voluntary. In other words, in order to practice as an Oracle database administrator, for example, one does not need to be certified in order to do the job. However, in some situations it may be advisable and advantageous to be certified.

**Accreditation**: This is a designation that an organization receives which demonstrates that standards and abilities are in place for public safety, welfare and confidence. *Universities and Colleges provide accredited* programs. For example, McMaster University provides an various baccalaureate programs that are accredited by the Ministry of Education and Training of Ontario. The Professional Engineers of Ontario recognize this accreditation and can therefore streamline applicants for the P.Eng. licensing process.

**Licensure**: Licensure is the *most restrictive form of professional and occupational regulation*. Under licensure laws, it is illegal for a person to practice a profession without first meeting the standards.

A common way to remember the differences between these terms is to consider the following. "People are certified.", "Institutions are accredited.", and "Medical Doctors are licensed."

## Why Are There So Many Certifications?

There is no denying that the computer industry is a highly competitive one and is growing increasingly so every year. Part of the reason why there are so many IT certifications is due to

the different emphasis of knowledge and skills by institutions versus industries. Generally, industry wants: (1) responsible independent problem solvers; and (2) *trained workers for specific business tasks*.

On the other hand, institutions (post-secondary schools, College and Universities) focus primarily on breadth of knowledge and theory. The emphasis of material covered in ministry approved programs (diplomas, degrees) typically does not focus on specific technologies. As a result, there is a distinct *gap* between industry and the public institutions in this country. As a result, there are many certification programs offered by industry-specific training organizations that attempt to fill this gap and provide training on extremely specific technologies in the IT sector. The net result is a proliferation of hundreds of different certifications.

Despite the different focus between industry and educational institutions, recently there has been an increasing in the number of institutions becoming involved with industry certification. For example, at the Sheridan Institute of Technology and Advanced Learning there are approximately seven different computing related certifications all based on specific technologies commonly used in the IT industry. The Ministry of Education, Ministry of Training, College and Universities carefully watches over these programs to ensure that there are significant differences between the vendor-specific programs and the publicly-funding institutional-based programs. This ensures that the overlap of knowledge and skills is not too great and there is no conflict of interest between the institution and industry training programs. Some of the large industry vendors are particularly aggressive in working with institutions. For example, Oracle has a special partnership program entitled the Oracle Workforce Development program, which offers the institution the following; (1) all of the various Oracle software; (2) free 24/7 technical support; (3) 50% off vouchers for instructors to take Oracle courses; (4) assistance in setting up the labs; and (5) the most current "Oracle University" curriculum [9]. IBM also has a similar program with emphasis on a wide range of curriculum including web-programming, middle-tier management, and database administration using DB2. Sheridan has been approached many times by Oracle, IBM, Sun Microsystems and other vendors to "buy-in" to the perspective training curriculum programs and software. It is assumed that other Colleges and Universities have been similarly approached.

## Certification vs. Competency

The words "certified" and "competent" definitely do not mean the same thing. In reality most certifications merely set a base standard of competency and certify that the candidate managed to reach that standard. Many people incorrectly think that if a candidate completes the requirements for a certificate that they are competent to perform the associated job [10]. Competency for a specific IT job involves a number of different factors. Some factors are: (1) how good is the training vendor or organization, (2) how high is the competency standard (i.e., how difficult is it to pass), and (3) the obvious, how much experience does the candidate already have in the area of certification? For example, one IT expert said that he passed the

"Sun Microsystems Certified Java Programmer" exam and it said that it was a real challenge – he had to really work hard to earn that certification [10]. Conversely, this same expert said that to earn the "Macromedia Certified Coldfusion Developer" certification required extremely little effort [10].

Low competency standards for certifications not only allow people to "just-get-by" without becoming competent; they also devalue the qualification's prestige, and ultimately detract from its usefulness and credibility. Unfortunately, there are more than a few certifications out there that fall into this category [11].

There are a number of different certifications available for IT practitioners – over 400 at this time. This includes certifications by Microsoft, Oracle, Cisco, Novell, Sun Microsystems, A+, and many others [12]. In this paper only the most valued[1] and popular ones are presented: Microsoft and Oracle.

## Microsoft Certifications

Stating the obvious, Microsoft is the largest software company in the world. With so many different products that Microsoft develops and supports it is not surprising that this vendor has over 10 different certification programs – more than any other IT vendor in the world. The following is a list of the core IT certifications offered by Microsoft:

- Microsoft Certified *Desktop Support Technician* (MCDST);
- Microsoft Certified *Systems Administrator* (MCSA);
- Microsoft Certified *System Engineer* (MCSE);
- Microsoft Certified *Database Administrator* (MCDBA);
- Microsoft Certified *Trainer* (MCT);
- Microsoft Certified *Application Developer* (MCAD); and
- Microsoft Certified *Solution Developer* (MCSD).

### Microsoft Certification Benefits

Microsoft is so large and has such a command of the IT marketplace that it is very subjective to state the benefits of a certification from this vendor. However, there are a number of independent studies that have been conducted that consistently list Microsoft as one of the top 10 certifications in North American over last several years... "An independent study found that companies with at least 25% staff holding Microsoft certifications offer their company: (1) 15% increase on projects deployed on time and on budget; (2) 17% decrease in frequency of downtime; and (3) 14% increase in IT end-user satisfaction" [11, 13-16]

[1] The value of a certification is subjective. In this context the author is trying to be as objective as possible by simply referring to the Top 10 Certification list posted on [12].

Microsoft states the benefits of their certifications are:

- industry recognition of knowledge and proficiency with Microsoft products and technologies;
- discounts on MSDN subscription during the first year of certification;
- access to exclusive discounts on products and services;
- access to technical and product information through the Member Site;
- a Microsoft Certified logo, certificate, wallet card, and lapel pin to identify you as an Microsoft Certified Professional (MCP) to colleagues or clients;
- invitations to Microsoft conferences, technical training sessions, and special events; and
- free access to Microsoft Certified Professional Magazine Online, a career and professional development magazine.

Since there are so many certifications this vendor offers, only two are selected for this review: the Microsoft Certified Application Developer (MCAD) and the Microsoft Certified Solution Developer (MCSD).

### Microsoft Certified Application Developer (MCAD)

This certification is designed to attract computing practitioners who develop, test, deploy, and maintain department-level applications, components, Web or desktop clients, or database and network services. The recommended requirement before attempting certification is 1 to 2 years experience building, deploying, and maintaining applications. To earn the MCAD the candidate must pass four core exams and one elective exam. Each test is approximately $125 USD and may be taken at any one of the thousands of Prometric testing centres across Canada and the U.S.A. [17].

### Microsoft Certified Solution Developer (MCSD)

The MCSD is aimed for computer practitioners who are much the same as Professional Engineers, in that they analyze and design leading-edge enterprise solutions. For this certification, Microsoft recommends the candidate to have at least 2 years experience in a lead developer role analyzing business and technical requirements, and defining solution architecture. Similar to the MCAD certification, to earn the MCSD, the candidate must pass four core exams and one elective exam. Figure 2 depicts the two certification programs and their relation to the software development cycle.

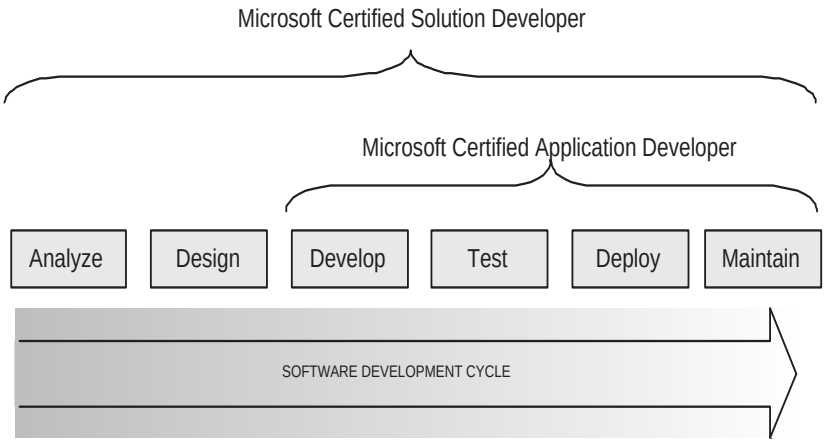



*Figure 2.* Microsoft Certified Solution Developer and Application Developer Certifications.

### Disadvantages

Like most certifications of this nature, these have a short lifespan. In other words, re-certification and re-training is necessary in order to keep up with the recent versions of the software. As a result, this takes time since it requires reading additional books, possibly taking instructor-led courses and preparing for the new exams. Understandably, this re-certification process is costly as virtually every new version of software that the vendor produces will require an upgrade-path to be taken if the candidate is interested in maintaining the validity of the certification. Although certifications typically do not expire, they do however, loose their credibility over time–particularly if they represent competency of skills for a software package that undergoes rapid changes that are constantly released as full versions to the public.

Compared to licensure, this may be considered a significant disadvantage of certification. However, it should be noted that this characteristic of certifications is not unique to Microsoft. Virtually all certification programs share this disadvantage.

## ORACLE

Oracle is a major database software vendor. It is currently the second largest software company in the world. Compared to Microsoft, there are surprising few certification tracks available for the IT practitioner. The most popular certification is the Oracle Certified Professional (OCP) designation which involves completing the 4 exams in the Oracle 9i version:

- Introduction to Oracle: SQL;
- Database Administration: Fundamentals I;
- Database Administration: Fundamentals II; and

- Database Performance Tuning

The OCP designation was ranked among the top certifications in 2002 with the fastest growing return on investment (ROI) [11, 13]. Currently, it is one of the most sought after marques of credibility for expertise in the Information Technology marketplace [16].

**Requirements and Disadvantages**

The requirements to earn any of the Oracle certifications are much the same Microsoft's certifications. The candidate must pass the required exams which are typically located at a Prometic testing centre at the cost of $125 USD per test.

The disadvantage to earning an Oracle certification is much the same as Microsoft's. The lifespan for these certifications is very short. Thus, the candidate must further invest time and money to become re-certified so that his/her credentials will be validated again. Within the last five years Oracle's database product line has gone through three distinct versions (i.e., Oracle 8i, 9i and 10g). It is interesting to note that when Oracle first began certification (Oracle 8i) there were 5 mandatory exams that the candidate required to pass in order to become certified with the Oracle Certified Professional designation. Today, with version 10g, there are only 2 exams required to earn the exact same credential.

## 15.4 Conclusion

This paper presented a review of the advantages and disadvantages of popular computing licenses and certifications. It included the eligibility requirements, fee structures, and membership restrictions for the different types of credential-issuing organizations. The main goal of the paper was to aid the reader in determining if these computing credentials are desirable for him/herself and to promote awareness of the various issues associated with each type of credential.

Although there are advantages to licensing and certification, there are relatively few computing professionals that pursue these credentials and even fewer are pursuing licensure. In the computing industry it is becoming less obvious why licensing is a significant credential for the practising of IT related work. Struggles similar to the Professional Engineers of Ontario and Microsoft on the issue of enforcement in the use of "software engineer" send clear messages to the public and computer practitioners. The IT industry is a rapidly moving entity with standards changing very quickly.

There are abundant differences between certifications however they are not as popular as forecasted by experts years ago. In other words, not every computer practitioner is running out and getting the latest and greatest certification. Some reasons why this is so can be attributed to the extremely short lifespan of IT certifications, the cost (measured in time and finances) of certification, and the constant need to re-certify when new versions of the product are released. Other reasons why certification may not be overly popular is because of increased competition. Colleges and other educational institutions are offering

more "certification-like" programs that result in the candidate earning a "certificate" on vendor-specific material even though it is not officially recognized by the vendor. An additional reason why there are not as many people pursuing certifications may be due to the fact that there are simply so many to choose from. With over 400 different certifications at this time how many more will there be in two years hence, or five years from now? There comes a point of saturation in the IT sector – and when this happens the credibility of all certifications will be compromised. For example, there are a number of computer specialists who make a game of certification – they try and earn as many as possible thinking this is the best way to enhance their career [3]. Such approaches are not wise yet common in the IT sector [3]. The result is a collection of people who have diverse knowledge and skills yet their depth of knowledge and experience is shallow.

As a result, the author predicts the future of certification will experience a significant decline until a reasonable level of credibility is established – perhaps with the assistance of the Ministry of Education and Training. The certification market will then stabilize and move forward in a more gradual fashion.

Regarding the future of licensure for the computing professional it appears that the decline has already begun [5]. Licensure has significant benefits that must be cultivated in the IT practitioner including ethical behaviour, honesty, and professionalism [1]. However, licensure lacks flare, and cannot confirm a person's practical skill or knowledge on a specific technology. Unfortunately, this is primarily what industries want in order to fulfill their specific business requirements. These reasons combined with the intense competition of vendors offering certifications, and private and public institutions offering their own "certificates" make it a challenge for licensing associations to grow in today's IT marketplace.

# Bibliography

[1] C. Abberfoil. "PEO's Legislated Mandate," 2005. *Retrieved from:* http://www.peo.on.ca/

[2] H. Bell. "The Professional Engineer: Registration and Regulations," vol. 2005. *Retrieved from:* http://www.answers.com/topic/professional-engineer

[3] T., J. Smith. "The Canadian Council of Professional Engineers," 2005. *Retrieved from:* http://www.ccpe.ca/e/index.cfm

[4] D. Pearce. "Licensing as a Professional Engineer: Answers to Frequently Asked Questions for Software Practitioners," 2005. *Retrieved from:* http://www.peo.on.ca/

[5] R., T. Jackson. "The Structure of the Canadian Information Processing Society," vol. 2005 *Retrieved from:* http://www.cips.ca/about/structure/

[6] K. Brown. "Top 10 Reasons Why Companies Hire I.S.P. holders," 2005. *Retrieved from:* http://www.cips.ca/standards/ispcert/#trademark

[7] T., S. O'Connor. "MSCD Certification Benefits," 2005. *Retrieved from:* http://www.microsoft.com/learning/mcp/mcsd/benefits.asp

[8] R., J. Price. "Certification Road Map: The Journey and the Destination," 2005. *Retrieved from:* http://www.computer.org/certification/cert_for_you.htm

[9] J., R. Ahmed. "Oracle Workforce Development Program," 2005. *Retrieved from:* http://workforce.oracle.com/

[10] A. Grant. "IT Certification: The Perks and Pitfalls," 2005. *Retrieved from:* http://www.sitepoint.com/print/it-industry-certification

[11] M. Villano. "2004 Certification Study," 2004. *Retrieved from:* http://crn.com/sections/special/certification/certification.jhtml

[12] B. Salam. "2004 Certification Study: Most Important Certifications (over the next six to 12 months)," 2004. *Retrieved from:* http://www.crn.com/sections/special/certification2004/ important.jhtml

[13] B. Nagel. "10 Hottest Certifications for 2002," 2002. *Retrieved from:* http://certcities.com/editorial/features/story.asp?EditorialsID=37

[14] B. Nagel. "10 Hottest Certifications for 2003," 2003. *Retrieved from:* http://certcities.com/editorial/features/story.asp?EditorialsID=55

[15] B. Nagel. "10 Hottest Certifications for 2004," 2004. *Retrieved from:* http://certcities.com/editorial/features/story.asp?EditorialsID=76

[16] B. Nagel. "10 Hottest Certifications for 2005," 2005. *Retrieved from:* http://certcities.com/editorial/features/story.asp?EditorialsID=86

[17] G. Young. "Thomson Prometric – Test Sites," 2005 *Retrieved from:* http://securereg3.prometric.com/Welcome.aspx

# Chapter 16

# Olivier Dragon and Mark Pavlidis: A Comparison of Requirements Specification Methods—Tabular Specifications vs. Statecharts

The importance of precisely defined requirement specifications has been documented in numerous references, and the number of methods to express the specifications does not fall far behind. In this paper, we evaluate two such methods – tabular specifications and statecharts – against a set of criteria, then contrast to the two methods based on this analysis. The evaluation is based on the partial specification of an actual system, a magnetic levitation feedback control system. In the conclusion, we discuss the merits of each method, the suitable roles for each. Finally, we propose possible improvements to the two methods that leverages the benefits of each for an improved methodology.

## 16.1 Background

Having precise formal requirements for an embedded control system is essential for its proper design and implementation. The specification includes not only hardware requirements but also for the software that controls the hardware. In this paper we compare two formal methods of describing software requirements for a magnetic levitation control system. The first method is one that follows David Parnas' ideas of tabular specification which has been used in software controls for airplanes and nuclear power plants [7]. The second method is statecharts, as described by David Harel [1] and widely used particularly in the automotive industry. We will be using Matlab's Statflow as the modelling tool and therefore adhere to its syntax and semantics for statecharts.

While there are other methods of formally defining requirements, these will not be compared. The scope of this paper will be restricted to tabular specifications and statecharts as these are two adequate and popular methods of formally and precisely defining requirements.

## 16.2 Motivation

The initial question we asked ourselves before delving deeper into the issue was why is there no *best practise* standard tool or method that is used by software engineers to create formal requirements for safety or mission-critical control systems? Both have been successfully used in real-life application yet no one best practise method is recognized. Thus we have set out to identify what are the benefits that have made each method successful, and the limitations that have prevented their widespread use and standardization.

The goal of this investigation is finding the benefits and limitations of each methods (tabular specifications and statecharts) in describing strictly functional requirements. We also wish to compare the two methods and hopefully find if one is superior.

In order to perform this comparison we will use a small part of a magnetic levitation control system. We will reduce the scope of the comparison by focusing only on the feedback control loop specification.

## 16.3 Hardware and Experiment Background

Figure 16.1 shows the big picture of the control system. The main idea is to use the magnetic field measurements from the hall effect sensor and implement a feedback controller that adjusts the current and its direction through the solenoid to either push or pull the permanent magnet.

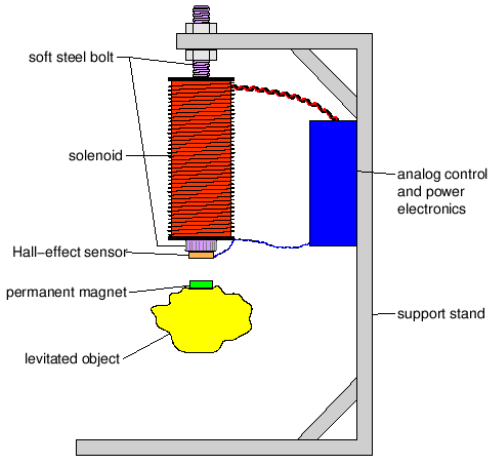



Figure 16.1: The magnetic levitation hardware setup[4]

## Comparison Criteria

The following criteria are used to evaluate the two specification methods. The evaluation for each criterion will be discussed for each method, then used in the comparison of the

methods.

### Completeness and Consistency

A specification that completely determines the externally visible response of the system, from the possible states of the system and any acceptable stimuli, is considered complete. This criterion ensures that every possible case is covered. A specification is consistent if only one response can be determined for a specific pair of state and stimuli. This ensures that each case is disjoint.

### Abstraction

This criterion evaluates the methods ability to precisely specify the behaviour of the system without imposing design level details. That is, the input-output relation is maintained as Mills type black-box without revealing how the output response is derived from the input stimuli. The imposition of design details must be excluded from specifications, unless defined as a physical constraint on the system.

### Ambiguity and Understandability

A precise requirements specification should only be able to be interpreted one way, thus eliminating the opportunity for differing designer interpretation. This criterion also evaluates the understandability of the specification. In particular, the ease of determining the defined response of the system, and tracing responses based on a state and set of stimuli. The ease of understanding what the system should do will aid in the verification and validation of the system.

### Executability

Given a precise specification, the ability to automatically execute it, or transform the specification into executable source code increases productivity and avoids the errors that are introduced by manually performing the implementation task. An executable specification that is verified correct and used for the implementation validation will further have less systematic errors that would have been introduced in the stages that are automated.

### Traceability

The traceability of a specification is critical to a readers ability to follow the flow of a requirement during the software lifecycle by linking a requirement to a design decision or a code block. Adequate traceability allows for propagating the inevitable changes in the requirements due to the natural evolution of the system or significant changes in the hardware.

#### Verification and Validation

The capability for verification (i.e. formally proving correctness) of the specification. The additional work and tools available to allow for verification is a point of discussion. The structure of the specification directly impacts the quality of the results from inspection and testing. Discussion and evaluation will focus on how well the methods facilitate validation.

## 16.4 Informal Software Controller Requirements

This section contains a list of requirements for the magnetic levitation controller. The formulation of the requirements expressed in prose were obtained by gathering all the requirements for the project. An analysis of the compiled requirements, including filtering out all but those specific to the magnetic levitation controller, organized the software requirements into the functional and non-functional categories. Within each category of requirement, the requirements are further decomposed as necessary. Each level discusses the details for the decomposition at that level.

### Functional Requirements

The functional requirements of the magnetic levitation controller consists of all the actions the controller must perform in order for the physical system to satisfy its operating conditions (i.e. maintain the levitation of the object).

The functional requirements are decomposed by the different modes of operation. A mode is a superstate of a grouping of states into subsets of similar states. A state is a specific set of values of the system variables. In this case (as is the case with with all deterministic controllers), the set of values is sufficient to determine future behaviour.

As previously mentioned we will be only focusing on the feedback control mode of the system for simplicity.

#### Mode: Initialization

The initialization mode is active when the device is first powered on, or reset. This mode is responsible for safe state start-up of the device.

1. The controller shall check that inputs to ensure each is in its proper operating range.

2. The controller shall set all outputs to the fail-safe values at start-up.

3. The controller shall initialize the state values of the PID controller.

4. The controller shall not power the solenoid during initialization.

**Mode: Polarity Detection**

The polarity detection mode is active when the device is normally operating, but it does not currently sense an object in the region of levitation. This mode is responsible for determining the polarity of the permanent magnet attached to the levitated object. The polarity will be used in order to determine correct control output.

1. The controller shall not power the solenoid while detecting the polarity.

2. The controller shall determine the polarity of the permanent magnet by monitoring the Hall Effect sensor input.

3. The controller shall identify a North pole facing upward when the Hall Effect sensor detects a negative magnetic field.

4. The controller shall identify a South pole facing upward when the Hall Effect sensor detects a positive magnetic field.

5. The controller shall begin feedback control to magnetically levitate the object when the polarity of the magnet it detected.

**Mode: Magnetic Levitation Feedback Control**

The magnetic levitation feedback control mode is active when then controller is actively controlling the outputs to maintain the levitation object by the magnetic field powered by the electric current running through the solenoid.

Figure 16.2 shows the control regions. It gives position of the permanent magnet with respect to how we want the controller to behave upon it.

1. The controller shall push the object into the operating region when it is detected that the object is too close to the solenoid.

2. The controller shall pull the object into the operating region when it is detected that the object is too far from the solenoid.

3. The controller shall allow the PID control constants to be tuned in order to improve the stability of the system.

4. The controller shall continuously execute the PID control computations and modify the system outputs accordingly.

5. The controller shall detect when the object has been removed from the region of control, cease control operations, and reset control state values.

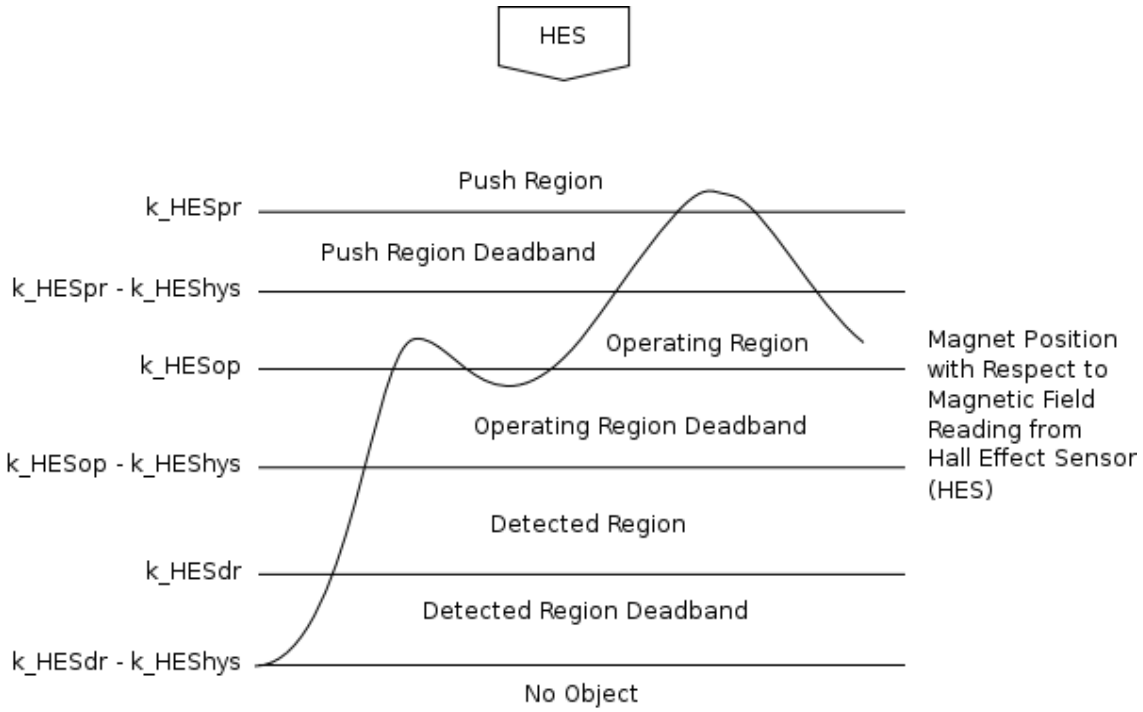



Figure 16.2: Control regions upon which the controller must change its control policy

**Mode: Safe Shutdown**

The safe shutdown mode is active when either a hardware problem/failure is detected, the reset or power off buttons are pressed.

1. The controller shall pull the object toward the solenoid until it is detected to be held in place.

2. The controller shall stop all control functions when shutting down.

3. The controller shall monitor the hardware fault that triggered the shutdown, and if restore normal operation if the fault no longer in effect.

## Variables and Constants

The following list of variables and constants will be used in both methods below in order to maintain consistency in the expression of the requirements. This convention is taken from [10]. Monitored variables (system inputs) are prefixed with $m\_$, controlled variables (system outputs) with $c\_$, and constants with $k\_$, $e\_$ and $y\_$ for scalar values, enumerated values and types respectively. Finally function values are prefixed with $f\_$.

**f_HESflux** Detected magnetic flux

**f_PIDout** Computed PID controller output magnitude

**f_PIDdir** Computed PID controller output direction

**c_PWM** Pulse-width modulated output current magnitude to the solenoid

**c_Dir** Direction of the output current to the solenoid

**k_ZeroDuty** Value of c_PWM when the controller should do nothing

**k_FullDuty** Value of the c_PWM when the controller should output maximum current

**k_HESop** HES limit to decide when an object is in a stable operating region

**k_HESdr** HES limit to decide when an object's presence is detected

**k_HESpr** HES limit to decide when an object is too close to the solenoid

**k_HEShys** Hysteresis value used to maintain a safety dead-band

**e_Push** Value of c_Dir when the solenoid should push the magnet away

**e_Pull** Value of c_Dir when the solenoid should pull the magnet closer

**y_DutyCycle** Type of the c_PWM values

**y_SolenoidPolarity** Type of the c_Dir values

## 16.5 Statechart Representation

Statecharts were introduced in 1984 by David Harel [1] as an extension of conventional finite state machines. For the simple reason of tool availability we settled on using the MathWorks Stateflow syntax and semantics, for which more information can be found in the Stateflow User's Manual [5]. The Stateflow representation of statecharts is reasonably similar to Harel's description.

The requirements expressed in statcharts are completely visual and resemble a simple flowchart. Each state of the system consist of one rounded-corner box in the picture. We also have superstates which are high-level abstract states that include more discrete sub-states. State transitions can have three pieces of information attached to them: a boolean expression condition upon which the transition can be taken, an event that triggers a particular transition of state and finally an action performed during the transition from state to state. Junctions (circles) are used to group state transitions. Finally functions (square boxes) are used to abstract from specific mathematical function details.

### Statechart Example

See Figure 16.3 for the example of a statechart specification for the magnetic levitation control system.

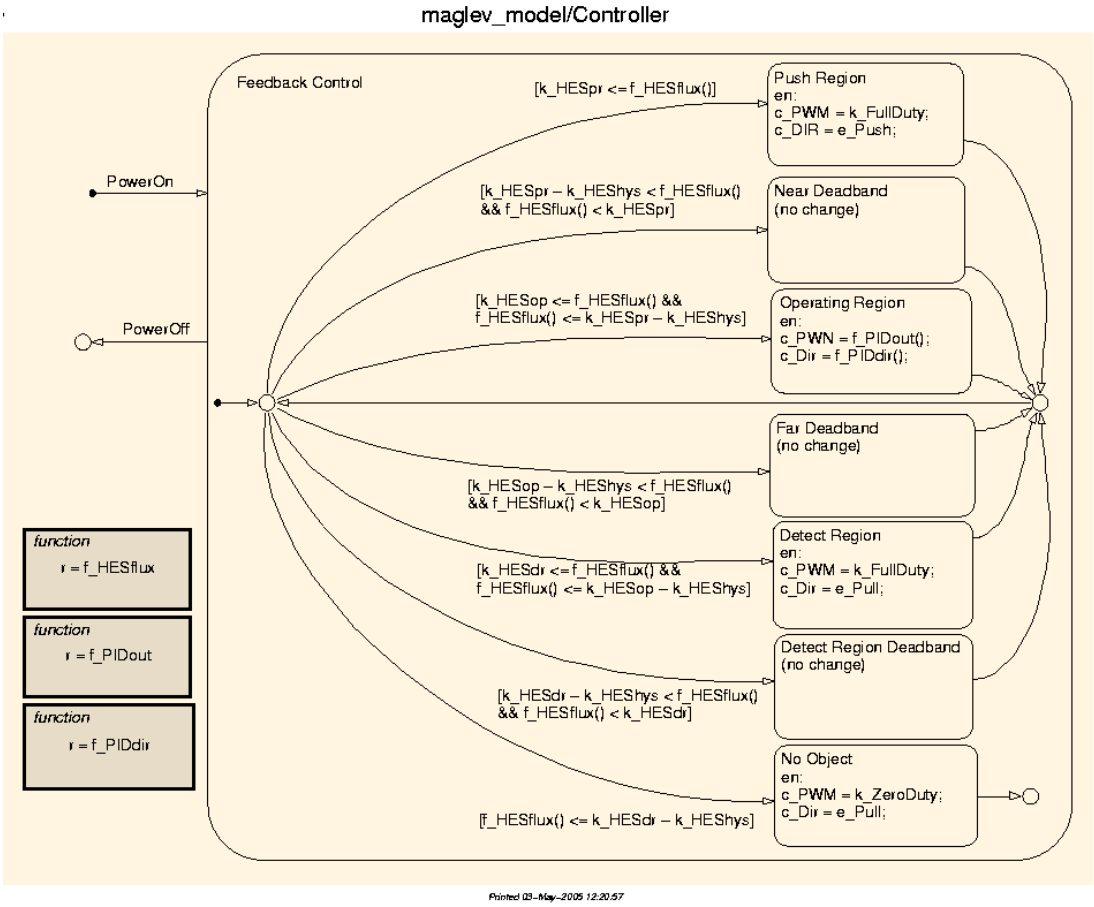



Figure 16.3: The Statechart model for the feedback control requirements

## 16.6 Statechart Criteria Evaluation

The following is the evaluation of the statechart method against the comparison criteria defined in Section 16.3.

#### Completeness and Consistency

Statecharts do not lend themselves well to creating complete and consistent requirements. From Harel's paper, dealing with such issues should be taken care of by the person making the statechart. However the subjective visual aspect of statechart makes it difficult to ensure

from the start that specifications are complete and consistent. By this we mean that two different statecharts capturing the same requirement, in the same functional way but with a different visual look may improve or worsen the ease of creating complete and consistent requirements.

Completeness and consistency is an understandability problem because if the requirements cannot be easily understood fully then these are more likely to be incomplete or inconsistent. Things may work in one case easily understood by the user but it may not work in all cases.

### Abstraction

Statecharts are very good when it comes to abstraction. They allow for a "mode" perspective very common in other engineering fields by using states and superstates. This modular abstraction helps with understandability (see below). It is also possible to keep away from design and implementation details by hiding these in "subcharted states," a particular feature of Stateflow that allows to hide (visually) substates and calculations for high-level statecharts.

One particular abstraction problem is with the use of the Stateflow application. The application itself does not enforce abstraction since anyone can go into subcharts of states and see the design details. Since these design details are often necessary for providing executability or useful code generation, using the Stateflow application as the means to read or view the requirements gives rise to leaks in the abstraction.

### Ambiguity and Understandability

There are many points which can be discussed about this criterion. Many of these, especially for understandability, are subjective. However we will try to provide arguments which although subjective we feel most people will agree with.

Statecharts, as mentioned in the previous section, are easy to understand in particular by non-software people as the system state or mode metaphor is common in other engineering fields. This makes statecharts an excellent way to convey ideas to people who have little knowledge of software design.

One major issue of ambiguity with Stateflow's statecharts in particular is the possibility for non-determinism to creep in the requirements unnoticed. This is because Stateflow doesn't check for completeness or consistency but instead uses the *12 o'clock rule* [5]. This rule dictates that if more than one state transition can be taken exiting a state, then the one closer clockwise to the vertical axis (12 o'clock) will be taken. This can be a major cause of ambiguity if the chart is modified or if the design uses a different transition priority rule.

Statecharts also have other understandability issues. One is the visual real estate required to draw complex system may come as a hindrance to creation, understandability and maintenance of requirements. Stateflow's subcharted states come in handy here but may interfere with abstraction.

Related to this real estate issue is description of variables. By simply looking at a statechart one cannot know the meaning of various variable or constant names. Stateflow

allows for associating documentation for each but this do not show in the statechart. Doing so may clutter chart and make it larger than it should be. This definitely takes away from the "self-explanatory" understandability factor.

Another is the fact that conditions, events and actions must use a programming language's syntax; easily understood by machines but less so by humans. These cannot be typeset in common mathematical syntax and may appear cryptic to non-software people.

#### Executability

This is an area where statecharts and Stateflow really shine. The tool support for executability is two fold: first the statechart can be executed using Simulink to view state transitions. Input values can be easily changed for testing and outputs can be monitored. Stateflow can moreover generate a C code program that will be based directly on the requirements. On the other hand for this program to be of any use some design decisions must be made which may lessen the level of abstraction.

#### Traceability

Stateflow itself does not provide capabilities to easily trace requirements to design decisions *unless* the design is itself also created using Stateflow via subcharted states or otherwise. In fact, Stateflow itself because of the code generation and state automata execution is more geared towards design than requirements elicitation. There may be other tools out there to help cope with this issue but none could be found for the purpose of this analysis.

#### Verification and Validation

Here we will only talk about automated verification. Manual verification was addressed in section 16.6.

Stateflow itself provides automatic syntactic verification but no verification of semantics, completeness or consistency. The executability *may* help a user find such problems, but as has been said numerous times before: testing shows the presence of bugs but not their absence.

On the other hand PVS has the abilities to deal with state transition system and thus it may be possible to validate Stateflow statecharts using it. The process of converting the statechart to PVS notation however may be a tedious and error prone activity. Automation may be possible to provide a more robust validation process.

## 16.7 Tabular Specification Representation

The tabular specifications are based on the Parnas-style function tables [2]. The tables are formatted based on the guidelines used by the Shutdown System One (SDS1) requirements engineers at Ontario Power Generation (OPG) that developed the Trip Computer Design

Requirements (TCDR), as described in [9], for the Darlington Nuclear Generating Station software shutdown system. The variation on the tables are that they are formatted and labelled in such a way that permits both software experts as well as domain experts to read and understand the documents.

The requirements are modelled as a finite state machine with an arbitrarily small clock-tick. It describes an idealized behaviour where monitored variable values and computed responses are determined instantaneously. The documentation uses the 4-variable model described in [8]. The documentation labels all system environmental stimuli as monitored variables, prefixed with $m\_$, and responses from the system as controlled variables, prefixed with $c\_$. When a natural decomposition of a subset of the REQ relation of monitored and controlled variables, as defined in [2], better communicates to the reader the specified behaviour, internal function tables, called Supplementary Function Tables [10] are used. The internal function results are prefixed with $f\_$. Further naming conventions are prefixes $k\_$, $e\_$, $y\_$ for constants, enumerated tokens, and types, respectively.

### Tabular Specification Example

The following, Table 16.1, is an example of the tabular expression method of specifying the informal requirements from Section 16.4. It specifies the outputs in terms of the controlled variables of the system controller (i.e. the pulse-width modulated current duty cycle, and the current flow direction) based on the input value of the Hall Effect sensor.

## 16.8 Tabular Specification Criteria Evaluation

The following is the evaluation of Tabular Expressions method against the comparison criteria defined in Section 16.3.

#### Completeness and Consistency

The structure and precision of tabular specifications lends itself to being verified for coverage and disjointness of the table predicates. The verification of simple tables may be by manual inspection, which is facilitated by the structure of the tables, where it is easy to check the completeness and consistency of the specification.

Automatic verification of completeness and consistency can also be done using theorem provers. For example, SRI's Prototype Verification System (PVS) supports input of Parnas-style function tables to generate Disjointness and Completeness proof obligations that must be discharged by theorem proving. PVS can, for all but very complex proofs, automatically prove the obligations. More importantly, when the built in proof strategies fail, an unprovable sequent results. The unprovable sequent provides a counter-example to when the specification does not hold, aiding in the identification of the incomplete or inconsistent specification.

## 16.7.3 Magnetic Levitation Control

### 16.7.3.1 Inputs/Natural Language Expressions

| Input | NL Expression | Reference |
|---|---|---|
| f_HESflux | Detected magnetic flux | TBD |
| f_PIDout | Computed PID controller output magnitude | TBD |
| f_PIDdir | Computed PID controller output direction | TBD |

### 16.7.3.2 Initial Value

| Output | Initial Value | Reference |
|---|---|---|
| $c\_PWM_{-1}$ | k_ZeroDuty | TBD |
| $c\_Dir_{-1}$ | e_Pull | TBD |

### 16.7.3.3 Output Type

| Output | Initial Value |
|---|---|
| c_PWM | y_DutyCycle |
| c_Dir | y_SolenoidPolarity |

### 16.7.3.4 c_PWM, c_Dir

| *Condition* | *Results* **c_PWM** | **c_Dir** |
|---|---|---|
| f_HESflux $\leq$ k_HESdr - k_HEShys<br>{*Hall Effect sensor does not detect an object*} | k_ZeroDuty | e_Pull |
| k_HESdr - k_HEShys $<$ f_HESflux $<$ k_HESdr<br>{*Object in the detection deadband region*} | $c\_PWM_{-1}$ | e_Pull |
| k_HESdr $\leq$ f_HESflux $\leq$ k_HESop - k_HEShys<br>{*Object is detected, and is pulled in to operating region*} | k_FullDuty | e_Pull |
| k_HESop - k_HEShys $<$ f_HESflux $<$ k_HESop<br>{*Object in the far operating region deadband*} | $c\_PWM_{-1}$ | $c\_Dir_{-1}$ |
| k_HESop $\leq$ f_HESflux $\leq$ k_HESpr - k_HEShys<br>{*Object in the operating region*} | f_PIDout | f_PIDdir |
| k_HESpr - k_HEShys $<$ f_HESflux $<$ k_HESpr<br>{*Object in the close operating region deadband*} | $c\_PWM_{-1}$ | $c\_Dir_{-1}$ |
| k_HESpr $\leq$ f_HESflux<br>{*Object detected to be in the push region, close to the electromagnet*} | k_FullDuty | e_Push |

Table 16.1: Tabular Specification of the Magnetic Levitation Controller

### Abstraction

The tabular specifications are a set of mathematical expressions that have a simple and intuitive meaning. They describe the functional behaviour, like any mathematical function, by not revealing in any manner how the result is obtained, but it has a precise meaning to know what is functional result. Thus, this method of specification is adequate for maintaining an abstract specification that is precise but does not delve into the details that should be left the designers.

### Ambiguity and Understandability

The precise nature of a complete and consistent tabular specification more than adequately satisfies the criteria of ambiguity. The tables are a set of simple expressions that have a structured presentation. The strategy of divide and conquer allows one to take a complex specification expression and, in combination with the mathematical language used, break it up into small precise expressions. A common fault for some unambiguous specification methods is that, in order to unambiguous, the expression is difficult to understand. The manner that the expression is broken up and presented with tabular notations allows one not to have to read the whole expression to use it. Further, the pieces of the expressions have been decomposed in the structure of the table, thus the reader need not mentally parse an expression. Therefore, tabular specifications are well suited for unambiguous yet understandable requirement specifications.

### Executability

The precision and expressivity of tabular specifications provide a basis for host of tool support that could include simulation of the system based on the requirements. At the present time, the tool support for tabular specifications is limited to development as academic prototypes or in-house industrial tools. There is not currently widespread commercial tools that provide for simulation of tabular specifications.

### Traceability

The lack of a complete set of tools for using the tabular expressions methodology, means it does not inherently have traceability support. The current manual construction of tables in word processors, or PVS specification language, does not have the capability to provide automatic traceability support. This task must be defined in additional tasks of the core methodology, and performed manually. The decomposition of the tabular specification structure affords the methodology the ability to adequately support traceability, but ability must be provided by tool support.

### Verification and Validation

The tool support, by PVS, for verification of tabular expressions is discussed in 16.8. The use of tabular notation allows specifications to be fully verified. The majority of the verification can be automated by PVS, and the remainder can be verified interactively with PVS. Full details of using PVS with tabular expressions is covered in [6], and its application for requirements expands on the material in [3].

Verification of the implementation by inspection is facilitated by the structure of the tabular specifications. The implementation is constructed from the tabular expressions, so to verify an inspector only needs to produce the table defined by the code, and compare that to the tabular specification. The precise, complete and constant nature of the specification allows for the tables to be simulated, or used as test oracle for the validation of the implementation.

## 16.9 Comparison of Specification Methods

The following section compares and contrasts the two specification methods against the evaluation criteria. When one method is clearly superior for a particular criteria, it is identified as such. Otherwise the relative strengths of the methods are discussed, and the choice is left to the reader to determine which method is best suited for a given system.

### Completeness and Consistency

The structure and tool support for verification of tabular expressions results in a far more superior methodology when it comes to completeness and consistency. It is quite unfortunate that statecharts have few ways to deal with such issue as it is a very important part of safety or mission-critical systems. It has been widely noted, in particular by Wassyng and Lawford in [10] that the primary reason tables are useful in the specification of requirements is the ease by which disjointness and coverage conditions are enforced.

### Abstraction

Both methods provide the necessary constructs to achieve a good level of abstraction for the specification of system requirements. The tabular method enforces the discipline of ensuring design level details are not included in the requirements. This is unlike the statecharts method that requires design level details in order simulate or execute the statechart.

### Ambiguity and Understandability

Both methods are precise and leave little room for ambiguity. The Stateflow's 12 o'clock rule does leave much to be desired, as it semantics are precise mechanically but may be less obvious to the reader. Understandability more often than not in the eye of the beholder. Both methods seem to present themselves well depending on what you are looking for. The

main understandability advantage of statecharts over tabular specification is the flow chart-like visual aspect which explicitly exposes system state transitions and is easily understood by non-software people. The main advantage for tabular specification is conciseness and rigorous structure.

#### Executability

The lack of tool support for the simulation of tabular specifications in comparison to the multitude of applications available for statecharts (e.g. Stateflow, Statemate) gives the latter an unmistakable advantage. This alone has most likely played a crucial role in the method's popularity in industry. If the value of a shorter development cycle that is provided by these tools outweighs the value provided by the criteria where tables excel then statecharts are an ideal specification methodology.

#### Traceability

As far as traceability of requirements to design decisions is concerned neither method excels. For both absence of adequate tools is the root cause for their lacking in this crucial area. The structure of tabular specifications provide the possibility for robust traceability but current tools are rudimentary at best.

#### Verification and Validation

The integration of tabular specification verification with a theorem prover, such as PVS, gives the method a definite advantage over statecharts. Statecharts tools only provide syntactic verification, and testing through executability. The executability can improve the confidence in the requirements' correctness but cannot ensure consistency and completeness. If Stateflow, or other statechart tools, would be able to harness the PVS capability of verifying state automata the evaluation of these methods would be equivalent for this criterion.

## 16.10 Conclusion and Recommendations

Decomposition of the requirements into modes facilitates statecharts since a mode is a superstate of a group of similar substates. This gives a readable pictorial in-the-large view of the system. For tabular specifications, the divide and conquer style aids in keeping the tabular expression simpler since it only considers the set conditions affecting the current mode, instead of all the conditions affecting every output in all modes. This gives a good in-the-small view of the system. Where only the details of the function in the table at hand need to be understood.

An improvement to current statechart tools, or future table tools, would be to leverage the benefits of the two methodologies. Both methodologies can be modelled as finite state machines using inputs (monitored variables) and timing events as transitions. A relation

could be developed to map the tabular expressions to conditions and actions of transitions in a statecharts. Thus a system could be specified using either method or both. That is high-level mode transitions are well suited for the graphical statechart where a large system view is of greater value. And tables are well suited for the more complex expressions used for those of specific state transitions within the modes. The verifiability that is the major benefit from tabular specifications can therefore be augmented by the ability to graphically connect components of the system. The automatic code generation capabilities of the statechart tools would further be improved by the correctness of the requirements and design.

In summary, both tabular specifications and statecharts methods for requirements specifications provide differing benefits and limitations. The more suitable method depends the specific project, and the goals and constraints of the project. By combining the benefits of the two methods the limitations of each could be compensated by the benefits of the other; providing thus a specification method that allows for a higher degree of correctness and understandability throughout the software lifecycle.

# Bibliography

[1] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[2] R. Janicki, D.L. Parnas, and J. Zucker. Tabular representations in relational documents. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected papers by David L. Parnas*, chapter 4, pages 71–87. Addison Wesley, 2001.

[3] M. Lawford, P. Froebel, and G. Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. Submitted to Formal Methods in System Design. `http://www.cas.mcmaster.ca/~lawford/papers/tables.pdf`, 2001.

[4] Katie A. Lilienkamp and Kent Lundberg. Low-cost magnetic levitation project kits for teaching feedback system design. Submitted to the 2004 American Control Conference. `http://web.mit.edu/klund/www/papers/ACC04_maglev.pdf`.

[5] MathWorks. *Stateflow User's Guide*. World Wide Web, `http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/stateflow.html?BB=1`, Published Year N/A. Last Viewed May 2005.

[6] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Revised May 1996.

[7] D.L. Parnas. Inspection of safety-critical software using program-function tables. In Daniel M. Hoffman and David M. Weiss, editors, *Software Fundamentals: Collected papers by David L. Parnas*, chapter 19, pages 371–382. Addison Wesley, 2001.

[8] D.L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[9] A. Wassyng and M. Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: International Symposium of Formal Methods Europe*, pages 133–153. Springer-Verlag, 2003.

[10] A. Wassyng and M. Lawford. Software tools for safety-critical software development. Accepted for publication in International Journal on Software Tools for Technology Transfer (STTT), 2005.

# Chapter 17

# Ramez Mousa: Software Failures

## 17.1 Introduction

A software failure, or a software disaster, occurs when an error or a bug in a software component, or its specification, leads to unexpected actions, and the system, upon which the software component operates, diverges from its intended actions. These failures result in substantial social and economical losses, and in some instances, the loss of human life. Thus, software failures can be very costly, and it is our duty and responsibilty as software developers, to gurantee that no such failures or disasters occur, and to ensure the saftey and wellbeing of others who may be affected by our work.

This paper disucsses five signiciant software failures that occured within the past two decades. It investigates and identifies the cause and reason that led to each failure, for example, incomplete or ambiguous requirements, implementation did not satisfy rquirements, or lack of or inadequate testing, etc, and provide recommendations on how to solve the problem and prevent it from occuring again in the future. It will conclude the discussion on each failure by identifying the lessons to be learned from the mistakes that caused the failure. The five failures chosen for analysis in this paper are the Thearc-25 which occured between 1985-1987, Ariane 5 in 1996, Patriot Missile in 1996, Mars Climate Orbiter in 1999, and the Columbia Space Shuttle in 1981. These failures were due to different reasons and caused by different mistakes from the developers as will be demonstrated throughout this paper. This choice of failures for discussion emphasizes the importance of each stage in the software development process since an error in any stage of the development process could lead to catastrophic results. Thus, proper care and attention must be given at each stage.

The purpose of this paper is not to point fingers of blame or criticize developers and companies who were involved in the development of the failed software components, but to identify and understand the lessons that should be learned from these mistakes and failures and to ensure that no such mistakes or errors ever occur again in future software components. An important part of learning and improving for the future is first understanding the mistakes of the past and the aim of this paper is to point out to new software developers the mistakes of their predecessors in order to help them learn from past experience and better prepare

them for the future.

## 17.2 Therac-25

The Therac-25 was a computer controlled radiation therapy machine developed in 1985 to treat cancer patients by irradiating them with protons or electrons at computer-controlled energy levels. The machine however, had a number of flaws and errors which resulted in six patients having massive overdoses of radiation between the time period of June 1985 to January 1987. In fact, the radiation levels were even up to thirty times the amount desired and resulted in the death of three of the six patients and permenant injuries to the others. These incidents have been described as the worst series of radiation accidents in the 35-year history of medical accelerators.

Investigation of the code of the machine found several errors that were mainly due to the machine running in a multitask environment with concurrent access to shared memory. Many taks ran nearly perfect in insolation, but it was not until the tasks started running asynchronously and communicating without proper access to shared data that problems were discovered. Many race conditions, initiated by unsynchronized access to shared memory, passed incorrect data to various tasks.

This paper focusses in specific on a concurrency problem in the code of the Therac-25 known as the Tyler problem or Tyler error (because it took place in Tyler, Texas). The Tyler error was mainly concerned with errors in the data entry routines that allowed the machine to be set-up for a patient's treatement before the operator of the machine has completed entering the full prescription or making later modifications to the entered prescription. For example, if the the operator of the machine made an incorrect entry for the energy level needed to treat a patient in the user's menu and attempted to edit the entry within 8 seconds, the modifications are displayed to the operator on the screen, *but* are not updated in the operating parameters of the machine.

We now investigate the code of the data entry routines to detect its errors while also suggesting improvements to the code in order to fix these problems. The following explanation is obtained primarily from [3]. Figure 1 shows the tasks and routines in the code blamed for the Tyler accidents. The treatment monitor task, Treat, controls the various phases of treatment by executing its eight subroutines as shown in the diagram. The treatment phase indicator variable, Tphase, is used to determine which subroutine should be executed. Following the execution of a particular subroutine, Treat reschedules itself.

One of Treats subroutines, called Datent, which is used during data entry, communicates with the keyboard handler task, which is a task running concurrently with Treat, via a shared variable called Data-entry completion flag. The Data-entry completion flag is used to determine if the operator has completed entering the prescription data for a patient. The keyboard handler recognizes the completion of data entry and changes the Data-entry completion flag accordingly. Once the Data-entry completion flag is set, the Datent subroutine detects the variables change in status and changes the value of Tphase from 1, which represents Data Entry, to 3 which represents Set-Up Test.

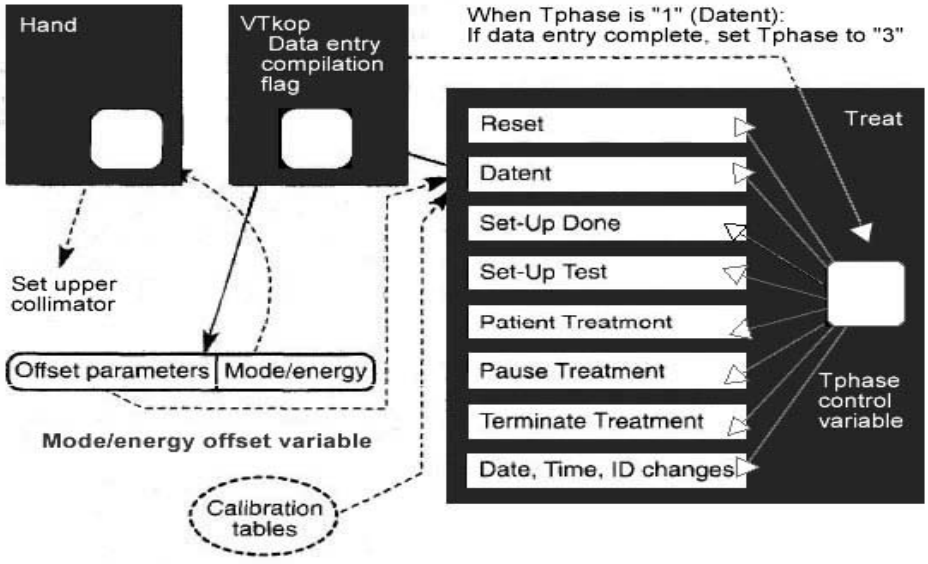



Figure 17.1: Tasks and routines blamed for Tyler error [3]

The data entry process forces the operator of the machine to enter the mode and energy values for the patient. The keyboard handler parses the mode and energy level specified by the operator and places an encoded result in another shared variable, the 2-byte mode/energy offset variable, MEOS, as shown in Figure 1. Datent uses the high-order byte of the MEOS variable in order to set several operating parameters. The operator can later edit the mode and energy levels separately. But, a potential serious problem can now occur if the keyboard handler sets the data-entry completion variable before the operator edits the data in MEOS. Datent will not detect the changes in MEOS since it has already exited and will not be reentered again. Therefore, an operator could be tricked into thinking that the edited input was accepted by the machine when infact it was completely ignored since Datent already exited. This is the first significant problem with the data entry routines, any edits performed to the mode/energy levels after the Datent routine has completely exited will not be detected [3]. In order to fix this problem, the data-entry completion flag should signal to the Treat task that the operator edited the data in order to reinvoke Datent. Although this was a serious problem, it was still not the only error in the data entry routines.

Another error in the data entry routines could be observed by closely studying the code of the Datent routine reproduced in Figure 2. The first thing that Datent does when entered is to check whether the mode/energy has been set it MEOS. If so, it sets all the operating parameters accordingly. Datent next calls the Magnet subroutine, which sets the bending magnets used in the patient radiation process. Setting the bending magnets takes about 8 seconds.

Magnet routine calls a subroutine called Ptime to introduce a time delay. Since several magnets need to be set, Ptime is entered and exited several times. A flag to indicate that bending magnets are being set, called bending magnet flag, is initialized upon entry to the

```
Datent:
  if mode/energy specified then
    begin
      calculate table index
      repeat
        fetch parameter
        output parameter
        point to next parameter
      until all parameters set
      call Magnet
      if mode/energy changed then return
    end
  if data entry is complete then set Tphase to 3
  if data entry is not complete then
    if reset command entered then set Tphase to 0
  return

Magnet:
  Set bending magnet flag
  repeat
    Set next magnet
    Call Ptime
    if mode/energy has changed then exit
  until all magnets are set
  return

Ptime:
  repeat
    if bending magnet flag is set then
      if editing taking place then
        if mode/energy has changed then exit
  until hysteresis delay has expired
  Clear bending magnet flag
  return
```

Figure 17.2: Code for the Datent (data entry) routine [3]

Magnet subroutine and cleared at the end of Ptime. Furthermore, Ptime checks a shared variable, called editing taking place, which is set by the keyboard handler. It indicates the presence of any editing requests from the operator. If there are edit requests, then Ptime clears the bending magnet variable and exits to Magnet, which then exits to Datent. However, the edit change variable is checked by Ptime *only* if the bending magnet flag is set. Since Ptime clears it during its first execution, *any* edits performed during each succeeding pass through Ptime will not be recognized. This is precisely the second significant problem with the data entry routines, any edits performed before Datent exits but during the setting up of the bending magnets, which take 8 seconds, will not be detected since the magnet bending take about 8 seconds and Magnet does not recognize edits after the first execution of Ptime. Thus, an edit change of the mode or energy, although reflected on the operators screen and the mode/energy offset variable, will again not be sensed by Datent, and therefore, the operating parameters will not be updated. The problem could be fixed by clearing the bending-magnet variable at the end of Magnet, after all the magnets have been set, instead of at the end of Ptime [3].

The above discussion explains the significant errors in the data entry routines relating to the Tyler problem. However, the Therac-25 also had other concurrency-related problems, such as the Yakima error (took place in Yakima, Washington), which was caused by errors in the code of the Therac-25 that allowed the machine to be activated in an error setting (failure of software interlock). These problems are outside the scope of this paper, however, they all similar to the first Tyler problem dicussed above and arise from tasks running in a multi-task environment.

There are many important lessons to learn from the failure of the Therac-25. First and foremost, is the importance of testing. Many of the errors and bugs in the code of the Therac-25 could have been easily eliminated through proper and adequate system testing. For example, the keyboard handler error, where the keyboard sets the data-entry completion variable before the operator finishes editing the data in MEOS (a concurrency problem), could have been easily exposed through testing. It was later revealed through an investigation committee created to determine the cause of the failure of the Therac-25 machine, that the machine was tested as a whole for only 2400 hours. For machines of such criticality and magnitude, experts suggest that it should have been tested for at least twice that duration and testing should have included various scenarios (editing of the prescription at different times and from different places, and so on). It is quite evident that testing was not sufficient on this machine. Moreover, it was also revealed that there was no unit testing performed at all on separate modules. The second error discussed in the Tyler problem above (bending magnets flag error) could have been detected through proper unit testing of the Datent routine. One of the test cases on the Datent routine could have been the situation where the operator enters some data and attempts to change immediately. This would have exposed the bending magnets flag error and shown that under such conditions, the magnets of the machine would not be configured properly. Thus, unit testing is very important and very effective in detecting many errors that lie within a module.

There are still other reasons, besides insufficient and inadequate testing, for the failure of

the Therac-25 machine that we can learn alot from. First, the code for the Therac-25 machine was written in its entirety by one programmer. This was a fatal error because the Therac-25 was far too large and complex to be developed in its entirety by one programmer. More programmers should have worked together on a system of such criticality and magnitude since many bugs and errors can be easily overlooked if only one programmer is responsible for the entire system. The principles of software engineering encourage team work and division of tasks in such large systems. Moreover, it was also the same programmer who also did all the testing and documentation! This resulted in very little documentation during development and poor testing since the tasks of the one programmer were unbearable. Proper documentation is very important and makes the code easier to understand, more clear, and also aids in the process of debugging and detecting erros. If more programmers were involved in the development of the machine, then the quality of the code, testing, and documentation would have improved significantly and many problems would have been eliminated. It is quite evident that the programmer who developed this system in its entirety has been given too many responsibilites and too many tasks to complete, which resulted in a poor overall system. We must avoid such situations and ensure that team work and work division is adequate, therefore, yielding a much better overall product. Finally, a last important point to make is that the code of the Therac-25 machine was not inspected in any way. Code inspection is crucial and helps detects minor bugs that some test cases might not be able to detect. Code inspection and testing complement each other and together they can help eliminate hidden bugs in the code.

## 17.3 Ariane 5

The Ariane 5 was a rocket developed by the European Space Agency (ESA) in the middle 1990s in order to deliver a satellite into space. The ESA established an outstanding record of success with their Ariane-series of rockets and the Ariane 5 was expected to be the most successful one yet. However, on June 4, 1996, the Ariane 5 rocket exploded exactly 42 seconds after liftoff when the rocket veered of its flight path and broke up.

A board of inquiry was immediately set up in order to investigate the cause of the failure and after extensive investigation, the origin of the failure was narrowed down to the flight control system. Fortunately for the inquiry board, two primary computer-controlled Inertial Reference Subsystems (IRS), which determine the rocket's altitude and position, and transmit this information to the main control computer, were recovered. The two IRS units were completely indentical and redundant. The primary IRS unit provided the data for the main control computer, while the secondary IRS unit was a backup for the primary and stood-by ready to immediately replace the primary in case of failure [1].

After exntensive investigation of the two IRS units, the source of the failure was determined to be a software error within the two subsystems. In specific, an integer overflow error occured when the primary IRS attempted to convert a 64-bit floating point number to a 16-bit integer. The floating point number, which measured the horizontal velocity if the rocket, was simply too large to be represented as a 16-bit integer [1]. The overflow error

caused the computer in the primary IRS to shut down and attempt to switch control to the backup, redundant, secondary IRS unit. Control was switched to the secondary unit, however, since the secondary unit was redundant and identical, it had already experienced the same overflow error itself and had already shut down when the primary unit attempted to transfer control to it [1].

Furthermore, the inquiry board noted that the IRS software code for the Ariane 5 was derived directly from that of the the Ariane 4 where it was used successfully. The IRS software code worked flawlessly with the Ariane4 because the rocket had a completely different initial trajectory from that of the Ariane 5, which produced much smaller horizontal velocity values. Thus for the Ariane 4, the floating point number, which as previously mentioned, stored the horizontal velocity of the rocket, was easily converted to a 16-bit integer without causing an overflow error [5]. However, this was no longer the case with the Ariane 5 since its trajectory produced a much higher horizontal velocity that could no longer be represented as a 16-bit integer. Such a detail (Ariane 5 has a much higher horizontal velocity than Ariane 4), although minor, was completely overlooked, but it was the main reason for the explosion for the Ariane 5 rocket which cost a massive $7 billion dollars to design, develop, and build [5].

There are many lessons to learn from the explosion of the Ariane 5 rocket. First and foremost, is again the importance of testing. Any flight simulation, or preflight software testing, would have detected the unexpected higher values of the horizontal velocity of the Ariane 5 rocket, and the resulting overflow error it caused. Next we note that hardware redundancy is not a perfect approach to providing fault tolerant software. It may be appropriate when dealing with hardware failures, however, it is ineffective in dealing with software failures. The two IRS units were identical and redundant, so when the primary IRS unit failed, it was inevitable that the secondary unit will do likewise. The two units should have been different in case there is a hidden error in one of them, then the second could properly take over control from the first and maintain the system in a normal, safe mode, thereby providing proper fault tolerance since it is highly unlikely, that both units would suffer from the same errors if they were design and developed in completely different ways.

We also note that proper updates and modifications must be applied to existing software components before integrating them into a new system. The IRS code for the Ariane 5 was reused from the Ariane 4 without first updating and properly modifying it which was the main reason for the failure. While software reuse fits nicely within the principles of software engineering, it may still be quite tricky since minor details can be easily overlooked when reusing components. Therefore, special care and attention is required and proper modification needs to be applied to existing components before software can be reused. In fact, software update is an undocumented precondition to software reuse. Finally, we note the importance of the proper use of exception handling. The inquiry board of the Ariane 5 noted that while exception handling was present in the code of the IRS, it was not used properly. The exception handler simply shut down the system once the overflow error occured when it should have attempted to recover from the error. Exception handling is very important, and can the system recover from many errors that may arise and return the system to a

safe mode. Unfortunately it is quite often misused, as in the Ariane 5, or forgotten about altogether.

## 17.4 Patriot Missile

The Patriot Missile was developed by the U.S. Department of Air Defence in order to intercept Iraqi Scud Missiles during the Gulf war of 1991. The Patriot Missile batteries were initially widely hailed for their effectiveness in intercepting Iraqi Scud Missiles as the Pentagon announced that approximately 90% of all Scud Missiles were being intercepted by the Patriot Missiles. However, in the months following the end of the war, the ineffectiveness of the Patriot began to appear. In fact, one specific critical failure, when a Scud Missile hit a U.S. military barrack in Saudi Arabia on February 25, 1991, killing 28 U.S. soldiers and injuring many others, was being blamed on the ineffectiveness of the Patriot and its failure to intercept the Scud [4].

A careful analysis of the Patriot Missile system revealed that there was a problem in the system's weapons control computer that slowly reduced the system's accuracy as time went by. The Patriot Missile maintained a "time since last boot" timer expressed as an integer in units of tenths of seconds. In order to predict where the Iraqi Scud Missile will appear next, the timer needs to be converted to a single precision floating point number. The exact problem of the system occured when attempting to store the converted floating point number into the registers of the Patriot Missile. The registers of the Patriot were only 24-bits long. Thus, the floating point number must be rounded-off in order to be in the registers. This rounding-off of the floating point number, which as previously mentioned stores the "time since last boot" timer, resulted in loss of precision causing less accurate time calculation that became worse the longer the system operated [4].

In fact, after the system has been running continuously for 8 hours, time calculation drifted by about 0.0275 seconds, which was enough to yield a 55 meter error in the prediction of where the Scud will appear next, thereby, completely missing the Scud. At the time the Scud Missile hit the U.S. barrack, the system has been running for 100 hours increasing the time error by about 0.3433 seconds, yielding 687 meters of timing inaccuracy [5]. Thus, the effectiveness of the Patriot was closer to 10% rather than the 90% announced by the Pentagon [4].

The main lesson to learn from the failure of the Patriot Missile is the importance of verifying and validating assumptions. Extra care must be taken when making assumptions. The developers of the Patriot Missile assumed that the floating point number could be simply rounded-off and stored in 24-bits long registers without taking into consideration that timing calculation is very sensitive and a simple round-off calculation could cause drastic effects. They should have been more careful when making this assumption and should have verified and ensured that such rounding-off calculation would not affect the accuracy of the system. Another important lesson to point out is again related to testing. Load testing of the Patriot Missile, testing under prolonged periods of time, which is vital for systems that run continuously such as the Patriot Missile, would have easily detected the inaccuracy of the

system and prevented the failure and saved the lives of the American soldiers who died when hit by the Scud Missile.

## 17.5 Mars Climate Orbiter

The Mars Climate Orbiter was a spacecraft launched by NASA to Mars to study the Martian weather, climate, and water and carbon dioxide budget in order to understand the reservoirs, behaviour, and atmospheric role of volatiles, and to search for evidence of long term and episodic climate changes. However, the spacecraft was lost and subsequently destroyed when a navigation error caused the spacecraft to miss its intended 140-150 km altitude above Mars during orbit insertion, instead entering the Martian atmosphere at about 57 km. At this very low altitude, the spacecraft was destroyed by the atmospheric stress and friction [2].

NASA appointed a review board to determine the cause of the failure of the Mars Climat Orbiter. After a thorough investigation, the board reported that the main cause of the loss and destruction of the spacecraft was the failed translation of Imperial measurements (inches) into Metric measurements (meters) in a segment of ground based, navigation related, software. In specific, the astronautics team in Colorado, U.S., working on the navigation of the spacecraft, submitted acceleration data in Imperial units to the jet propulsion team in California, who entered the data into a computer that assumed Metric units. This error threw off the program that calculates how slight changes in the spacecraft's angular momentum affects its path towards Mars. As time went by, the slight discrepancies due to the difference in the units used, built up in NASA's projection of the Climate Orbiter's course eventually leading the Orbiter to miss its intended altitude by around 85 km upon reaching Mars [2, 8].

Engineers that developed and coded the ground based, navigation related, software later admitted that the measurements system to be used in the software was completely overlooked. It was a very minor detail, however, it was not never taken into account and it solely led to the destruction of the spacecraft which cost $125 million to design and build. The difference in the system of measurement to be used propogated through the stages of the development process of the system to the point where it resulted in a major difference in the various development teams' understanding of the spacecraft's path as it approached Mars [2]. Thus, this failure demonstrates how minor details could lead into fundamental misunderstandings and cause catastrophic results. Quite often small details are easily and completely overlooked, so software developers should give them special attention.

The failure of the Mars Climate Orbiter should also alert software developers to the importance of proper documentation and communication among all teams involved in developing a system. It is quite evident that the system had incomplete and improper documentation. The documentation should clearly state the system of measurements used so that developers are clear on such an important issue. It is also evident that the two teams involved in developing the navigation related software had minimal communication throughout the development of the Orbiter since they were both using different measurement units. The system of measurement was never discussed when in fact it should have been discussed in the early stages of the development process. Had the teams discussed the system of measure-

ments earlier and properly documented it, the whole failure would have been avoided and millions of dollars saved. Thus, communication among development teams and proper documentation are very important and they help clarify all confusions that may exist between different teams. Thereby, avoiding any disasters that may occur.

## 17.6 Columbia Space Shuttle

The Columbia Space Shuttle was developed in the early 1980s for a mission in outer space. There were a few minor bugs in the Shuttle Mission Simulator, SMS, of the space shuttle, which was developed by IBM's Federal Systems Division, that the software developers did not know about before the scheduled liftoff of the space shuttle. Fortunately though for the software developers of the space shuttle, there was some fuel accidently spilled on the body of the shuttle just before the scheduled liftoff that forced the development team to postpone and reschedule takeoff of the shuttle until all fuel spill is cleaned up. The clean up of the fuel was estimated to take a full month. The development team, observing that they had nothing to do for a month until all clean up is complete, decided to spend more time working on the SMS [6].

They decided to simulate a "Transatlantic abort sequence", a backup plan used when the shuttle can can neither return to the launch site nor acheive orbit. In such a scenario, the mission is aborted and the shuttle is redirected to an early landing in Spain after dumping excess fuel. When the crew issued the mission abort command, all four of the redundant flight control computers simultaneously locked up and became completely unresponsive. This exposed the hidden bugs in the SMS to the development team. Had this occurred during a real flight, it is unlikely the shuttle could have been safely landed [6].

The development team, shocked from the outcome of the test simulation, but quite relieved that such error did not occur during a real mission, spent a long time investigating the cause of the failure. They narrowed the fault down to a multi-purpose module that is used during shuttle takeoff, landing, and dumping of excess fuel at several points of the trajectory in preparation for an early landing. The module was first invoked during shuttle ascent, it went through its process correctly and completely and even terminated successfully. However, when it was invoked again from a different point in the software in order to open the gas tanks and dump the excess fuel for the early landing, there were some counters in the code that had not been reinitialized. In specific, one of the counters that were not reinitialized was a variable used as a basis for a GOTO branch in the code. The code was expecting this counter to have a value between within a specific range, but because the counter was not reinitialized, it started out with a much higher value. Eventually, the code encountered a value beyond the expected range, which caused it to branch out of its logic to a memory address containing no code. This in turn caused all four redundant and indentical flight control computers to crash simultaneously [6].

A simple fix took care of the problem. However, the development team was not satisfied because they were committed to producing only the best possible code. They wanted to ensure that this bug, or any other bug for that matter, are completely eliminated from

their code. So, they came up with a systematic way to eliminate such generic bugs. They developed a list of seven questions that had a high probability of isolating the bugs [7]. A random group of programmers then applied these questions to the incorrect multi-purpose fuel dump module as well as other modules in order to see if they would indeed detect any problems. This process worked incredibly well, detecting 17 previously unknown bugs in the code! One of which would have caused catastrophic results! Many of these newly detected problems included counters that were properly reinitialized in several other modules. This is software engineering at its best. Instead of quickly fixing the problem in the code and moving on quickly, as most developers do, the development team in this case transformed the defect into an opportunity to improve their code [7].

The damage caused by the bugs in the SMS was luckily minimal because it was discovered during an unplanned simulation, however, the bugs could have just as easily arisen during a real mission if the original takeoff was not postponed and led to an unfortunate disaster with a large social and economical cost. But despite the low cost of the failure, there is still alot to learn from the development team of the Columbia Space Shuttle. First and foremost, is the committment to produce only the best possible code. Once the development team discovered a bug in the code, they did not simply fix the bug and move on, but they derived an approach to help eliminate all similar generic bugs. Their sole goal was to produce an error-free software. Software developers should follow the excellent standards set by this development team and likewise ensure that only the best possible software is delivered. Moreover, nowadays more and more software is used in saftey-critical systems, which over emphasizes the importance of producing highest quality software since the safety and wellbeing of others is at stake.

Another important lesson to learn from the Columbia Space Shuttle is that quite often a developer commits the same mistake or error at several different places in the code. Thus, when an error is detected, its always worthwhile to go back and check other places in the code where similar errors may have occured and ensure that they are non-existant. The 17 errors later detected by the development team of the space shuttle were mostly counters that were not reinitialized. It was the same error occuring in several places in the code. This demonstrates that developers could easily make the same error several times. The space shuttle development team realized this fact and wisely reviewed all modules where similar erros may occur to ensure correctness of the code. Finally, as mentioned previously in the discussion of the Ariane 5 explosion, hardware redundancy is not an effective approach to providing fault tolerant software. The two flight control computers were redundant and running identical software.

## 17.7 Concluding Remarks

This paper discussed five important software failures that occured in the past two decades. It analyzed the problem that led to the failure, gave recommendations on how to the solve the problem and prevent such problems from occuring in the future, and identified the lessons to be learned from these failures. The most important and most valuable lesson is

the importance of testing. Every failure discussed in this paper could have been eliminated through adequate and sufficient testing. Testing improves the quality of software and helps detected hidden errors in the code. A study on software testing done in 2002 discovered that software bugs cost the U.S. economy $59.5 billion annually. The same study however, found that more than a third of that cost, around $22.2 billion, could be eliminated by improving testing. Thus, testing can help detect errors, eliminate problems and most importantly save precious human lives.

Another important lesson to learn from the failures discussed is the importance of various software engineering principles such as team work, communication between team members and different teams, clear and complete requirements, sound design, correct implementation, and most importantly, the committment to producing only the best possible software. Nowadays, more software is used in safety critical systems. There is a movement towards including software in many more systems. This was never the case a few decades ago. So, software developers have now become responsible for the safety and wellbeing of others, which may not have occured so frequently in the past. Thus, software developers must take this new responsibilty and ensure that all software engineering principles are followed, all proper saftey measures are taken in producing the software, and that the safety of others is ensured. Lets learn from the mistakes of the past and follow the example and high standards set by the development team of the Columbia Space Shuttle and take every error or defect in our code as an opportunity to improve our software, and commit ourselves to producing only the best possible product!

# Bibliography

[1] Arnold, Douglas N. (1996). *Two disasters caused by computer arithmetic errors.* Retrieved April 10, 2005, URL: http://www.ima.umn.edu/ arnold/455.f96/disasters.html.

[2] Booyle, Adam. (2000). *Mars Climate Orbiter.* Retrieved April 15, 2005, URL: http://techcenter.davidson.k12.nc.us/Group3/marsorbiter.htm.

[3] Leveson, Nancy, and Turner, Clark S. An Investigation of the Therac-25 Accidents. *Computer: Innovatice Technology for Computing Professionals*, 26(7):18–41, 1993.

[4] Skeel, Robert. Roundoff Error and the Patriot Missile. *SIAM News*, 25(4):11, 1992.

[5] Social Themes: Risks in Numeric Computing. Retrieved April 10, 2005, URL: http://cs.furman.edu/digitaldomain/themes/risks/risks_numeric.htm.

[6] Spector, Alfred and Gifford, David Case Study: The Space Shuttle Primary Computer System. *Communications of the ACM*, 27(9):874-900, 1984.

[7] The Ganssle Group. Disaster! *Embedded Systems Programming*, 1998.

[8] Wikipedia Free Encyclopedia. (2005). *Mars Climate Orbiter*. Retrieved April 14, 2005, URL: http://en.wikipedia.org/wiki/Mars_Climate_Orbiter.