

Topics in Software Design
Volume 2

Emil Sekerinski (Ed.)

SQRL Report 35
McMaster University

April 2006

Contents

Introduction	1
1 Lei Hu: Design Recovery of Specifications	3
1.1 Introduction	3
1.2 Reverse Engineering Review	4
1.3 Foundation of Design Recovery	6
1.4 Classification of Design Recovery Techniques	11
1.5 Constructing Formal Specifications from Source Code	13
1.6 Conclusion	20
1.7 Problems	20
Bibliography	21
2 Jiacong Zhang (Kevin): Theories of Refactoring	23
2.1 Introduction	23
2.2 Background	23
2.3 Refactoring Tools	27
2.4 Refactoring Theories	31
2.5 Concluding Remarks	36
2.6 Exam Question	36
Bibliography	39
3 Saba Aamir: Memory Management Strategies In Programming Languages	41
3.1 Introduction	41
3.2 Explicit Memory Management Strategies	42
3.3 Semi-Automatic Memory Management Strategies	43
3.4 Automatic Memory Management Strategies	47
3.5 Programmers Point Of View	61
3.6 Conclusion	61
3.7 Exam Questions	62
Bibliography	62

4	Andi Huang: Fault Tolerance	65
4.1	Techniques For Achieving Software Fault Tolerance	66
4.2	Techniques For Measuring Software Fault Tolerance	74
4.3	Conclusion	78
4.4	Questions	78
	Bibliography	78
5	Jay Parlar: Dynamic Languages	81
5.1	Introduction	81
5.2	History of Dynamic Languages	82
5.3	Type Systems	83
5.4	Disadvantages of Dynamic Languages	88
5.5	Advantages of Dynamic Languages	91
5.6	Interactive Interpreter	95
5.7	Conclusion	98
5.8	Exam Questions	99
	Bibliography	99
6	Yu Wang: A Survey of Software Distribution Formats	103
6.1	Fat Binary	103
6.2	Application Oriented Virtualization	104
6.3	Platform Oriented Virtualization	108
6.4	Distribution In Source Code	109
6.5	Other Distribution Formats	113
6.6	Concluding Remarks	114
6.7	Exam Question	115
	Bibliography	116
7	Ayesha Kashif: History of Statecharts	119
7.1	Introduction	119
7.2	STATEMATE	120
7.3	ECSAM	126
7.4	UML Statecharts	129
7.5	Modecharts	131
7.6	Plug and Play	133
7.7	Hypercharts	135
7.8	Concluding Remarks	135
7.9	Exam Question	136
	Bibliography	136

8	Salvador Garcia: Misuse Cases	139
8.1	Introduction	139
8.2	Basic Concepts	140
8.3	Misuse Case Diagrams	141
8.4	Method Guidelines	142
8.5	Methodology	143
8.6	Template	144
8.7	Eliciting Exceptions and Test Cases	145
8.8	Example	145
8.9	Conclusions	148
8.10	Exam Questions	149
	Bibliography	149
9	Software Design Teaching Methods	151
9.1	Introduction	151
9.2	Integrating Testing and Design Methods for Undergraduates: Teaching Software Testing in the Context of Software Design	153
9.2.1	Advantages of the Method	156
9.3	Teaching Software Design with Open Source Software	157
9.4	Teaching Undergraduate Software Design in a Liberal Arts Environment Using RoboCup	159
9.5	Teaching Software Engineering Using Lightweight Analysis	162
9.6	Teaching Software Engineering Through Simulation	163
9.7	Conclusion and Discussion	166
9.8	Exam Questions	166
	Bibliography	166
10	Lutfi Azhari: Software Documentation Environments	169
10.1	Javadoc	170
10.2	Doxygen	171
10.3	DOC++	173
10.4	SODOS	175
10.5	SLEUTH	177
10.6	Variorum: A multimedia-Based Program Documentation System	180
10.7	Literate Programming	183
10.8	Design and Architecture Documentation	184
10.9	How much documentation is enough?	185
10.10	Summary	185
10.11	Exam Questions	186
	Bibliography	186

11 Ming Yu Zhao: Object-Oriented Literate Programming	189
11.1 Introduction	189
11.2 Object-Oriented Design	190
11.3 Literate Programming	193
11.4 Complement and Conflict	194
11.5 An Example: A Banking System	196
11.6 Conclusion	238
11.7 Exam Questions	238
Bibliography	238
12 Hongqing Sun: History of Tabular Expressions	239
12.1 Introduction	239
12.2 History and Real Word Tables	240
12.3 Decision Tables from 1950s	242
12.4 First Large Application of Tables: A-7E Aircraft Program Requirements	243
12.5 Why Tabular Expressions?	244
12.6 A Milestone of Tabular Expressions	245
12.7 Theoretical Ripeness in SERG	248
12.8 TTS: Table Tool System of SERG	253
12.9 Other Table Notations Projects	258
12.10 Tabular Expressions Today	260
12.11 Exam Questions	261
Bibliography	261
13 Jorge Santos: Architecture Description Languages	265
13.1 Introduction	265
13.2 Characteristics of ADLs	266
13.3 Differences Between ADLs and Other Languages	267
13.4 Some Sample ADLs	267
13.5 Concluding Remarks	275
13.6 Exam Questions	275
Bibliography	275
14 Dan Zingaro: On the Practice of B-ing Earley	277
14.1 Languages and Recognizers	277
14.2 General Context-Free Recognizer Machine	278
14.3 State Sets	280
14.4 Refinement 1 – Intuition Behind State Sets	281
14.5 Refinement 1 – Refinement Machine	281
14.6 Refinement 1 – Linking Invariant	284
14.7 Refinement 2 – Earley’s Algorithm Revisited	284
14.8 Refinement 2 – Refinement Machine	285
14.9 Refinement 2 – Linking Invariant	287

14.10	Next Steps	287
14.11	Exam Questions	288
	Bibliography	288
15	Marwan Abdeen: Use Cases, Scenarios, Sequence Diagrams and Message Sequence Charts	289
15.1	Use Cases	289
15.2	Scenarios	298
15.3	Sequence Diagrams	299
15.4	Message Sequence Charts	303
15.5	Sequence Diagrams Versus Message Sequence Charts	309
15.6	Exam Questions	312
	Bibliography	312
16	Jie Gui: Models for Configuration Management	315
16.1	Introduction	315
16.2	Models for SCM	317
16.3	Implementations of SCM Tools	330
16.4	Concluding Remarks	341
16.5	Exam Question	341
	Bibliography	342

Introduction

This collection of papers is produced by participants of the graduate course CAS 703 Software Design, winter term 2005/06, at McMaster University. The course was divided into two parts. In the first part the instructor gave seminars on fundamental topics in software design. For the record, these were:

1. Elements of Programming
2. Modularization
3. Abstract Programs
4. Testing
5. Exceptions
6. Functional Specifications
7. Object-Oriented Programs
8. Object-Oriented Modelling
9. Requirements Analysis
10. Object-Oriented Design
11. Reactive Programs
12. Configuration Management
13. Software Development Process

For the second part, students selected a topic for which they reviewed the literature, gave a presentation, and wrote a paper. This report consists of those papers, in order of presentation. Some of the articles are surveys and some develop new ideas; they are all beyond the material found in textbooks on software design. The topics range from issues in programming languages to programming tools, design principles, pedagogical issues, and managerial issues. All papers are sound starting points for further research.

Emil Sekerinski
April 2006

Chapter 1

Lei Hu: Design Recovery of Specifications

The objective of this paper is to describe how to recover specifications from source code during the design recovery process. We first outlines some basic reverse engineering concepts in §1.1 and §1.2, which are followed by the description of process of design recovery in §1.3. In §1.4 we explore the spectrum of design recovery techniques and conduct a comparison on various aspects. In §1.5 we concentrate on two formal approaches based on formal methods to derive specifications from source code are introduced. Finally, we give the conclusion and future work.

1.1 Introduction

During the last three decades, the emergence of large software systems has created a great impact on modern society. These large software systems, such as e-commerce, banking system and telecom system, have been developed and applied in various domains of our daily life. However, the rapid growth of software systems in both size and complexity has resulted in a dramatic increase of investment in maintenance. Software maintenance has been recognized as one of the most costly phases in the software development. Studies on the effort of the software development show that the portion spent on software maintenance has increased dramatically from 49% in 1977 to more than 90% in 1995 [6]. For example, it was estimated that Nokia Inc. used about \$90 million just for preventive Y2K-bug corrections [14].

The first step in the software maintenance phase is to try to understand the design that already exists. To keep the maintenance phase to be as efficient as possible, we need to obtain appropriate design abstractions to facilitate program maintenance and program understanding. Ideally, these design abstractions should be acquired from software documentation and original software developers. Nevertheless, it is often the case that such documentation is out-of-date or nonexistent and original software developers have moved away. As a result, the maintainers are faced with the task of recovering the intent of the original author without any guidance but the analysis of the source code. These situations demand the techniques

and tools of reverse engineering.

In this survey, we first give a brief introduction to reverse engineering and some related terms in software engineering. Then design recovery is discussed. Finally, we focus on the derivation of formal specifications used in design recovery.

1.2 Reverse Engineering Review

Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create the representations of the system in another form or at higher levels of abstraction [4].

In fact, the term "reverse engineering" stems from the process of the analysis of hardware whose objective is to analyze a competitor's products or to duplicate the system. As Rekoff mentioned in a landmark paper on the topic, he defines reverse engineering as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system". Compared with the hardware systems, in the software system, the goals of reverse engineering is to gain a sufficient design-level understanding to aid maintenance, strengthen enhancement or support replacement and migration [4].

Reverse engineering is required especially when it would take us long time to understand a software system because of the incorrect and antique documentation, the complexity of the system and the insufficient knowledge of the maintainer of the system. By performing reverse engineering, we can recover the lost information, facilitate the migration between platforms, improve or provide documentation, provide alternative views, extract reusable components, cope with complexity, detect side effects and reduce maintenance efforts. Design recovery and redocumentation are the two primary subareas of reverse engineering.

- **Redocumentation** is the creation or revision of a semantically equivalent representation within the same relative abstraction level [4]. The precondition of the redocumentation is the design documentation of the existing subject system. What we do is to use other views to represent the system (for example, dataflow, control flow and data structure). Redocumentation provides an easy way to visualize the structure of the system component.
- **Design recovery** is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observation of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself [4].

We note that reverse engineering, a process of examination, does not change the subject system.

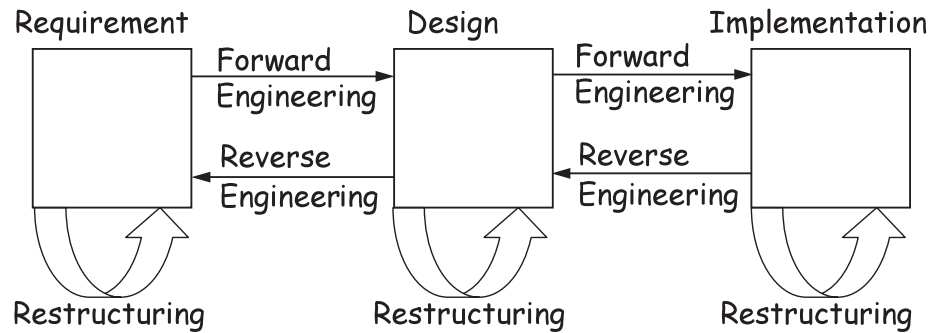


Figure 1.1: Relationship between terms.

Some Related Terminology

There are some confusable terms in the software life cycle. To avoid the confusion and to fully understand the concept of reverse engineering, we give the corresponding definitions of these terms. Figure 1.1 [4] shows these terms and their relations to each other and to three major phases: requirements, design, and implementation in life-cycle model of a software system.

- **Forward Engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system [4]. Here we use the word "forward" to distinguish it from reverse engineering.
- **Reengineering** also called renovation or reclamation, is the examination and alteration of a subject system to reconstruct it in a new form and the subsequent implementation of the new form [4].
- **Restructuring** is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's functionality. Restructuring does not need to understand the semantics of the system. It simply transforms an unstructured code into a structured one.

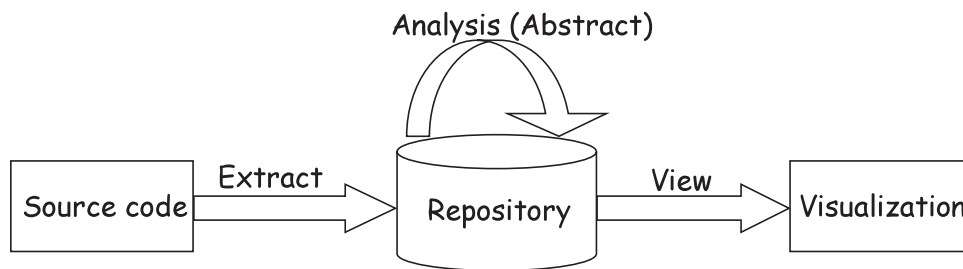


Figure 1.2: Process of Design Recovery

1.3 Foundation of Design Recovery

Being a subset of reverse engineering, design recovery occurs throughout the software life cycle, from the software development to maintenance. To develop new software, we always prefer to spend some time trying to understand the structure of similar mature systems. Likewise, when we perform software maintenance, we always face such kinds of questions like what a program does, how it does it, why it does it, and so forth. To answer these questions, we need to understand the system's structure through design recovery.

Process of Design Recovery

Current research in software reverse engineering mainly focuses on two areas: representing software systems and analyzing software systems. Therefore, we can divide the process of design recovery into extract, analysis (abstract) and view as shown in Figure 1.2.

Source codes are extracted by extractor, and then stored into a software repository. By different analysis techniques, these data are abstracted to provide a deeper understanding of

the software system. Abstractions of the system are then viewed by appropriate visualization ways. The extract \rightarrow analysis (abstract) \rightarrow view process provides a classical framework for the reverse engineering techniques. Reverse engineering tools like Rigi [13], Bookshelf [9] and DALI [15] all follow this process.

Representing Source Code

Source code is often the only accurate and reliable source for us to start the reverse engineering activities. Although source code contains some important information, some design information cannot be directly acquired from the program codes. For example, to describe software architecture which emphasizes to lay out the modular structure and relationship. In contrast, a detailed dataflow and controlflow analysis requires a fine grained representation of each source code object (variables, methods, paragraphs) and their occurrences in program statements. Therefore, we need to find certain appropriate ways to represent the code to facilitate design recovery process.

In reverse engineering, we always choose graphs to represent the software artifact, since graphs own the strong mathematical foundation and abundant efficient algorithmic support. In the following part, we will introduce some famous graph models such as, TA [12], RSF [13], and TGraphs [6], which have been applied in software reverse engineering successfully. We also give a simple example using TGraphs to represent a small program.

1. TA, the Tuple-Attribute language, allows us to store information about certain types of graphs conveniently. This information includes:
 - (a) Nodes and edges in the graph, and
 - (b) Attributes of these nodes and edges.

The information of program, represented by TA, can be interpreted to be a graph data base.

2. RSF, comes from the reverse engineering tool Rigi, created at the University of Victoria. A RSF file consists of a sequence of triples and one triple on a line. A RSF triple can represent an arc between two nodes like arcType, startNodeName and endNodeName. For example, using a domain model that has function and data type nodes interconnected by call arcs, a token-level RSF stream then contains triples like:

arcType	startNodeName	endNodeName
call	main	printf
call	main	listcreate
data	main	FILE
data	listcreate	List

3. TGraphs, are directed graphs, whose nodes and edges may be attributed and typed. Each type can be assigned an individual attribute schema specifying the possible at-

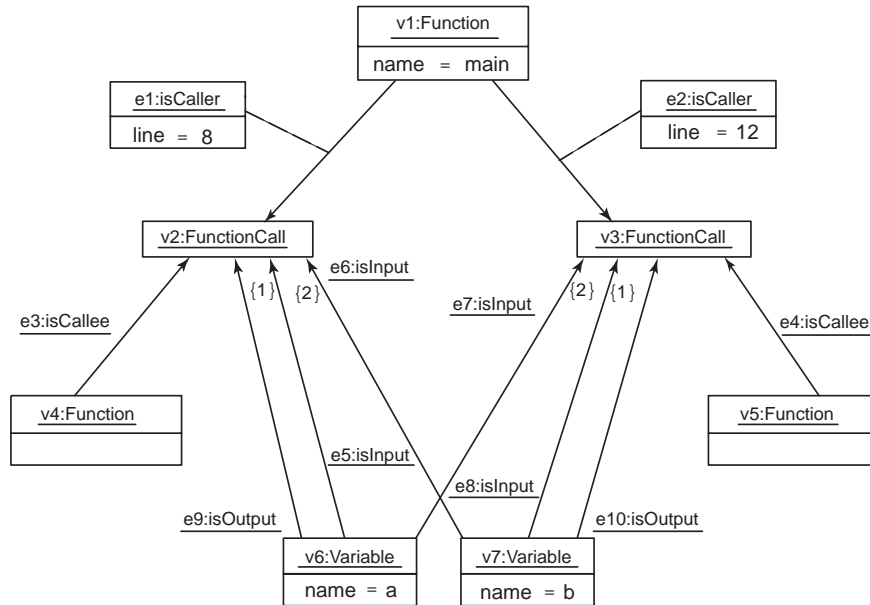


Figure 1.3: A program fragment represented by TGraph [6]

tributes of nodes and edges. Here, we give a simple example to show how to represent a program fragment using TGraphs.

```
int main()
{
    int a;
    int b;
    a=max(a,b);
    b=min(a,b);
}
```

In Figure 1.3, the functions main, max and min are represented by the nodes of type function. These nodes are attributed with the function name. FunctionCall nodes represent the calls of functions max and min. They are associated with the caller by isCaller edges and to the callee by isCallee edges. The isCaller edges are attributed with a line attribute showing the line number which contains the call. Input parameters (represented by variable nodes that are attributed with the variable name) are associated with isInput edges. The ordering of parameter lists is given by ordering the incidences of isInput edges pointing to FunctionCall nodes. The first edge of type isInput incident to function call v2 (modeling the call max(a,b)) comes from node v6 representing variable a. The second edge of type isInput connects to the second parameter b (node v7). The incidences of isInput edges associated with node v3 model

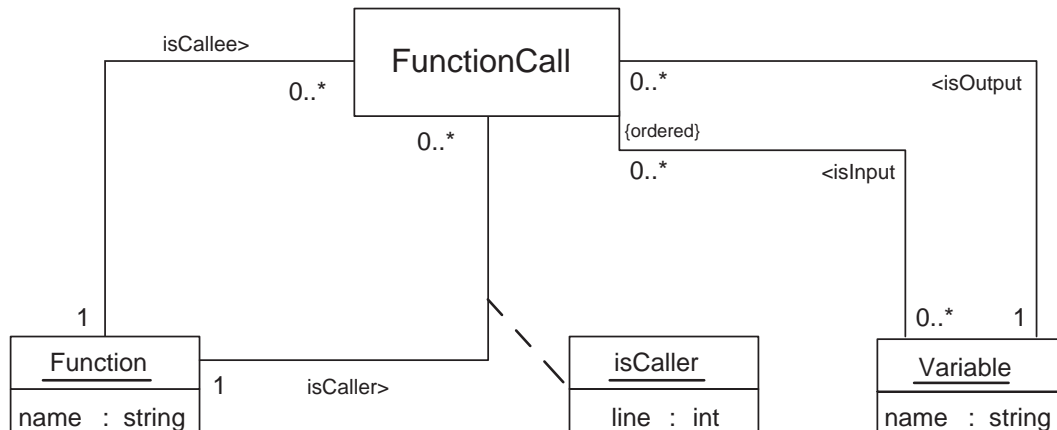


Figure 1.4: Graph class definition

in the reversed parameter order. Output parameters are associated with their function calls by `isOutput` edges.

Describing Graph Classes by UML Class Diagrams

As we have seen above, TGraphs provide a simple structural graph-based means to represent source code. However, for different reverse engineering tasks, we need different TGraph structures. As we mentioned before, if we want to describe software architecture with emphasis on the modular structure and relationship, we need to redefine the meaning of the nodes and edges to represent the software system as an attributed relational graph at a higher-level of abstraction. UML class diagrams offer a convenient declarative language to define a graph class with respect to a given application context.

Figure 1.3 offers a fine grained representation of program structures, focusing on the description of functions, function calls, variables and their interdependencies. Figure 1.4 shows a possible graph class definition of this graphs, depicted as an UML class diagram [6]. Node classes (`FunctionCall`, `function`, and `variable`) are defined by classes. Edge classes (`isCallee`, `isInput`, and `isOutput`) are defined by associations. Attributed edge classes (`isCaller`) are described by association classes. The orientation of edges is depicted by an arrow. Multiplicities denote degree restrictions. Ordering of incidences is indicated by the keyword `ordered`.

Extracting Facts from Source Code

All program understanding and analysis activities in reverse engineering rely on the data stored in the repository. Therefore, extracting accurate and reliable facts from source code then storing them into the repository is an important step in the design recovery process. The extracting step is conducted by an extractor in the design recovery process. An extractor is a

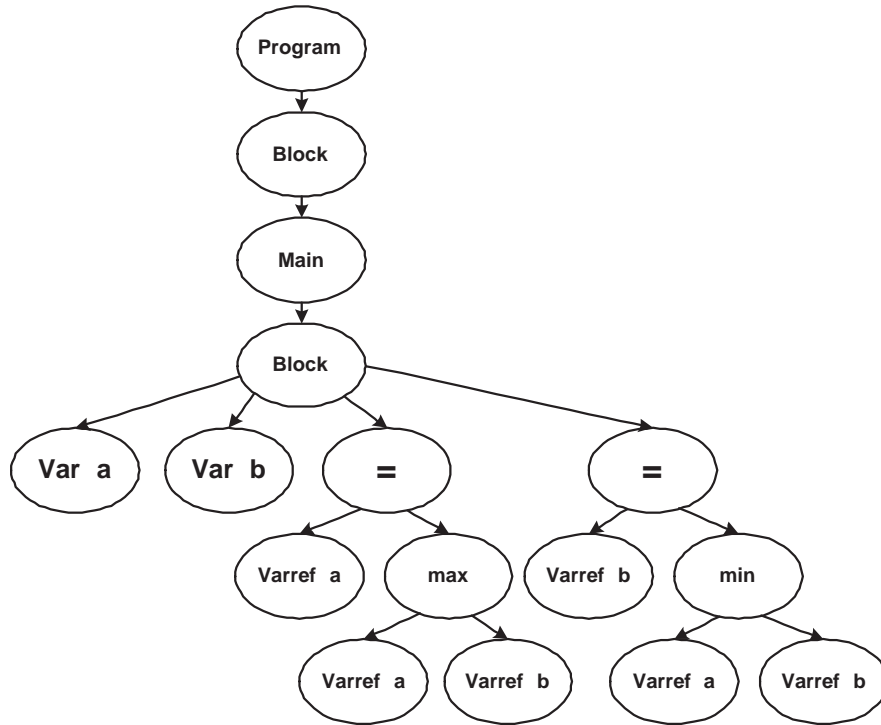


Figure 1.5: Abstract Syntax Tree

program which processes source code and outputs facts about the code in a software exchange format (SEF). CPPX, a C++ Extractor, was developed by the team of Dean, Malton and Ric in 2001. This Open Source tool is based on GNU'S GCC front end and produces information according to the Datrix schema [18]. A schema is derived from a domain model, a description that relates entities in the schema to their real-world counterparts, thus providing a basis for meaningful interpretation of the data [8]. The extracting process mainly include the following steps.

1. Parse the source code to extract abstract syntax tree as shown in Figure 1.5.
2. According the given graph class (schema), extract graph representation of the system by abstract syntax tree traversal.

Analysis

In the analysis process, for different reverse engineering tasks, a variety of techniques are used to recover design abstractions from a combination of source code, existing design documentation (if available) and general knowledge about application domain. The recovered design abstraction must include conventional software engineering representations such as formal

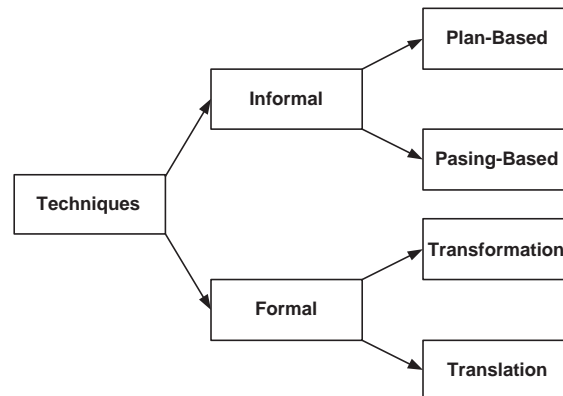


Figure 1.6: Classification of Reverse Engineering Techniques

specifications, module breakdowns, data abstractions, dataflows, and program description language [1]. In the context of design recovery, we can classify the design abstraction into structural abstractions and functional abstractions. A structural abstraction is a description of a software system based on the syntactic properties of a programming language. For example, the abstraction of module is a structural abstraction. In contrast, the functional abstraction is a description of a software system that is based on the semantics of a program. In other words, a functional abstraction describes program behavior. For instance, abstraction of formal specifications is a functional abstraction.

1.4 Classification of Design Recovery Techniques

The analysis techniques in design recovery process can be classified according to the underlying approach used to analyze softwares. The classification of techniques is shown in Figure 1.6.

1. Informal techniques

Informal techniques are those methods that rely on pattern matching and user-driven clustering techniques based on the syntactic structure of code [10]. The informal techniques facilitate the derivation of structural and functional abstractions. The informal techniques can be decomposed into two additional sub-categories: plan-based and parsing-based.

- **Plan-based techniques** rely primarily on using pattern matching to identify plans within source codes. A program plan is a description of a computational unit contained within a program where a computational unit performs some abstract function [10].

- **Parsing-based approach** analyzes program through the properties of the syntactic structure of a programming language [10]. In general, the parsing-based approach constructs a high-level structural abstraction of the source code. This high-level structural abstraction typically includes data flow diagrams and some other graphical representations of the design.

2. Formal techniques

Formal techniques are those techniques that are based on using some type of formal analytical method to derive a specification from source codes [10]. Mathematical logic is the basis of formal techniques. Each step in the reverse engineering process can be formally verified. In the reverse engineering, we always use formal techniques to get the formal specification from source code.

Formal techniques can also be subdivided into two categories: techniques using a knowledge-base or transformation library to derive formal specifications from code, and techniques using derivation or translation to derive formal specifications from code.

- **Transformation** is a method for changing a specification from one form to another while preserving the semantics of the specification. In the context of programs, a program transformation is a means for changing a program from one form to another while preserving the semantics of the program. Each program transformation typically changes a group of programming statements at a once, where the group is determined by the author of the particular transformation.
- **Translation** is also a method for changing a program from one form to another while preserving semantics but at an atomic level of granularity. In the context of program reverse engineering, a translation technique is usually applied to translates a program into an equivalent formal specification according to some simple rules.

The primary difference between transformation and translation is the degree to which high level knowledge about a problem domain or programming language is incorporated into the transformation or translation rules. In the case of transformation, the rules typically involve transforming aggregations of programming statements into simpler, equivalent sequences of statements or concise formal specifications. In many cases, a large library of transformations is required to capture the many different possible code constructions. The translation, in contrast, involves much simpler rules that are based on single atomic statements such as assignments, alternations, and iterations, thus requiring fewer rules.

1.5 Constructing Formal Specifications from Source Code

Definition of Formal Specification

In software engineering, a formal specification is a specification expressed in a language whose vocabulary, syntax and semantics are formally defined. A specification is formal if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring the useful information from the specification (the proof theory) [19].

Applying Formal Specification in Software Development

Developing and analyzing a formal specification can reduce the costs of software development. When a conventional software development process is used, validation costs are about 50 percent of development costs and implementation and design costs are about twice the costs of specification. With formal specification, specification and implementation costs are comparable and system validation costs are significantly reduces [17]. Since requirements problems can be discovered by developing the formal specification, we are able to avoid to rework to correct these problems after the system has been designed.

Motivation of Formal Specification Abstraction in Reverse Engineering

Sometimes we want to re-implement an existing system using latest software technology. To keep the functionality of existing systems, we need to extract specification from the system as the new starting point for system improvement, including redesign, re-architecture and re-development. Several techniques and tools have been developed to perform the formal specification abstraction in reverse engineering, we can classify these techniques into two subareas: using weakest precondition and using strongest postcondition. We discuss these two approaches in the following section.

Using Strongest Postcondition

First we describe an approach to abstract formal specification through the strongest postcondition predicate transformer sp [7], and Hoare triple [11]. Following this, we introduce AUTOSPEC [3], a tool developed to support this approach to derive formal specifications from program code. Before our discussion of the approach, we first give some prime definitions.

- **Precondition** is a condition or predicate that must always be true prior to the execution of some section of code or before an operation in a formal specification.

- **Postcondition** is a condition or predicate that must always be true just after the execution of some section of code or after an operation in a formal specification.
- **Hoare triple** is of the form $Q \{S\} R$, where Q and R are precondition and postcondition respectively.
- **Predicate transformer** is a total function mapping between two predicates on the state space of a program.
- **wp (Weakest Precondition)** the weakest precondition $wp(S, R)$ is the set of all states where the statement S can begin to execute and terminate with postcondition R hold.
- **sp (Strongest Postcondition)** the strongest postcondition $sp(S, Q)$ is a predicate transformer, which is the set of all states in which there exists a computation of S beginning with Q true [7]. In other words, given that Q holds, if S terminates, execution of S results in $sp(S, Q)$ being true.

Primitive Construction

In this section, we describe the abstraction of formal specifications from several primitive program constructs, such as assignment, alternation, sequences and iteration. For every primitive constructs, we first give the semantics of predicate transformer sp . Then we will see how to get the formal specification by using the Hoare triples $Q\{S\}R$ where $R = sp(S, Q)$.

- **Assignment**

Suppose we have an assignment statement $x := E$; where x is a variable, and E is an expression. The definition of sp of an assignment statement is:

$$sp(x := E, Q) \equiv (\exists v. Q_v^x \wedge (x = E_v^x))$$

where Q is the precondition. Using the Hoare triple notation, a formal specification of assignment statement is constructed as:

$$\{Q\} x := E; \{sp(x := E, Q)\}$$

- **Alternation**

Suppose we have an alternation statement like

$$if B_1 \longrightarrow S_1; \dots \parallel B_n \longrightarrow S_n; fi;$$

Then formal specification of alternation is :

$$\begin{aligned} &\{Q\} \\ &if B_1 \longrightarrow S_1; \dots \parallel B_n \longrightarrow S_n; fi; \\ &\{sp(S_1, B_1 \wedge Q) \vee \dots \vee sp(S_n, B_n \wedge Q)\} \end{aligned}$$

- **Sequence**

For a given sequence of statements $S_1; \dots; S_n$, we can find that the postcondition for some statement S_i is the precondition for the subsequent statement S_{i+1} .

$$sp(S_1; S_2, Q) \equiv sp(S_2, sp(S_1, Q))$$

If we have the sequence $S_1; S_2$ with respect to the precondition Q , using a Hoare triple notation, a formal specification of the sequence statement is constructed as:

$$\begin{aligned} &\{Q\} \\ &S_1; \\ &\{sp(S_1, Q)\} \\ &S_2; \\ &\{sp(S_2, sp(S_1, Q))\} \end{aligned}$$

- **Iteration**

Given an iterative form like:

$$do B \longrightarrow S od$$

The loop is executed when B is true. The strongest postcondition semantics for iteration is $sp(DO, Q) = \neg B \wedge (\exists i. i \geq 0 \wedge sp(IF^i, Q))$ [7]

Here the notation IF^i indicates the execution of "*if* $B \longrightarrow S$ *fi*" i times and DO represents the loop statement. Using the Hoare triple notation, a formal specification of the iteration statement is constructed as:

$$\begin{aligned} &\{Q\} \\ &do B \longrightarrow S od \\ &\{\neg B \wedge (\exists i. i \geq 0 \wedge sp(IF^i, Q))\} \end{aligned}$$

The following example shows how construct the formal specification of iteration statement $do i < n \rightarrow i := i + 1 od$.

The unrolled form of the iteration have the following form:

$$if i < n \rightarrow i := i + 1 fi$$

.

.

.

$$if i < n \rightarrow i := i + 1 fi$$

Assume the precondition is $\{(i = start) \wedge (start < n)\}$. By applying the sp semantics for each alternation statement, we get

1. $\{(i = start) \wedge (start < n)\}$
2. $if i < n \rightarrow i := i + 1 fi$

3. $\{sp(i := i + 1, (i < n) \wedge (i = start) \wedge (start < n))\}$
4. \vee
5. $\{sp(skip, (i \geq n) \wedge (i = start) \wedge (start < n))\}$
6. \iff
7. $\{(i = start + 1) \wedge (start < n)\}$
8. $if\ i < n \rightarrow i := i + 1\ fi$
9. $\{sp(i := i + 1, (i < n) \wedge (i = start + 1) \wedge (start < n))\}$
10. \vee
11. $\{sp(skip, (i \geq n) \wedge (i = start + 1) \wedge (start < n))\}$
12. \iff
13. $\{((i = start + 2) \wedge (start + 1 < n))\}$
14. \vee
15. $(i \geq n) \wedge (i = start + 1) \wedge (start < n)\}$
16.
17. $\{((i = start + (n - start - 1)) \wedge (start + (n - start - 1) - 1 < n))\}$
18. \vee
19. $(i \geq n) \wedge (i = start + (n - start - 2)) \wedge (start + (n - start - 2) - 1 < n)\}$
20. \iff
21. $\{((i = n - 1) \wedge (n - 2 < n))\}$
22. $if\ i < n \rightarrow i := i + 1\ fi$
23. $\{sp(i := i + 1, (i < n) \wedge (i = n - 1) \wedge (n - 2 < n))\}$
24. \vee
25. $\{sp(skip, (i \geq n) \wedge (i = n - 1) \wedge (n - 2 < n))\}$
26. \iff
27. $\{i = n\}$

In the construction of specifications of this iteration statement, we need to make a human specifier. In line 17, the inductive assertion that " $i = start + (n - start - 1)$ " is made. In fact, we make this assertion according the loop invariant " $i \leq n$ ". However, comparing with this simple iteration statement, to determine an invariant for a complicate program is a non-trivial thing and the manual application can be prone to error. Therefore, we need to employ some tools to facilitate us to detect the invariant of the program.

Using Daikon to Detect Invariant

Daikon [5] is an invariant detector developed by Program Analysis Group in MIT. By running a program, Daikon observes the values that the program computes, and then reports properties that were true over the observed executions. The output of Daikon is a set of formulas called an operational abstraction which states properties about a program's data structures. The operational abstraction can be written in an assert statement or a formal specification. By applying Daikon in the construction of formal specifications, we can get the loop invariant directly rather by a human specifier.

Driving More Abstract Specification

The specifications that are constructed using the strongest postcondition (sp) are called "as-built" since they are derived from source code, and thus represent behavior based on the

final product rather than the original design. An approach has been defined to derive more abstract specifications from as-built specifications by requiring that the derived abstraction and the as-built specification satisfy a abstraction matching which is defined as follows:

Abstraction Match

Let I be a program with specification i such that the corresponding precondition and postcondition are i_{pre} and i_{post} , respectively. Let l be an axiomatic specification with precondition l_{pre} and postcondition l_{post} . A match is an abstraction match if $i \preceq l$, so that $(l_{pre} \rightarrow i_{pre}) \wedge (i_{post} \rightarrow l_{post})$ [3].

To preserve an abstraction match relation, we can weaken the postcondition of a specification. For example, suppose I is a specification with precondition I_{pre} and postcondition I_{post} . Let I' be a specification with precondition I'_{pre} and postcondition I'_{post} . If $I_{post} \rightarrow I'_{post}$, we can get $I \preceq I'$ by

$$((I'_{pre} \leftrightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \Rightarrow ((I'_{pre} \rightarrow I_{pre}) \wedge (I_{post} \rightarrow I'_{post})) \quad (1.1)$$

Expression (1.1) provides a basic idea to derive a more abstract specification by weakening the postcondition. In Table 1.1 several options are provided to weaken the postcondition.

Operation	I_{post}	I'_{post}
Delete a conjunct	$A \wedge B \wedge C$	$A \wedge C$
Add a conjunct	$A \wedge B$	$(A \wedge B) \vee C$
\wedge to \rightarrow	$A \wedge B$	$A \rightarrow B$
\wedge to \vee	$A \wedge B$	$A \vee B$

Table 1.1: Weakening the postcondition [3]

AUTOSPEC (Semi-automated specification and generation system) is a system using the semantics of the strongest postcondition predicate transformer to construct formal specifications from source code. It was developed by Gannod and Cheng. The high-level design of the AUTOSPEC system is shown in Figure 1.7 [3]. The AUTOSPEC system interacts with three entities: the user, a specification editor called SPECEDIT, and a theorem prover called TPROVER.

The process of execution is as follows: AUTOSPEC system reads a file according to the user's decisions on how a source file analysis should proceed. Then, the system generates formal specifications based on the use of strongest postcondition, and annotates the original source code with those specifications. The user also interacts with the AUTOSPEC indirectly via the use of the SPECEDIT and TPROVER tools.

The main component of the AUTOSPEC system is the analysis, or SP component. The SP component is responsible for constructing formal specifications from programming construct according to the semantic definitions we described in previous part.

1. SPECEDIT: a specification editor with a graphical user interface front-end that supports the user to construct or modify specifications. It allows the user to save the

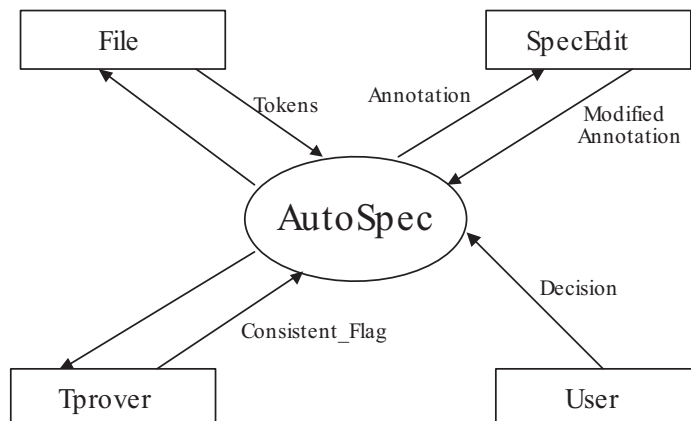


Figure 1.7: Design of AUTOSPEC System

specifications to files for later modification or for incorporation into other tools that use first-order logic as an input language [3]. The SPECEDIT is composed by two components: a parser and a user interface. The user interface facilitates users to construct a syntactically correct specification. The parser is responsible for checking the syntax of the specifications modified by users.

2. TPROVER: a tableau theorem prover that verifies the consistency of specifications modified by a user [3]. In the process of extracting specifications, there exist many interactions between the user and the AUTOSPEC system. The interactions include the modification of a specification by the user and the incorporation of the modified specifications into the current analysis. The TPROVER component can verify whether modified specifications are consistent with the former specifications generated by the SP components. The input of TPROVER is the source file containing a first-order logic specification.

Using Wide Spectrum Language (WSL)

Wide Spectrum Language(WSL)

WSL is a "Wide Spectrum Language" which includes both low-level programming constructs and high-level abstract specifications within a single language [20]. Such a language forms an ideal tool for deriving formal specification from source code in reverse engineering. By using WSL, we are able to use transformations not only for restructuring at the same abstraction level, but also transforming from low-level source code to a high-level abstract specification. A transformation is an operation which maps any program satisfying the applicability conditions of the transformation to an equivalent program [20]. Since all the transformations in WSL have been proved correct, we only need to apply it in reverse engineering process. The proof is based on the semantics of weakest precondition. It was

shown in that two programs $S1$ and $S2$ are equivalent if and only if the corresponding weakest precondition $wp(S1, R)$ and $wp(S2, R)$ are equivalent formulas for any formula R [20]. If we ensure that each step in the reverse engineering process consists of proven transformations, the abstracted specification is guaranteed to be a formal specification of source code. To abstract the specifications from source code using WSL, we usually need to perform the following steps:

First Step: Restructure and Simplify

First, we perform restructuring and simplification transformation to source code. For instance, eliminating expression of switch by representing the complicated control flow as a simple double-nested loop.

Second Step: Using Abstract Data Types

After applying restructuring and simplification transformation to source code, we replace the low-level procedures and data structures with the higher level abstraction using some appropriate abstraction data types like stacks, sequences and random access files [2].

Third Step: Restructure and Simplify again

Since in step 2 we use abstract data types to replace the low-level data structure, we can possibly perform a further simplification and restructuring.

Fourth Step: Abstracting Specification

In the last step, we use higher-level operators based on [16] to describe the functionality of the low-level expression.

FermaT: A tool for Reverse Engineering

FermaT is a program transformation system developed by M.P Ward. The system is implemented in MetaWSL, an extension to WSL and C language. The basic idea of FermaT is program refinement and equivalence [21]. The prototype of FermaT, called "Maintainer's Assistant" was designed to test the idea rather than be a practical tool. Although "Maintainer's Assistant" include a large number of transformations, it is far less efficient than desired. Based on "Maintainer's Assistant", FermaT is constructed by making some improvement described below:

1. By adding domain-specific constructs, WSL is extended to MetaWSL, a language for writing program transformation.
2. By adding an abstract data type, programs can be represented as tree structures.
3. Adding constructs for pattern matching, pattern filling and iterating over components of a program structure.

The system structure is composed by

1. A parser for MetaWSL
2. A interpreter for MetaWSL
3. A translator from MetaWSL to C

4. A small C runtime library
5. A WSL runtime library

Comparison between using WSL and Strongest Postcondition

The two approaches we discussed in §1.5 both belong to the formal techniques in reverse engineering. The main difference between them is that the approach using Strongest Postcondition directly applies the strongest postcondition predicate transformer to the code to construct formal specifications, while the latter takes the weakest precondition predicate transformer as a guideline for constructing formal specifications.

1.6 Conclusion

In this report we have given an overview of reverse engineering and design recovery, and discussed two approaches to abstract formal specifications from existing systems. As we have seen, reverse engineering is a process of analyzing and understanding software system by recovering its design and formal specifications. The recovery of formal specification is an essential stages in a reverse engineering application. Over the past two decades, several techniques and tools for formal specification abstraction have been introduced. These approaches have been proved successful through a number of challenging small case study programs. Our future investigation should focus on automating the process of abstracting formal specifications from source code.

1.7 Problems

1. Compare these two specifications for the statement S , tell which one is more abstract, why?

Specification1:

$$\{P\}$$

$$S$$

$$\{Q_1 \wedge Q_2\}$$

Specification2:

$$\{P\}$$

$$S$$

$$\{\neg Q_1 \vee Q_2\}$$

2. Compare the reverse engineering, reengineering and inverse engineering.

3. Determine the strongest postcondition $\{R\}$ for the following program segment: $\{x = 2\}$
 $y := 3 * x^2 - x; x := (y * x)/5 \{R\}$.

Bibliography

- [1] Ted J. Biggerstaff. Design Recovery for Maintenance and Reuse. *Computer*, 22(7):36–49, July 1989.
- [2] T. Bull. An Introduction to the WSL Program Transformer. *Conference on Software Maintenance*, November 26-29 1990.
- [3] Gerald C. Gannod and Betty H. C. Cheng. A Suite of Tools for Facilitating Reverse Engineering Using Formal Methods. In *Proceedings of the 9th International Workshop on Program Comprehension*, page 221, September 2001.
- [4] Elliot J. Chikofsky and James H. Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
- [5] Daikon. <http://pag.csail.mit.edu/daikon/>. *MIT Program Analysis Group*.
- [6] Jurgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO Generic Understanding of Programs. Technical report, Institute for Software Technology (IST), University of Koblenz-Landau, June 2002.
- [7] CS Scholten EW Dijkstra. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.
- [8] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimothy. Towards a Standard Schema for C/C++. In *Working Conference on Reverse Engineering*, pages 49–58, 2001.
- [9] P. Finnigan, R. Holt I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, 1997.
- [10] Gerald C. Gannod and Betty H. C. Cheng. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Proceedings of the 6th Working Conference on Reverse Engineering*, pages 77–88. IEEE, Oct. 1999.
- [11] C.A.R. Hoare. An Axiomatic Approach for Computer Programming. *CACM*, 12:576–580, 1969.
- [12] JR. C. Holt. An introduction to TA: the Tuple-Attribute Language. Technical report, School of Computer Science, University of Waterloo, March 1997.
- [13] JR. C. Holt. <http://www.rigi.cs.uvic.ca/>. Technical report, Department of Computer Science, University of Victoria, June 2005.

- [14] Jussi Koskinen. Software Maintenance Cost. Technical report, June 2003.
- [15] Kazman R. Tool Support for Architecture Analysis and Design. *Joint Proceedings of the SIGSOFT '96 Workshops (ISAW-2)*, pages 94–97, 1996.
- [16] R.Bird. Lectures on Constructive Functional Programming. *Oxford University, Technical Monograph PRG-69*, September 1988.
- [17] Ian Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2001.
- [18] Ric Holt Thomas R. Dean, Andrew J. Malton. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2-5 2001.
- [19] Axel van Lamsweerde. Formal Specification: a Roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 147–159, Limerick, Ireland, June 2000.
- [20] M. P. Ward. Abstracting a Specification from Code. *J.Software Maintenance: Research and Practice*, 5:101–122, June 1993.
- [21] M. P. Ward. Specifications from Source Code – Alchemists’Dream or Practical Reality? *4th Reengineering Forum*, pages 147–159, September 19-21 1994. Victoria, Canada.

Chapter 2

Jiacong Zhang (Kevin): Theories of Refactoring

Refactoring is a practical and meaningful technique, which is the process to restructure software without changing its original external behavior. In this report, we will discuss the refactoring skills, current refactoring tools and the theories for its correctness.

2.1 Introduction

Every programmer does refactoring no matter they are on purpose or not. After finishing coding a program, they would do some clean up work: moving a method from one class to another more related class; extracting a big chunk code and make them as a new function; even renaming a variable to make it more meaningful. The purpose is to make software easy to understand and extend, meanwhile keep its original behavior. Experienced programmers normally do refactoring by hand. But later automatic refactoring tools appeared to make it more efficiently. At the same time some theories also have been researched to support to prove the correctness of refactoring. In chapter two, some background knowledge about refactoring are introduced; in chapter three, we will discuss something about refactoring tools, including the criteria of refactoring tools and some popular refactorings for different language; two refactoring theories are presented in chapter four.

2.2 Background

What is refactoring?

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Two points need to be emphasized here. First, we should make sure that the purpose to restructure the software is to make the software system more readable to people, more convenient to add new fea-

tures later, and after the refactoring process, the software system may even become better designed than before. Some changes that don't enhance the software's readability, flexibility, and improvability are not refactoring. For example, changing an array sort operation with replacing the insertion algorithm by quick sort algorithm or heap sort algorithm is not refactoring, because the change does not improve any readability of the system. On the contrary, the more complex algorithm may even decrease the readability, although it does improve the performance of the software system. The second point is that the external behavior does not change after refactoring. Users of the software system can not tell any interfacial and functional difference from the original system. It should be the "same" system for them.

Martin Fowler, the conspicuous advocator of software refactoring, gave the definition of refactoring in his book as follow [4]:

Refactoring (noun): a change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

Why do we need refactoring?

Refactoring refines the software design

Normally, software designing is finished before coding and refactoring is done after finishing the implement partial or all the functions of based on the design. So, how refactoring refine the design? So how can refactoring refine the software design? A poorly designed system usually takes more code to do the same thing. When developers want to modify some functions of the system, they have to change a lot of places to implement the modification. However, refactoring can eliminate these duplications and ensure everything is said only once, which is just the essence of a good design.

Refactoring makes code easy to understand, helps to find bugs

Refactoring makes the code easier to be understood for sure. That's just some purpose of doing refactoring. Nowadays, a programmer's code is often read by other developers by the programmer himself to continue development. So, making code easy to understand is very important. If code is easy to understand, everything is clarified very clearly, it is also difficult to miss a bug, because bug is also exposed obviously within the refactored code.

Refactoring helps to speed up programming

People will feel confused why refactoring can speed up programming. Let's imagine if a programmer wanted to add some new features on an existing software system, which was poorly designed. He has to spend lots of time to read and understand the existed code, and also spend lots of time to add new code in several places to do the same things. However, if

he did refactoring on the original code first, he would implement the new function in a very easy way, even in several minutes.

Refactoring and Test

Having a solid test is the essential precondition for doing refactoring. Even an experienced programmer cannot avoid making mistakes and introducing bugs when doing refactoring. So the best way is to do the test after doing any basic refactoring. Martin Fowler said that the correct rhythm of refactoring is "test, small change, test, small change, test..." [4]. The rhythm allows the refactoring to move quickly and safely. With the refactoring rhythm, the test program has to be run very frequently, so an automatic test program for checking the result by itself is necessary. It means that the test only output "ok" or "failure" and exception information. In this case, programmers don't have to spend lots of time in checking the result. It's hard to imagine that the hundreds of output string lines on console are checked one by one.

History of Refactoring

The term "Refactoring" most probably comes from the quote of Peter Deutch in 1989 that "*interface design and functional factoring constitute the key intellectual content of software and are far more difficult to create and recreate than code*" [2]. Based on this, William F. Opdyke and Ralph E. Johnson thought that "*If separating function into objects is factoring, then changing where the function exists must be refactoring*" [10]. In 1992, William F. Opdyke developed the first detailed written work on refactoring in his doctor thesis. Later, Don Roberts and John Brant, who are both Ralph Johnson's students, developed the first automatic refactoring tools "Refactoring Browser" for refactoring Smalltalk programs. In 1999, Martin Fowler wrote the book "*Refactoring: Improving the Design of Existing Code*" [4] to build a comprehensive catalog of refactoring skills. Recently, Tom Mens prompt the idea of "Formalizing Refactoring with Graph Transformations" [8] to prove the refactoring preserving software behavior.

Refactoring skills

An experienced object-oriented program can refactor his code efficiently. He knows where he should extract a method from a big chunk of code; where to extract superclass or create subclass in his hierarchy class design; where to move fields or methods between objects; where to change the inheritance to delegation and so on. That's because he really understand the object-oriented conception and know how to put it into programming practice. Since the article does not focus on the detailed operation for each refactoring, here I just list the outline of these refactorings and give some corresponding examples, which lets you have a visual understanding what refactoring is. Remember that refactoring is not dogmatic. Programmers refactor their code mostly based on their programming experience and understanding

of object-oriented conception.

Simple Example

When we find that subclass uses only part of its superclasses interface or do not want to inherit data from it. Then, we can create a private field for the super class, adjust methods to delegate to the superclass, and remove the subclass. Inheritance is a wonderful thing, but sometimes it isn't exactly what we want. Programmers often start inheriting from a class but then find that many of the superclass operations are really true of the subclass. In this case they have an interface that's not a true reflection of what the class does; or they may find that you are inheriting a whole load of data that is not appropriate for the subclass; or they may find that there are protected superclass methods that don't make much sense with the subclass. In this case, delegation can be used instead and make it clear that programmers are making only partial use of the delegated class. They can control which aspect of the interface to take and which to ignore.

Example

```
public class NodeList : ArrayList{
    public NodeList(){
    public void Add(Node item){
        base.add(item);
    }
    public new Node this[int index]{
        get{
            return (Node) base[index];
        }
    }
}
```

After replaced the inheritance with delegation, we get

```
public class NodeList {
    private ArrayList _nodelist;
    public NodeList(){
        _nodelist = new ArrayList();
    }
    public void Add(Node item){
        _nlist.Add(item);
    }
    public new Node this[int index]{
        get{return (Node) _nlist[index]; }
    }
}
```

2.3 Refactoring Tools

In previous chapter, we described how to manually refactor programs. *Yet manual refactoring really is like cutting down a forest with an axe - we need refactoring tools to take a real chain saw to the problem.* [4]

Why do we need refactoring tools?

An experienced object-oriented programmer can do refactoring for his code after he implements the software functional requirements, by which he makes his code more readable, easy to add new features and well-designed. But, doing factoring by hand is time consuming, because after refactoring, programmers have to check the safety, which means to check if the refactored code preserved the original code behavior. Future more, programmer cannot avoid making some careless mistakes or introducing bugs even with an exact test suite, especially when they are tired. Forgetting to remove a reference of a variable while refactoring just to delete this variable; renaming a method name or signature but not synchronizing a place that calls this method are most typical examples. Even though later, those mistakes were caught by compiler, test or code review, you need spent time fixing it. Yeah, it does waste time. The fact prevents programmers from making refactoring while they know they should. So, we need automatic refactoring tools.

Normally, all the refactoring tools can do the safety checking. A programmer who wants to refactor a program merely needs to ask the tool to check the refactored code. If it is safe, perform the refactoring. Using a refactoring tool can provide many benefits. It can make many simple but tedious checks and flag in advance problems that if left unchecked would cause the program to break as a result of refactoring.

As refactoring becomes less expensive, design mistakes become less costly. Because it is less expensive to fix design mistakes, less design needs to be done up front. Up front design is a predictive activity because the requirements will be incomplete. Because the code is not available, the correct way to design to simplify the code is not available. The correct way to design to simplify the code is no obvious. In the past, we had to live with whatever design we initially created because the cost to change the design was too great. With automatic refactoring tools, we can allow the design to be more fluid because changing it is much less costly.

Criteria for a refactoring tool

Here, let's further discuss some of the criteria that an automatic refactoring tool must have to be successful. Both technical criteria and practical criteria are important.

Technical Criteria [11]

1. **Program Database** One of the first requirements that we recognized was the ability to search for various program entities across the entire program. For example, a program might want to find all calls that can potentially refer to a particular method. Tightly integrated environments such as Smalltalk constantly maintain this database [12]. At anytime, the programmer can perform a search to find cross references. The maintenance of this database is facilitated by the dynamic compilation of the code. As soon as a change is made to any class, it is immediately compiled into bytecodes and the database is updated. In more static environments such as Java, the code is entered into text files. Updates to the database must be performed explicitly. These updates are very similar to the compilation of the Java code itself.
2. **Abstract Syntax Trees (ASTs)** AST is a kind of parse tree, which is a data structure that represents the internal structure of the method itself. We know that most refactorings have to manipulate portions of the system that are below the method level. These are usually references to program elements that are being changed. For example, if an instance variable is renamed (simply a definition change), all references within the methods of that class and its subclasses must be updated. Other refactorings are entirely below the method level, such as extracting a portion of a method into its own, stand-alone method, or assigning a common sub expression to a temporary variable and replacing all occurrences of the expression with the temporary. Any update to a method needs to be able to manipulate the structure of the method. To do this requires AST. Here is the AST example for the following method. see Fig. 2.1

```
public void hello(){
    System.out.println("HelloWorld");
}
```

3. **Accuracy** The refactorings implemented by a tool must reasonably preserve the behavior the programs. Total preservation of behavior is impossible to achieve. For example, what if a refactoring makes a program a few milliseconds faster or slower? The usually would not affect a program, but if the program requirements include hard real-time constraints, this could cause a program to be incorrect. However, refactorings can be made a reasonably accurate of most programs. As long as the cases that will break a refactoring are identified, programmers who use those techniques can either avoid the refactoring or manually fix the parts of the program that the refactoring tool cannot fix.

Practical Criteria [11]

1. **Speed** The analysis and transformations needed to perform refactorings can be time consuming if they are very sophisticated. The relative costs of time and accuracy always must be considered. If a refactoring takes too long, a programmer will never use the automatic refactoring but will just perform it by hand and live with the consequences. Speed should always be considered. Therefore, a few refactorings may not be implemented in refactoring tools because they cannot be implemented safely in a

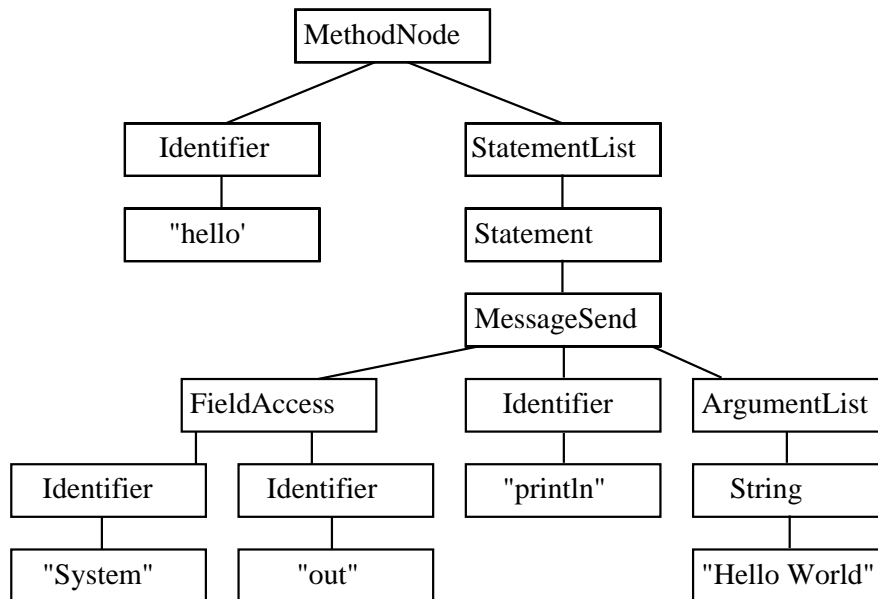


Figure 2.1: Abstract Syntax Tree for hello world.

reasonable amount time. But most refactorings are extremely fast and very accurate. Another approach to consider if an analysis would be too time consuming is to simply ask the programmer to provide the information. This puts the responsibility for accuracy back into the hands of the programmer while still allowing the refactoring to be performed quickly.

2. **Undo** Automatic refactoring allows an exploratory approach to design. You can push the code around and see how it looks under the new design. Because a refactoring is supposed to be behavior preserving, the inverse refactorings, which undoes the original, also is a refactoring and is behavior preserving. Without undo features, it is very difficult to find an old refactored version of program when we do some tentative refactorings. This is annoying and wasting time. With the addition of undo, we can explore with impunity, knowing that we can roll back to any prior version. We can create classes, move methods into them to see how the code will look, and change our minds and go a completely different direction, all very quickly.
3. **Integrated with Tools** In the past decade the integrated development environment (IDE) has been at the core of most development projects. The IDE integrates the editor, compiler, linker, debugger, and any other tool necessary for developing programs. Refactoring tools is also need to be integrated into IDE to make it convenient to do refactoring while you are doing development. Simply having them at your fingertips will make refactoring tools easy to be accepted and popularized. And it is also the reason why most current commercial refactoring tools of different languages are integrated

within the corresponding popular IDEs. We will show this in later section soon.

Language Features and Programming Style Supporting refactoring

Some language features and programming styles support refactoring, compared with those of others. For example, Java is one of such languages. In Java, we know that it has static typing feature, which makes it relatively easy to narrow possible reference to the part of the program that you might want to refactor. And also, let's say if you want to change a method name, since Java has class inheritance and protection-access control mode (public, default, protected, private) features, this makes it easier to determine where the references of the methods are, even if there exist some different methods with the same name. If the method to be renamed is declared private to the class, then references to that function can occur only within the class itself; If the method is declared protected, references can be found only in the class, its subclasses (and their descendents); Even if it is declared as public, the analysis is still limited to the classes listed for "protected" methods and operations on the instances of the class that contain the method, its subclasses, and its descendents. In Smalltalk, such analysis is more difficult. [9]

Several good design principles applied during initial development and throughout the software development process make it easier to do refactoring and evolving software. For example, defining member variables and most methods as private or protected is an abstraction technique that often makes it easier to refactor the internal of a class while minimizing changes made elsewhere in a program. (Recall one of Refactoring problems: Changing Published Interface) Using inheritance to model generalization and specialization hierarchies makes it fairly straightforward to later generalize or specialize the scope of member variables or functions using refactorings to move these members within inheritance hierarchies.

Therefore, refactoring tools is languages sensitive. Here we list some refactoring tools of some languages, whose features support doing refactoring.

Current refactoring tools for different languages [3]

Smalltalk

1. **Smalltalk Refactoring Browser** I mention it first because it is the original refactoring tool and still one of the most full-featured, which is developed by Don Roberts and John Brant. [13] They not only developed the first complete refactoring tool, but also proved the correctness of doing refactoring, which I will go further to discuss in next chapter.

Java

1. **RefactIT** It is the most popular refactoring tool nowadays. And it can restructure Java source-code by its over 30 refactoring operations and wizards; It can also be

integrated within almost all current Java development IDE such as Eclipse, JDevelop, JBuilder, NetBeans etc. It is the most popular refactoring tool nowadays. And it can restructure Java source-code by its over 30 refactoring operations and wizards; It can also be integrated within almost all current Java development IDE such as Eclipse, JDevelop, JBuilder, NetBeans etc.

2. **IntelliJ Idea** This is a fully fledged IDE whose abilities go far beyond refactoring. I think they have succeeded in really moving forward the state of the art for IDEs.
3. **Eclipse** The Java part of Eclipse, JDT, is able to perform several types of automatic refactorings on Java projects, classes, and their members. There are several ways to quickly select a refactoring for an element in a Java project.
4. **JFactor** A plug-in tool - works with JBuilder and Visual Age. Instantiations is a well respected outfit with a long history in Smalltalk and Java VM and compiler technology.

.NET

1. **ReSharper** ReSharper was created to increase the productivity of C# developers. It comes equipped with a rich set of features, such as intelligent coding assistance, on-the-fly error highlighting and quick error correction, unmatched support for code refactoring, and a whole lot more. ReSharper's tight integration with Visual Studio .NET provides quick and easy access to all of its advanced features right from the IDE.
2. **C# Refactory** C# Refactory 2.0 parses large solutions far more rapidly. A new refactoring "move member" is introduced. Extract super class and push up member refactorings have been refined. Toolbar/keyboard customizations are persistent, The screenshots below illustrate improvements to the user interface in version 2.0.
3. **Refactor! Pro** A general .NET refactoring tool that supports both C# and Visual Basic. A VB only version (below) is available as a free download from Microsoft. Works with VS 2005 only.

Most of the above are commercial refactoring tools, except Smalltalk Refactoring Brower and Eclipse. But frankly speaking, Eclipse doesn't have as many refactoring features as other tools such as RefactIT as well as quality and reliability, although it is already very robust.

2.4 Refactoring Theories

Refactoring must preserve software original observable behavior, and refactoring tools have to be fast and reliable. But how can we tell if a refactoring tool is reliable? Is there any theories to proof the tools can safely refactor a program?

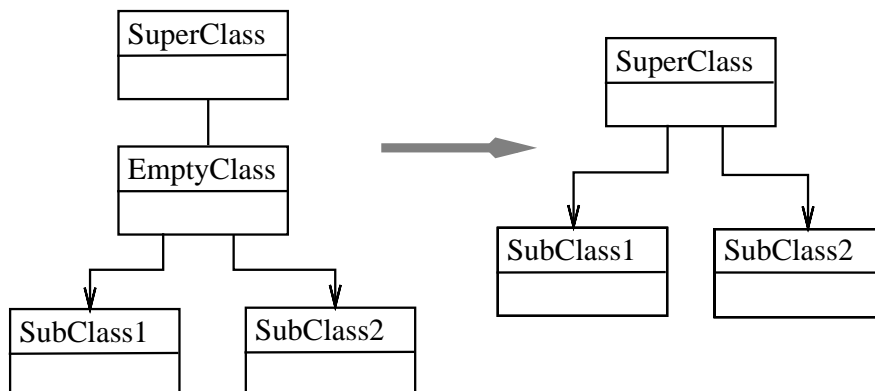


Figure 2.2: Remove Class Refactoring with an Empty Class. [11]

Refactor Correctness Proving Strategy

Refactoring occurs at different levels. In high level, refactoring is represented as big changes of the whole software system, such as changing the major design; in low level, refactoring is represented as many primitive changes. For example, just rename a variable. High-level refactorings can be implemented in terms of several low-level refactorings. So, if we can implement the low-level (primitive) refactorings correctly, then the high-level refactorings will be correct for sure. Then how can we prove the correctness of the low-level refactoring. William Opdyke first tried this strategy as follow. First, identified 23 primitive refactorings, for each primitive refactoring, he defined a set of preconditions that would ensure that the refactoring would preserve behavior.

Here is the example of the precondition of Remove an empty class primitive refactoring. There are two cases that the refactoring might happen. First is that the class is unreferenced and has no subclasses. The second is that the class is unreferenced, has subclasses, but has no methods or instance variables. In this case, the class is removed and all of its subclasses become subclasses of the original class's superclass.(see Fig. 2.2) He arrived at his collection of refactorings by observing several systems and recording the types of refactorings that OO programmers applied. His refactorings were defined in terms of C++, but many of them are applicable to other OO languages. The seven program properties that his refactorings preserved are defined as follow: [9]:

1. **Unique Superclass:** Every class must have exactly one superclass. Even though his research focused on C++, he only considered single inheritance systems.
2. **Distinct Class Names:** Every class in the system must have a unique identifier. Even in the presence of nested scopes or namespaces, this property must be true.
3. **Distinct Member Names:** This property enforces distinct member names within a

single class. The method can still be redefined in either superclasses or subclasses.

4. **Inherited Member Variables Not Redefined:** Classes cannot define variables that are inherited from their superclasses.
5. **Compatible Signatures in Member Function Redefinition:** In C++, it is critical that overriding methods have the same signature as the overridden method.
6. **Type-Safe Assignments:** After a refactoring, the left-hand side of every assignment must be of the type or a subtype of the type of the variable on the right-hand side.
7. **Semantically Equivalent References and Operations:** Semantic equivalence was defined operationally. The program had to produce the same value for a given set of inputs both before and after a refactoring.

Thus he got that the 23 primitive refactorings are correct. Then, he created 3 more complicated refactorings by composing the primitive refactorings. Since each primitive refactoring was behavior preserving, the composition was necessarily behavior preserving.

Use FOPL to Define and Prove Refactoring

Although, the strategy is there, proving the behavior preserving of refactoring is still very difficult, because the analyses that ensuring the property is difficult to computer and prove. So, later Don Roberts, who develop the first refactoring tools (Smalltalk Refactoring Brower), gave an alternative definition of refactoring.

First, as we already know that the precondition of refactoring is the assertions that a program must satisfy for the refactoring to be applied. Then the refactoring can be defined as program transformations that have particular preconditions that must be satisfied before the transformation can be legally performed.

Definition (without postcondition): [11] a refactoring is a pair $R = (pre; T)$ where pre is the precondition that the program must satisfy, and T is the program transformation.

Then Don suggested representing and storing the program with Abstract Typed Trees, which provides the way of expressing transformation on programs as tree-to-tree transformation rules. To ensure that a particular refactoring is legal, the program must meet certain criteria. A program must be analyzed to determine if it meets these criteria. So, he defined some Analysis Functions to describe the relationship between different software entities such as methods, classes and instance variables. Each transformation can be analyzed by Analysis Function and Abstract Syntax Tree function.

Analysis Functions is divided into two categories [11], primitive and derived. Primitive Analysis Functions are to form the basis for the program analysis that ensures valid refactorings.

These are the fundamental functions that a program analysis framework must compute to implement the refactorings.

e.g: Superclass(classname) The immediate superclass of classname

Derived Analysis Functions are useful to describe the preconditions of refactorings. They can be computed from the primitive analysis functions.

e.g: Subclasses(classname) The set of all immediate subclasses of classname.

$$\text{Subclasses}(class) = \{c \mid \text{Superclass}(c) = class\}$$

Then, he specifies refactorings' preconditions with the form of First Order Predicate Logic. By this way, he moved the transformation out of a language which is difficult to reason into a more tractable (First Order predicate calculus). Therefore, all the refactoring can be proved by First Order Predicate Calculus composed of Corresponding Analysis Functions and Abstract Syntax Trees' Function.

Recall that a basic FOL language $L = (F, P)$, where F is the set of function symbols and P is a set of predicate symbols. A model $M = (D, I)$ for language L is a pair where D is domain of discourse and I is a mapping which assign value from within D to the symbols with F and P . [7] Consider each program p induces a model M_p , evaluate the precondition pre of each refactoring, check if the model M_p satisfies the precondition pre . If pre is satisfied, we say that the refactoring is legal. It can be expressed as follow [11]:

A refactoring $R = (pre, T)$ is legal for a program p iff $\models_{M_p} pre$

Formalizing Refactorings with Graph Transformations

In view of the wide acceptance of graph-like representations in today's refactoring tools, it seems natural to use graph rewriting as the basis for the desired formal model. Indeed, most refactoring tools represent the source code by means of an abstract syntax tree augmented with extra links to represent frequently used relations (e.g., inheritance, method invocation and variable access). Since a refactoring is supposed to change that graph, a formal specification for a refactoring would quite naturally correspond with a number of graph rewriting rules regardless of programming language [14]. Tom Mens explored the use of graph rewriting for specifying refactorings and their effect on programs. He introduced a graph representation for programs and demonstrated that it is possible to prove that refactorings preserve certain program properties, and that graph rewriting is a suitable formalism for such proofs.

He validated this approach as following steps as follow:

1. Converting source code into a graph
2. Getting a formal graph representation of an abstract syntax tree augmented with variable access and method invocation relations (program graph)

3. Getting the graph of refactoring by graph production
4. Getting the graph of precondition of refactoring by graph production by using graph rewriting with negative application conditions [5] [6]
5. Verifying the preconditions and invariants in the graph representation by showing that a rewrite rule applied on a graph satisfying the necessary precondition actually preserves the necessary properties. In particular the properties of access, update and call preservation are considered

The main idea of the approach is to use graph formal representation to specify an abstract syntax tree augmented with the relation between different methods of a program, use graph rewriting rules to represent the program transformation (refactoring), and use graph rewriting with negative application condition to express precondition of refactoring. Due to it is not the focus of this report, I just give an example on the definition of programming graph to give you the idea how can software be represented as graph form.

Programming Graph Definition: [8]

Let Σ be a set of node labels and Δ a set of edge labels. A (labeled) graph over Σ and Λ is a 3-tuple $G = (VG, EG, nlabG)$, where VG is the set of nodes; $nlabG : VG \rightarrow \Sigma$ is the node labeling function and $EG \subseteq VG \times \Delta \times VG$ is the set of edges.

Thus programs are represented by typed, labeled, directed graphs. They are called program graphs. In a program graph, software entities (such as classes, variables, methods and method parameters) are represented by nodes whose label is a pair consisting of a name and a node type.

Example: LAN Simulation [1]

```
public class Node{
    public String name;
    public Node nextNode;
    public void accept(Packet p){
        this.send(p); }
    protected void send(Packet p){
        System.out.println(name + nextNode.name);
        this.nextNode.accept(p); }
}
public class Packet{
    public String contents;
    public Node originator;
    public Node addressee;
}
public class PrintServer extends Node{
```

```

public void print(Packet p){
    System.out.println(p.contents); }
public void accept(Packet p){
    if(p.addressee == this) this.print(p);
    else super.accept(p); }
}
public class Workstation extends Node{
    public void originate(Packet p){
        p.originator = this;
        this.send(p); }
    public void accept(Packet p){
        if(p.originator == this) System.err.println("no destination");
        else super.accept(p); }
}

```

Fig. 2.3 [8] shows the Node type set $\Sigma = \{C, M, MD, V, VD, P, E\}$, and the Edge type set $\Delta = \{l, i, m, t, p, e, c, a, u\}$, which form the graph of the above program.

Fig. 2.4 [8] shows the graph representation of the LAN simulation example, and Fig. 2.5 [8] give the method definition of Node class.

2.5 Concluding Remarks

1. Software refactoring should become a necessary part of software development process, just like designing, coding, debugging and testing.
2. Refactoring techniques depends on the Object-Oriented conception. Object-Oriented programming languages support the design of code reuse.
3. Current refactoring tools almost can do any refactorings to restructure software and refine the software design, but not one hundred percent. Programmers still have to do some refactoring by hand.
4. Directly proving the correctness of refactoring is difficult. We need to abstract refactorings of certain language and represent it with a formal specification, which is easy to analyze and prove. (FOPL calculus or graph)

2.6 Exam Question

1. Refactor the following code by decomposing conditional


```

if(data.before(SUMMER_START)||data.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceChange;

```

node type	description	examples
C	C lass	<i>Node, Workstation, PrintServer, Packet</i>
M	M ethod signature	<i>accept, send, print</i>
MD	M ethod D efinition	<i>System.out.println(p.contents)</i>
V	V ariable	<i>name, nextNode, contents, originator</i>
VD	V ariable D efinition	<i>public Node nextNode</i>
P	P arameter of a method definition	<i>p</i>
E	(sub) E xpression in method definition	<i>p.contents</i>
Edge type	description	examples
$l : M \rightarrow MD$	dynamic method lookup	<i>accept(Packet p) has 3 possible method definitions</i>
$V \rightarrow VD$	variable lookup	<i>::</i>
$i : C \rightarrow C$	inheritance	<i>class PrintServer extends Node</i>
$m : VD \rightarrow C$	variable membership	<i>variable name is defined in Node</i>
$MD \rightarrow C$	method membership	<i>method send is defined in Node</i>
$t : V \rightarrow C$	variable t ype	<i>String name</i>
$M \rightarrow C$	method return type	<i>String getName()</i>
$p : MD \rightarrow P$	p arameter definition	<i>send(Packet p)</i>
$P \rightarrow C$	parameter type	<i>send(Packet p)</i>
$e : MD \rightarrow E$	e xpression in method definition	<i>System.out.println(p.contents)</i>
$E \rightarrow E$	subexpression in method definition	<i>p.contents</i>
$c : E \rightarrow M$	(dynamic) method c all	<i>this.send(p)</i>
$a : E \rightarrow \{V \mid P\}$	variable or parameter a ccess	<i>p.contents</i>
$u : E \rightarrow \{V \mid P\}$	variable or parameter u date	<i>p.originator = this</i>

Figure 2.3: Node type set $\Sigma = \{C, M, MD, V, VD, P, E\}$ and Edge type set $\Delta = \{l, i, m, t, p, e, c, a, u\}$

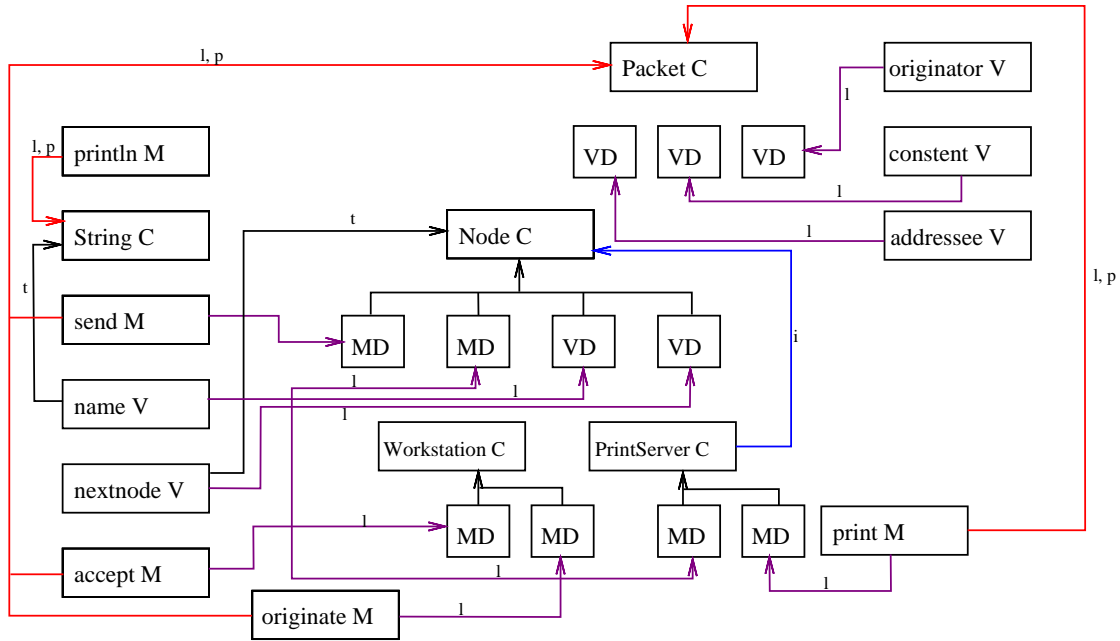


Figure 2.4: Static structure of LAN simulation.

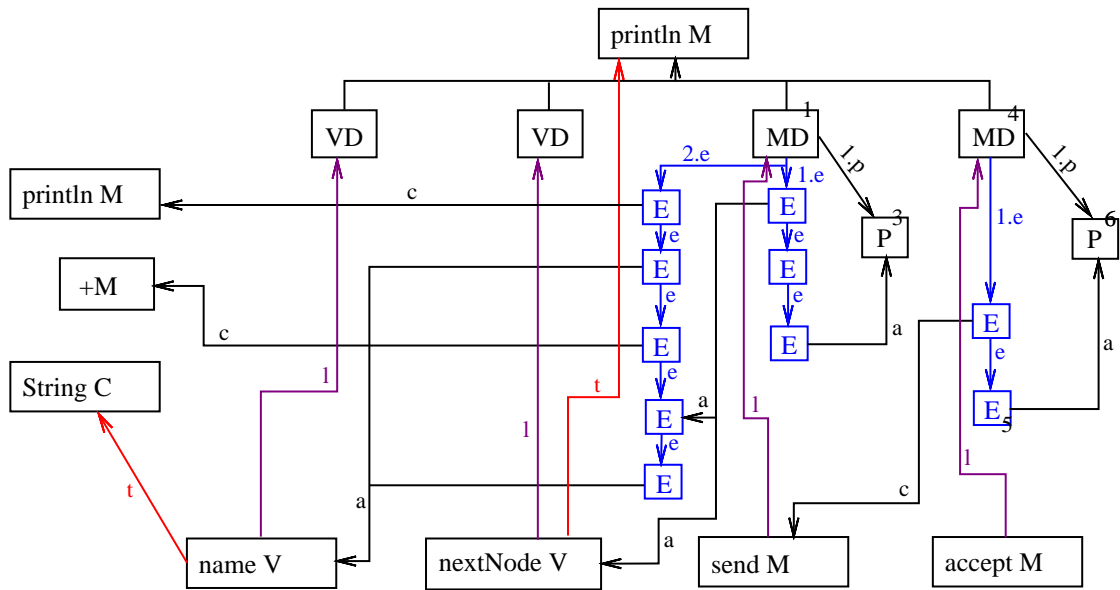


Figure 2.5: Method definitions in class Node

else

*charge = quantity * _summerRate;*

2. Explain the difference and relation between refactoring and design pattern.
3. Give the precondition of the primitive refactoring "create_empty_class".

Bibliography

- [1] Serge Demeyer, Dirk Janssens, and Tom Mens. Simulation of a lan. *Electronic Notes in Theoretical Computer Science*, 2002.
- [2] Peter Deutsch. *Software Reusability*, volume Application and Experience, chapter Design Reuse and frameworks in the Smalltalk-80 System. LOOKUP, 1989.
- [3] Martin Fowler. <http://www.refactoring.com>.
- [4] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
- [5] Annegret Habel, Reiko Heckle, and Gabriele Taentzer. Graph grammars with negative application conditions. Technical report, Technical University of Berlin, 1996.
- [6] Reiko Heckle. Algebraic graph transformations with application conditions. Master's thesis, TU Berlin, 1995.
- [7] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. "John Wiley & Sons", 1987.
- [8] Tom Mens. Formalizing refactorings with graph transformations. *Software Maintenance and Evolution*, 2004.
- [9] William Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
- [10] William Opdyke and Donald Roberts, editors. *Refactoring. Tutorial presented at OOP-SLA '95: 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Austin, Texas, 1995.
- [11] Donald Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.
- [12] Donald Roberts, John Brant, and Ralph Johnson. Theory and practice of object systems. Technical report, University of Illinois at Urbana-Champaign, 1997.
- [13] Donald Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Phys. Rev. Lett.*, 1999.

- [14] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

Chapter 3

Saba Aamir: Memory Management Strategies In Programming Languages

Memory management is a complex field of computer science and there are many techniques to make it more efficient. As software grows to be more complex, memory management has become a marathon task. Memory management involves the memory needed for a program's objects and data structures from the limited resources available, and re-cycling that memory for re-use when it is no longer needed.

Many languages manage memory by explicitly allocating and deallocating data on the heap. Examples of these types of languages include Pascal, C and C++. Some logical, functional and most object oriented languages (with the exception of C++) use garbage collection to manage the heap automatically. Examples include Smalltalk, Eiffel, Java, Oberon and some other languages. Some other languages e.g Modula-3 offer both explicitly and automatically managed heaps [17]. This paper discusses different techniques of memory management.

3.1 Introduction

Memory management is the process of coordinating and controlling the use of memory in a computer system or program. It can be divided into three areas:

- *Hardware Memory Management* deals with the electronic devices that actually store data. This includes RAM, memory caches, memory management unit (MMU) etc.. The MMU is a hardware device responsible for handling memory accesses requested by the main processor. This typically involves translation of virtual addresses to physical addresses, cache control, bus arbitration, memory protection, and the generation of various exceptions. Not all processors have an MMU.
- *Operating system memory management* is concerned with virtual memory and its protection. In the operating system, memory must be allocated to user programs, and

reused by other programs when it is no longer needed. Sometimes the operating system has to pretend that the computer has more memory than it actually does, and also that each program has the computers memory to itself. Both of these are made possible by using virtual memory. Many operating systems support protection of memory pages. Memory pages are fixed blocks of virtual memory used for transfer of data between virtual memory and disk. Individual pages may be protected against a combination of read, write or execute accesses by a process. A process which attempts a protected access will trigger a protection fault. Pages can be protected for a number of reasons: an operating system may want to protect pages for security, or to implement “copy-on-write” or “demand-zero-filled” pages.

- *Application memory management* involves obtaining memory from the operating system, and managing its use by an application program. Application programs have dynamically changing storage requirements. The application memory manager must cope with this while minimizing the total CPU overhead, interactive pause times, and the total memory used. It is a complex task to manage application memory such that the application can run most efficiently, as the operating system may create the illusion of nearly infinite memory. Ideally, these problems should be solved by tried and tested tools, tuned to a specific application.

In this research paper we are dealing with the third type i.e “Application Memory Management” unless specified otherwise.

3.2 Explicit Memory Management Strategies

Manual memory management is where the programmer has direct control over when memory may be recycled. Usually this is either by explicit calls to heap management functions like `malloc()` and `free()`, or by language constructs that affect the stack (such as local variables). The key feature of a manual memory manager is that it provides a way for the program to say, “Have this memory back; I’ve finished with it”; the memory manager does not recycle any memory without such an instruction.

The `malloc()` function allocates unused space of indeterminate value whose size is equal to the size of an object, which it takes as an argument. A pointer is returned if the allocation succeeds which points to the start (lowest byte address) of the allocated space. If the size of the space requested is zero, the behaviour is implementation-dependent; the value returned will be either a null pointer or a unique pointer. If the space cannot be allocated, either a null pointer is returned or an exception is raised depending upon the implementation.

The `free()` function releases an allocated block of memory. It takes a pointer that has been returned by `malloc()` as an argument and has no return type.

Strengths And Weaknesses Of Explicit Memory Management. The most important strength of manual memory management is that developers have complete control on

the allocation and de-allocation of memory. This way developers can decide exactly before which statement memory needs to be allocated and exactly at which statement and under which pre-conditions the memory will be de-allocated.

Manual memory management also has some problems associated with it, which follow.

- *Premature free or dangling pointer.* Many programs give up memory, but attempt to access it later and crash or behave randomly. This condition is known as premature free, and the surviving reference to the memory is known as a dangling pointer. This is usually confined to manual memory management.
- *Memory leak.* Some programs continually allocate memory without ever giving it up and eventually run out of memory. This condition is known as a memory leak.
- *Poor locality of reference.* Another problem with the layout of allocated blocks comes from the way that modern hardware and operating system memory managers handle memory: successive memory accesses are faster if they are to nearby memory locations. If the memory manager places far apart the blocks a program will use together, then this will cause performance problems. This condition is known as poor locality of reference.
- *Inflexible design.* Memory managers can also cause severe performance problems if they have been designed with one use in mind, but are used in a different way [7]. These problems occur because any memory management solution tends to make assumptions about the way in which the program is going to use memory, such as typical block sizes, reference patterns, or lifetimes of objects. If these assumptions are wrong, then the memory manager may spend a lot more time doing bookkeeping work to keep up with what's happening.

3.3 Semi-Automatic Memory Management Strategies

Reference Counting

Reference counting is a semi-automated memory-management technique which keeps track of the number of active references to an object. Many languages and applications have adopted algorithms based on reference counting, for example, early versions of the Smalltalk object-oriented language, InterLisp, Modula-2+ and the Adobe Photoshop program. Many operating systems also use this method, such as Unix, to determine whether a file may be deleted from the file-store. It is suitable for real-time programming and is used in distributed systems where tracing all pointers, as in the case of garbage collection, is impractical.

Algorithm for Reference Counting. In reference counting, each cell has an additional field, the reference count. The storage manager must maintain the invariant that the reference count of each cell is equal to the number of pointers to that cell from roots or heap

cells [17]. The initializing phase of the algorithm is to place all cells in a pool of free cells, which is usually implemented as a linked list along with a `free_list` pointer to the head of the chain.

The reference count of free cells is zero. On the allocation of a new cell from the pool, its reference count is set to one which is incremented each time a pointer is set to refer to this cell. The elimination of a reference to the cell decrements the count by one; Fig. 3.1 [17]. If this causes the reference count to drop to zero, the reference counting invariant implies that there are no remaining pointers to this cell so it can be returned to the list of free cells.

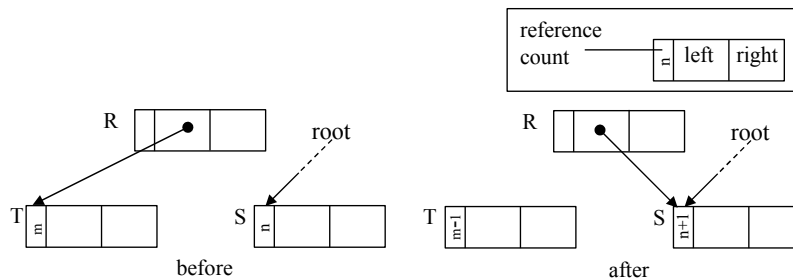


Figure 3.1: `Update(left(R), S)`.

```

allocate()
  newcell := free_list
  free_list := next(free_list)
  return newcell

New()
  if free_list = nil
    abort "Memory exhausted"
  newcell := allocate()
  RC(newcell) := 1
  return newcell

```

Every time on a modification of a pointer reference, the compiler must generate code to update the referenced object's reference count. `Update` overwrites the word in the heap that is its first argument, `R`, with its second argument, `S`, which is assumed to be a pointer, increments the reference count of `S` to take account of this new reference and removes the original pointer from `R` to its target, `*R`, so the reference count of `*R` must be decremented too. Suppose the pointer at `R` originally referred to node `T`. If this pointer was the last reference to `T`, `delete` can return `T` to the free list after recursively deleting any pointers from `T`.

```
free(N)
  next(N) := free_list
  free_list := N

delete(T)
  RC(T) := RC(T) - 1
  if RC(T) = 0
    for U in Children(T)
      delete(*U)
  free(T)

Update(R, S)
  delete(*R)
  RC(S) := RC(S) + 1
  *R := S
```

Strengths and Weaknesses of Reference Counting. The strength of the reference counting method is its incremental nature. Memory management overheads are distributed throughout the computation. Management of active and garbage cells is interleaved with the execution of the user program whereas in non-incremental schemes like mark-sweep, the execution of a program is suspended to perform garbage collection. This makes reference counting a suitable method in a highly interactive or a real-time system [9].

Due to the fact that few cells are shared and many are short-lived, (suggested by empirical studies of a wide range of languages like Lisp [10], Cedar, Standard ML [3], and C and C++ [5]) the standard reference counting method reclaims these cells as soon as they are discarded, in a stack-like manner, whereas under a tracing scheme the garbage collector is invoked reclaiming the memory only when the heap is exhausted. Immediate reuse of cells generates fewer page faults in a virtual memory system, and possibly better cache behaviour [21].

Another benefit of reference counting over garbage collection schemes is that its spatial locality of reference is likely to be no worse than that of its client program. A cell whose reference count becomes zero can be reclaimed without access to cells in other pages of the heap. This contrasts with tracing algorithms which typically need to visit all live cells before reclaiming dead ones [11].

The most serious disadvantage of reference counting is the high processing cost paid to update counters to maintain the reference count invariant. It requires significant assistance from the compiler and imposes overhead on the mutator. The term mutator is used for the user program from the perspective of the garbage collector [13]. In contrast, pointer updates have no memory management overhead under a simple tracing regime.

Another major drawback of simple reference counting algorithms is that they fail to reclaim cyclic structures. Examples of cycles include doubly-linked lists, and 'trees' in which

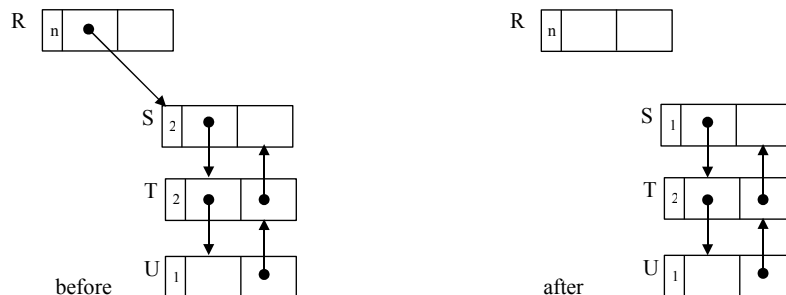


Figure 3.2: Reference counting cyclic data structures: after delete ($\text{right}(R)$) the cycle STU is neither reachable nor reclaimable.

leaf nodes contain a pointer back to the root node. Many implementations of lazy functional languages based on graph reduction handle recursion by using cycles.

Consider the structure in Fig. 3.2 [17]. Deleting the pointer $\text{right}(R)$ causes the reference count of S to decrement but it remains non-zero causing the control to return to the user program without reclaiming S , T and U . This causes memory leaks.

Memory Pools

Memory pools are also a semi-automate memory management technique. They are suitable for programs that go through specific stages, each of which has memory that is allocated for only specific stages of processing. For example, many network server processes allocate memory whose maximum lifespan is the life of the current connection, i.e., per-connection memory allocation. At the end of the stage, the entire memory pool is freed at once.

In pooled memory management, each allocation specifies a pool of memory from which it should be allocated. Each pool has a different lifespan. The world's most popular web server Apache uses memory pools.

There are two well known pools that can be used in a program. They are GNU libc's obstacks implementation which are included by default in GNU-based Linux distributions or Apache's Apache Portable Runtime which has a lot of other utilities to handle all aspects of writing multiplatform server software [6].

Strengths and Weaknesses of Memory Pools. The advantages of memory pools are that it is simple to manage memory for the application. There are standard implementations that are very easy to use. Memory allocation and deallocation is much faster, because it is all done a pool at a time. Allocation can be done in $O(1)$ time, and pool release is actually $O(n)$ time, but divided by a huge factor that makes it $O(1)$ in most cases [6]. For the safe recovery of program in case regular memory is exhausted, error-handling pools can be pre-allocated.

The disadvantages are that memory pools are only useful for programs that operate in stages. They often do not work well with third-party libraries. The pools may need to be modified with the changes of program structure, which may lead to a redesign of the

memory management system. The selection of a pool should be done with care since any wrong allocations can be hard to catch.

Smart Pointers

Smart pointers are pointers which are wrapped by classes to form pointer-like objects [21]. C++ does not offer any safe-guards to the programmers using raw pointers for memory management which may lead to memory leaks and heap corruption. It does not have a garbage collection mechanism as part of the language like other object-oriented languages have. Thus the approach to use smart pointers is adopted.

The C++ standard library provides a smart pointer called `auto_ptr` [2] as a template class. The constructor of `auto_ptr` allocates memory for whatever type is being pointed to and the destructor deletes whatever is pointed to whenever `auto_ptr` leaves its scope. Making use of `auto_ptr` objects instead of raw pointers ensures that memory leaks will not occur because as soon as `auto_ptr` objects go out of scope, their destructors will activate and deallocate the memory. This is particularly important in the context of exceptions.

Smart pointer owns the object it points to. They offer ownership management in different ways [2]. Some smart pointers can transfer ownership automatically by assigning one smart pointer to the other. In doing so the source smart pointer becomes null and the destination holds the ownership of the object. This behaviour is depicted by `auto_ptr`.

A number of additional smart pointers are available in the Boost library including `scoped_ptr`, `shared_ptr` and `weak_ptr`.

- `Scoped_ptr` is used for pointers local to a function or class. Unlike `auto_ptr`, it is non-copyable and therefore enforces sole-ownership.
- `shared_ptr` is a reference counted pointer which allows multiple ownership of the object pointed to. When the last `shared_ptr` goes out of scope, the object is deleted.
- `weak_ptr` is a non-owning observer of an object it points to. `weak_ptr`s do not affect the reference count when constructed or destructed. When the last `shared_ptr` goes out of scope then all `weak_ptr`s observing that `shared_ptr` are reset to null.

3.4 Automatic Memory Management Strategies

Automatic memory management is a service, either as a part of the language or as an extension, that automatically recycles memory that a program would not otherwise use again. Automatic memory managers (often known as garbage collectors, or simply collectors) usually do their job by recycling blocks that are unreachable from the program variables (that is, blocks that cannot be reached by following pointers). Garbage collection was invented by John McCarthy around 1959 to solve the problems of manual memory management in Lisp programming language.

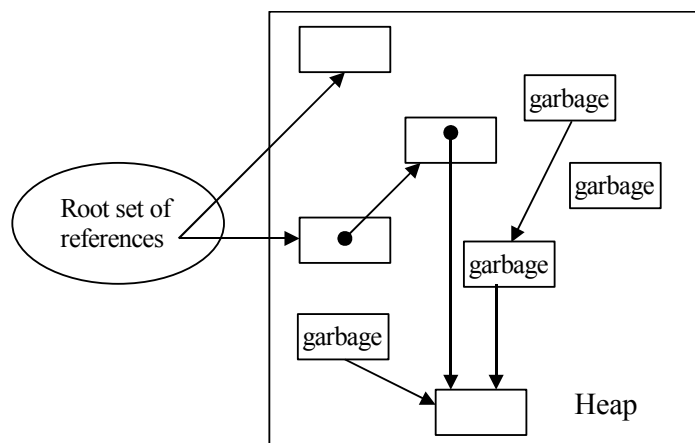


Figure 3.3: Reachable and unreachable objects.

Garbage collectors are usually run when the available memory drops below a specific threshold. Generally, they start off with a “base” set of data that is known to be available to the program which consists of stack data, global variables, and registers. They then try to trace through every piece of data linked through those. Everything the collector finds is good data; everything that it doesn’t find is garbage and can be destroyed and reused (see Fig. 3.3). To manage memory effectively, many types of garbage collectors require knowledge of the layout of pointers within data structures, and therefore have to be a part of the language itself.

Automatic memory management is often more efficient and allows the programmer to work on the actual problem. The module interfaces are cleaner. There are fewer memory management bugs as compared to explicit memory management. The drawback of using automatic memory management is that memory may be retained because it is reachable, but won’t be used again.

There are several basic strategies for garbage collection like mark-sweep, mark-compact and copying. There are also some incremental collectors that result in shorter collection pauses since the entire heap need not be collected at once and some concurrent collectors that can run while the user program runs [1]. Others must perform an entire collection at once while the user program is suspended (also called stop-the-world collectors). Hybrid collectors are also available, such as the generational collector employed by the 1.2 and later JDKs, which use different collection algorithms on different areas of the heap.

Regardless of the algorithm chosen, trends in hardware and software have made garbage collection far more practical. Empirical studies from the 1970s and 1980s show garbage collection consuming between 25 percent and 40 percent of the runtime in large Lisp programs. While garbage collection may not yet be totally invisible, it sure has come a long way [13].

Most modern languages use mainly automatic memory management: Basic, Dylan, Erlang, Haskell, Java, JavaScript, Lisp, ML, Modula-3, Perl, PostScript, Prolog, Python, Scheme, Smalltalk, etc.

The Mark-Sweep Garbage Collection

The mark-sweep or mark-scan algorithm was the first garbage collection algorithm to be developed that is able to reclaim cyclic data structures. Variations of the mark-and-sweep algorithm continue to be among the most commonly used garbage collection techniques.

When using mark-and-sweep, unreferenced cells are not reclaimed immediately. Instead, garbage is allowed to accumulate until all available memory has been exhausted. If a request is then made for new cell, the execution of the program is suspended temporarily while the mark-and-sweep algorithm collects all the garbage from the heap and return it to the pool of free cells. Once all unreferenced cells have been reclaimed, the normal execution of the program can resume. A call to `New` returns a pointer to a new cell from the `free_pool`.

```
New()
  if free_pool is empty
    mark_sweep()
  newcell := allocate()
  return newcell
```

Algorithm for Mark-Sweep Garbage Collection. The mark-sweep algorithm is called a tracing garbage collector because it traces out all live objects to determine which cells are available for reclamation. It consists of two phases.

```
mark_sweep()
  for R in Roots
    mark(R)
  sweep()
  if free_pool is empty
    abort "Memory exhausted"
```

In the first phase, the *mark phase*, all cells reachable from root are marked. Each cell contains an extra bit, the mark-bit, to be used by the garbage collector to record the liveness of the cell. As mark traverses all cells reachable from the roots, the mark-bit is set in each cell visited.

```
mark(N)
  if mark_bit(N) = unmarked
    mark_bit(N) := marked
  for M in Children(N)
    mark(*M)
```

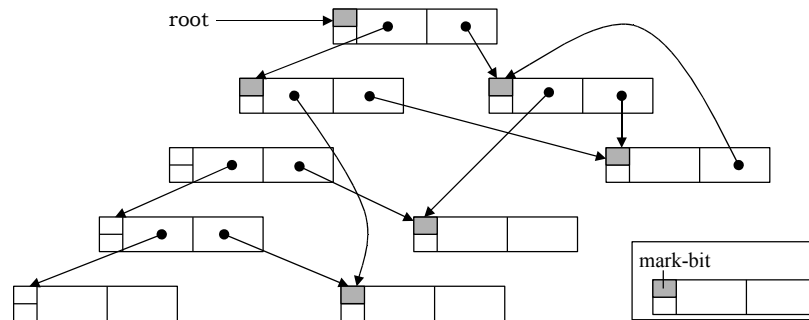


Figure 3.4: The graph after the marking phase. All unmarked cells (with unshaded mark-bits) are garbage.

Termination of the marking phase is enforced by not tracing from cells that have already been marked. When the marking phase has completed, all cells reachable from root will have had their mark-bits set. An example of marking is shown in Fig. 3.4 in which cells that have been marked are indicated by shading their mark-bits. Any cell that is left unmarked could not be reached from root, and hence must be garbage.

In the second phase, the *sweep phase*, the entire heap is scanned and all the unmarked cells are reclaimed. At the same time, the mark-bits of active cells is cleared in preparation for the next invocation of the collector. If the sweep phase fails to recover sufficiently many free cells, the heap must be expanded or the computation aborted. In the algorithm, free simply returns its argument to the free pool for recycling.

```
sweep ()
  N := Heap_bottom
  while N < Heap_top
    if mark_bit(N) = unmarked
      free(N)
    else
      mark_bit(N) := unmarked
      N := N + size(N)
```

Strengths and Weaknesses of Mark-Sweep. Mark-sweep is simple to implement, can reclaim cyclic structures easily, and doesn't place any burden on the compiler or mutator like reference counting does. These advantages of mark-sweep over reference counting have led to its adoption by some systems [17].

One of the disadvantages of mark-sweep is that collection pauses can be long, and the entire heap is visited in the sweep phase, which can have very negative performance consequences on virtual memory systems where the heap may be paged [4]. Non-interruptible, globally traversing mark-sweep algorithms are not practical for safety-critical, real-time, highly interactive or distributed systems.

The major drawback of mark-sweep is that every active (that is, allocated) cell, whether live or not, is visited during the sweep phase. A significant percentage of objects are likely to be garbage (objects that are active but not live), which means that the collector is spending considerable effort examining and handling garbage [13]. Thus the asymptotic complexity of this algorithm is proportional to the size of the entire heap including both live and garbage objects [20].

Mark-sweep collectors also tend to leave the heap fragmented. In a real memory system the effect on performance may not be great although benefits of caching could be lost. In a virtual memory system such fragmentation may lead to loss of locality between associated cells of a data structure and result in 'thrashing', that is, excessive swapping of pages to and from secondary storage [4]. In either case, fragmentation makes allocation more difficult as suitable 'gaps' must be found in the heap to accommodate new objects. It can also cause allocation failures even when sufficient free memory appears to be available but is not contiguous.

The Copying Collection

In a copying collector, another form of tracing collector, the heap is divided into two equally sized semi-spaces, one of which contains active data and the other is unused. When the active space fills up, the execution of a program is suspended. The collector then traverses the active data structure in the active space, Fromspace, copying each live cell into the inactive space, Tospace, when the cell is first visited, refer to Fig. 3.5 [18]. The roles of the spaces are then flipped, with the old inactive space becoming the new active space. Since garbage cells are simply abandoned in the old active space, Fromspace, copying collectors are often described as scavengers - they pick out worthwhile objects amidst the garbage and take them away.

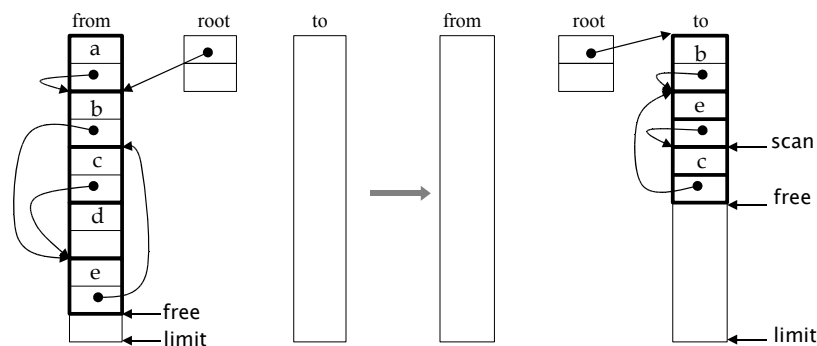


Figure 3.5: Copying live cells in Fromspace to Tospace. free points to the next free location

The copying collection compacts data structure into the bottom of Tospace resulting in low allocation costs. The out of space check is a simple pointer comparison and space is allocated by incrementing the free space pointer by the size of the object which is passed as

a parameter n to `New`. Thus compacting collectors can allocate objects much more efficiently than collectors that causes fragmentation.

```
init()
  Tospace:= Heap_bottom
  space_size := Heap_size / 2
  top_of_space := Tospace + space_size
  Fromspace := top_of_space + 1
  free := Tospace
```

```
New(n)
  if free + n > top_of_space
    flip()
  if free + n > top_of_space
    abort "Memory exhausted"
  newcell := free
  free := free + n
  return newcell
```

Algorithm for Copying Collection. First, the roles of `Tospace` and `Fromspace` are swapped by `flip`, which resets the variables `Tospace`, `Fromspace` and `top_of_space`. Each cell reachable from a root is then copied from `Fromspace` into `Tospace`. For clarity, a simple recursive algorithm [12] is used.

```
flip()
  Fromspace, Tospace := Tospace, Fromspace
  top_of_space := Tospace + space_size
  free := Tospace
  for R in Roots
    R := copy(R)
```

`Copy(P)` scavenges the fields of the cell pointed at by P . To preserve the topology of shared structures, a forwarding address, which is the address of the copy in `Tospace`, is left in the `Fromspace` object when it is copied. Whenever a cell in `Fromspace` is visited, space is reserved in the `Tospace` if it is not already being copied. Otherwise its forwarding address is returned. In this recursive copying algorithm, the forwarding address is set to point to this reserved memory before the constituent fields of the object are copied which ensures termination and that sharing is preserved.

The forwarding address might be held in its own field in the cell. More generally it can be written over the first word in the cell provided that the original value of the word is saved beforehand. In Algorithm [17], it is assumed that the forwarding address field of cell P is $P[0]$, and `forwarding_address(P)` and $P[0]$ are used interchangeably.

– Note: P points to a word, not a cell

copy(P)

```

if atomic(P) or P = nil          –P is not a pointer
    return P
if not forwarded(P)
    n := size(P)
    P' := free                    –reserve space in Tospace
    free := free + n
    temp := P [0]                –field 0 will hold the forwarding address
    forwarding_address(P) := P'
    P' [0] := copy(temp)
    for i := 1 to n-1            –copy each field of P into P'
        P' [i] := copy(P[i])
    return forwarding_address(P)

```

An Example. Consider the collection of the structure [18] shown in the Fig. 3.6. When the active space, from, fills up, i.e., when free reaches limit, garbage collection is initiated. Suppose P is a pointer in the from space pointing to from space. There are three possibilities.

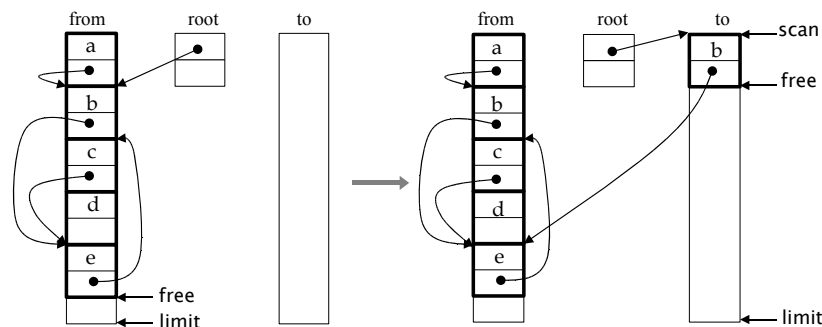


Figure 3.6: Copying objects reachable from roots

- If P points to a from-space object that has not been copied, then object pointed by P is copied to free in the to-space, P[0] is set to free and free is incremented.
- If P points to a from-space object that has already been copied to to-space, then P[0] is a special forwarding pointer that indicates where the copy is.
- If P points to an object elsewhere, then P is left unchanged in the to-space.

The collection starts by copying the objects reachable from the roots and setting scan to the start of the to-space, see Fig. 3.7.

Then all objects between scan and free are visited and all pointers in those are forwarded. Finally, all the live objects are copied to the to-space and the forwarded addresses are reset. Everything in the from-space is now garbage and can be reclaimed, see Fig. 3.8.

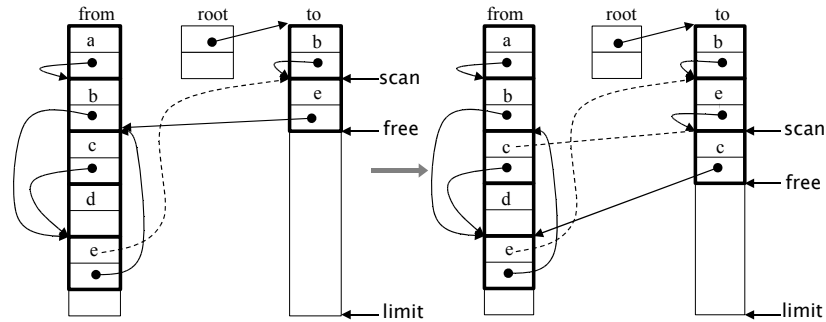


Figure 3.7: Setting the forwarding address

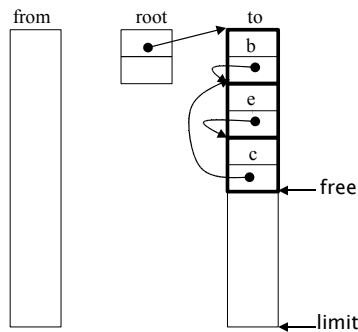


Figure 3.8: Copying all live objects completes the collection.

Strengths and Weaknesses of Copying Collection. Copying collection has the advantage of only visiting live objects. The duration of collection cycles in a copying collector is driven by the number of live objects [8].

In copying collection the set of live objects are compacted into the bottom of the heap. This improves locality of reference of the user program, eliminates heap fragmentation and reduces the cost of object allocation.

Objects that become garbage before the next collection cycle, the deallocation cost is zero, as the garbage object will be neither visited nor copied.

However, copying collectors have the added cost of copying the data from one space to another, adjusting all references to point to the new copy. In particular, long-lived objects will be copied back and forth on every collection.

The costs of the copying algorithm are twofold: First, the algorithm requires that all live objects be copied every time garbage collection is invoked. If an application program has a large memory footprint, the time required to copy all objects can be quite significant. A second cost associated with copying collection is the fact that it requires twice as much memory as the program actually uses. When garbage collection is finished, at least half of the memory space is unused [16].

Mark-Compact Garbage Collection

The mark-compact garbage collection combines the desirable features of both mark-sweep and copying collection. The copying algorithm has excellent performance characteristics, but it has the drawback of requiring twice as much memory as a mark-sweep collector. The mark-sweep algorithm has the unfortunate tendency to fragment the heap. The mark-compact algorithm eliminates fragmentation without the space penalty of copying but at the cost of some increased collection complexity.

Like mark-sweep, mark-compact is a two-phase process, where each live object is visited and marked in the marking phase. Then in the compaction phase, marked objects are copied such that all the live objects are compacted at the bottom of the heap and their pointers are updated. If a complete compaction is performed at every collection, the resulting heap is similar to the result of a copying collector. There is a clear demarcation between the active portion of the heap and the free area, so that allocation costs are comparable to a copying collector. Long-lived objects tend to accumulate at the bottom of the heap, so they are not copied repeatedly as they are in a copying collector.

The algorithm can be expressed as follows [16]:

for each root variable r

mark (r);

compact ();

Theory Behind Mark-Compact. Mark-compact collection is like mark-sweep with compaction capability like copying collection. It marks all the live objects starting from the root set; Fig. 3.9 [18].

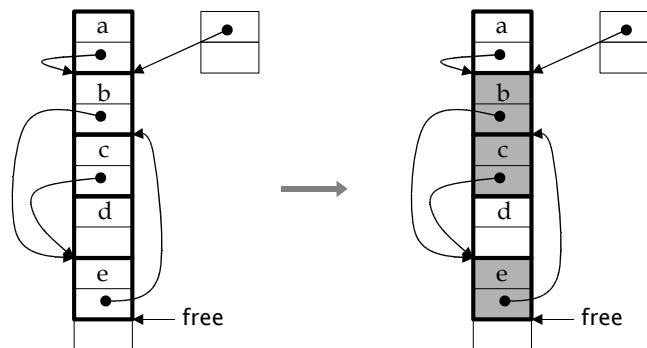


Figure 3.9: Marking phase of Mark-Compact.

In a compaction phase, Fig. 3.10, each object's new address, which is the sum of the sizes of all objects encountered so far, is calculated and is stored as forwarding field. All pointers are updated to point to the new locations.

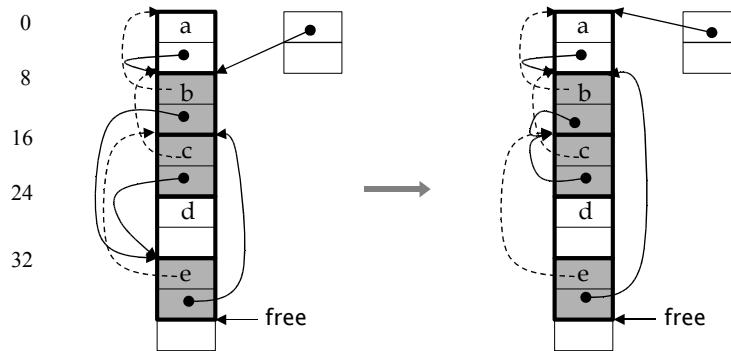


Figure 3.10: Calculating new addresses and updating pointers.

Finally, the objects are moved to their new location; Fig. 3.11, leaving the pointers unchanged. On this occasion, all objects are unmarked and all forwarding pointers become unused again [18].

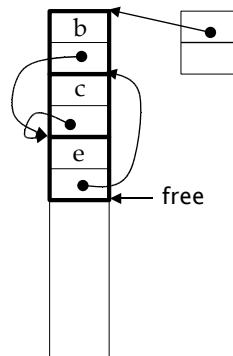


Figure 3.11: Compacting the live objects.

Strengths and Weaknesses of Mark-Compact Collection. Like copying collection, new objects are allocated by incrementing the free pointer; eliminating the need to maintain free lists. However, unlike copying collection, compaction supports locality and better caching by preserving the original order of objects. Copying collection always moves all the objects back and forth, whereas, in mark-compact, long-lived objects are accumulated at the bottom of the heap and stay there.

Mark-compact requires one extra pointer for forwarding in each object; whereas mark-sweep does not need any extra space and copying collections needs twice as much space. Like with mark-sweep, the marking phase traverses the live objects on the entire heap which results in swapping if it is larger than main memory. Like copying collection, only live objects are copied; sometimes only 5% of the heap objects are live.

Generational Collection

In any application heap, some objects become garbage shortly after their creation, some survive for a long time and then become garbage, and others can remain live for the entirety of the program's run. Empirical studies have shown that for most object-oriented languages, the vast majority of objects, as much as 98 percent, depending on the metric for object youth, die young [14]. Object's age can be measured in wall-clock seconds, in total bytes allocated by the memory management subsystem since the object was allocated, or by the number of garbage collections since the object was allocated. This also supports the weak generational hypothesis that "most objects die young" [19]. This plays a significant role in the choice of a collector.

A generational collector divides the heap into multiple generations. Objects are created in the young generation, and objects that meet some promotion criteria, such as having survived a certain number of collections, are then promoted to the next older generation. A generational collector is free to use a different collection strategy for different generations and perform garbage collection on the generations separately.

Of the objects that survive past their first collection, a significant portion of those will become long-lived or permanent. The various garbage collection strategies perform very differently depending on the mix of short-lived and long-lived objects. Copying collectors work very well when most objects die young, because objects that die young never need to be copied at all. However, the copying collector deals poorly with long-lived objects, repeatedly copying them back and forth from one semi-space to another. Conversely, mark-compact collectors do very well with long-lived objects, because long-lived objects tend to accumulate at the bottom of the heap and then do not need to be copied again.

Minor Collections. One of the advantages of generational collection is that it can make garbage collection pauses shorter by not collecting all generations at once. When the allocator is unable to fulfill an allocation request, it first triggers a minor collection, which only collects the youngest generation. If the minor collection frees enough heap space, the user program can resume immediately otherwise it proceeds to collect higher generations until enough memory has been reclaimed. In the event the garbage collector cannot reclaim enough memory after a full collection, it will either expand the heap or it will throw an `OutOfMemoryError` [3].

Intergenerational References. A generational tracing collector starts from the root set, but unlike traditional tracing collectors that visit all live objects, does not traverse references that lead to objects in the older generation, which reduces the size of the object graph to be traced. This leads to a problem; how to prevent minor collection from reclaiming an object if it is only reachable through an older generation object?

To address this problem, generational collectors must explicitly track references from older objects to younger objects and add these old-to-young references into the root set of the minor collection. There are two ways to create a reference from an old object to a young

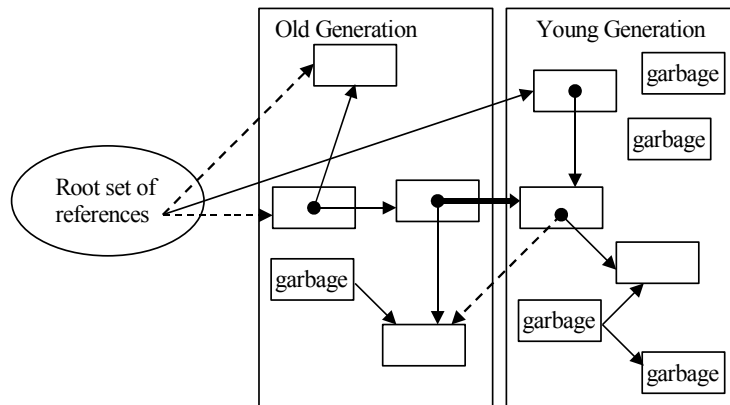


Figure 3.12: Intergenerational references.

object. Either one of the references contained in an old object is modified to refer to a young object, or a young object that refers to other young objects is promoted into the older generation.

In Fig. 3.12, the arrows represent references between objects in the heap. The bold arrows represent old-to-young references that must be added to the root set for a minor collection. The dotted arrows represent references to old objects, either from the root set or from the young generation, which don't need to be traced when collecting only the young generation.

Tracking Intergenerational References. The garbage collector needs to have a comprehensive set of old-to-young references when it wants to perform a minor collection. The mutator and garbage collector can work together to maintain this set of old-to-young references as they are created. When objects are promoted into the older generation, the garbage collector can note any old-to-young references that are created as a result of the promotion, which leaves only the tracking of intergenerational references created by pointer modifications.

The garbage collector can track old-to-young references that arise through modifying references held within existing objects in several ways. It could track them in the same manner as maintaining reference counts in reference-counting collectors (the compiler could generate additional instructions surrounding pointer assignments) or could use virtual memory protection on the old generation heap to trap writes to older objects. Other ways to handle intergenerational references are remembered list, card marking and page marking [18].

Influence of Memory Management Strategies on Software Design

Memory management strategies used in different languages has vast impact on modern software design. The basic problem in managing memory is knowing when to keep the data it contains, and when to throw it away so that the memory can be reused. There are two different views due to the strategies used in software design.

People in favor of automatic memory management argue that manual memory management conflicts with the principles of abstraction and modularity [17]. They argue that by using automatic memory management, programmers are relieved of the burden of book-keeping detail and their time is better spent on the higher-level details of the design.

Memory management by the run-time systems is adopted by all high-level programming languages for static and stack-allocated data. The programmers do not have to worry where to place the global data, or how to setup or take down procedure activation frames on the stack. This abstraction principle also applies to heap-allocated data. Automatic memory management also decreases the chances of one module to cause the failure of another module through space leaks, space overflows and premature reclamation of storage. The software developer doesn't have to worry about these anymore. This is very helpful in case of large-scale projects involving multiple teams of developers working on different modules.

For good design, every module should communicate with as few others as possible, and if any two modules do communicate, they should exchange as little information as possible [15]. Adding book-keeping detail to module interfaces weakens abstraction and reduces the extensibility of modules. Modifications to the functionality of a module might entail alteration of its memory management code. Since liveness is a non-local matter, changes to book-keeping code might radiate beyond the module being developed.

However, the people on the other side of the ring, i.e., those in favor of manual memory management, argue that real-time systems demand that the time spent serving memory requests will be very small and all memory requests will be satisfied which is not possible with the automatic memory management. It is also said that although automatic management removes the two major problems of explicit storage management, dangling pointers and space leaks, it is still vulnerable to other errors, and moreover raises debugging problems of its own.

Automatic management doesn't have any solution for the problem of data structures that grow without bound. Such data structures are "surprisingly common", with one example being the caching of intermediate results to avoid re-computation [17]. It should also be noted that developers by nature don't trust the code of anybody else and hence they try to use manual memory management where-ever possible. In selecting the strategy to manage memory, the following points should be considered.

1. Footprint of the final compiled file (i.e total space required by the final compiled file).

The footprint is the "size of the software in memory when it is running live" in the system memory. The footprint of the program using automatic memory management is larger than the same program using manual memory management. This is because automatic memory manager also includes the code with automatic memory allocaters and de-allocaters. Automatic memory manager also has to keep track of each and every memory allocations and changes and their deallocations in proper order. Code written using manual memory management doesn't have these components.

Automatic garbage collection is an integral part of some of the languages. The problem with using these languages in real time applications is that all mission critical systems

(routers, switches and all medical equipments etc) have a limited size of memory residing within their assembly. Hence one has to be very careful so that the final compiled file has a smaller size and can fit in the memory on that device. Smaller size gives it a better chance to fit on all sorts of low memory platform or devices.

2. Pause or delay while collecting garbage.

This is also a very critical factor while selecting any language. In languages using automatic memory management, the garbage collection task or tasks run in the background at the expense of other tasks running at that time. This is the reason that a small pause or delay in program execution is felt during the garbage collection.

Care should be taken to confirm that a small pause will not cause problems in program execution. For example a person browsing internet can wait a few seconds more for his GUI to appear on the PC. On the other hand a person on ventilator cannot wait extended period of time and can expire. Care should also be taken that time consumed by garbage collection will not cause any other task to time-out.

3. Selection of exact time when destructor or de-allocator are called.

Manual memory management enables the software developer to control that after which statement “destructor” is called and exactly under what terms and conditions the de-allocator is called. Unfortunately with automatic management there is no guarantee when the de-allocator will be called. It is called automatically by the system and there may be situations when other higher priority tasks are called instead of calling the de-allocator. This delays the return of memory to the system, whereas the developer might be under false impression that this memory has been returned to the system.

4. External fragmentation.

A poor allocator can do its job of giving out and receiving blocks of memory so badly that it can no longer give out big enough blocks despite having enough spare memory. This is because the free memory can become split into many small blocks, separated by blocks still in use. This condition is known as external fragmentation.

How This Influence can be Avoided

One way of avoiding the problems like above is to write one’s own allocator and de-allocator [6]. This confirms that one will have no surprises while calling and using somebody else’s code. A well-designed memory manager can make it easier to write debugging tools, because much of the code can be shared. Such tools could display objects, navigate links, validate objects, or detect abnormal accumulations of certain object types or block sizes.

Why is This Influence Un-Avoidable

This influence is un-avoidable in some instances as it is totally against the nature of the problem being resolved. For example Smalltalk cannot be used for Real Time programming.

The reason being that Smalltalk does a lot of automatic garbage collection which will cause all sorts of code footprint increase, execution pauses and delayed response times.

Similarly, C/C++ is not very suitable for GUI based applications, reason being that C/C++ doesn't have much automatic memory management support, which is critical in GUI refreshment and other GUI based tasks.

3.5 Programmers Point Of View

Programmers generally tend to keep things in their hands. They tend to take care of all the aspects of the program themselves and seldom rely and believe in other peoples code. This is the reason that most of the times programmers hate to let go the power in their hands and use some third party tool to make important decisions about their code.

It is a very common complain of Java programmers that for instance, once an object moves to native method, optimizer should think that it escapes, but in reality it might happen that it does not.

Another important point which C++ developers point to is that Java doesn't have any destructors. Destructors are automatically called when an object of its type is being destroyed/deleted.

The "finalize()" method in Java can not be a replacement of destructor, since it is not known when it will be actually called. Sometimes some resources must be released just before exiting the method's frame. Even if it is 100% sure that object that allocates those resources does not escape, one can not rely on that optimizer will allocate this object on the stack and will call finalize() method just after method ends.

So "finalize" method should not be used for resource deallocation, instead some other method like freeResources() should be defined having construction like this:

```
try{MyObjectWithResources o = new MyObjectWithResources();
    ...
    finally{
        o.freeResources();
    }
}
```

which is manual memory management (or manual resource management) anyway. What this means is that stack-live objects exist and are used, and it could be nice if I could use "stacknew" operator instead of "new" and do not write code like above. However I agree that it is not safe (one can use this in wrong cases), and should not be in Java.

3.6 Conclusion

There are numerous patterns of memory management at one's disposal to match the project requirements. Each pattern has a wide range of implementations, each of which has its

Strategy	Allocation speed	Deallocation speed	Cache locality	Ease of Use	Generality	Usable in real time
Custom Allocator	Depends on implementation	Depends on implementation	Depends on implementation	Very difficult	None	Depends on implementation
Simple Allocator	Fast for small memory usage	Very fast	Poor	Easy	Very	No
GNU malloc	Moderate	Fast	Moderate	Easy	Very	No
Reference Counting	N/A	N/A	Excellent	Moderate	Moderate	Yes (depends on malloc implementation)
Pooling	Moderate	Very fast	Excellent	Moderate	Moderate	Yes (depends on malloc implementation)
Garbage Collection	Moderate(slow when collection occurs)	Moderate	Poor	Moderate	Moderate	No

Table 3.1: Comparison of memory allocation strategies.

benefits and drawbacks. Comparing several memory management strategies, see also Table 1.1 [6], I conclude that there's a tradeoff between numerous optimization possibilities, such as performance, speed, ease-of-implementation, and ease-of-use.

3.7 Exam Questions

1. The efficiency of a garbage collection scheme is the rate at which memory is reclaimed. Let M be the size of the heap and let f be the fraction of the heap occupied by live data. Estimate the running time, amount of space reclaimed and efficiency of the mark-sweep and copying garbage collection schemes as a function of f and M .
2. What are the drawbacks of explicit memory management?
3. How are intergenerational references handled in generational garbage collection?

Bibliography

- [1] Neville Harris Ahmed El-Habbash, Chris Horn. *Garbage Collection in an Object Oriented, Distributed, Persistent Environment*. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.
- [2] Andrei Alexandrescu. *Modern C++ Design - Generic Programming and Design Patterns Applied*. C++ In-Depth Series. Addison Wesley, 2001.

- [3] Andrew W. Appel. *Simple generational garbage collection and fast allocation*. *Software Practice and Experience*, 19(2):171–183, 1989.
- [4] H. D. Baecker. *Garbage collection for virtual memory computer systems*. *Communications of the ACM*, 15(11):981–986, 1972.
- [5] David A. Barrett and Benjamin G. Zorn. *Using Lifetime Predictors to Improve Memory Allocation Performance*. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 187–196, Albuquerque, NM, June 1993. ACM Press.
- [6] Jonathan Bartlett. *Inside memory management - The choices, tradeoffs, and implementations of dynamic allocation*. 2004.
- [7] Bill Blunden. *Memory Management - Algorithms and Implementation in C/C++*. Wordware Publishing, Inc., 2002.
- [8] David R. Chase. *Garbage collection and other optimizations*. PhD thesis, Rice University, 1987.
- [9] T. W. Christopher. *Reference count garbage collection*. *Software Practice and Experience*, 14(6):503–507, 1984.
- [10] Douglas W. Clark and C. Cordell Green. *An Empirical Study of List Structure in Lisp*. *Communications of the ACM*, 20(2):78–86, February 1977.
- [11] Robert Courts. *Improving locality of reference in a garbage-collecting memory management system*. *Communications of the ACM*, 31(9):1128–1138, 1988.
- [12] Robert R. Fenichel and Jerome C. Yochelson. *A Lisp Garbage Collector for Virtual Memory Computer Systems*. *Communications of the ACM*, 12(11):611–612.
- [13] Brian Goetz. *Java theory and practice: A brief history of garbage collection*. 2003.
- [14] Brian Goetz. *Java theory and practice: Garbage collection in the HotSpot JVM*. <http://www-106.ibm.com/developerworks/java/library/j-jtp01274/>, 2004.
- [15] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- [16] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. John Wiley & Sons, 1999.
- [17] Rafael Lins Richard Jones. *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1997.
- [18] Dr. Emil Sekerinski. *Lecture Notes of Compiler Construction*. 2004.

- [19] David M. Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.
- [20] Paul R. Wilson. *Uniprocessor Garbage Collection Techniques*. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
- [21] Benjamin Zorn. *The Effect of Garbage Collection on Cache Performance*. Technical Report CU-CS-528-91, Department of Computer Science, Boulder, Colorado, 1991.

Chapter 4

Andi Huang: Fault Tolerance

Computer based systems have developed in scope and complexity greatly. Meanwhile, highly reliable software is significantly required in many areas. For instance, in railway software system, nuclear reactor control system, aircraft control system, the high reliability becomes more and more important, since the cost and consequences of these systems' failing will lead to serious results. Ideally, software could be developed without errors. However, the current state of the practice is that fewer errors are introduced, but creating or obtaining components without faults are almost infeasible. It would be very dangerous to assume the software developed is error-free. Thus, we must accept that errors will happen, and build additional redundancy so that errors in the operation of a system are detected and corrected. Our goal is to correct an erroneous internal state of the system before it propagates to cause an externally observable failure. To achieve this, we turn to software fault tolerance. The classic definition [18], [4] of software fault tolerance is: using a variety of software methods, faults are detected and recovery is accomplished.

This paper surveys the techniques in achieving software fault tolerance, which includes recovery blocks (RcB), N-version programming (NVP), distributed recovery blocks (DRB), consensus recovery blocks (CRB), acceptance voting (AV), retry blocks (RtB), N-copy programming. Also, techniques in measuring software fault tolerance such as reliability models and fault injection technique are surveyed.

During the exploration of this survey, Dhiraj's book [13] and Laura's book [16] provided a comprehensive overview of software fault tolerance; while Lee's book [22] gave a deeper analysis; Jeffrey's book [12] and Daniel's book [20] were really good on measuring fault tolerance. Of course, all those paper and other books listed in the bibliography provided a lot of valuable information to understand fault tolerance techniques.

4.1 Techniques For Achieving Software Fault Tolerance

There are four phases in achieving software fault tolerance: error detection; error confinement and assessment; error recovery; fault treatment. Based on the recovery strategy, software fault tolerance can be classified into two classes: backward recovery and forward recovery.

Backward recovery: after an error happens, backward recovery techniques will try to recover the system to an error-free state (checkpoint) by rolling back the system to a previously “good”, error free state. The advantages of backward recovery are: it provides a general recovery scheme (a uniform pattern of error detection and recovery); it can handle unpredictable errors caused by residual design faults if the errors do not affect the recovery mechanism; it can be used regardless of the damage sustained by the state; it requires no knowledge of the errors in the system state. The disadvantages of backward recovery are: it requires many resources such as time and stable storage to perform checkpointing and recovery; it often requires the system to halt temporarily.

Forward recovery: after an error happens, forward recovery technique will try to recover the system to an error-free state by finding a new state from which the system can continue to operate. This state may be a degraded mode of the previous error-free state. The advantage of forward recovery is: it is efficient in terms of the overhead (time and memory), which can be crucial in real-time applications where the time overhead of backward recovery can exceed stringent time constraints. The disadvantages of forward recovery are: it requires knowledge of the error; it is application-specific and can only remove predictable errors from the system state.

In fault tolerance, *redundancy* is a key supporting concept. It can be in several forms such as hardware, software, information, and time, providing the additional capabilities and resources needed to detect and tolerate faults. In this paper, we will focus on software and information(data) redundancy.

As we know, simple redundancy such as simply replicate the identical software or use the same information(data), the same error will probably still be there. It is necessary to implement diversity in redundancy. As a result, design diversity and data diversity become the key techniques in achieving software fault tolerance.

Design Diversity

Design diversity [5] is the provision of identical services through separate design and implementations. The goal of design diversity is to make the modules as diverse and independent as possible, with the ultimate objective being the minimization of identical error causes. We want to increase the probability that when the software variants fail, they fail on disjoint subsets of the input space. In addition, we want the reliability of the variants as high as

possible in order that at least one variant will be operational at all times.

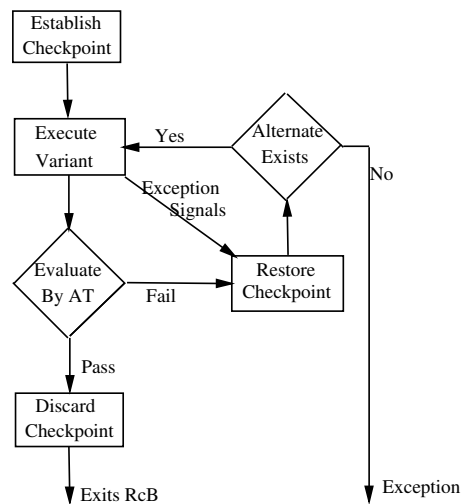
Recovery Blocks (RcB): RcB technique is one of the two original design diverse software fault tolerance techniques. It was introduced in 1974 by Horning, et al. [9]. The RcB technique is categorized as a *dynamic technique* [16]. In dynamic software fault tolerance techniques, the selection of a variant result to forward as the adjudicated output is made during program execution based on the result of the *acceptance test (AT)*. The AT is used to verify that the system’s behavior is acceptable based on an assertion on the anticipated system state.

Usually, there are many ways to realize a program function through different algorithms and designs. These differently implemented programs, which are called variants, will have different degrees on efficiency in many criteria such as reliability, execution time and memory utilization. In RcB, these variants are arranged depending on their efficiency in a decreasing order. In other words, the most efficient variant, which is called primary try block, will be placed in the first place and the less efficient variants, which are called alternate try blocks, will be placed serially after the primary try block. Also, acceptance test (AT) and backward recovery are hired in the RcB. The general RcB pseudocode and structure [16] are:

```

ensure acceptance test
by primary try block
else by alternate try block 1
else by alternate try block 2
...
else by alternate try block n
else error
    
```

Recovery block structure:



The RcB pseudocode above states that the technique will first try to ensure the AT, using the primary try block. If the result does not pass the AT, then n alternate try blocks will be tried sequentially. During the process, if any try block’s results passes the AT, then RcB finishes and returns. Otherwise, an error occurs.

The RcB technique provides a unifying framework to achieve software fault tolerance, incorporating strategies for error detection by AT; backward error recovery provided automatically by a mechanism; fault treatment which simply uses an alternate module; no strategy for damage assessment is needed because of the backward recovery.

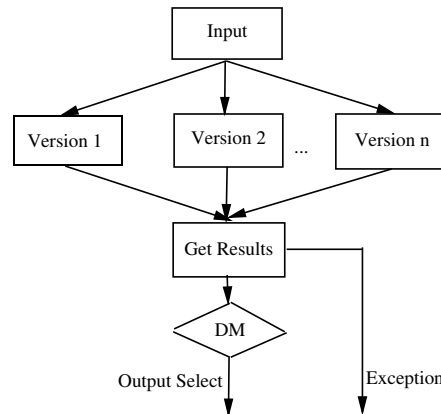
It should be pointed out that a highly effective acceptance test (AT) is required for the RcB technique. If an error cannot be detected by the AT, then this error may lead to the failure because those recovery mechanisms cannot be triggered with a “poor” AT. However, there is a conflict at some point: on the one hand, it is desirable to have an AT as comprehensive as possible; on the other hand, there is a need to keep the AT simple so that its run-time overheads is reasonable and the AT itself is not prone to design faults.

Two augmentations to the basic RcB technique have been suggested: in some implementations of RcB, especially for real-time applications, there is a *watchdog timer* (WDT) [8], which can provide extra control on the time for the executive; the other augmentation is to use an *alternate routine execution counter*. This counter is used when the primary fails and alternate takes the execution. The counter controls the times to execute the alternate, providing the ability to take the primary out of service for “repairing” while the alternate continues the algorithm execution tasks.

N-Version Programming(NVP): NVP is the other original design diversity software fault tolerance techniques. NVP was suggested by Elmendorf in 1972 [7] and developed by Avizienis and Chen [5], [6]. NVP is categorized as a *static technique* [16]. In static software fault tolerance techniques, a task is executed by several programs and a result is accepted only if it is decided as an acceptable result, usually through a majority vote. The technique is called *static* because the various programs executing the task will execute in the same way, regardless of which result(s) was decided as acceptable by the decision mechanism (DM).

In NVP technique, there must be over two variants, which have been independently designed to satisfy a common specification. These variants will run parallel and the *best* results will be decided by the DM. The general N-Version Programming pseudocode and structure [16] are:

```
run Version 1, Version 2, ..., Version n
Result if (Decision Mechanism (Result 1, Result 2, ..., Result n ))
return Result
else Error
```



N-Version Programming structure

The NVP pseudocode above states that the technique invokes n versions in parallel. The results of these executions are provided to the DM, which will determine the result. If a correct one can be adjudicated, it will be returned. Otherwise, an error occurs.

Here is an example: to perform sort, there are some variants, which can be implemented with different algorithms such as heap sort, quick sort, bubble sort and so on. The input data will be executed simultaneously on these variants and the results will be gathered and adjudicated by the DM. Here is DM can be a majority voter.

When a DM can be successfully implemented, the NVP is a simple and attractive technique for fault tolerance: error detection is provided by DM; error recovery involves ignoring the values identified as erroneous by the check; and the fault treatment simply results in the versions determined to have produced erroneous results be identified together with a reconfiguration to avoid reinvoking failed versions subsequently.

One thing should still be pointed out that even though NVP utilizes the design diversity principle, it cannot be guaranteed that the variants have no common residual design faults. If this occurs, the purpose of NVP is defeated. The DM may also contain residual design faults. If it does, then the DM may accept incorrect results or reject correct results, NVP is defeated again.

Two augmentations to N-Version Programming have been suggested, just by using a different decision mechanism than the basic majority voter: one optional DM is the *dynamic voter*. It has the ability to handle a variable number of result inputs, since the basic DM can only cope with fixed number inputs; the other augmentation is voting on the results as each version completes execution, instead of waiting for the completion of all versions. This scheme is more efficient than the basic NVP, assuming the versions have different expected execution times. Since the NVP will exit as soon as any acceptable result is found.

Distributed Recovery Blocks (DRB): DRB technique was developed by Kane Kim [11] as a means of integrating hardware and software fault tolerance in a single structure. As the name suggests, it is a modification of the RcB. The difference between DRB and basic RcB is that the primary and alternate recovery block are resident on two or more nodes interconnected by the network.

This technique is mainly applied in real-time applications, distributed and parallel computing systems, and handles both hardware and software faults. Although DRB uses recovery blocks, it implements a forward recovery scheme, consistent with its emphasis on real-time applications. The general pseudocode of DRB [16] is:

```
run RB1 on Node 1, RB2 on Node 2
ensure AT on Node 1 or Node 2
by Primary on Node 1 or Alternate on Node 2
else by Alternate on Node 1 or Primary on Node 2
else Error
```

The above DRB pseudocode states that the technique executes the recovery blocks on both nodes concurrently, with one node executing the prime and the other executing the alternate. It first tries to ensure the AT with the primary recovery block on node 1. If this result fails with the AT, then the DRB tries the result from the alternate recovery block on node 2. If neither passes the AT, then backward recovery is used to execute the alternate on Node 1 and the primary on Node 2. The results of these executions are checked in the same way as before. If neither of these results passes the AT, then an error occurs. If any of the results are successful, the result is passed on.

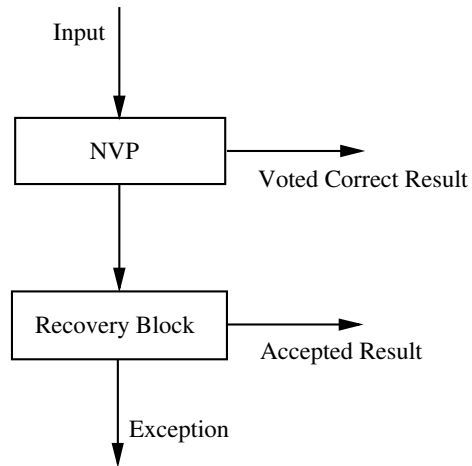
DRB technique has the following useful characteristics [11]: regardless of whether a node fails due to hardware faults or software faults forward recovery can be accomplished in the same way; the recovery time is minimal since maximum concurrency is exploited between the primary and the shadow nodes; the increase in the processing turnaround time is minimal because the primary node does not wait for any status message from the shadow node.

Consensus Recovery Block (CRB): CRB technique was suggested by Scott, Gault, and McAllister [19]. It's a kind of hybrid fault tolerance strategy, which combines both RcB and NVP fault tolerance techniques. Briefly, CRB technique processes in NVP style, which is then followed by RcB style. If either of them succeeds, the result is passed to the successor computing station. If neither succeeds, then the system raises an exception. Meanwhile, in CRB technique, the importance of the AT used in the RcB is reduced. Also, it can handle cases where NVP would not be appropriate because of *multiple correct results* (MCR: two or more dissimilar correct answers exist for the same problem, for the same input, which will cause DM do the wrong decision).

The general pseudocode and structure for CRB [16] are:

```

run Ranked Variant 1,
    Ranked Variant 2, ...,
    Ranked Variant n
if (Decision Mechanism (Result 1,
                        Result 2, ...,
                        Result n ))
return Result
else
ensure Acceptance Test
by Ranked Variant 1 [Result]
else by Ranked Variant 2 [Result]
..
else by Ranked Variant n [Result]
else raise failure exception
return Result
    
```



Consensus Recovery Block Structure

In CRB, all n variants are ranked by their reliability and service. These n variants first are invoked concurrently in NVP style. If the DM can produce a correct result, then CRB exits. Otherwise, the result of the highest rank variant will be sent to AT. If that variant's results fail on AT, then the next highest rank variant's result will sent to the AT, and so on, until an acceptable result passes the AT or no variant is left.

In the RcB part of the CRB technique, the existing results of variant execution, the ones that just failed to result in a majority decision, can be run through the AT again, which provides some kind of tolerance to transient fault. The system fails if both the NVP and the RcB portions of the technique fail to come up with a correct result.

A general disadvantage of all hybrid strategies such as the CRB is that an increased complexity of the fault-tolerance mechanism will result in the increasing of the probability of existence of design or implementation errors.

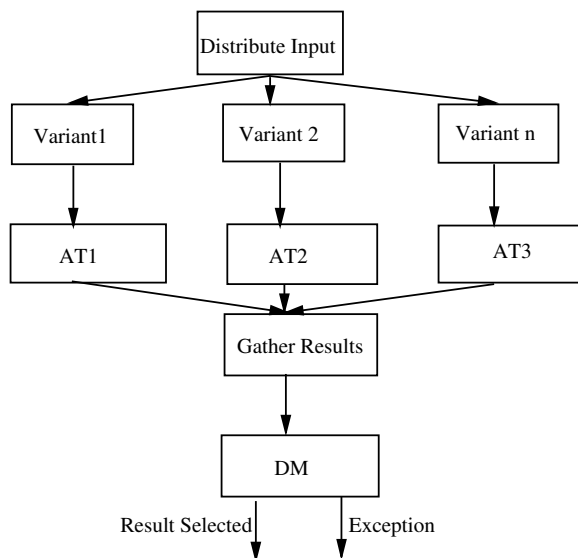
Acceptance Voting(AV): the AV technique was proposed by Athavale [3]. It is the other kind of hybrid fault tolerance strategy, which incorporates both AT and voting-type DM technique. The AV technique is also implemented with forward recovery. In brief, in AV technique, the output of each module is first presented to an acceptance test and only those results that pass the acceptance test are to be used by DM (vote). A dynamic voting algorithm is required in this technique, since the DM may see varying number of results.

For example, if two results pass the AT, they are compared. If three results pass, they will be voted upon. If no results pass the AT, then the system fails. It also fails if the dynamic voter cannot select a correct result. The general pseudocode and structure for AV [16] are:

```

run Variant 1, Variant 2, ..., Variant n
ensure Acceptance Test 1 by Variant 1
ensure Acceptance Test 2 by Variant 2
...
ensure Acceptance Test n by Variant n
Result i, Result j, ..., Result m pass
the AT
if (Decision Mechanism
(Result i, Result j, ..., Result m ))
return Result
else
return failure exception

```



The Acceptance Voting Structure

In the AV fault tolerance technique, the reliability of AV is quite important to the success of this fault tolerance technique. As one can see from the structure, AV acts as a pioneer, which means if it allows erroneous results to be accepted, then the advantage of catching potential related faults prior to being assessed by the voter-type DM is minimal at best. Also, as a kind of hybrid fault tolerance strategies, a general disadvantage such as increasing in the probability of existence of design or implementation errors will also apply to AV technique.

Data Diversity

To complement the design diversity techniques in achieving software fault tolerance, Ammann and Knight [2] proposed the use of data diversity. Data diversity involves obtaining a related set of points in the program data space, running the same software on those points, and then using a decision algorithm to decide the resulting output. In data diverse techniques, those “diverse” data are obtained through *data re-expression algorithms* (DRA). The goal of DRA is to produce data points which are outside of the failure region.

Data diversity and design diversity are analogous at some point: design diversity techniques use different alternate to run on the same data; data diversity, however, use different expressions (version) of data to be run by the same program.

Retry Blocks (RtB): similar to RcB in structure, the RtB technique also uses acceptance tests (AT) and backward recovery to achieve fault tolerance. It is categorized as dynamic fault tolerance technique too. Here is the general pseudocode for Retry Blocks technique [17]:

```

ensure Acceptance Test
by Primary Algorithm(Original Input)
else by Primary Algorithm(Re-expressed Input)
else by Primary Algorithm(Re-expressed Input)
...
Deadline Expires
else by Backup Algorithm(Original Input)
else failure exception

```

In basic RtB, there are one DRA and a watchdog timer (WDT). The function of the WDT is to trigger the backup algorithm if the primary algorithm cannot produce any acceptable result within a specified period of time.

The pseudocode above states that the technique will first try to ensure the AT, using the primary algorithm on the original input. If the algorithm results pass the AT, then the RtB is complete. Otherwise, it will hire DRA to re-express the original input and run the same algorithm again until AT finds an acceptable result or the deadline of WTD is reached. If it is the latter case, a backup algorithm will be invoked on the original input data. If this backup algorithm is not successful, an error occurs.

To achieve effective RtB, highly effective AT is required. The reason is quite similar as what has been discussed in RcB technique. Also, the success of this techniques highly depends on the efficiency of the re-expression algorithm used. DRAs are very application dependent and in-depth knowledge of the algorithm is required. There is no general rule for the derivation of DRAs for all applications. Usually, a simple DRA is more desirable than a complex one because the simpler algorithm is less likely to contain design faults.

N-Copy Programming(NCP): similar to NVP in structure, the NCP technique also uses a decision mechanism (DM) and forward recovery to accomplish fault tolerance. It is categorized as static fault tolerance technique too. The general pseudocode for NCP [17] is:

```

run DRA 1, DRA 2, ..., DRA n
run Copy 1(result of DRA 1),
Copy 2(result of DRA 2),
...,
Copy n(result of DRA n)
if (Decision Mechanism (Result 1, Result 2, ...,Result n))

```

```
return Result
else failure exception
```

In the NCP technique, there is one or more DRAs, a DM and at least two copies of a program. DRAs are hired to run on the original inputs to re-express and try to get diverse data, standing outside the failure region. The copies execute in parallel using the re-expressed data as input except one use the original inputs. A DM is hired to examine the results of the copy executions and selects the “best” result.

The NCP pseudocode above states that the technique will first run different DRA on the same original input and then execute those re-expressed data on program copies concurrently. The results of the copy executions are provided to the DM. If a correct result can be adjudicated by the DM, then the result is returned. Otherwise, error happens.

To achieve effective NCP, highly reliable and efficient DM and DRA are required. If DM does contain residual design faults, then the DM may accept incorrect results or reject correct results. Also, a simple DRA is more desirable than a complex one since the simpler algorithm is less likely to contain design faults itself.

Other Techniques

Besides the techniques for achieving software fault tolerance, which have been discussed above, there are still many other techniques. Since the available size of this paper, we will just list these techniques without detail analysis: *Two-Pass Adjudicators(TPA)* [15] technique which combines both data and design diversity; *Robust software approach*, which does not use any form of redundancy, such as monitoring techniques, atomicity of actions, decision verification, and exception handling to partially tolerate software design faults; *Resourceful Systems* [1]; *Byzantine fault tolerance* [10].

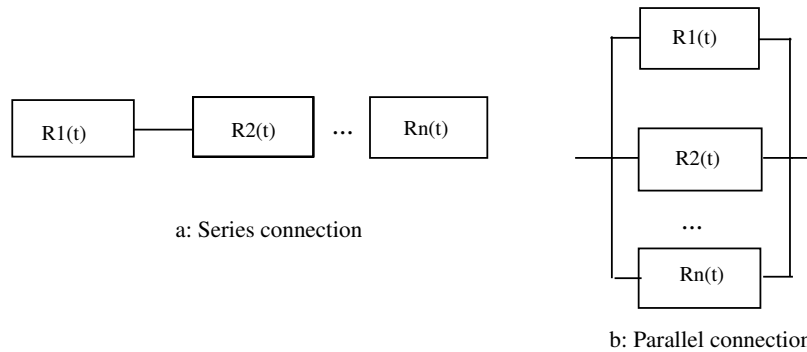
4.2 Techniques For Measuring Software Fault Tolerance

In order to evaluate the efficiency of fault tolerance mechanisms, techniques in measuring software fault tolerance such as reliability models and fault injection have been introduced.

Conventional reliability Models

Reliability models [13] are intended to illustrate the effect of failures of elements on the over-all system, particularly where some elements are redundant or incorporate fault tolerance provisions. In the conventional reliability model, redundant elements are placed in parallel and all essential functions required for the system are placed in series. The failure probability of a function that consists of two redundant elements is the production of the failure probabilities of the individual elements. The reliability of a system is the product of the

reliability of all its functions. Here are the figures of conventional reliability models [21]:



If $R_i(t)$ is the reliability of module i and if the model are assumed independent, then the overall system reliability is:

$$R_{serial}(t) = \prod_{i=1}^n R_i(t)$$

$$R_{parallel}(t) = 1 - \prod_{i=1}^n [1 - R_i(t)]$$

Fault tree diagrams (FTDs) and reliability block diagrams (RBDs) are two of the most commonly used approaches based on this conventional reliability model. Fault trees represent all the sequences of individual component failures that cause the system to failure, in a treelike structure. It is a pictorial representation of the combination of events that can cause the occurrence of system failure; In RBDs, the system reliability is represented by the reliability of each component in a series-parallel logical structure. The most fundamental difference between FTDs and RBDs is that RBDs work on system success combinations, while FTDs work on system failure combinations.

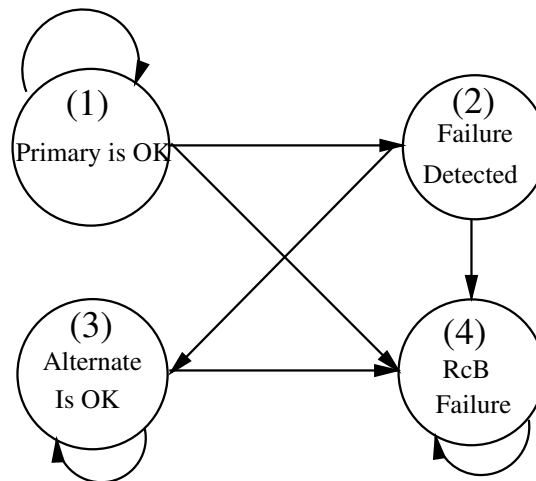
While this conventional modeling technique can be used to depict the structure of fault tolerance, it is not suitable for quantitative evaluation because it does not account for the two most significant factors that contribute to the system failure after fault tolerance is implemented: undetected failure and correlated faults. In NVP, the undetected failure includes those that never reach the Voter, while in the RcB approach, they arise from inadequate coverage of the Acceptance Test (AT). Correlated faults are those faults present in two or more versions of a program.

State Transition Model

Compared to the conventional reliability model, state transition model [14] does consider the undetected failures and correlated faults. It will be illustrated here by evaluating the reliability of a RcB system.

The figure below is a representation of these potential causes of system failure for a recovery block with a single alternate. A given recovery block can be one of the following four states:

1. Primary routine operates satisfactorily
2. A failure in the primary software has been detected
3. Alternate routine operates satisfactorily
4. Recovery block fails



State Transition Model

The system failure probability is represented by the probability of all paths that originate from state 1 and terminate at state 4. These paths are:

1. 1 to 4: undetected failure of primary routine
2. 1 to 2 and 2 to 4: detected failure of primary routine and a correlated fault in the alternate that prevent state 3 from being achieved
3. 1 to 2, 2 to 3 and 3 to 4: detected failure of the primary routine followed by satisfactory operation of the alternate and then followed by an uncorrelated fault in the alternate.

The transition probability table for this system is:

From/To	1	2	3	4
1	1-f	cf	0	f(1-c)
2	0	0	u	1-u
3	0	0	s	1-s
4	0	0	0	1

c = coverage(error detection capability)

f = failure probability of the primary routine

s = success probability of the alternate routine

u = probability of no correlated faults exist in the primary and alternative

The probability failure of the RcB, $P[F]$, is the sum of the probabilities leading to state 4:

$$\begin{aligned}
 P[F] &= P[1, 4] + P[1, 2]P[2, 4] + P[1, 2]P[2, 3]P[3, 4] \\
 &= f(1 - c) + cf(1 - u) + cfu(1 - s) \\
 &= f(1 - cus) \\
 &= f(1 - E)
 \end{aligned}$$

Where E is to represent the effectiveness of the fault tolerance provisions. If there is no fault tolerance, $P[F] = f$.

The example above shows that state the transition model is able to provide a quantitative reliability evaluation to those system which is incorporated with fault tolerance. Meanwhile the importance of the efficient AT and diverse variants in RcB technique have been showed.

Fault injection

Fault injection [12] consists of the deliberate insertion of artificial faults in a computer system or component to assess its behavior in the presence of faults and allow the characterization of specific dependability measures. It improves our ability to observe the software in the most stressful circumstances and gains insight into how the software fault tolerance has achieved. There are four main methods for fault injection:

1. Messages-based: this approach corrupts the messages between components as they execute.
2. Memory/Storage-based: this approach corrupts the values of information either in database, in memory, or on a disk.
3. Debugger-based: this approach uses a debugger to inject errors into a running process
4. Process-based: this approach affects the state of the system by manipulating processes, for example, by having high-priority process affect system state.

Here is an example of injecting fault on a NVP system (with 3 versions): the injected fault is common design fault. The goal is to predict whether problems that manifest themselves in individual versions can propagate out the NVP system: the fault is applied to all the three versions, what we are interested in measuring is that how often the simulated fault will produce identical results which can confuse the DM. Given two incorrect but equivalent result, the voter will pick a result that is different from what it would have selected had fault injections not occurred.

Fault-injection technique allows us to simulate imaginable and unimaginable event to show

how tolerant the system is. We can take the appropriate steps to improve the tolerance of the software if the results from the simulation are unacceptable. In this manner, fault injection plays an important role of measuring fault tolerance.

Other Techniques

Besides the basic techniques for measuring software fault tolerance above, there are still some other techniques proposed: Monte Carlo simulations; Markov chains / Markov graphs [20].

4.3 Conclusion

We have surveyed the traditional techniques of achieving and measuring software fault tolerance. In the achieving part, techniques belonging to design diverse and data diverse are mainly surveyed; in the measuring part, basic reliability models and fault injection are surveyed.

Since the growing demand for high reliability in many systems, software fault tolerance techniques will play an even more important role in the near future. We should select different techniques to achieve and measure software fault tolerance, depending on the application's requirements, available resources and characters of each different techniques.

4.4 Questions

1. In the RcB (recovery blocks) technique, why are variants arranged depending on their efficiency in a decreasing order?
2. Please compare RcB and NVP on the four phases in software fault tolerance.
3. In reliability models, why do we say state that the transition model is “better” than conventional reliability model at some point?

Bibliography

- [1] R. J. Abbott. “resourceful systems for fault tolerance, reliability, and safety,”. *ACM Computing Surveys*, 22(3):35–68, 1990.
- [2] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.
- [3] A. Athavale. “performance evaluation of hybrid voting schemes”. Master’s thesis, North Carolina State University, Department of Computer Science, 1989.

- [4] A. Avizienis. On the implementation of n-version programming for software fault-tolerance during execution. *IEEE Transactions on Software Engineering*, pages 149–155, 1977.
- [5] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *IEEE Computer*, 17(8):67–80, 1984.
- [6] L. Chen and A. Avizienis. “n-version programming: A fault-tolerance approach to reliability of software operation,”. In *Proceedings of FTCS-8*, 1978.
- [7] W. R. Elmendorf. ““fault-tolerant programming,””. In *Proceedings of FTCS-2*, pages 79–83, 1972.
- [8] H. Hecht. Fault-tolerant software for real-time applications. *ACM Comput. Surv.*, 8(4):391–407, 1976.
- [9] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell. A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium*, pages 171–187, London, UK, 1974. Springer-Verlag.
- [10] Shostak Lamport and Pease. The byzantine generals problem. *ACM Transaction on Programming Languages and Systems*, 1982.
- [11] Michael R. Lyu, editor. *Software Fault Tolerance*, pages 189–209. John Wiley & Sons, Inc., New York, NY, USA, 1995.
- [12] Jeffrey M. Voas & Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors. in Quality Software Project Management*, page 177. Prentice Hall, 2001.
- [13] Dhiraj K. Pradhan, editor. *Fault-tolerant computer system design*, page 451. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [14] Dhiraj K. Pradhan, editor. *Fault-tolerant computer system design*, page 453. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [15] L. L. Pullum. “a new adjudicator for fault tolerant software applications correctly resulting in multiple solutions”. In *Proceedings: 12th Digital Avionics Systems Conference*, 1993.
- [16] Laura L. Pullum. *Software fault tolerance techniques and implementation*, chapter 4. Artech House, Inc., Norwood, MA, USA, 2001.
- [17] Laura L. Pullum. *Software fault tolerance techniques and implementation*, chapter 5. Artech House, Inc., Norwood, MA, USA, 2001.

- [18] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.
- [19] J. W. Gault Scott, R. K. and D. F. McAllister. “the consensus recovery block,”. In *Proceedings of the Total Systems Reliability Symposium*, 1983.
- [20] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems (2nd ed.): design and evaluation*, pages 305–342. Digital Press, Newton, MA, USA, 1992.
- [21] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems (2nd ed.): design and evaluation*, pages 285–286. Digital Press, Newton, MA, USA, 1992.
- [22] T.Anderson/P.A.Lee. *Fault Tolerance - Principles and Practice*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1981.

Chapter 5

Jay Parlar: Dynamic Languages

Dynamic languages have a long history in computation, going back as early as the initial Lisp implementations in the 1950s. More recently, dynamic languages have been labeled “scripting languages”, and their use in production systems has been limited. One reason for this is the inherently slower execution speed of many dynamic languages. However, various type inference techniques have been developed which drastically increase the execution speed of some dynamic languages. Even when such techniques are not possible, there are still many reasons why a dynamic language would make a good choice for program development.

5.1 Introduction

Dynamic Languages are typically viewed as those languages that are dynamically typed, and high level [3]. In general, a dynamic type system is one where the type of any given variable or method will not be known at compile time.

Instead, all type information is determined dynamically at run time. In fact, the type information for a variable can change many times during a single execution of the program. In addition, explicit declarations of variable types are not made (as would be the case in traditional statically typed languages like C++, Java and Pascal).

For example, we have the following arbitrary Ruby code:

```
def somefunction(x)
  if x > 1 then
    y = 5
  else
    y = "abc"
  end
  return y
end
```

There are four key points to note about this example, that mark it as code from a dynamic language:

1. No type is given for the function argument x
2. No variable declaration for y is given
3. Depending on the result of the conditional $x > 1$, y will be assigned an R-value [25] of type String or type Integer.
4. No return type is given for `somefunction`. In fact, depending on what y is, the function could return an integer value, or a string value.

It should be noted that points 1, 2 and 4 could also apply to polymorphic functions in statically typed languages. However, it is point 3 in combination with the other points that strongly identifies this as a dynamically typed language.

Strachey [25] defined this nicely as:

...types are to be attributes of R-values only and that any type of R-value may be assigned to any L-value.

Having to wait until runtime implies a stronger possibility of having something going wrong with program execution. However, with most dynamic languages, the flexibility allowed by their dynamism will, in many cases, outweigh the cost of losing compile time type checking.

5.2 History of Dynamic Languages

If one's definition of a dynamic language is simply that type information of a variable is not known at compile time, then assembly language could be viewed as a dynamic language! Assembly language works purely with bit strings in computer memory, and what those bit strings represent at any time is only known to the programmer.

Bit strings in computer memory are known as an “untyped universe” [6]. Other examples [6] include S-expressions in pure Lisp, λ -expressions in λ -calculus and sets in set theory.

However, most true dynamic languages do have an actual type system, where the type is stored with the R-value.

Probably the oldest dynamic language is Lisp, which had its first implementation in 1958 [11]. Lisp has just a few basic types, namely lists and atoms, and from atoms, numbers and symbols are defined [27]. From these basic types, higher level types can be defined. Even with higher level types, Lisp still manages to operate without explicit type declarations (although using Common Lisp's optional declarations [17], one can introduce optional type declarations [16]).

Smalltalk, one of the earliest object oriented languages, and one of the earliest dynamic languages, saw its first version in 1972, with another major release (Smalltalk-76) in 1976 [12]. Smalltalk is still in use today, being used to develop modern systems, such as web application frameworks [24]. Interestingly, Smalltalk was always used to design “real” systems. In

later years though (esp. the early 1990s), most dynamic languages were called “scripting languages”, and not used to design full programs. Instead, they were used for simple scripting tasks.

Traditionally, dynamic languages have been called “scripting languages”, as in the past they have only been used for scripting predefined components of a system. Probably the most famous and heavily used of these is Perl, created in 1987 by Larry Wall [15]. Perl was developed by Larry Wall out of a need to perform various scripting tasks on text files. This heritage still holds today.

For whatever reason (probably much to do with the typical uses of Perl), many people in the 1990s would not even contemplate using dynamic languages for “production” or “enterprise” tasks. There was a common notion that they were too slow and/or unsafe to use in such environments [19].

In just the past few years though, there has been a resurgence in the interest of dynamic languages (including Lisp), lead greatly by languages such as Python and Ruby, which were designed as “general purpose” languages, as opposed to specialized scripting languages [3].

5.3 Type Systems

In general [6]:

A type system has as its major purpose to avoid embarrassing questions about representations, and to forbid situations where these questions might come up. In mathematics as in programming, types impose constraints which help to enforce correctness.

So the main goal of a type system is for protection of the programmer, to help prevent errors. However, it must be stressed that no type system can guarantee that no errors will occur at runtime [14]:

Type systems are always prey to the Halting Problem. Consequently, a type system for a general-purpose language must always either over- or under-approximate: either it must reject programs that might have run without an error, or it must accept programs that will error when executed.

In addition to helping to avoid variable representation problems, a type system can also be used to increase performance. For example, by identifying integer variables at an early stage, a compiler can ensure that arithmetic is performed on them at the assembly level [7].

As a simple example, note the following C code:

```
int x,y,z;
x = 1; y = 2;
z = x + y;
```

Using GCC3.3, on PowerPC, the compiler outputs (without optimization) the following assembly fragment:

```
li r0,1
stw r0,32(r30)
li r0,2
stw r0,36(r30)
lwz r2,32(r30)
lwz r0,36(r30)
add r0,r2,r0
stw r0,40(r30)
mr r3,r0
```

The compiler knows that all variables in the code are simple integers, thus no runtime type checking needs to be done, and the basic assembly instruction `add` can be used to add them together.

Compare this with the following Python code:

```
def simple_int_addition():
    x = 1
    y = 2
    z = x + y
```

In the Python interactive interpreter, one can view the byte code that will be generated for this method (an excerpt from the interpreter session is shown):

```
>>> import dis
>>> dis.disassemble(simple_int_addition.func_code)
 2          0 LOAD_CONST          1 (1)
          3 STORE_FAST          1 (x)

 3          6 LOAD_CONST          2 (2)
          9 STORE_FAST          0 (y)

 4          12 LOAD_FAST          1 (x)
          15 LOAD_FAST          0 (y)
          18 BINARY_ADD
          19 STORE_FAST          2 (z)
```

On first glance, this seems roughly equivalent to the assembly code generated for the C program. However, the `BINARY_ADD` opcode does not simply cause an assembly level `add` to be performed [23]. Instead, it will cause the Python interpreter to analyze the types of both variables, check (through dynamic lookup) if an addition method is defined for those two types, and then perform the addition.

As a small example of the potential performance benefits of typing, in [10] there is a comparison between a prime number generator written in Python, and one written in C. With Python, one has the option of coding performance critical modules and functions in C, and calling those C modules from Python.

For the prime number example, two versions were written, one in pure Python, the other a Python program calling a C module.

The time taken for the pure Python version to calculate the first 1000 prime numbers was 0.317926 seconds. The version that had a Python program importing a C prime number generator took only 0.027067 seconds, a 12 times speed increase.

Not all dynamic languages (including Python) will be this slow in all cases, but the example does give a good idea of what kind of performance one can expect.

Static and Dynamic Typing. In a static type system, types are “are associated with constants, operators, variables, and function symbols” [6]. The implication of this is that at all times during program execution, the type of *anything* in the program is known, without having to do any explicit checking.

Some statically typed languages, such as C, C++ and Java, require that the programmer enter the type information themselves, while languages like ML and Haskell can infer types at compilation time (this will be discussed in greater detail later).

Dynamic typing is a system in which no type information is explicitly given for variables, and none is automatically inferred. Instead, variables are created “on-the-fly”, with type information attached to the R-value. A variable can freely change its type by giving it a new R-value with different type information attached.

Another signature of a dynamic type system is that function signatures do not have type information associated with their arguments. Take the following Python code as an example:

```
def foo(x):
    return x + 1
```

There is no type information given in the signature of `foo` to indicate the type of object `x` is supposed to be. In this example, one must inspect the function itself to determine what kind of objects are acceptable (in this case, that being any object that can be summed with an integer).

It must be noted that the type information *is* present at runtime, otherwise the Python interpreter would not be able to determine *how* to add together `x` and the integer 1. This can be explicitly seen in the following example, which again shows captured output from the interactive interpreter:

```
>>> def foo(x):
...     print "Type of x is", type(x)
...     return x + 1
...
>>> y = 5
```

```
>>> type(y)
<type 'int'>
>>> print foo(y)
Type of x is <type 'int'>
6
```

Type Inferencing. Having a purely dynamic type system usually implies certain performance constraints. In general, the type of an object not being known until runtime prevents the kind of compile time optimizations one can achieve with a static language.

However, there have been many attempts at addressing these performance constraints, particularly through the use of type inference.

Type inference is defined as [5]:

... the process of finding the most accurate type information for a program that does not explicitly state the types of variables at compile-time, all without inferring any inaccurate information

Compile time type inferencing is already used successfully in many statically typed functional languages, such as Haskell and Standard ML. However, compile time type inferencing in dynamic languages presents a slightly different problem, as in many cases it is impossible to know ahead of time what type a variable will be (usually thanks to `eval` and its cousins).

Even the notion of type inferencing a static *imperative* language has different concerns than that of a functional language (mostly due to the ability to have assignment statements). Suzuki [26] solved this imperative type inference problem in the Smalltalk language, basing his research off the work done by Milner [18] for the ML language.

It is theoretically possible to write a program which asks a user to type in a new class name, and the program will then dynamically create a class definition and instantiate a new variable of that class. Obviously, a type inferencer cannot anticipate that, not without being able to read a user's mind.

However, there are occasions where type inferencing can work. In the `simple_int_addition` example above, a type inferencer should be able to determine at compile/analysis time that `z = x + y` will only ever represent the addition of two integers, due to the two integer assignments of `x` and `y` that occur directly above it.

Brett Cannon attempted to build a compile time type inferencer for Python as part of his Master's thesis [5]. The conclusion he came to was that without restricting *any* of the dynamism of Python, and without changing the language's syntax or semantics, the most type inferencing that could be done is of atomic variables in local namespaces (ie. local variables in method bodies).

So if a method as follows was given:

```
def dosomeinference(x):
    a = 5
    b = a * x
    return b
```

The only aspect of that method that could be inferred is the type of `a`, as it is a local variable, and works with an atomic type (integer). The variable `b` involves a variable of unknown type, thus its evaluation would still have to be done dynamically at runtime.

The result of his experiment adding the type inferencer to the Python interpreter was a speedup of less than 2%, which at the same time added a great deal of complexity to the source code of the Python interpreter.

Not all dynamic languages have such problems with type inferencing. The Self language has now been using type inferencing for some time [2, 7].

The original Self compiler performed Cartesian Product type inference [2] only. Cartesian Product type inference works on method calls to improve performance of polymorphic methods.

It works by analyzing all occurrences of a method call, and computing a Cartesian product of the types of the arguments, of all calls. Then each tuple in the Cartesian product can be analyzed separately, essentially creating a list of all the *monomorphic* method calls that are possible.

In [7], enhancements were made to the Self compiler to add iterative type analysis. Iterative type analysis performs analysis on method bodies, as opposed to method calls. The addition of iterative type analysis to the Self compiler caused a two-fold speed increase over the previous compiler.

The reason that these algorithms worked so well for Self, but not for Python, has to do with runtime capabilities of the languages. The example given earlier of a program that prompts a user for a new class name illustrates this perfectly. At compile time, all the information that will be used at runtime is not necessarily present in a Python program.

Cartesian product type inference and iterative type analysis require all source code to be available at compile time, and that is not guaranteed with a Python program, or any program written in a language that can generate code at runtime.

The Python community has responded to these issues with a few separate projects, but three of the most interesting are:

1. Psyco [23]
2. ShedSkin [8]
3. PyPy¹

Psyco. The Psyco project is essentially a Just-in-time [4] (JIT) compiler for Python. Instead of analyzing source code before execution, it analyzes running code during execution. When it sees sections of code that can be replaced by its own specially written assembly instructions, it will step in, and run the assembly code in place of having the interpreter perform its normal execution.

¹<http://codespeak.net/pypy/dist/pypy/doc/news.html>

In some cases (particularly integer-arithmetic-intensive programs), the speedup has been as much as 100 times. However, there are cases where no portions of a program can be aided by the Pscyo JIT, and the overall result is a slower program.

Luckily, a programmer has the option of profiling their code, and having Psyco only apply itself to the sections of the program they feel would benefit from the JIT.

ShedSkin. ShedSkin is a project aimed at producing a high-speed Python-to-C++ compiler. The project uses various type inference techniques to analyze Python source code, and generate highly efficient C++ code.

Developed as part of the author's Master's thesis, the author has developed it with a strong awareness of the existing literature (taking advantage of many of the references given here, [26, 18, 2]. It is still in its early stages, but sample programs have already seen as much as a 65-fold speed increase ².

However, because of its early stages, the compiler can only operate on a small subset of Python code. No code from the standard library can be used, metaclasses cannot be used, nor can any of Python's powerful reflection capabilities.

PyPy. The PyPy [21] project is an attempt to implement the Python language, in Python itself!

The reason for this is that current Python interpreters are written in C and Java, both of which are lower-level languages than Python (due to their limited basic data types, forced explicit type declarations, early binding, etc.). An implementation of Python written in a higher-level language, like Python, would result in a code base for the interpreter that's much easier to understand and work on.

A second goal of the PyPy project is for automated interpreter generation. With that, people could automatically generate Python interpreters written in, for example, Lisp, or C++, or JavaScript, etc.

To facilitate these goals, the PyPy project has specified a subset of Python called "RPython". This subset eliminates many of the issues that prevent type inferencing from working on regular Python code. With this, the new Python interpreter can be written in RPython, and then type inferencing can be combined with the automated backend generators to produce reasonably fast Python implementations.

PyPy probably has the strongest chance of becoming the community accepted high-speed compiler. The PyPy community has strong ties to the core Python developer community, and they have been awarded funding from the European Union to continue their development [8].

5.4 Disadvantages of Dynamic Languages

Throughout the history of dynamic languages, they have traditionally had a number of disadvantages compared to more traditional static languages.

²<http://shed-skin.blogspot.com/2006/01/006-update.html>

Execution Speed. The topic of Execution Speed has been covered quite thoroughly in section 5.3. However, it bears repeating. Typically, dynamic languages will be slower, in terms of raw execution speed, compared to a statically compiled language performing the same task, assuming both languages have adequate interpreters/compiler [3].

In applications where raw performance is required (including many numeric applications, high quality graphics rendering, etc.), the choice to use a dynamic language may result in problems down the road.

Somewhat related to execution speed is tasks that are very low-level (drivers for an OS, etc.). It is usually preferable to write these types of applications in C or assembly, as one often needs access to parts of a system that a high-level dynamic language might abstract away (such as memory management, CPU registers, etc.).

Type Errors. There are a certain class of errors that occur in dynamic languages that will not occur in static languages, namely type errors.

An example of a Python interactive interpreter session is shown, illustrating the simplest example of this:

```
>>> def f(x):
...     return 1 + x
...
>>> f("abc")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in f
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

So a function `f` was created which accepts a single argument `x`, and sums that argument with 1 (or more precisely, applies 1 and `x` to the `+` operator, which is polymorphic).

The `+` operator is not defined when the two objects passed to it are a string and an integer. Because of this, a `TypeError` is raised at runtime.

There would be no way to detect this error without actually running the program. Thus, if a given program ended up being released, without certain code execution paths being tested, it is possible that a type error could occur at runtime.

Another class of runtime error that is closely related has to do with typos, or missing variables. As an example:

```
>>> def g():
...     y = 1
...     x = 2
...     return y + z
...
>>> g()
Traceback (most recent call last):
```

```

File "<stdin>", line 1, in ?
File "<stdin>", line 4, in g
NameError: global name 'z' is not defined

```

In this example, we define a function `g` that defines two local variables, and is *supposed* to sum them and return the result. However, the programmer made a mistake, and instead of typing `y + x`, they typed `y + z`.

The interpreter lets the function be defined without complaint, and it is not until some piece of code actually executes the function that we will see a mistake has been made.

As a final example of a problem a compiler would catch:

```

>>> z = 5
>>> def g():
...   y = 1
...   x = 2
...   return y + z
...
>>> g()
6

```

Here the same function `g` has been defined, but a variable `z` is available in the global namespace. Because of this, the addition goes through without a runtime error. While many static languages would allow the addition with a global variable, most would emit a compiler warning that the variable `x` had gone unused in the function.

Scalability. A common criticism of dynamic languages is that they do not scale very well. For example both Ruby and Python [13] can only utilize one processor at a time, when programming with a threaded model. So even on a multiprocessor computer, only one Ruby/Python thread (per process) will be able to run.

Of course, this does not prevent a programmer from using a multi-process model (as opposed to multi-threaded), creating separate instances of the interpreters for each process.

The other issue of scalability that is often considered is in terms of the system size [3]. Many believe that as systems get larger, static interface checking will be the only way to make sure everything still works correctly together.

The fact that method signatures give no type information can be hinder-some, for example, consider the following Ruby class:

```

class DataModel
  def addMoreData(data)
    ... do some stuff ...
  end
end

```

By the method signature of `addMoreData`, there is no syntactic way to tell what *type* of data should be passed in. In some ways, this breaks classic information hiding [20] strategies, as one must go into the code of the method to ascertain what is required.

5.5 Advantages of Dynamic Languages

While as with any technology, dynamic languages present a number of disadvantages, there are a number of advantages to using dynamic languages, which have made them popular for so long now.

Scripting. The classic use for dynamic languages, and the one most people are familiar with, is for scripting [19].

Due to the fact that type declarations are not required, dynamic languages are usually incredibly flexible, making them perfect for small scripting tasks.

One trait most scripting languages possess is easy-to-use file and string processing constructs. Many scripting tasks deal with parsing over some files, and checking the string contents of them, so scripting languages try to make those tasks as easy as possible.

Consider the following Python script which iterates over all the lines of all the files named on standard input, and replaces the string “foo” in those lines with the string “bar”:

```
import fileinput,sys
for line in fileinput.input(inplace=True):
    if "foo" in line:
        line = line.replace("foo","bar")
    sys.stdout.write(line)
```

Now try to imagine how much longer the equivalent program would be in C++, Java, or most other static languages. Just the syntactic support for checking substrings (ie. “foo” in line) is something that gives Python an advantage over *most* static languages, languages which would usually require some explicit function calls and comparisons. The Python approach is closer to natural English, and easier to read.

Or consider the following Tcl command (from [19]):

```
button .b -text Hello! -font {Times 16} -command {puts hello}
```

This command dynamically creates a new button object, which displays a string of text in a 16 point Times font, and prints out the message “hello” when a user clicks on it.

An equivalent command in C++ using Microsoft Foundation Classes would require 25 lines of code, and multiple functions. Even simply setting the font is a tedious task [19]

```
CFont *fontPtr = new CFont();

fontPtr->CreateFont(16, 0, 0,0,700, 0, 0, 0, ANSI_CHARSET,
```

```

OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
DEFAULT_PITCH|FF_DONTCARE, "Times New Roman");

buttonPtr->SetFont(fontPtr);

```

So for simple scripting tasks, dynamic languages have proven themselves to be ideal, with most static languages being much too heavy-weight relative to the importance of the job.

Built-in Types. Most dynamic languages come with high-level built-in types, such as hash tables, dynamically extensible lists, strings with dozens of useful methods, etc. In static languages, types like this are usually implemented as classes in a library, if at all.

Python, for example, has built in dictionaries (hash tables). These are a part of the language definition, and have their own special syntax to create them, namely `{}`. The reason this is so important is that having syntactic support for high-level data-types that will be used over and over reduces the amount of “finger-typing” needed. Code becomes more compact, concise, and easier to read.

Python dictionaries can take almost any object as keys (and all the keys do *not* have to be of the same type), and they can take *any* object as the paired values, and again, the values in a dictionary do not have to be of the same type.

```

>>> contain = {}
>>> contain[3] = "ABCD"
>>> contain["foo"] = 9999
>>> "foo" in contain.keys()
True
>>> "bar" in contain.keys()
False
>>> 9999 in contain.values()
True
>>> contain[3]
'ABCD'
>>>

```

This short example showed a few of the interesting features of Python dictionaries.

1. A dictionary named `contain` was initialized.
2. A value "ABCD" was added, with a key of 3
3. A value 9999 was added, with a key of `foo` (notice that the value has a different type than the value added in the previous step)

4. `"foo" in contain` asks if the key `"foo"` is in the dictionary
5. `"bar" in contain` asks if `"bar"` is in the dictionary. Since we haven't added it, `False` is returned.
6. `contain[3]` asks the dictionary for the value paired with key `3`

This example only showed a few of the uses and methods of dictionaries, but it is easily seen how usable they are, with minimum effort.

Code Compactness. Dynamic languages, for a variety of reasons, tend to involve higher degrees of code compactness than many static languages. In general, this relates to the fact that dynamic languages tend to be higher-level, with higher-level constructs available to aid in reducing code size. The widely accepted estimate is that a static language will require 5-10 times the number of lines of code than a dynamic language implementing the same functionality [9].

One study [22] did an empirical comparison of scripting languages versus traditional static languages (Java, C, C++) and found that the working with the scripting languages took no more than half the time than doing an equivalent program in one of the static languages, and the resulting programs were half as long.

As an example (from [9]) of this, we will show a simple class and its constructors in Java, and compare that with equivalent code in Python.

Java:

```
public class Employee
{
    private String myEmployeeName;
    private int    myTaxDeductions = 1;
    private String myMaritalStatus = "single";

    //----- constructor #1 -----
    public Employee(String argEmployeeName)
    {
        myEmployeeName = argEmployeeName;
    }

    //----- constructor #2 -----
    public Employee(String argEmployeeName,
                    int    argTaxDeductions)
    {
        myEmployeeName = argEmployeeName;
        myTaxDeductions = argTaxDeductions;
    }

    //----- constructor #3 -----
    public Employee(String argEmployeeName,
                    int    argTaxDeductions,
                    String argMaritalStatus)
    {
        myEmployeeName    = argEmployeeName;
        myTaxDeductions   = argTaxDeductions;
        myMaritalStatus   = argMaritalStatus;
    }
}
```

Python:

```
class Employee(object):
    def __init__(self,
                  argEmployeeName,
                  argTaxDeductions = 1,
                  argMaritalStatus = "single"):

        self.EmployeeName = argEmployeeName
        self.TaxDeductions = argTaxDeductions
        self.MaritalStatus = argMaritalStatus
```

This class creates an `Employee` object, with three attributes:

- `myEmployeeName`
- `myTaxDeductions`
- `myMaritalStatus`

Each of the attributes is given a default value. We want, on construction of the class, to *require* that a name is given to the instance, but allow the `argTaxDeductions` and `argMaritalStatus` status fields to be optional.

Obviously, the Python equivalent is much shorter. Instead of having to build separate constructors, the optional arguments to the constructor have been declared as *keyword arguments*. This means they are given a default value if not explicitly used when creating the instance.

So for example, the following instantiations could be performed:

```
>>> bob = Employee("Bob")
>>> jane = Employee("Jane", argMaritalStatus="married")
>>> bill = Employee("Bill", argTaxDeductions=0, argMaritalStatus="married")
```

While this is obviously just one particular example, showing how fewer constructors are usually required, it accurately represents the general trend of dynamic languages. Namely, that over the course of a program, the number of lines required will end up being much fewer than in a static language (for a variety of reasons).

5.6 Interactive Interpreter

A final aspect of many dynamic languages, that should be discussed in more detail, is the interactive interpreter.

An interactive interpreter is a program that provides the user with a simple prompt. From that prompt, they can begin executing statements of the language. Programs that work almost exclusively from the prompt of an interactive interpreter include Maple³ and Matlab⁴.

The basic Python interactive interpreter prompt, on startup, will look something like this:

```
Python 2.4.1 (#2, Mar 31 2005, 00:05:10)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

³<http://www.maplesoft.com/>

⁴<http://www.mathworks.com>

The >>> characters indicate that the prompt is waiting for user input. In Ruby and Common Lisp, respectively, we see:

```
irb(main):001:0>
```

and

```

i i i i i i i      ooooo  o      oooooooo  ooooo  ooooo
I I I I I I I      8      8  8      8      8  o  8      8
I \ '+' / I        8      8      8      8      8      8
\ '-+-' /          8      8      8      ooooo  8oooo
  '-_||_-'         8      8      8      8      8
      |             8      o  8      8      o  8  8
-----+-----    ooooo  8ooooooo  ooo8ooo  ooooo  8

```

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
 Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
 Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
 Copyright (c) Bruno Haible, Sam Steingold 1999-2000
 Copyright (c) Sam Steingold, Bruno Haible 2001-2005

```
[1]>
```

Dynamic languages are not the only languages though that provide an interactive prompt. Many functional languages, such as Haskell, also provide an interactive prompt. However, the interactive prompts of functional languages do not always give the programmer the full power of the language. For instance, some Haskell interactive prompts do not let you define new functions, while some allow new functions to be defined, but only with the `let` statement.

Dynamic languages, because of their very nature, do not require this. Take Common Lisp for example, where we can define new functions right inside the interpreter:

```
[1]> (defun add5 (x) (+ x 5))
ADD5
[2]> (add5 10)
15
```

Or take Python, where one can even define entirely new classes in the interpreter, and create instances of them right there:

```
>>> class Test(object):
...     def foo(self):
...         print "Someone called 'foo'"
```



```
...
>>> t = Test()
>>> t.foo()
Someone called 'foo'
```

The power of this feature should be obvious: It allows the programmer to quickly prototype algorithms and data structures they wish to use in their actual program. This enables a form of iterative development that is not directly possible with most static languages.

In addition, the interpreters usually let the programmer import files they've created for some program, and then perform interactive tests on the structures and functions contained in those files.

Dynamic Typing. Just as dynamic typing has been shown to have disadvantages, it also comes with several strong advantages which make it quite appealing to have.

Often times, dynamic typing is called “duck typing” [1]. The basic premise behind duck typing, is that as long as an object has a method of the correct name, that takes the correct number of arguments, then it will be called, no matter what type the object itself is. In essence, “If it looks like a duck, and quacks like a duck, it must be a duck” [1].

An example of this is as follows:

```
>>> class A(object):
...     def text(self):
...         print "Class A 'text'"
...
>>> class B(object):
...     def text(self):
...         print "Class B 'text'"
...
>>> def call_text(x):
...     x.text()
...
>>> call_text(A())
Class A 'text'
>>> call_text(B())
Class B 'text'
>>>
```

In this example, we declared two different classes, A and B. Neither inherits from the other, and they have no common superclass (other than `object`, which is the superclass of every class in Python). The function `call_text` takes a single argument, and on that argument attempts to call the `text` method.

The point here is that `call_text` doesn't care at all about the type of `x`. All it cares about is that `x` has the `text` method. If that method is not present, then a runtime `AttributeError` will be raised.

The key reason one would want this functionality is flexibility. While in a language like Java or C#, one could define either a base class, or even better, an interface, the end result is not exactly the same.

The reason is that one does not always know which methods they may want to use, beforehand. In the Java SWT library, a large number of widgets have a `Text` property. The Java designers decided that it is a reasonable field name, and have stayed consistent with it in defining the SWT library. A Java interface `IText` could be created, but then every class that implements `Text` would have to also declare that it implements that interface, in addition to every other interface the class might already implement.

An interface could instead be declared that encapsulated the `Text` property and other properties, for the purpose of reducing the number of interfaces required. Then, *every* class that wanted to have `Text` would also have to implement the other methods in the interface, whether or not they were actually useful.

While this might be viewed as “safer”, it is restricting to the programmer. A key notion of dynamic languages in general is flexibility, trying to prevent the compiler from “fighting” the programmer as is often the case in static languages.

Whether exchanging some perceived safety for increased flexibility is a good thing, is up to each individual programmer to decide. However, many dynamic language advocates have years of personal empirical [22] data to show that the benefit of flexibility vastly outweighs the cost of lost safety, in the long run.

That is the question in general with dynamic languages. It is not one feature in particular that makes dynamic languages “better” or “worse” than static languages. Instead, it is the combination of features available in dynamic and static languages that will affect how an individual programmer views them, and decides how to use them.

It should be noted that there are some static languages (notably Objective Caml) that provide type inferencing systems which bring their static type systems very close to dynamic type systems, in terms of flexibility. However, these languages also tend to be functional languages, which for a variety of reasons (good and bad), have not become a viable option for a majority of programmers.

5.7 Conclusion

A brief introduction to dynamic languages, and what separates them from more traditional static languages, has been given. To accompany this, a short history of dynamic languages has been given.

The key separating feature is the type system. Static type systems can often be used to increase execution speed of a program, and this is one of the reasons that dynamic languages are typically viewed as “slow”. However, some dynamic languages can take advantage of various type inferencing techniques, to greatly improve run-time execution speed.

Not all languages can though. Python, for instance, has proved to be *too* dynamic to work well with most type inferencing techniques. Type inference can be applied reasonably successfully, but only if certain aspects of dynamism are removed.

In addition, a bevy of other disadvantages and advantages of dynamic languages have been covered, including run-time safety and built-in types, respectively.

A brief introduction to the uses of interactive interpreters was given. While interactive interpreters are often present in statically compiled languages, they tend not to be as flexible as the interactive interpreters of purely dynamic languages.

Finally, the benefits of dynamic typing for flexibility, as opposed to interfaces and subclassing, were discussed. Many people find that losing some potential safety is more than made up for by the flexibility provided by dynamic type systems.

5.8 Exam Questions

1. Explain one disadvantage of using a dynamic language
2. Explain one advantage of using a dynamic language
3. Explain why type inferencing can't work in some dynamic languages

Bibliography

- [1] Python tutorial. Technical report, Python Software Foundation, 2005. <http://docs.python.org/tut/node18.html>, Last accessed March 24, 2006.
- [2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, London, UK, 1995. Springer-Verlag.
- [3] David Ascher. Dynamic languages - ready for the next challenges, by design. Technical report, ActiveState, 2004. http://www.activestate.com/Company/NewsRoom/whitepapers_ADL.plex, Last accessed March 21, 2006.
- [4] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.
- [5] Brett Cannon. Localized type inference of atomic types in python. Master's thesis, California Polytechnic State University, 2005.
- [6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17:471–522, 1985.
- [7] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *LISP AND SYMBOLIC COMPUTATION: An International Journal*, 1991.
- [8] Mark Dufour. Efficient implementation of modern imperative languages; application to python. June 2005.

- [9] Stephen Ferg. Python and java: a side-by-side comparison. http://www.ferg.org/projects/python_java_side-by-side.html, Last Accessed March 16, 2006, February 2004.
- [10] Pradeep Kishore Gowda. Speed up python with pyrex. *DeveloperIQ*, April 2005. http://www.developeriq.com/articles/view_article.php?id=469, Last Accessed March 21, 2006.
- [11] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of lisp. *ACM SIGPLAN Notices*, 28, March 1993.
- [12] Alan C. Kay. The early history of smalltalk. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 69–95, New York, NY, USA, 1993. ACM Press.
- [13] Ken Kinder. Event-driven programming with twisted and python. *Linux Journal*, 2005(131):6, 2005.
- [14] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2006. <http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/PDF/plai-2006-01-15.pdf>, Last accessed March 21, 2006.
- [15] Tom Christiansen Larry Wall and Jon Orwant. *Programming Perl: 3rd Edition*. O'Reilly & Associates Inc., 2000.
- [16] LispWorks. *CLHS: Declaration TYPE*, 2005. <http://www.lispworks.com/documentation/HyperSpec/Body/d.type.htm>, Last accessed March 21, 2006.
- [17] LispWorks. *CLHS: Symbol DECLARE*, 2005. <http://www.lispworks.com/documentation/HyperSpec/Body/s.declar.htm>, Last accessed March 21, 2006.
- [18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [19] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31:23–30, March 1998.
- [20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [21] Samuele Pedroni. Compiling dynamic language implementations. Technical report, 2006. <http://codespeak.net/pypy/dist/pypy/doc/dynamic-language-translation.html>, Last accessed March 21, 2006.
- [22] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.

- [23] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.
- [24] A. Lienhard S. Ducasse and L. Renggli. Seaside - a multiple control flow web application framework. In *Proceedings of ESUG Research Track 2004*, pages 231–257, September 2004. <http://www.iam.unibe.ch/~scg/Archive/Papers/Duca04eSeaside.pdf>, Last accessed March 21, 2006.
- [25] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13:11–49, 2000.
- [26] Norihisa Suzuki. Inferring types in smalltalk. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 187–199, New York, NY, USA, 1981. ACM Press.
- [27] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP 3rd Edition*. Addison-Wesley, 1989.

Chapter 6

Yu Wang: A Survey of Software Distribution Formats

Nowadays various computer architectures and operating systems are developed. Software development does not focus on any single of them anymore. Instead, software companies and distributors concerns more about how their software system can be deployed as universal as possible.

We call both architectures and operating systems *platforms*. Without a multi-platform solution, assume we have M instruction set architectures, to design a software that is executable over all the possibilities, we will have to compile the software M times or even more, due to combination of different hardware components. Whereas with a multi-platform solution, the software needs only to be written once, and by techniques of the solution, it is able to achieve the same functionality as if the code is compiled M times in previous case.

We define *software distribution format* to be any form in which software is distributed. In this article, we give discussion on several software distribution formats, which are *fat binary*, *software for virtual machines* and *source code distribution*, where the topic of virtual machines are divided into *application oriented* and *platform oriented*. In term of compilation, these distribution formats can be considered as *fully compiled*, *half compiled* and *not compiled*.

At the end, comparison on these formats is drawn in concluding remarks.

6.1 Fat Binary

An immediate solution to universal software distribution is to have the source code compiled into several binaries against different architectures, and have all of them available to the computer of end user, so that the binary selectively installed based on the architecture of that computer. This solution is known as *fat binary* [25] or *universal binary* [1], such that compiled binaries are archived and compressed into a single package and allow the system to choose which binary should be installed according to its architecture at the install time.

Due to multi-architecture packaging in fat binary, one disadvantage is that the size of the package to be distributed tends to be larger than other solutions introduced in this article.

A fat binary for software with size s often ends up with a size of $M \times s$, for M instruction set architectures. But because fat binary format is easy to be applied technically when producing software, this technique is still widely used, such as the instance that it makes Apple Computer's migration from PowerPC architecture to X86 architecture in 2005 much smoother [26].

A similar solution to fat binary is the deployment of PocketPC applications using *Microsoft Installer* format (MSI) [2]. A PocketPC is a handheld computer running WindowsCE based operating system by Microsoft and was originally developed in November of 1996. Typical steps of installing an application into a PocketPC are that first have the setup package installed on a desktop workstation, and then transfer the *cabinet binary file* (CAB) when synchronizing with the PocketPC device when it gets connected [14].

Due to historical reason, current PocketPC devices come with processors in different architectures, such as Intel XScale¹, Hitachi SH3² and MIPS³, which raise the problem of compatibility when distributing software applications. Since the computing performance of handheld devices are not as powerful as desktop computers, the solution should be as much independent of the processor as possible. In this case, fat binary is most appropriate.

In term of compilation, fat binary can be considered as *fully compiled* distribution format. Other than universal distribution for different architectures, fat binary is also used for other purposes, such as applications with different localized versions. Such cases are rare and are out of the scope of our topic.

6.2 Application Oriented Virtualization

To make the software adoptable for different platforms, one can consider a machine that virtually exists on top of each platform, such that the software only needs to be written for this virtual machine and it is able to be executed on every platform from the point of view from users. Here we classify modern virtual machines into two types, *application oriented virtualization* and *platform oriented virtualization* which is discussed in next section.

We define *application oriented virtual machine* to be virtual machine which provides a set of non-native instructions and allows applications, which is compiled against to this instruction set, to be launched and executed in the virtual machine. The language of such instruction set is commonly referred as intermediate language. The code in intermediate language is called *intermediate code*. An application oriented virtual machines bridges the underlying platforms and its applications, to allow applications running on multi-platforms.

Modern application oriented virtual machines are either *abstract stack machines* or *register-based machines*. A stack machine model and manipulate memory spaces as stacks, where structured data and function calls are pushed to the stacks and are popped as needed. A register-based machine calls operations against finite registers such that input data is com-

¹See www.intel.com/design/intelxscale

²See <http://www.superh.com>

³See <http://www.mips.com>

puted results are placed in the registers. To narrow the scope of our topic, register-based machine is not discussed here. For detailed differences between two types of machines, one can refer to [18].

As software is distributed in intermediate code, eventually it has to be compiled again, by the built-in compiler from virtual machine, into native machine language to be understood by computers. We often refer the compilation from intermediate language to native machine language as code generation. Depending on the time when the compilation takes place, two types of code generation are considered. The first one is called *install-time* code generation. As named, the intermediate code is compiled during the time when the software get installed. After this stage, software program is completely compiled into machine language of target platform, and will be running natively.

One major weakness of install-time code generation is that, if the software to be compiled is relatively large in code size, it will probably take a long time for the installation stage to complete its work. If this is the case, we change strategy to another type of code generation, called *Just-in-time* (JIT) compilation. In this mode, intermediate code are selectively compiled during the running time of the software.

The condition of selective compilation is that the JIT compiler only compiles intermediate code encountered in current program state, such as the code after a conditional branch. Since the time taken for compiling a block of code is much shorter than compiling all the code, large software as mentioned above can be load and executed faster, with its usability guaranteed.

In JIT compilation, once the intermediate code is compiled, compiler output is saved in memory for possible calls in future. As the program life time progresses, more and more intermediate code are translated. Since the native code is directly executable on the underlying hardware, it runs faster than the code blocks which are not yet compiled. This leads to a situation that the performance of a software running in JIT mode gradually gets improved, with respect to the beginning when the program is loaded. Eventually all the code blocks are compiled into native code, and the performance reaches its maximum. We call the stage before the intermediate code completely compiled *JIT warm-up* stage. In term of compilation, software for application oriented virtual machine can be considered as *half compiled* distribution format.

It is suspected in [4] that the earliest proposal that comes up with the concept of JIT is [13] back in 1960s. The author believed that the compilation of function code into machine code can be done on the fly, thus no compiler output needs to be saved in physical storage, which conforms the idea of JIT. From that time on, several implementations of JIT compilers for different languages are researched, such as [15], [10] as well as [22].

Java Virtual Machine

The actual time when JIT got well known, is when Sun Microsystems released Java in 1990s. It contains a virtual machine is called *Java Virtual Machine* or JVM, which is designed to run applications only written in Java language originally. There does exist third-party implementations that compile course codes in other programming language into

intermediate code executed by JVM[11], but its designers have integrated JVM tightly with whole Java development environment, and thus supports object model of Java directly, such as inheritance and interfacing. Low level methods of objects include *static methods*, *virtual methods* and *interface methods*.

Aggregate data such as object is stored only after the memory is dynamically allocated for it, and is collected when it is no longer accessible. Scalar data is stored in either local variable, structure field or on the stack of abstract machine. When invoking methods, scalar data and/or reference to aggregated data are pushed onto the evaluation stack, and the returned value of the methods are presented on the top of the stack.

In file system, compiled Java intermediate code exists as *class* files, or a single compressed JAR file. The filename of each class file has to be the same as the public class name within the file. Classes can be organized using the directory structure from the file system. Each class file is in *byte code* format, where JVM instructions are presented as one-byte opcodes ranging from 0 to 255. Hence there are about 250 instructions by which JVM intermediate language is formed. These opcodes contains instructions such as *load and store*, *arithmetic*, *type conversion*, *object creation*, *operand stack management*, *control transfer*, *method invocation* and etc.

In JVM, instructions are type specific, such that type checking is necessary before a value is passed to the instruction. Primitive types include `byte`, `short`, `int`, `long`, `char`, `float` and `double`. Type specific instructions with the same functionality differ in the first character of the instruction name. For instance, `iadd` and `fadd` are both arithmetic instructions that calculate the sum of two values, but one take integer type and the other one take float point type.

JVM is considered to be *architecture neutral* in [19], because it aims to emulate low level instruction in byte code for its virtual architecture, to in order to minimize the gap with the underlying actual architecture. Because of this, on some architecture, it is even possible to execute the byte code instructions directly in machine level, such as ARM926EJ-S 32-bits RISC CPU⁴.

On September 30, 2004, Sun Microsystems released Java 1.5, which adds language features including *generics*, *metadata*, *autoboxing/unboxing* and *enumerations*. For detailed specification of JVM, one can refer to [21].

Common Language Runtime

Based on the experience of Java Development Kits, Microsoft released its competitive .NET Framework in early 2002, which contains a virtual machine mechanism known as *common language runtime* or CLR, where common language is the new name of intermediate language for its virtual machine. Similar to JVM, it is an abstract stack machine with instructions specific to stack operations, as previously described.

In contrast to the architecture neutral JVM, CLR does not restrict to single programming language, as long as a language can be compiled to the common language. Therefore

⁴See <http://www.arm.com/products/CPUs/families/ARM9Efamily.html>

CLR is considered to be *language neutral* and it makes possible that objects are not tied to a particular object oriented language, but are generally defined by the common language. Because all data structures and message passing are based on a set of rules, known as *common language specification* (CLS), the interoperability allows that one object can exchange information with another object where both of them are originally implemented in different languages [6].

Applications for CLR, or .NET applications, are mostly presented as one or more *portable executable* (PE) files, which is originally an executable format for native programs under Windows operating systems, can be dynamically loaded. To fit the applications into virtual machine domain, *metadata* is include in each PE file.

The information contained in metadata gives the description of the application, such as types, members, references and class information, which can be used by the runtime for memory allocation, method invocation, object location, code verification and etc. Based on class information, a class can be loaded as either value class such as `struct` or reference class such as `class`.

Same as JVM, CLR uses byte code when representing its virtual machine instructions. The instructions are specific to stack manipulations, such that values and methods are pushed to and popped from the abstract stack during the execution. The difference is that, unlike JVM, instructions in CLR is not type specific. When calling `add` instruction, JIT compiler will automatically correspond the value stored in the stack slots to the correct types, as information about variable types are already included in the metadata. Having designed the common language in this way, it widens the value passing semantic, hence multi-language interoperability is well supported [11].

When we say CLR, we refer to the implementation of the virtual machine. while the specification is defined in *common language infrastructure* or CLI. It is now an international standard accepted by Ecma International⁵, which allows anyone other than Microsoft to implement the corresponding CLR environment, such as Mono Project⁶ and DotGNU Project⁷. Their implementation does not contains Windows targeted libraries such as WinForm.Net, which is not a part of CLI specification.

Garbage Collection In Virtual Machines

While program is running, occupied memory space is not always freed by the program itself, while actually it is a responsibility of the program. This is often due to bad programming habit from developers. Without a mechanism that automatically cleans the memory, if this happens, memory will be eventually consumed undesirably and no other programs are able to access the memory resource anymore after that. Such mechanism is called *garbage collection*.

In object oriented virtual machines such as JVM and CLR, whether a memory slot of some object is garbage is decided by the its reachability. During the runtime, virtual system

⁵See <http://www.ecma-international.org/publications/standards/Ecma-335.htm>

⁶See <http://www.mono-project.com>

⁷See <http://www.gnu.org/projects/dotgnu/>

scans through the list of objects in the hash table periodically, to see if the inspected memory space belongs to any object or its sub-objects. If no object is found, the memory space is considered to be *unreachable* and can be collected with no problem.

In general cases, it is not realistic to scan through the memory completely, as it might affect the usability of the program. To more efficiently have the task done, modern virtual machines divide the memory segments into *generations*, where generation 1 is the newest generation and so on. It is by observed experience that newer allocated memory tends to be more possible to be unused during the program life cycle; while an older memory segment tends to be more useful in future. Basic algorithm is that,

- after the garbage collector scans through the objects for the first time, survived memory slots can be labeled as generation 2 and leaving remaining memory slots labeled generation 1;
- In the coming rounds of collection, only garbage in generation 1 is scanned and collected, until no memory can be release anymore [7].□

While distributing cross-platform software applications running on virtual machines, it is particularly important to have memory garbage collection well facilitated in terms of the performance and stability of the running application, since the memory management techniques at the bottom layer may vary on different platforms.

6.3 Platform Oriented Virtualization

While application oriented virtual machines allows applications running correctly without knowing about the architecture it is running, in some other cases, people do concern about the architecture for particular purposes. For software engineers and developers, it is possible that the software they are implementing is to be tested on different operating systems. For users, some of them want running a second operating system without partitioning their hard drive again. There are many other cases can be thought of, and we define virtual machine for such purposes to be *platform oriented virtual machine*.

To distinguish platform oriented virtual machines from application oriented virtual machines easier, we use the name *platform emulator* instead. This is actually the essential difference between a platform oriented virtual machine and application oriented virtual machine, where a platform emulator is implemented to provide an virtual computer environment whose hardware components are either mapped from the physical system or virtualized by software modules.

We call each of such computer environment *guest computer* and the actual computer running the emulator is called *host computer*. A guest computer is supposed to be no different from the actual physical computers, such that it follows the standard steps to power on, boot and shutdown. In most cases, any software can be executed on the guest computer with no problem, including operating systems.

To a host computer, guest computers are just software instances running in different processes independently. This is done by separating resources, such as memory space and storage space for each of the guest computer, so that more than one guest computer are allowed to be running on the host computer. In some emulators, storage space is usually saved as a separate partition in the hard-drive of the host computer or a disc image in the file system, such as VMware [24]. Before the guest computer is turned on, the platform emulator maps the image file as a hard drive device to be accessed by the guest computer.

The file image containing the software system is called *virtual appliance*, which provides a great convenience for distributing large software systems [17]. Software companies which produce operating systems, firewalls, databases can distribute the corresponding virtual appliances for users to try and run, without offending the existing platform⁸.

Other commercial platform emulators such as Microsoft Virtual PC⁹ provides the same functionality as VMware. The emulator Win4Lin¹⁰ only provides emulation specific to Windows systems under Linux. Open source platform emulators include bochs¹¹ and QEMU¹², which provide multi-architecture emulations. For PowerPC emulation, one should consider PearPC¹³ which is specific to MacOS X operating system. More emulators can be found at [28] which also provide a good comparison between these emulators.

6.4 Distribution In Source Code

A special architecture-neutral format is source code distribution. To software companies and distributors, it is important to gain as much business clients and end users as possible on different architectures and platforms, to in order to increase the market share of their software. Ignoring the concern of copyright and protection of intellectual properties, the scenario of distributing software in the form of source code is considered.

The scenario is usually found in the community of free and open source software, or FOSS in shorthand. Such software systems and applications are mostly licensed under BSD license, GNU General Public License and MIT License[12], which in common let their source codes viewable in the public domain and allow people to study and even make modifications to the source. It is believed such licenses reduces significant amount of time and manpower for software developers building new software by either using or learning the opened source code [3].

There are two types of source codes. One is to be compiled, such as code written in C and C++; the other one is to be interpreted, such as code written in Perl and Python. We call the code to be interpreted as script in the following paragraphs, to differ from code to

⁸Virtual appliances for VMware can be downloaded at <http://www.vmware.com/vmtn/appliances/>. QEMU appliances can be downloaded at <http://free.oszoo.org>

⁹See <http://www.microsoft.com/windows/virtualpc>

¹⁰See <http://www.win4lin.com/>

¹¹See <http://bochs.sourceforge.net>

¹²See <http://fabrice.bellard.free.fr/qemu/>

¹³See <http://pearpc.sourceforge.net>

be compiled.

Code To Be Compiled

Consider the compiled code whose binary only runs natively on specific architecture or operating system instead of running on a virtual machine. It turns out that during the software deployment, it might be more feasible to distribute its source code, in addition of distributing this program in binary form, if the code follows particular standards. Such standards are supported by the operating systems on which the program are intended to be executed. One example is POSIX, which stands for Portable Operating System Interface, with the X standing for the application programming interfaces, or API, inherited from Unix.

Incorporated with ANSI C standard, POSIX standard is widely supported by Unix systems, as well as non-Unix systems such as Linux. Windows NT based operating systems support POSIX only in real-time part [27]. For applications that is not Windows native but POSIX accordant to run under Windows, one can try installing one of Cygwin environment¹⁴ and Windows Service for UNIX¹⁵, which provide more POSIX compatibilities to Windows operating system.

Any source code conforms POSIX standard is able to be compiled into corresponding native binary format. It is very similar to install-time compilation from virtual machines, except that instead of intermediate code, source code is compiled. It is not necessary to use the same compiler to achieve this, as long as the compiler being used supports ANSI C standard. One famous example is the Hello World program, as shown below, which can be compiled almost anywhere.

```
#include <stdio.h>
int main (void) {
    printf ("Hello, World!\n");
    return 0;
}
```

As we have discussed previously, executable and linkable binaries have limitation on running anywhere due to lack of cross platform support. Even some software applications can be running on virtual machines, and each platform has its own implementation of such virtual machines, it still does not broaden the limitation. This is because there is a large amount of software applications have not yet been ported to virtual machines.

The idea of source code distributions is as illustrated above, such that software companies and distributors provide the standard compliant source code to the end users, and allows users to compile the source code on their computers. We have given an example in later section.

¹⁴See <http://www.cygwin.com>

¹⁵See <http://www.microsoft.com/windows/sfu/>

Script To Be Interpreted

Comparing with compiled programs, scripts are typically more portable and tend to be smaller in file sizes. It is because the statements in scripts are usually at a higher level than low level code in compiled programs, with the same semantic information remained. The difference between a compiler and an interpreter is that, the compiler translates the given code into machine code which has the same semantic as its source code, and then to be executed directly by the computer; whereas the interpreter goes through each statement of the give script and meanwhile it executes that statement without knowing what statement is going to be executed next.

During the running time of a script, interpreter accesses the run-time information, such as input and output and conditional branches, and keeps the program states synchronized with the semantic specified in the statements of the script. In most cases, source code is the only form in which a software, written in interpretive language, presents. In other words, the script itself is the software.

Scripts can be executed on multi platforms, if their interpreters are implemented on these platforms. Generally, the interpreters can be categorized into two difference kinds, one is stand alone interpreter and the other one is integrated interpreter. For example, most interpreters for languages Perl and Python previously mentioned are stand alone interpreters. These two languages are widely used in operating systems which support POSIX standard, mainly for system administration and small application. Provided that the interpreters are pre-installed, a script can be written so that it can be directly called. An example of Hello World program written in Perl is given below.

```
#!/usr/bin/perl
print "Hello, World!\n";
__END__
```

The first line in hw.pl specifies the path of desired interpreter, so that when the script is called, the shell environment, automatically locates the interpreter for the script to be interpreted. Similar syntax applies to Python. Under operating systems that does not natively provide a shell environment for the purpose, one can either installing a shell environment from third party, such as previously mentioned Cygwin environment, or pass the path of the script as a parameter to the interpreter.

Integrated interpreters are mostly referred to the script engines built within browsers. Gecko based browsers such as Mozilla Firefox¹⁶, provide support of JavaScript. Microsoft's Internet Explorer supports both JavaScript and VBScript, where the latter one is derived from Visual Basic language¹⁷.

Comparing with Internet Explorer, Gecko browsers are more dependent on JavaScript. Binding with the *XML User Interface Language*, or XUL, which is a markup language for building graphical user interfaces within the browser, it allows applications other than web

¹⁶For more information about JavaScript, visit <http://www.mozilla.org/js/>

¹⁷For more information about VBScript, visit <http://www.microsoft.com/vbscript/>

applets to be created[29]. Examples include the many dialogs from Mozilla Firefox browser are their selves written in XUL. Normally, these applications are shipped in a particular format, so called *Cross-Platform Install*, or XPI in shorthand. With supported browsers, installation from XPI file is extreme easy, by just clicking on the link to it and the installing process is automatically launched¹⁸.

Adaptive Compilation

For given source code to be successfully deployed in a group of computers with different platforms, the code has to follow supported standards. Moreover, during the stage of installing or deploying, we want to help the build system to tweak the process adaptively, so that the source code can be successfully compiled and thus executable by the system. The problem we want to solve is that, for the same functionality, the corresponding API on different platforms might differ. For instance, the `memcpy()` function found on GNU C Library, is named as `bcopy()` in BSD System Library with their arguments reversed to each other.

Such tweaking usually involves the steps such that, after inspecting the system environment, related identifiers are defined and passed to the code preprocessor by the build system. Based on the defined identifiers, the preprocessor conditionally selects correct branches inside the source code, to ensure the correct API is being used. See the following code segment in C language as an example.

```
...
#ifdef BSD_MEM
#define memcpy(_dest, _src, _l) bcopy(_src, _dest, _l)
#endif
...
```

The above statements are called macros, which is supposed to be understood by C preprocessor. These macros are saying that, if identifier `BSD_MEM` is define, then define `bcopy()` as `memcpy()`, with the destination and source pointers swapped. In the scenario that the build system finds out that the current operating system uses BSD System Library, it might compile the source code with the parameter `-DBSD_MEM` passed to the preprocessor.

Almost all compilers for C language support this option, and the preprocessor is automatically launched before each compilation. Now, with the source code preprocessed, the compiler translates all the occurrences of `memcpy()` into `bcopy()`, hence the compatibilities is guaranteed.

The example above describes a solution for codes to be compiled, while the same technique applies to code to be interpreted, or script, too. In both case, build environment information, such as architecture type and paths to the required libraries, is detected before preprocessing. To detect the build environment, GNU tools *Autoconf*, *Automake* and *Libtool*, which is included in the GNU Coding Standards [16], are used to interact with operating system for obtaining the information [23]. Among steps of building,

¹⁸For more XUL applications, visit <http://addons.mozilla.org>

- Autoconf generates portable shell scripts of tweaking build parameters, such as macro identifiers shown above;
- once the shell scripts are generated, Automake produces `configure` script based on the shell script for generating makefiles;
- `configure` script now can be called to tweak compiler parameters and makefiles are generated according to the environment information;
- to compile the source code, use `make` command which execute makefiles, with the help of Libtool for producing portable libraries.

Adaptive compilation can be found on highly customizable operating systems from open source communities. Popular distributions such as Gentoo Linux¹⁹ and FreeBSD²⁰ both provide similar source code package releasing systems named *Portage* and *Ports System* respectively. Each system stores a directory of software as a telephone book and allow end user search for particular applications. Source code of selected application will be retrieved remotely, compiled and installed in steps above. In term of compilation, source code can be considered as *not compiled* distribution format.

There is a project, called *Linux From Scratch* or LFS²¹, even provide no actual code at all. Instead, it provides manuals of how to build and configure a complete operating system by obtaining the required source code manually. Even the compiler itself has to be downloaded and compiled by an existing compiler from somewhere else.

6.5 Other Distribution Formats

Slim Binaries

In 1997, *slim binaries* is introduced in [9], which proclaimed that source code can be translated into intermediate code represented by tree structure. When executing, the time taken for accessing mechanical storages, which are mostly hard drive and floppy drive, can be used for compiling intermediate code into native binary code. It is a form of just-in-time compilation. Comparing with byte code, tree presentation is convenient for storing semantics, such as conditional branches.

We have defined that application oriented virtualization is basically to have an abstract machine executing byte code, which is an analogy to an real computer running native machine code. Thus the mechanism of slim binaries does not belong to this category. It is neither a distribution format in source code, since source code should be human readable in our definition.

¹⁹See <http://www.gentoo.org>

²⁰See <http://www.freebsd.org>

²¹See <http://www.linuxfromscratch.org>

Slim binaries is believed to be a replacement of Java in [8]. The author pointed out that a virtual machine is not able to verify its byte code efficiently, since semantics have to be analyzed again and it is redundant; where with code stored in tree presentation, it is easy to verify the semantic structure for slim binaries. Another advantage of slim binaries is that the code size is even smaller than native machine code, because of the semantics represented in tree structure, mentioned in [5].

Architecture Neutral Distribution Format

Mentioned in both [9] and [11], there used to be a specification commissioned by Open Software Foundation which is called *Architecture Neutral Distribution Format* (ANDF). It was an attempt to distribute software in the form of intermediate code, to executed on stack based virtual machine. Instead of being compiled just-in-time, the intermediate code is to be compiled at its installation stage, but install-time code generation is not as time efficient as JIT, as mentioned in section 6.2 on page 105.

In modern platforms, static variables and functions are saved in corresponding memory offsets in binary files, as well as intermediate code for virtual machines. One major reason that ANDF got faded after 90s, is that variables and functions are symbolically saved in the intermediate code, which eases the reverse engineering for its source code. For commercial software companies, using such format is identical to disclosing its intellectual properties to the public.

After 2000, free and open source software became much more popular. FOSS developers at this point concerns more on how wide their software can be distributed, rather than the protection of intellectual properties. Because of this, other than distributing software in form of source code, ANDF can be a good choice [20]. Actually, there are ANDF based projects still running healthy, such as the TenDRA project²², which provides C/C++ compilers for ANDF.

6.6 Concluding Remarks

In above sections, we have discussed four forms in which software can be distributed, known as software distribution formats, which are fat binary, intermediate code for application oriented virtual machine, appliance for platform oriented virtual machine and software distributed in source code. These software distribution formats can be commonly found in existing software distributions.

Fat binary include fully compiled binaries for target platforms in a single file. It typically archives more than one binary files, and hence the file size is relatively large. Since the only operation required before execution is extraction of the binary corresponding to correct platform, it is considered to be a convenient for average users. It is used for software

²²See <http://www.tendra.org>

distribution on Mac OS X operating systems from Apple Computer and PocketPC devices promoted by Microsoft.

Application oriented virtual machine provides an environment running applications in form of half compiled intermediate code. The intermediate code requires to be compiled either at install time or in time for execution. Just in time compilation is considered to be efficient, as only code branches need to be compiled while running. Popular examples include Java Virtual Machine designed by Sun Systems, and Common Language Runtime from Microsoft .Net Framework.

Platform oriented virtual machine loads appliances as guest computers, running on top of a host computer. Guest computers are separate process instances and can be running without interfering each other and their host computer, which is good for risky task such as product testing. Since appliances are computer images, which has nothing to do with compilers, so they are neither fully compiled, half compiled nor not compiled distribution format. Number of appliances are provided and can be found at VMware website and FreeOSZoo website listed above.

Software in the form of source code, which is not compiled, can also be used when distributing. File size of source code is the smallest comparing to all other distribution formats, because it is written in plain text and can be compress efficiently. The weakness is that before running the software, compilation is required. Distributions in source code are commonly found in open source communities, such as Gentoo Linux, FreeBSD and Linux From Scratch, and their users tends to be advanced.

There are other software distribution formats not introduced here, such as Flash Media from Adobe²³. It is a format of universal distributable multimedia applications, which is mainly formed by scripts, intermediate codes and usually multimedia contents. Similar to application oriented virtual machines, a Flash file is to be executed by Flash Player, which can be either executed stand alone or as a browser plug-in.

Because of various software distribution formats as we mentioned, multi-platform support for software is made possible. As computer performance improved along with time, the gap of efficiencies in both compilation and execution between these formats is believed to be narrowed in future. Combinations of existing formats are expected.

6.7 Exam Question

1. What are fat binary and slim binaries?
2. What are the differences between application oriented virtual machine and platform oriented virtual machine?
3. Compare all formats discussed in the paper.

²³Originally designed by Macromedia which is a part of Adobe now. See <http://www.macromedia.com>

Bibliography

- [1] Inc. Apple Computer. *Universal Binary Programming Guidelines*. 1 Infinite Loop, Cupertino, CA 95014, 408-996-1010, 2 edition, 3 2006.
- [2] Ralph Arvesen. Developing and deploying pocket pc setup applications. <http://msdn.microsoft.com/library/en-us/dnnetcomp/html/netcfdeployment.asp>.
- [3] Sami Asiri. Open source software. *SIGCAS Comput. Soc.*, 33(1):2, 2003.
- [4] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003.
- [5] Arpad Beszedes, Rudolf Ferenc, Tibor Gyimothy, and Andre Dolenc. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, 2003.
- [6] Microsoft Corporation. Automatic memory management, 2006. [Online at MSDN; accessed 7-March-2006].
- [7] Microsoft Corporation. Common language runtime overview, 2006. [Online at MSDN; accessed 7-March-2006].
- [8] Michael Franz. The Java Virtual Machine: A passing fad? *IEEE Software*, 15(6):26–??, November / December 1998.
- [9] Michael Franz and Thomas Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.
- [10] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
- [11] K John Gough. Stacking them up: a comparison of virtual machines. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 55–61, Washington, DC, USA, 2001. IEEE Computer Society.
- [12] Michael K. Johnson. Licenses and copyright. *Linux J.*, 1996(29es):3, 1996.
- [13] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.
- [14] Brad A. Myers. Using handhelds and pcs together. *Commun. ACM*, 44(11):34–41, 2001.
- [15] T. Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 150–152, New York, NY, USA, 1987. ACM Press.
- [16] Arnold Robbins. What's gnu? gnu coding standards. *Linux J.*, 1995(16es):8, 1995.

- [17] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.
- [18] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163, New York, NY, USA, 2005. ACM Press.
- [19] Jeremy Singer. Jvm versus clr: a comparative study. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 167–169, New York, NY, USA, 2003. Computer Science Press, Inc.
- [20] Paul Tanner. Software portability: still an open issue? *StandardView*, 4(2):88–93, 1996.
- [21] Frank Yellin Tim Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley Professional, 2 edition, 1999.
- [22] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.
- [23] Gary V. Vaughan, Ben Elliston, Tom Tromeey, and Ian Lance Taylor. *GNU Autoconf, Automake and Libtool*. pub-NEW-RIDERS, pub-NEW-RIDERS:adr, 2000.
- [24] Brian Walters. Vmware virtual platform. *Linux J.*, 1999(63es):6, 1999.
- [25] Wikipedia. Fat binary. http://en.wikipedia.org/wiki/Fat_binary.
- [26] Wikipedia. Comparison of virtual machines — wikipedia, the free encyclopedia, 2006. [Online; accessed 17-March-2006].
- [27] Wikipedia. Posix — wikipedia, the free encyclopedia, 2006. [Online; accessed 18-March-2006].
- [28] Wikipedia. Universal binary — wikipedia, the free encyclopedia, 2006. [Online; accessed 17-March-2006].
- [29] Louie Zhao, Jay Yan, and Kyle Yuan. Mozilla accessibility on unix/linux. In *W4A '05: Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, pages 90–98, New York, NY, USA, 2005. ACM Press.

Chapter 7

Ayesha Kashif: History of Statecharts

Statecharts were first presented in 1984 [8] by David Harel of The Weizmann Institute of Science, Rehovot, Israel. Since then many variants of statecharts have been proposed in the literature. One of the variant is Modecharts [16], a specification language for real-time systems presented by Farnam Jahanian and Aloysius K. Mok, for the first time in 1988. It is similar to statecharts in some ways, but it is tailored to represent time constraint systems. Another one is ECSAM (Embedded computer system Analysis and Modeling method) [19] which was first used in 1980 by Jonah Z. Lavi and his team and further development in 1983 by a cooperative effort between Dr. Lavi and Prof. Harel. OMT [18] incorporated statecharts in UML in 1997. UML has adopted statecharts and has given them semantics that differ from Harel's in several points. However, the definition of state machines and their behavior in the OMG definition of UML [18] is vague and incomplete in several points. Then in 2002, Christian Prehofer [21] introduced a new approach for modular design of highly-entangled software components, called Plug and Play.

7.1 Introduction

The development life cycle of a system involves many tasks and is carried out by a professional team. The classical waterfall model [22] has the following steps: requirements analysis, specification, design, implementation, testing, and maintenance. Over the past 35 years, many techniques of this model were proposed with different approaches to the life cycle development. The system development life cycle contains a requirement analysis phase. To correct the specification errors and misconceptions if discovered in a later stage, then it will become very expensive, so the system behaviors should be carried out as early as possible. The main part of the specification stage is the construction of the system model.

Statecharts are especially effective for reactive systems in contrary to the transformational system [8]. A reactive system continuously interacts with its environment, using inputs and outputs that are either continuous or discrete and these inputs and outputs can be asynchronous. It has many possible states depending on the modes of operation, the current value of variables and the past behavior. Some of the examples of reactive systems include

online interactive systems such as automatic teller machines (ATMs), computer-embedded systems such as avionics, automotive and telecommunication systems and control systems, such as chemical and manufacturing systems. The main problem is to describe the behavior of a reactive system, in such a way that it understandable as well as formal enough to add computer simulations, which is the set of inputs and output events, conditions, actions and few additional constraints.

The modeling approach, the Statecharts language [9], extends the state-transition diagrams. The three basic elements of these statecharts are hierarchy, concurrency and communication. So these make the state diagram highly structured, compact and expressive that can express complex behavior as well as compositional and modular approach.

7.2 STATEMATE

The first article that introduced the language of statecharts [10] presented only a brief discussion of how its semantics could be defined. Since then many variants of statecharts have been proposed in the literature. Harel in 1989 [4] presented rigorous semantics for the first time. This section discusses the semantics of statecharts as a STATEMATE approach [15]. So the actual semantics of statecharts presented by David Harel were actually for STATEMATE [4].

Semantics of Statecharts

A semantic definition of a language to specify the behavior must be detailed enough to show how the model will execute or how the system will react to the inputs from the environment to give the outputs. The main part of the system model is an activity-chart [15] in which the functional capabilities of the system are captured by activities with the data elements and the signals that can flow between them.

There are three [15] types of states in statecharts: OR-states, AND-states and basic states. OR-states have sub-states which are related by "exclusive-or". AND-states have orthogonal components which are related by AND. Basic states do not have any substates or do not have any parent like the root.

Transition is labeled [15] as " $e[c]/d$ " where e is the event that trigger the transition, c is the condition which if true only then e will occur, and a is the action. Events and conditions are closed under the Boolean operations or, and and not. The expression $e[c]$ is interpreted as " e and c ". Beside appearing along transitions, actions can also appear with the entrance to or exit from the state.

The behavior [14] of a system is a set of possible runs, each representing the response of the system to a sequence of external stimuli by the environment. A run consists of a series of snapshots of the system's situation; such a snapshot is called status. The first is the initial status and then each subsequent one is obtained from its predecessor by executing a step. see figure 1. A step take zero time to execute and the time interval between two consecutive steps is not the part of the step semantics but depend on the execution environment and the

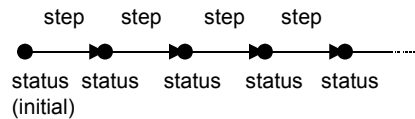


Figure 1

time model. An event generated in step n may be sensed in step $n+1$ only and not in the steps following $n+1$ step even if those steps also execute at time t .

Basic System Reaction

For a given root state R , a configuration related to R is the set of state A obeying the following rules:

1. A contains R .
2. If A contains a state B of type OR, then it must also contain one of B 's substate.
3. If A contain a state C of type AND, then it must also contain all of C 's substates.
4. The only states in A are those required by the above conditions.

As shown in figure 2. $B1, C1$ is not maximal, the full configuration is B, C, A and R . Also the configuration is closed upwards [15] means that when the system is in any state A , it must be in A 's parent state which is R in this example.

Figure 3 illustrates three cases of Chain reaction each with two steps. All start with the system in A when the external event E occurs. In Figure 3 (a), an event G is generated by reaction E/G in one compound state A and triggering another reaction in the orthogonal component B . In figure 3 (b), the subsequent step in the chain takes place, triggered by the derived event $ex(A1)$ indicating an exit from $A1$. In the third case, figure 3 (c), the reaction triggered by E causes the system the system to move to B and as a result the transition event F will take the system to state C .

Multiple external changes can occur exactly at the same time so multiple reactions may get enable and perform at the same time too. Like in figure 4, while in $A1$ and $B1$ when E occurs takes the two transitions at the same time.

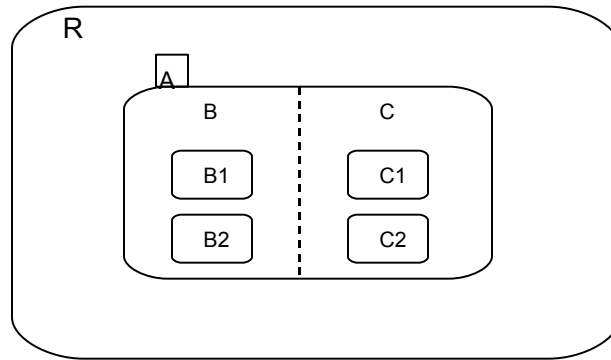


Figure 2.

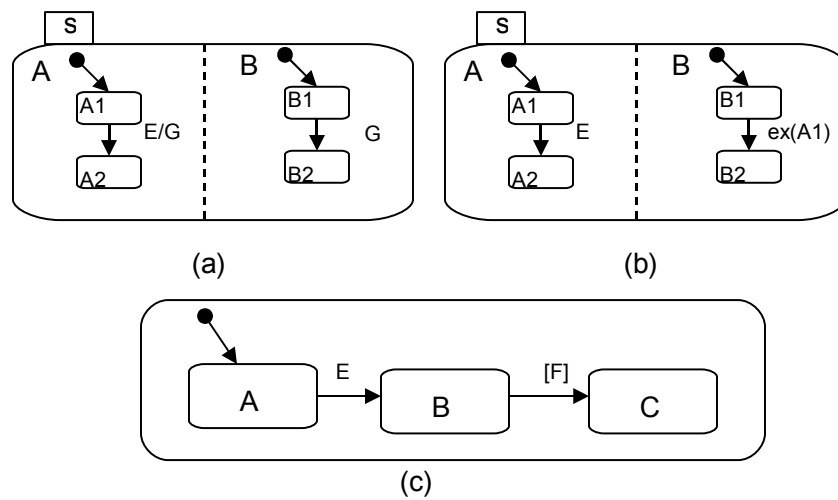


Figure 3.

Here comes the problem of order in which the action are performed. In figure 5, when E occurs , both actions are performed in the same step. The value of Y after carrying out $Y:=X$ in this step depends on whether or not the assignment of ! to X was performed before. This can be resolve by postponing the actual value update until the end of the step, when they are carried out at once.

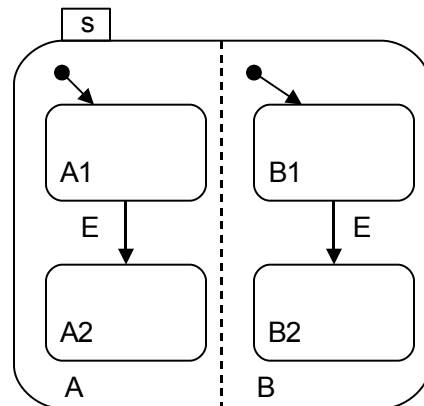


Figure 4.

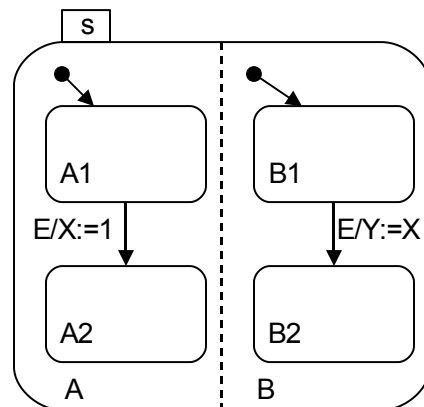


Figure 5.

Handling Time

Execution consists of steps triggered by external changes and the advancement of time which may cause chain of steps. The time calculated in dealing with the explicit time expressions appearing in timeout events and scheduled actions is measured in terms of some abstract time unit common to the entire statechart and different statecharts have different time units.

There are two time schemes [15]. In synchronous time scheme, the system executes a single step in every unit. It is best suited for modeling digital systems where the execution is synchronized with clock signals. The asynchronous time scheme is flexible as it allows several steps to take place within a single point in time. So change can occur any moment between steps and several changes can occur simultaneously. This set of steps is called a super-step [1].

Dealing with History

As explained earlier, Statecharts have two kinds of history connectors H and H*. As an example, consider figure 6, in which E1 is to be taken. If the system was in A1 when it was most recently in A, then E1 is taken as if its target state was A1 and the full transition is E1,E2. But if the system was in A2 most recently the target state of E1 is then A2 and the full transition is E1,E3. If these both were not the case or A's history was cleared then E1 is treated as if its target is A and the full transition is E1,E4,E2. In this last case E4 is taken so any actions associated with E4 are executed. The actions that erase the history of

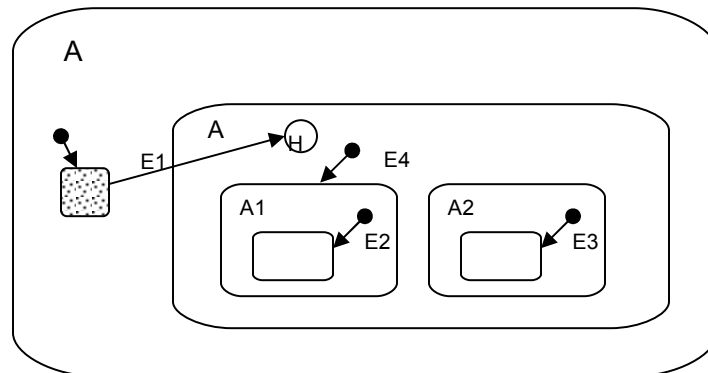


Figure 6.

a state are called $\text{history-clear}(S)$ [2] for S and $\text{deep-clear}(S)$ also clears the history of S with the history of all its descendent states too.

STATEMATE: Working Environment

STATEMATE is a commercial tool, designed for the specification and design of real life reactive systems. It specifies and analyzes the system under development (SUD) [2] from three points of view: structural, functional and behavioral. For these views STATEMATE provides three graphical languages.

Structural View

It is described using the language of module-charts, which describe the SUD [2] modules or the physical components, the environment modules and the clusters of data and/or control signals. Encapsulation is used to capture the submodule relationship.

Conceptual View

It is captured by activity-charts which are quite similar to module-charts. Here we have shapes for activities or functions carried out by the system. It also contains two additional kinds of objects: data-stores and control activities. Data-stores are for representing database, data variables and data structures. Where as control activities gives the behavioral view of the system so appear empty here and they will be described in the third graphical language.

Behavioral View

Statecharts are used here to give the repeated decomposition of states into sub-states in an AND/OR fashion, combined with the broadcast communication mechanism and all the other features described earlier in this paper.

Queries, Reports and Documents

STATEMATE provides a querying tool called the object list generator [15] to give user the data from the database. In addition to fixed format reports, STATEMATE also has document generation language with which user can make their own documents by writing programs with particular structure, contents and appearance. Other then the reports, there are different data dictionaries, interface diagrams and tree versions of various hierarchies (called N squared -diagrams [15]).

Execution and Dynamic Analysis

STATEMATE can carry out a step of the SUD's [2] dynamic behavior with all the consequences. The most basic way to run the SUD is in step-by-step fashion. At each step the user generated external events, changes conditions and carries out other actions such as changing variables. Thus all of these will happen in a single step. When the user gives the "go" command, STATEMATE responds by transforming the SUD into the new resulting status. A special "simulation control language" (SCL) [15] is designed to enable the user to retain control over how the execution goes. Program in SCL are just like conventional programs in a high-level languages with breakpoints.

Code-generation

STATEMATE can translate the model of the system into code of a high-level programming language. Currently Ada and C [15] are incorporated. The code generated will be a prototype code as reflects only the design decisions. Debugging mechanism is also present with which, user can trace the executing parts of the code back up to the STATEMATE model.

7.3 ECSAM

ECSAM is an Embedded computer system Analysis and Modeling method. It was first used in 1980 at the Israel Aircraft Industries (IAI) by Jonah Z. Lavi and his team. Further development in 1983 by a cooperative effort between Dr. Lavi and Prof. Harel which resulted in the development of the statechart formalism and the subsequent development of the STATEMATE. Dr. Lavi continued development of ECSAM and introduced E-level and S-level model concepts, see [19].

Overview

The ECSAM approach addresses conceptual and design models and defines relationship between them. Both are used concurrently to analyze and design the system and to represent the behavior and dynamics characteristics of the system, as shown in the Figure 7 below: ECSAM can be viewed as a kind of prism that splits the system description into a conceptual

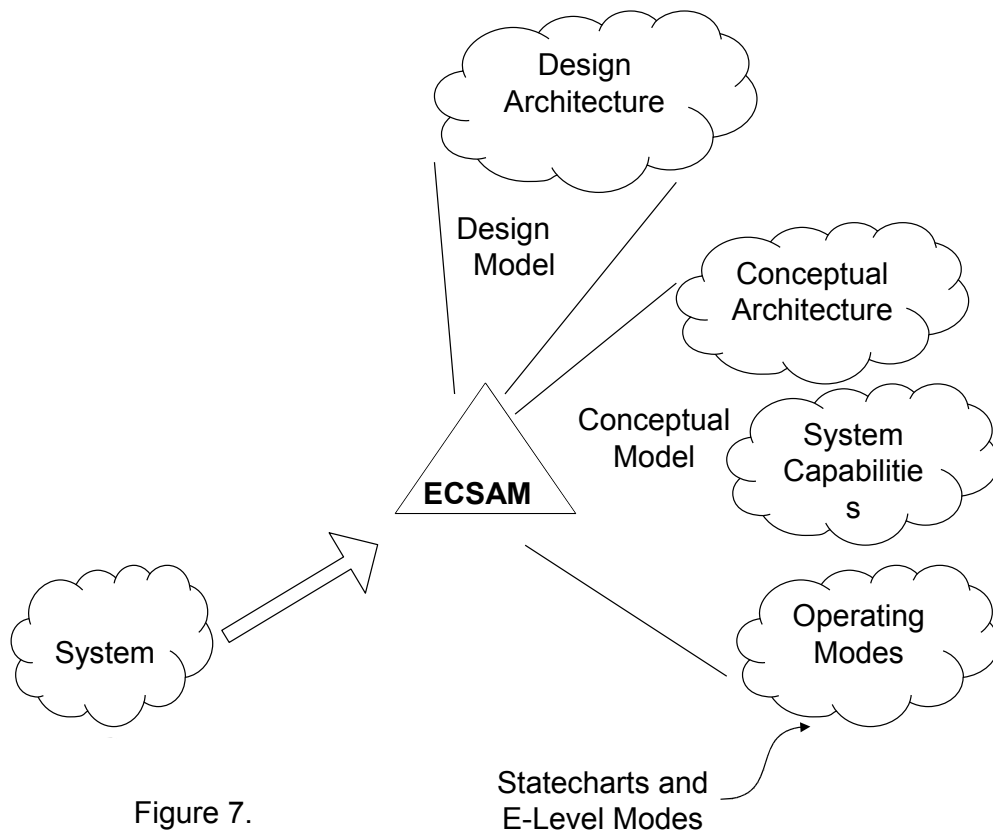


Figure 7.

model and a design model.

Design model

It describes the design architecture of the system, the structure of its hardware and software subsystems and their mutual relationships.

Conceptual model

It describes the dynamic behavior of the system by three views:

1. Conceptual Architecture View: internal conceptual subsystems and their functionalities, and the external and internal interfaces of the system.
2. System Capabilities View: dynamic process expressed in terms of the transformations it performs, its data-flow, and the associated process control.
3. Operating Modes View: main operating modes of the system and the transitions between them.

Modeling Process Overview

Modeling of the system is carried out as a sequence of steps from the major phases.

1. System's environmental model (E-level model): which describes the system's external structure and behavior (statecharts) as seen by operators and users.
2. Conceptual model (S-level model): which describes the system's internal conceptual structure and dynamics.

Statecharts and E-level Models

The E-level model describes the modes (states) of the E-level system, the transitions between them and their graphical representation using statecharts [8]. Typical examples of the system E-level modes (states): **Off, On, Deployment, Standby**. Transitions between modes are sequential operations and finite. When an event occurs, the system can react by performing a specific action, such as generating a signal, setting the value of a variable, and/or making a transition to another state. Events that cause reactions are called triggers.

Unlike state transition diagrams, the sources of triggering events are specified in the augmenting view that deal with conceptual structure and dynamics process.

ECSAM adopted statechart notations to describe the system's behavior by:

1. the system's operating modes, the transitions between them, and the control of the system's capabilities (invocation, termination, suspension, and resumption)
2. the process control charts that describe the process dynamics, including logic and timing

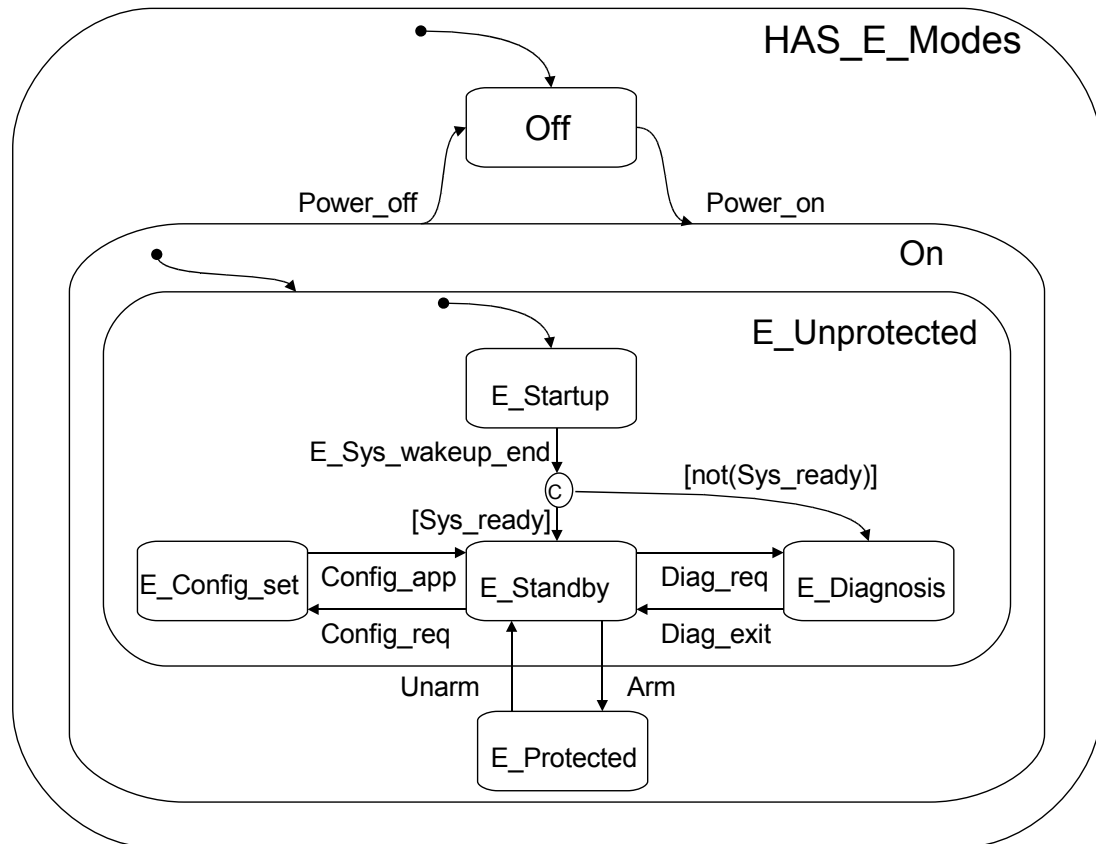


Figure 9.

The basic features of statecharts used here can be shown by an example of home alarm system (HAS). as shown in the figure 9. Initial mode is **off**. By *Power_on* event, the system changes its mode(state) from **off** to **on** and by *Power_off* event, system returns to **off** mode. **On** is the *parent state* of *Sys_modes* and *Pwr_source* or these two are the sub-states or sub-modes of on created by an **and-decomposition**.

Extension of statecharts notations

ECSAM extends the basic statechart notations and semantics to express more complicated situations.

Encapsulation of states: Statecharts has a top-down embedding and bottom-up encapsulation. Usually it is advisable to start with a top-level description and specify the internal details later but when the internal decomposition of the state is more complex,

a @ operator also known as is-a-chart operator, can be inserted before the name of the state to indicate that the state is specified in a separate chart.

Conditions and element expressions: Statecharts have C-connector and the arrows emanating from the C-connector must be mutually exclusive and more than two arrows can emanate from one C-connector. But if the conditions are not mutually exclusive then nondeterminism may occur. In ECSAM, there is a way to present transitions that have common triggering events. The arrow leading *to* the C-connector is labeled by the common triggering event, while the arrows emanating *from* the C-connector are labeled by the different conditions.

History entrances: The history entrance in statecharts indicates entry to first-level states only. If for example, have two substates then the "memory" of the history entrance would not "remember" in which of two substates the system was last, and the entrances would be the one specified as default. ECSAM extends the history down to the lowest level, the H-connector can appear with an asterisk attached (**H***), indicating entry to the most recently visited state on the lowest level. This connector is called a *deep-history* connector.

Supporting Tools

ECSAM models and reports can be created using widely available word processing programs that provide graphical support. Over the years, *CAS²E* tools providing varying levels of support of the ECSAM method have been developed. Currently, Statemate MAGNUM (developed by I-Logic) supports most aspects of the ECSAM method. Previously, tools such as Teamwork (developed by Cadre) and STP (developed by IDE) supported specific aspects of ECSAM.

7.4 UML Statecharts

Harel's statechart is mainly designed for function-oriented structure. OMT [18] incorporated statecharts in UML in 1997. The primary authors were Jim Rumbaugh, Ivar Jacobson, and Grady Booch, who originally had their own competing methods (OMT, OOSE, and Booch).

Overview

UML extends statecharts for object oriented structure with some semantic modifications to the statecharts. It is a discrete language that emphasizes on the dynamic behavior. The most common things in the UML to model with explicit state machines are classes and use cases. A use case is a named piece of functionality visible in a context that returns a result visible to one or more objects in that context (called actors). UML defines twelve types of diagrams, divided into three categories:

Structural Diagrams: includes the class diagram, object diagram, component diagram, deployment diagram

Behavior Diagrams: includes the use case diagram, sequence diagram, activity diagram, collaboration diagram, statechart diagram

Model Management Diagrams: includes packages, subsystems, models

The UML provides two different kinds of state machine formalisms: statecharts and activity diagrams. They differ in the kinds of situations to which they are applied. Statecharts are used when the transition from state to state takes place primarily when an event of interest occurs. Activity diagrams are appropriate when the object (or operation) changes state primarily upon completion of the activities executed within the state rather than the asynchronous occurrence of events.

Statecharts and UML

The definition of state machines and their behavior in the OMG definition of UML is vague and incomplete in several points [7].

Broadcasting events: Statecharts [8] are based on event broadcasting. This is simple and convenient for small models but difficult for large ones, because it results in global coupling of all components. UML is quite unspecific in this issue: the way events are transported from their source to the event queues of the state machines where they should take effect is undefined. Within a state machine, a dispatched event is broadcast.

Synchronous event processing: Statecharts [8] employ synchronous event processing. This means that the state machine immediately reacts to an external event and does all state transitions and processing of events triggered by state transitions instantaneously, i.e. in zero time. In particular, all reactions to an external event are completed before the next external event can happen. UML queues events instead of immediately reacting to them. However, once an event is dequeued, it is processed synchronously. Synchronous event processing may have nice formal properties, but it comes with a bunch of semantic problems (see [5]) and may lead to counter-intuitive behavior.

Kinds of actions: Harel statecharts have a quite simple action scheme: actions are triggered by state transitions and work synchronously, i.e. in zero time. UML, on the other hand, has introduced an elaborate action scheme, distinguishing entry actions (triggered and completed prior to entering a state), exit actions (triggered upon exiting a state and completed before proceeding to the next state) and do actions (executed while the system is in a particular state). For modeling requirements, it suffices to have two kinds of actions: those that are triggered and completed during a state transition and those that are executed while the system is in a particular state. Having these two kinds, one can employ a simple and powerful action triggering system.

Event queues: The UML event queues are an implementation oriented concept that can be omitted for requirements models, thus yielding much simpler event semantics.

History Both Harel and UML statecharts provide a history mechanism that allows easily re-entering that substate of a statechart which had been the last active one before the statechart was left.

State transition trigger conditions: Both Harel and UML statecharts use events and guard predicates for controlling the triggering of a transition. Usually, the trigger conditions and the triggered actions are written as annotations of the state transition arrows.

Integrating statecharts: Harel models behavior and functionality as two separate models which both have a decomposition hierarchy of their own. The models communicate by references to variables and by invocation of operations. UML, on the other hand, considers statecharts as auxiliary models that are embedded in the specification of classifiers in order to describe their internal behavior. UML thus integrates the models of a classifier and of its behavior, which makes it easy to model local behavior. However, as UML has no true composition of components (where the composite is a higher-level abstraction of its components, see [2]), it is awkward to specify global behavior in UML

UML Statecharts are a powerful visual formalism for capturing complex behavior and apply well to both functionally decomposed systems and to object-oriented ones. Objects are composite entities consisting both of information (attributes) and operations that act on that information (methods). Statecharts add a number of useful extensions to the traditional flat Mealy-Moore state diagrams, such as nesting of states, conditional event responses via guards, orthogonal regions, and history.

7.5 Modecharts

Modechart is a specification language for real-time systems. The semantics of Modechart is defined in terms of Real time logic RTL [17] which is a logic specially for absolute timing of events. Modecharts uses the concepts of modes from the work of Parnas *et al.* at the Naval Research Laboratory. Modes can be thought of as partitions of the state space of a system. Although it is similar to statecharts in some ways, but it is tailored to represent time constraint systems. From Statecharts it borrows the *compact representation of large state machines*.

Modes

Modechart specifications consist of modes, transitions and external events. Modes can be thought of as hierarchical partitions of the state space, and are related by a containment

relation, as dictated by how they are syntactically composed (sequentially or in parallel). The basic modes with no children are called atomic modes.

A serial relationship among several modes (children of a serial mode) indicates that the system operates in exactly one of the modes, at any time. Several modes can also be in parallel, in which case the system operates in all modes simultaneously. A mode is depicted graphically as a box, with all its child modes represented inside it.

Modes vs State variable

Despite the similarities, there are important semantic distinctions. A state variable represents information about data whereas a mode represents some control information about the system. The value of state variable is changed explicitly by completing the execution of an action which takes nonzero units of time to perform. Consequently, a state attribute S cannot become true and then become false at the same instant of time, i.e., the two events can not happen at the same time. A mode entry or exit is implicit in that it does not require the execution of an action; a mode transition is taken when a certain condition is satisfied so can happen at the same instant of time.

Transitions

Transitions allow the system to switch from one mode to another (possibly in different levels within the hierarchy). Transitions can be triggered by events, which can be external or internal. All the events in the system are instantaneously broadcast and transitions, once triggered, are taken instantaneously. Serial modes may designate one mode as their initial mode, which is instantaneously entered once the serial mode is entered (unless some other child has been entered explicitly with a transition crossing or terminating at it). Transitions between modes in parallel are not allowed. Entry of a parallel mode forces the system to enter all its children. A transition out of one mode requires exit out all modes in parallel with it. A transition is depicted as a directed edge, departing the source mode and ending in the destination mode.

Each transition has a *transition condition* associated with it. There are two types of transition conditions in Modechart, namely *timing conditions* and *triggering conditions*. Timing conditions are those that have the form (lb, ub) , where lb is a lower bound and ub is an upper bound. Triggering conditions are of the form, $p_1 \wedge p_2 \wedge \dots \wedge p_n$, where each p_j specifies a condition for taking the transition.

Tool

Modechart has been developed as a graphical specification tool in SARTOR (Software Automation for Real-Time Operations) [20], a design environment for hard-real-time software currently under development at the University of Texas at Austin. The goal of SARTOR is to mechanize the analysis and synthesis of real-time software from systems specification.

An implementation of Modechart serves as a front-end for SARTOR which provides a suit of tools to analyze a specification for satisfaction of safety requirements.

7.6 Plug and Play

In 2002, Christian Prehofer [21] introduced a new approach for modular design of highly-entangled software components using statecharts. He structure the components into features which will become usable and self-contained services and then he model each feature individually using statecharts. For the composition of components between the features, he consider the interaction between them. Then the full component descriptions are created automatically in a plug-and-play fashion by combining the statecharts for the required features and their interaction. Also he develop different classes of statecharts and showed the interactions on a case-to-case basis.

Statecharts

Statecharts are used for graphical description where transitions are labeled by the functions which trigger these transitions. An external function call triggers a transition labeled with this function depending on the current state of the statechart [3]. The following notation is used for labeling transition:

called_{function}()[*condition*]/*action*

Here the *called_{function}()* is the external event which initiate the transition. Note that all three labels may be empty then in this case it will be an internal transition without an external event, also called spontaneous transition.

An example is the statechart in Figure 8 describing the basic functionality of an answering machine in a feature called BasicAnsMachine. There are two states, Waiting (for a phone call) and Answering. The initial state is Waiting.

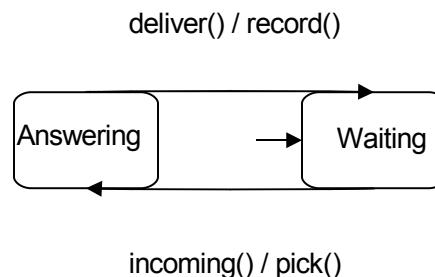


Figure 8. The BasicAnsMachine

Semantics

The semantic model uses an external black-box view of the component. It is based on the function calls from the outside which trigger transitions. Only the input and output is considered and not the internal states. A possible run can be specified by a trace of the externally called functions and the resulting actions of the statechart [13]. For example the trace for the statechart in figure 8 of `incoming()` and `deliver()` transitions is triggered by the external function calls.

Input sequence: `incoming(),deliver(),incoming(),deliver(),...`

Output sequence: `pick(),record(),pick(),record(),...`

Refinement

Here the refinement of statecharts is by specifying a component in more detail to reduce the under-specification. So a step-wise refinement by adding more specific behavior. The main benefit of this graphical refinement rules is the ease of use. These rules are based on syntactic input and output events.

Following are the elementary statecharts refinement operations:

1. Add new behavior which was not specified before like adding a new state or transition to a state.
2. Eliminate alternative transitions, if any exists. This will reduce the non-determinism and specifies the behavior more precisely.
3. Add internal or compatible behavior, which does not change the original output. So the new behavior can abstract from the additional behavior and the old behavior remains the same.
4. Eliminating transitions for exceptional cases. So refinement only holds if some exceptional case does not occur.

Modeling Features and Interactions by Statecharts

A feature is like a class in an OO-design which is an interface with functions and encapsulates internal state. Plug-and-play approach describes both features and interactions by partial statecharts to describe a high-level view of their behavior. The key point is that the features and interactions are as the fragments of statecharts. For a concrete feature combination, it uses extensive use of hierarchical statecharts and parallel composition of statecharts.

There are three kinds of modeling features with statecharts:

1. Base features which a complete statechart, including initial and final states.
2. Transition-based features which refine a transition locally and do not add externally visible in out transition to a statechart.

3. State-extending features which add global states and externally-visible transitions. These features extend the states of other features and also the external visible interface.

7.7 Hypercharts

Hypercharts is an extension to statecharts to support Hypermedia specification 1984 [6]. The specification of hypermedia applications is a complex task as it requires the use of suitable models capable of capturing the synchronization requirements related to the presentation of dynamic data such as audio and video. Such models must be able to specify both the conventional navigation elements of hypertext and the complex timing and sequencing relationships of multimedia presentations.

A hyperchart is a conventional statechart extended with three new set of notations [6]: timed history, timed transitions and synchronization mechanisms.

Timed history

The assignment of the timed history symbol - a clock icon - to a transition t with a state s as its destination means that if s has been active in past, then the firing of t will recover the value of the timed history register of s , in addition to recovering the last configuration of s in terms of its sub-states. Similarly to the history symbols defined in conventional statecharts, the timed history symbol has a recursive version implying that the timed history registers of s and all its sub-states, recovered by conventional recursive history.

Timed transitions

Hypercharts provides a temporal notation which allows the specification of transitions whose firing is determined by the time progress during the active time period of its source states. Using timed transitions it is possible to specify multimedia presentation requirements such as delay and jitter.

Synchronization Mechanisms

It is provided by M:N synchronized transitions. An M:N synchronized transition is grouped into five components : source states, source arcs, target states, labels and a synchronization type.

7.8 Concluding Remarks

Statecharts were introduced as a development [11] over modeling techniques such as traditional state machine modeling. The weaknesses inherent in state machine modeling such

as overly complex diagrams for larger systems, and the lack of concurrent support were addressed by the statechart method.

Statecharts overcame these weaknesses [12] by introducing concepts such as orthogonal regions and and-states. These concepts allow for the specification of far more complex real time systems than traditional state machine modeling. The strength of statechart approach lies in the similarity to state diagrams and availability of direct reconstruction and re-implementation, introduction of and-state, direct relevance to embedded software system with possible extensions.

Several variants of statecharts have been introduced and there have also been several attempts to underpin the intuitive meaning of statecharts with precise semantics.

7.9 Exam Question

1. Describe the main three elements of statecharts.
2. Describe what a history indicator is used for in statechart diagrams.
3. A vacuum cleaner of a company has the following design: If it is activated, it starts vacuuming the offices, afterwards the canteen and at last the laboratory. If it has finished cleaning the laboratory, it starts again in the offices. If it is deactivated, it returns to its locker and stops there. Every time on changing his operational area, it empties his dust bin into some container. If it runs low on energy, he goes to a recharging station, and recharges its batteries. Afterwards it continues the cleaning where it has stopped. Draw a statechart diagram modeling the robot's behavior.

Bibliography

- [1] D. Harel B. P. Douglass and M. Trakhtenbrot. *Statecharts in Use: Structured Analysis and Object-Orientation*. 1998.
- [2] A. Naamad A. Pnueli M. Politi R. Sherman A. Shtul-Trauring D. Harel, H. Lachover and M. Trakhtenbrot. *STATEMATE: A Working Environment for the Development of Complex Reactive Systems*. 1990.
- [3] H. Kugler D. Harel and R. Marelly. *The Play-in/Play-out Approach and Tool: Specifying and Executing Behavioral Requirements*. 2002.
- [4] J. Schmidt D. Harel, A. Pnueli and R. Sherman. *On the Formal Semantics of Statecharts*. 1987.
- [5] Von der Beeck. *A Comparison of Statechart Variants*. In H. Langmaack, W.-P. de Roever, J. Vytopil (eds.): *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, 1994.

- [6] Oliveria Fabiano B Paulo, Paulo Masiero. *Hypercharts : Extended statecharts to support hypermedia specification. IEEE*, 1997.
- [7] Martin Glinz. *Statecharts For Requirements Specification-As Simple As Possible, As Rich As Needed. Proceedings of the ICSE 2002 International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, Orlando*, 2002.
- [8] D. Harel. *Statecharts: A Visual Formalism for Complex Systems. Preliminary version: Technical Report CS84-05*, 1984.
- [9] D. Harel. *Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming*, 8(3):231–274, 1987.
- [10] D. Harel. *On Visual Formalisms. Comm. Assoc. Comput. Mach.*, 31(5), 1988.
- [11] D. Harel. *Some Thoughts on Statecharts, 13 Years Later. Proc. 1996 Asian Computer Science Conf., Singapore*, 1996.
- [12] D. Harel. *On Modeling and Analyzing System Behavior: Myths, Facts and Challenges*. 1997.
- [13] D. Harel. *From Play-In Scenarios To Code: An Achievable Dream. Computer*, 34(1), 2001.
- [14] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. page 258, 1998.
- [15] David Harel and Amnon Naamad. *The STATEMATE semantics of statecharts. ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [16] F. Jahanian and A. Mok. *Modechart: A specification language for real-time systems. IEEE Transactions on Software Engineering*, 1988.
- [17] F. Jananian and A.K Mok. *Safety analysis of timing properties in real-time system. IEEE Trans. Software Eng.*
- [18] Ivar Jacobson Jim Rumbaugh and Grady Booch. *The Unified Modeling Language Reference Manual*. 1998.
- [19] Joseph Kudish Jonah Z. Lavi. *Systems Modeling and Requirements Specification Using ECSAM: An Analysis Method for Embedded and Computer-Based Systems*. DORSET HOUSE PUBLISHING, 2005.
- [20] A. K. Mok. *SARTOR-A design environment for real-time systems. Proc. 9th IEEE COMPSAC, Chicago, IL*, page 174181, 1985.
- [21] Christian Prehofer. *Feature Interactions in Statechart Diagrams or Graphical Composition of Components*. 2002.

- [22] W. W. Royce. *Managing the Development of Large Software Systems*. *Proceedings of IEEE WESCON*, 1970.

Chapter 8

Salvador Garcia: Misuse Cases

One of the most important parts in one system is the security. Security requirements are not enough to analyze all the security in a system, it just considers the functional part. Use cases are not designed to analyze non-functional requirements, so, they are not enough to analyze all the security related with a system. For that is necessary to view the system from a negative way, using hostile cases: misuse cases.

8.1 Introduction

Use cases are *a scenario-based technique for requirements elicitation* [10]. They are a modeling technique for analyze and specify functional requirements in an early stage [7]. They provide scenarios about how the system works when the user is interacting, and help to describe the requirements in a graphical way. A use case defines a goal-oriented set of interactions between external actors and the system. The main advantages of use cases are:

1. They are reusable.
2. They are easy to understand.
3. They represent interaction with the user.
4. They can be integrated with other analysis tools.
5. They are a useful technique for requirement's analysis.
6. They can serve as the basis for the estimating, scheduling, and validating effort.

Actors are users outside the system that interact with the system. A use case is initiated by a user with a particular goal in mind, and completes successfully when that goal is satisfied. It describes the sequence of interactions between actors and the system necessary to deliver the service that satisfies the goal.

However, use cases are focused in the analysis of functional requirements, but not necessarily with non functional requirements like all of those that are related with operational availability, performance, reliability, reuse and security [9] and [3]. It is easy to make some assumptions that can be very important within the project [6], and if we ignore them, some problems can be generated during the next stages, deriving a premature implementation of the design.

Sindre and Ophal [9] defines security as "the prevention of, or protection against, access to information by unauthorized recipients, and intentional but unauthorized destruction or alteration of that information", i.e. the protection of the information against a misuse of the system. On security terms, it is important to consider what are the weakest points in the system. There can be some security holes that because a lack of design are not considered, and, it can be a possible vulnerable point for a security attack.

There are some methods to deal with the security of the system, for example the i-Framework [4], a goal oriented method that works with three elements: actors, intentional elements and links. Some methods create security requirements [9], while others just deal with the necessities of the primary requirements. But, they can be very inconsistent, they are not analyzing exactly in what cases can be applied, they are focused just in the point of view of the user, and sometimes for a overview of a program in general is very hard to discover in what specific parts they are effective.

Also, to deal with the security problems, there are some cryptographic approaches that can help but, if they are not based in a very well designed system, they are not enough to build a secure application. They must work together with a well designed application, that can cover all the situations where they can apply [5].

8.2 Basic Concepts

Use cases describe some functions that the system should do; when they are related with security issues, they model mechanisms to protect the system. For example, in an instant teller machine, one security requirement is the verification of the card holder and his PIN number. It specifies that each time an user introduces his card, he must introduce a PIN to validate that is the cardholder. However, these requirements, are vulnerable to security threats. An undesired user, can try to guess the PIN number, and access illegally to the system.

One approach to address security threat analysis is the development of misuse cases [3]. A set of use cases are taken from a positive "point of view of the user"; misuse cases threat security requirements within a negative scenario [1]. All security requirements exist because people that create methods to attack the systems; employing use and misuse cases can improve security by helping to mitigate threats.

Sindre and Opdahl [9] introduce the concept of Misuse cases to analyze the system in an hostile way. Formally, a misuse case is "a special kind of use case, describing behavior that the system/entity owner does not want to occur". A misuse case is the opposite of an use case; it represents all the things that can attack a system, but, preserving all the properties of use cases.

Misuse cases analyze the interaction between applications and the misusers whom seeks to break the security in the system (misactors). In the uses cases, an actor plays the role of the users that interact with the system; in the misuse cases, a misactor plays the function of the users and situations that can break the system. For example, in a car security system, a possible actor, is the owner of the car; possible misactors are thieves and the weather.

Misuse cases help to analyze threats, but, are not very helpful for the specification of security requirements. Donald Firesmith [3] proposes to include use and misuse cases in the analysis of the system, but remarking the differences between them (Table 8.1).

Table 8.1: Differences between misuse cases and security use cases

	Misuse Cases	Security Use Cases
Usage	Analyze and specify security threats	Analyze and specify security requirements
Success Criteria	Misuser Succeeds	Application Succeeds
Produced by	Security Team	Security Team
Used by	Security Team	Requirements Team
External Actors	Misuser, User	User
Driven by	Asset Vulnerability Analysis	Misuse Cases

8.3 Misuse Case Diagrams

There are three ways to represent misuse cases diagrams (Fig. 8.1). The first way (a) is representing it in terms of normal use cases. It has the advantage that it is in the same terms; both, because both cases are represented at the same time it can be confusing. The second approach (b) is to separate totally the use cases and misuse cases, it easy to see how works each part, but, it is hard to identify the relations among them. It is necessary to see the whole case or requirement.

For a better understanding of the diagram, Sindre and Ophal [9] proposes to draw in black color all the issues related with the misuse cases (c). Representing them in this way, permits

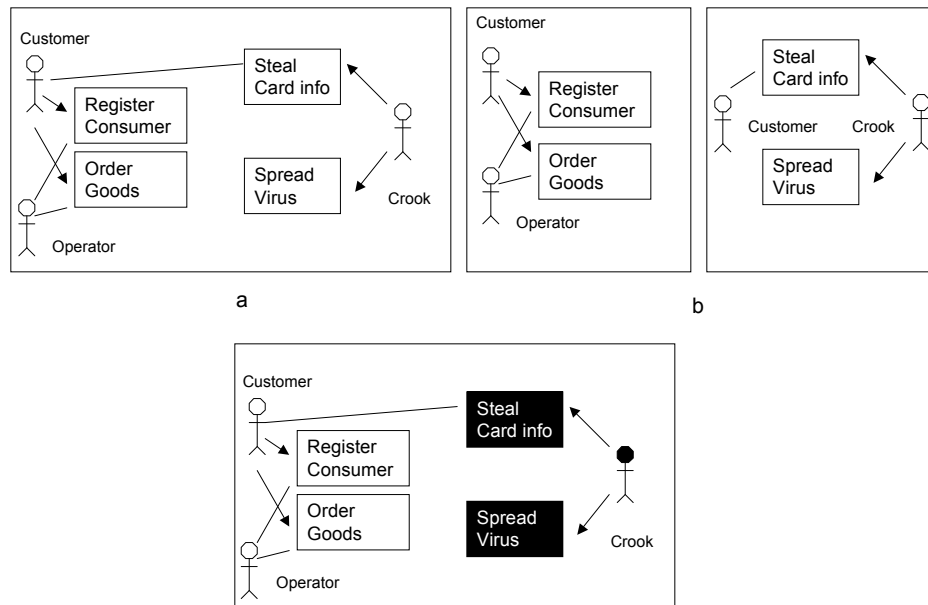


Figure 8.1: Diferent representations of a misuse case diagram.

to the designer to identify in a fast way how a misuse case and a misactor are related with the rest of the system. In this way the diagram is simple and legible. Its easy to distinguish what part of the diagram corresponds to the use and misuse cases and also, all their relations can be included.

Traditionally, an use case represents functional requirements, and misuse cases represent non-functional requirements. Another way of looking at the role of misuse cases is to observe that the typical response to a threat is for the designers to create a subsystem whose function is to mitigate that threat.

8.4 Method Guidelines

The best approaches for choosing the correct and most important cases, are given when the designer thinks about what situations are permitted or not in a system and when possible patterns are analyzed [5], and also, it is necessary to think like an attacker, i.e., from a misactor's point view. For choosing the best approach, McGraw suggests the next guidelines:

1. Brainstorming. Get as many ideas as possible.

2. Answer to general questions applied to all the systems. Try to answer question like these help designers to identify the problem before it is created. Some example questions can be:
 - How can the system distinguish between good and bad input?
 - Can it tell whether a request is coming from a legitimate action?
 - Where the misactor can be positioned?
3. Follow attack patterns. Some examples can be:
 - Make the client invisible
 - Target programs that write to privileged OS resources
 - Attack user-supplied configuration files that run commands to elevate privilege
 - Leverage configuration file search paths
 - Manipulate terminal devices
 - Perform simple script injection
4. Simplicity. Misuse case diagrams should not contain all the possible misuse cases. It is necessary to divide them on simpler diagrams, but, not separating them from the use cases.

8.5 Methodology

Once the main components in a system had been analyzed, Sindre and Ophadl suggest to follow a methodology that helps to put all the pieces together and to discover new components in the system's analysis. [9]

1. Concentrate in normal actors. The design must be focused mainly on the principal actors, regardless of security issues; a clear concept of the system in general will help to get a better security analysis.
2. Introduce the major mis-actors. Potential misactors are persons, but, there can be more possibilities around the system. In an online system two possible misactors can be a hacker and a competitor.
3. Investigate potential relations between misuse cases and cases. Analyze how possible attacks can affect to the system. For example, how a "flood system" in a car security system can affect the functionality in general.
4. Introduce new cases. Considering security issues can derive new cases, and at the same time, new mis-uses.

5. Continue with a more detailed requirements documentation. If there are more details about the requirements, or if , if there are new, we can have a wider view about what new options can be considered regarding to the security issues.

8.6 Template

The template suggested is very similar to an use case template [8], but, we need to focus it into a misuse case interpretation, and also add some extra properties. Following the e-commerce system given in Fig. 8.2, a template for the access control could be defined as follows:

- Case Name: The name of the case.
- Summary: A brief description of the case.
- Author: Person who analyzed the case.
- Date: Date of the case analysis.
- Basic Path. The principal ways that misuse can take to pursue its goal.
- Extension/alternative paths/exceptional paths. Ways that the misuse can take, but, that are not very common. Because sometimes are not very visible, they can be the key point for an attack.
- Preconditions. Preconditions can be of three types:
 - Triggers: Entry criteria, what initiates the use case.
 - Assumptions: Conditions that must be true, but is not guaranteed by the system.
 - Preconditions: Conditions that are ensured by the system.
- Postconditions: They can be of three types:
 - Worst case threat: Describe the outcome if the misuse succeeds.
 - Prevention guarantee: Describes the guaranteed outcome whatever prevention path is followed.
 - Detection guarantee: Describes the guaranteed outcome whatever detection path is followed.
- Related business rules. Particular points depending of the system and the business.
- Iteration. Covering the necessity of a superficial analysis and after, more detailed descriptions. Begin in a upper level, an analyze each step in a more detailed way.
- Primary Actor or misuse profile.

- Scope. The scope of modeling, for example, an entire business, an information system, and so on.
- Level. The level of abstraction. It can be a summary, user goal, or subfunction.
- Stakeholders and risks. List of the various stakeholders and what their motivations are. Depending of the abstraction level, risks can be just described, or try to quantify in costs.
- Technology and data variations. Mention variations without giving the path for each case.

8.7 Eliciting Exceptions and Test Cases

A program failure leads to exceptions. The system under design should respond to undesirable events, and through it, prevent possibly failures [7]. Through misuse case analysis is possible to discover exceptions, and handle them in a proper way. The candidate exception scenarios must be generated and requirements to prevent system failures from a proven list of exception classes must be elicited [1].

Any possible scenario can lead to a test case and misuse cases can help to explore and analyze conditions and exceptions. Products of use / misuse case analysis that can contribute to effective test planning include.

- Specific failure modes. Useful for real time, embedded and safety - related systems.
- Security threats. Useful for distributed commercial and government systems.
- Exception handling scenarios. Always useful.

A test engineer could view misuse cases as existing purely to ensure better system testing. A quality engineer could equally well argue that their purpose is to improve the quality of delivered systems [1].

8.8 Example

Misuse case diagrams are represented within use case diagrams. Fig. 8.2 shows an example about an automated teller machine [3]. The three main requirements are deposit funds, withdraw funds and query balance. There are security cases to control access, ensure privacy, integrity and to ensure nonrepudiation of transactions. These user case are specified to protect the actors (users) from three security threats (misuse cases): spoof user, invade privacy and perpetrate fraud, that can be done by a cracker or a thief (misactors). It is important to remark that a user can activate or not a use case, it is represented by lines and

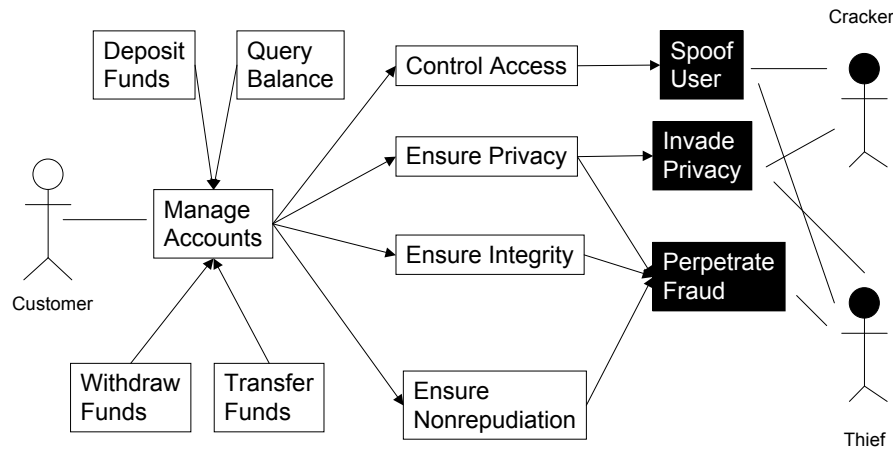


Figure 8.2: Diferent representations of a misuse case diagram.

arrows. When the case must be activated by an user or case, it is represented by an arrow. When it could be activated by an user or case, it is represented by a line. The following table documents the case Access Control, for further information, [3] presents the documentation of the other cases

- Case Name: Access Control.
- Summary: Access control is the extent to which a business enterprise, application, component or center controls access by its externals. It consists of identification, authentication and authorization.
- Author: Donald G. Firesmith
- Date: June 2003.
- Basic Path.
 - Attempted spoofing using valid user identity.
 - Attempted identity and authentication theft.
 - Attempted spoofing using social engineering.

- Alternative paths.
 - The misuser can quit at any point.
- Preconditions.
 - The misuser has no valid means of user identification.
 - The misuser has no valid means or user authentication.
 - The misuser has a valid means of user identification enabling the impersonation of a valid user that is authorized to use a protected resource.
 - The misuser does not have an associated valid means of user authentication.
 - The misuser has basic knowledge of the organization.
- Postconditions:
 - The system shall not have allowed the misuser to steal the user's means of authentication.
 - The system shall not have authenticated the misuser as a valid user.
 - The system shall not have authorized the misuser to perform any transaction that requires authentication.
 - The system shall have recorded the access control failure.
 - The system shall have prevented the misuser from stealing the user's means of identification and authentication.
 - The system shall have identified and authenticated the user
 - The system shall not have authenticated the misuser.
 - The system shall not have authorized the misuser to access the protected resource.
 - The system shall have recorded the access control failure.
- Related business rules. All the rules to realize a deposit, user data, privileges, etc.
- Iteration. No iterations.
- Primary Actor. Cracker and thief.
- Scope. Instant teller machine.
- Level. Summary
- Stakeholders and risks. External damage
- Technology and data variations. Model of the teller machine.

8.9 Conclusions

Use cases is a very good design tool for analyzing functional requirements in an early stage; but, they are not focused in the analysis of non-functional requirements. One of the most important non-functional requirements that must be considered in a system is security. Use cases analyze security requirements, they are considered as functional requirements because the user has a direct interaction and the client asked for them, some of them can be access interfaces, security algorithms, etc.

Security requirements analyze security issues from a positive and straight forward point of view. They just set what security processes that must have a system, like check user and password; but they do not analyze all the possible negative scenarios that can affect or break the system's security. For analyze security threats, Sindre and Ophal [9] suggest a new extension for the use cases: Misuse Cases.

One of the main advantages that they offer is the system analysis from a negative point of view, trying to do exactly what the system does not want. With that, it is possible to see another panorama to protect the system, placing a mis-actor to play the role of a normal actor who tries to attack the system. Also, there is a tool (Scenario Plus) that already had implemented this technique. It had the possibility to draw the diagram and it creates automatically the links between misuse and use cases through searching for underlined use case names with simple fuzzy matching [2].

But, misuse cases techniques are still in a development stage; there is not enough documentation for a deep analysis, and is not clear how can they be implemented during the other stages of the software design; there is not a clear method to get all the misuse case and a big part of the analysis depends on the designer's criteria.

Misuse cases have not been evaluated in practical applications, just in examples. However, the industry could be interested on use them because:

- Misuse cases is a close approach to use cases; they are using the same terms as in UML use cases diagrams.
- Security requirements are a main part in new system, specially those oriented to e-commerce.
- Misuse cases are small, simple and easy to understand.

The research group of the Norwegian University of Science and Technology, and from University of Bergen, already have been contacted some software development companies in Norway. They have been interested in this approach. This area is still in development, but it is already supported, and they are doing the first implementations in real cases, it is not going to be surprising if in few years it is adopted in the design cycle process.

8.10 Exam Questions

1. Explain what is a misuse case and a misactor?
2. Explain the main differences between security requirements and misuse cases.
3. The use case diagram of car security/safety requirements includes the next cases: Drive the car, lock the car, lock the transmission, control the car, control traction and control braking with ABS. Draw and explain the possible misuse cases, misuse actors, and the relations between them.

Bibliography

- [1] Ian Alexander. Misuse cases: Use cases with hostile intent. *IEEE-Software*, 20(1):58–66, 2003.
- [2] Ian F. Alexander. Initial industrial experience of misuse cases in trade-off analysis. *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 61–70, 2002.
- [3] Donald G. Firesmith. Security use cases. *Journal of Object Technology*, 2(3):53–64, 2003.
- [4] Eric Yu Lin Liu and John Mylopoulos. Analyzing security requirements as relationships among strategic actors. *SREIS'02*, 2002.
- [5] Gary McGraw. Misuse and abuse cases: Getting past the positive. *IEEE Security and Privacy*, May:90–92, 2004.
- [6] Joshua J. Pauli and Dianxiang Xu. Misuse case-based design and analysis of secure software architecture. *ITCC*, pages 398–403, 2005.
- [7] Emil Sekerinski. *Computer Science 703: Software Design*. McMaster University, 2006.
- [8] G. Sindre and A. Opdahl. Templates for misuse case description. *Foundation for Software Quality*, 2001.
- [9] G. Sindre and A.L. Ophahl. Eliciting security requirements by misuse cases. *Requirements Engineering*, 45:34–44, 2005.
- [10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6th edition, 2001.

Chapter 9

Software Design Teaching Methods

Despite the flurry of the research activity in software design, the pedagogical aspects of teaching method of software design courses have not been as forth coming. This paper surveys some of the successful methods for teaching software design courses which have been used in the computer and software engineering departments.

9.1 Introduction

In the present day although most of the universities include a software design course as a required or core course of an computer science or software engineering major, some institutions and colleges do not for various reasons.

In this paper we survey some of the software design teaching methods used in computer science departments. It should be noted that although the notion of “Software Engineering” is a relatively young discipline, the issues, models, terminology, notation, and emphasis have varied largely on this important subject. Therefore, teaching software design can be viewed in many different ways. This problem is also caused by the diversity of the models and the resultant diversity of the applications in software engineering.

Before we start to introduce some of the successful methods of teaching of software design used in the computer science departments, we would like to give some interesting notes and remarks taken from the talk given by Mike Godfrey in Cornell University in 1997. Following three question were proposed by Mike Godfrey in his discussion: “On the nature of Software Design:”

1. What is software design?
2. What is good software design?
3. Can good design be taught?

Following explanations are mostly taken from the slides of the talk given by Mike Godfrey, see [22]. Mike Godfrey points out that in order to answer the question “What is software

design?”), we must take into account the references such as process, artifacts, and appearances of software design. In other words, he says that one can examine the structure of a solution to a problem: how to implement what? Mike Godfrey claims that this question involves the process/methodology followed in developing software, the artifact produced, and how the system actually appears to various people. He says that approaches to software design should be careful to allow for the following items:

1. repeatability;
2. tolerance of creativity;
3. flexibility, modularity, adaptability, re-usability. These are necessary to allow for the possibility that needs may change, and things will inevitably go wrong;
4. scalability;
5. scrutability.

We are learning from his discussion that the phases of software design run from the more abstract levels of designing subsystems and sorting out dependencies to the actual coding process of data structure design and algorithm development. The ways doing the things can sometimes be very different from how they appear to be done. Choices to be made include:

1. top-down vs. bottom-up;
2. functional decomposition vs. data-centric;
3. object oriented or not.

Again in the same talk he says that it should be noted that the role of documentation is also very important. Parnas suggests that designing software is similar to proving mathematical theorems. In other words, it is a very messy business, that when finished is then cleaned up and documented nicely.

In his talk, Mike Godfrey discusses the question “What is good software design?” as follows: First of all, the definition of good/quality should be made. In fact this is a philosophically hard question and can be thought analogously as follows: What is good software design? What is quality? The answer depends on the values of evaluators. For instance: Is end-system performance of primary importance? Or are you more interested in ease of maintenance? Or perhaps ease of use? It is clear that the quality will mean different things to a computer company, for instance, than the Department of Defense. About creative design, Godfrey says the following things: In a creative software design, there are conflicting impulses. Even though well-defined, rigid methodologies tend to stifle creativity and slow the design process, such methodologies will tend to entail repeatable results. In fact, there are many instances in which creativity is not a goal. These include things like DoD projects, traffic light controls, nuclear reactor software and so on. In such projects, the main goals will be reliability, robustness, security and the like as opposed to creativity.

With regard to the final question “Can good design be taught?”, although there are some people who answers this question as: “No, it can not be taught, it is a black art”, most of the people in the present day believe that: “yes, it can be taught in a way.” The people who believe the latter claims that there is some structure which can be taught, in particular: well-defined methodologies and good engineering practice. Godfrey says that teaching creativity and innovative design presents different problems. Ben Schneiderman has suggested eight heuristics or “Golden Rules” for user interface design which may also apply to software design in general. These include: simple error handling, informative feedback, and consistency. More information about his interesting discussion can be obtained [22].

In this paper we will basically introduce the following software design and engineering teaching methods.

1. Integrating Testing and Design Methods for Undergraduates: Teaching Software Testing in the Context of Software Design;
2. Teaching Software Design with Open Source Software;
3. Teaching Undergraduate Software Design in a Liberal Arts Environment Using RoboCup;
4. Teaching Software Engineering Using Lightweight Analysis;
5. Teaching Software Engineering Through Simulation;
6. A brief introduction of a course met on the net.

9.2 Integrating Testing and Design Methods for Undergraduates: Teaching Software Testing in the Context of Software Design

In this section we would like to introduce a paper motivated from an industry complaint that students do not directly/easily fit into robust software processes. In this paper a teaching approach is presented, which is basically to combine design concepts with testing methods. In the paper it is proposed that integration of testing and design topics allow students to experience different approaches to the design concepts, and reinforces the life-cycle value of testing. The paper outlines the positive sides and benefits of the approach.

Most of the following notes are taken from the paper [21]. The best description of this course can be given as follows: “An advanced treatment of methods for producing a software design, the testing of that design and ensuing code. Focus is on Object-Oriented analysis and design methods, black-box (functional) testing techniques. The method includes treatment Unified Modeling Language (UML) techniques and their application to software development.”

In the paper it is pointed out that: This method, integrating testing and design, provides many opportunities to improve student learning both in testing and design concepts. One

of the pedagogical reasons for the integration of testing and design is the fact that design and testing fit together well. This means that good design includes test design prior to implementation, and this results in students having a stronger mix of design skills to apply to project works.

From both pedagogical and theoretical point of view, the author lists the reasons for integrating software testing topics with software design topics as follows:

1. Pedagogical: need to provide enough motivation for software testing topics;
2. Pedagogical: need for students to have skills in software and test design prior to their application in project courses;
3. Theoretical: desire to demonstrate the inherent relationship between design methods and test design methods.

Following table indicates the course outline:

1. Review of object orientation;
2. Introduction to testing;
3. Combinational test methods;
4. Structural modeling;
5. Dynamic modeling;
6. State machine design;
7. Inheritance and Polymorphism;
8. Class testing patterns;
9. Architectural modeling;
10. Integration testing and test patterns.

The author points out that the proposed integration topics allows students to experience different approaches to the design concepts, and reinforces the life-cycle value of testing (for example, early test planning). This results in students having a stronger mix of design skills to apply to project course work.

Basically, the approach given in this paper focuses on integrating of Object Oriented testing and design topics with the aim of students enter their project courses by knowing firstly, the application of design and analysis techniques to problems, and secondly, the ability to formulate and execute test plans from the design models they develop, see [4]. The author of the paper believes that the integration also brings with it learning advantages in both design and testing. Especially, students can experience first-hand the motivation for

developing test plans early in the life-cycle, and the ability to use their test-case designs to help analyze the quality of their design artifacts.

The objectives of this course are listed as follows:

1. Develop a suitable object-oriented design based on requirement;
2. Identify strengths and weakness of different software designs;
3. Develop and apply appropriate tests for object-oriented designs.

The author of the paper tells us that finding an appropriate starting point is a crucial issue for successful assimilation of the course material, since much of the knowledge and skills applied in each unit is dependent on the knowledge and skills in previous units and prerequisite OOP courses.

The main purposes of the course is to teach the students fundamental design techniques, and to help them have very good skills and techniques for testing: that testing can become a mental discipline that can help them at all phases of the software life-cycle. Next table summarizes the key motivational points for the testing material: moving up the stages of “Mental Discipline” [3].

1. Testing = Debugging: testing exists to support debugging;
2. Show that software works;
3. Show that software does not work;
4. Reduce perceived risk of software failure;
5. Mental discipline: Plan testing before, during design.

From the paper we are learning the fact that when the students taking the class have the necessary and enough testing fundamentals, the course moves back to design topics and UML notation. These topics include structural (class) modeling, dynamic (interaction) modeling and state machine (State Chart) modeling. It is said in the paper that this integration leads to: State machine design precedes FSM testing, inheritance and polymorphism precedes class testing, architectural modeling precedes integration testing.

The author points out that: the issue of dependency and sequencing among course topics is a crucial issue for the success of this integration method. However, the proper synchronization of topics and assignments is also one of the most important strengths of this method, particularly in helping students understand and absorb their knowledge and experience to grow in their “mental discipline” as testers. It should be noted that even though topics like State Machine Testing has the prerequisite knowledge of State Machine Design, Structural (Class) modeling, and Combinatorial Test Models, the advantage with this method is that most of the dependency amongst the more complex topics is on lower cognitive levels of understanding, such as terms and definitions. Therefore, students may not figure out well a design topic prerequisite, but will have another opportunity to learn that topic while assimilating the equivalent testing pattern.

9.2.1 Advantages of the Method

The author says that this integrated approach provides students with large opportunities to learn and experience these benefits first hand. The testing techniques force a detailed analysis of each of the designs under test, hence students may not figure out the design concept well in the beginning, however, the test model follows this, which requires students to decompose the problem while providing a structured means of assessing the quality (testability) of their design artifacts.

Furthermore, the author says that the integration of design and testing topics improves several subtopics that are more difficult to learn in separate courses. According to him, from a design perspective, system approaches to design are increased and improved by the fact that students learn to plan for testing during (or even before) design. In this way, students are forced to consider system views: Inputs and outputs, system and subsystem responsibilities and to review these issues. From a testing perspective, integrating the material has the advantages of clearly demonstrating the value of the effort of early test planning and design. Hence, students can experience the value of an advanced testing “Mental Discipline”. Another advantage of this integration of topics is in the related area of testing and quality assurance. Since many OOT patterns rely on OOA/D models as a prerequisite, a more “traditional” software testing course must either require design as a background, or can only teach general techniques. As a conclusion, the author states that from an overall learning perspective, the integration of the design and testing material helps prepare the students for project work.

Conclusion and Learnt Lessons

The author tells us that one of the very interesting lessons learnt in this course is that: in some occasions, students who completed the course reported offers for permanent positions based on what they were able to explain of their background in the design, testing and UML. An interesting story of a success of a student took the course can be seen in the paper [21].

We would like to finish the survey of this paper by pointing out following conclusions: The author of the paper says that: This integration method provides many cognitive and theoretical benefits for students. It helps prepare them better for project course experiences, and for successful starts in careers as software development professionals. In particular, integrating testing and design leads to at least five significant learning improvements in topics that are more difficult to achieve in separate courses: These are listed as follows: Design and Testing Topics Improved at the end of the course:

1. (For Design) Systems Approaches;
2. (For Design) Design Quality;
3. (For Testing) Testing as mental discipline;
4. (For Testing) Test planning;

5. (For Testing) Testing and Quality Assurance.

More information about this interesting and nice method can be obtained from the paper [21].

9.3 Teaching Software Design with Open Source Software

Now we would like to introduce another teaching method known as “teaching software design with open source software” by David Carrington and Soon-Kyeong Kim [12].

(Most of the following notes are taken from the paper [12]) In this method open source software tools are used as the main objects of study. In the paper it is pointed out that open source software provides very good resources for teaching software engineering. Students taking the course liked the opportunity to use and explore real world software and to study with open source software tools. The aim of the course was to exposure students realistic software systems and give them experience dealing with large amounts of code written by other people.

After we did these introductory explanations about this course, now we will try to describe the context, goals, teaching way, results of this course as we did in our previous method.

The paper reports the experiences of the authors revising a second year undergraduate software engineering course on design and testing [5] for a class of more than 260 students. The authors say that since many open source projects do not provide design documentation, students can undertake reverse engineering activities to understand the structure of such tools. The emphasis in the course was given to reverse engineering and maintenance activities wishing that students gain experience reading and understanding codes of other people.

We are learning from the paper that open source projects provide a large amount of materials for students to study. Not all are exemplars but it is valuable to expose students to a variety of programming styles and encourage their critical abilities. Hunt and Thomas [10] refer to this type of study as “software archaeology”, which is similar to the medical student pathology laboratory analogously.

The authors point out that incoming students to the course have previously completed two programming courses using Java and are studying a data structures and algorithms course in parallel. The course introduces students to the practical aspects of configuration management using CVS (Concurrent Versions System - an open source version control system).

Assessment for the course consists of four team assignments, a time monitoring exercise [6], and a final open-book exam. The goals of the course can be listed a follows:

1. Demonstrate the concepts and practice of software design and testing, the UML notation, software design patterns,code refactoring, and configuration management;

2. Extend student's programming experience, particularly in terms of program size but also in the use of additional program structures and development methods;
3. Provide students with positive experiences of collaborative learning and some appreciation of the need for life-long learning skills;
4. Expose students to open source software tools and real world code written by other developers.

Now let us introduce the teaching mechanism of the course: Before the teaching semester started, the following list of open source software engineering software tools were prepared with the purpose of that the tools should be potentially relevant to the software engineering activities of students.

1. ArgoUML: a UML modeling tool;
2. Eclipse: an extensible IDE;
3. JEdit: a programmer's text editor;
4. JRefactory: a tool to perform refactoring on Java source code;
5. JUnit: a regression testing framework.

During the teaching semester, the given assignments required a deep knowledge and understanding about the selected open source tool. The first assignment asked each team to select a tool. Each team was required to install and use their tool before giving a five minute presentation to their tutorial class that explained the issues i-) the purpose of the tool, ii-) the installation process, iii -) how to use the tool. We are learning from the paper that the main goal of this assignment was to get teams started and to overcome the difficulties with tool installation and use. A secondary purpose was to display a range of software engineering tools to students so that they were aware of the tools being studied by their peers, and so they could use them if they identified a need.

The second assignment represented investigation of the source code for the selected Java tool and offer an opportunity to receive feedback on the team's progress. In the assignment 2, students had to deliver: i -) source code descriptions, ii -) CVS and UNIX build evidence, iii -) a team learning journal, iv -) a plan for future research.

In the third assignment, each team had to prepare and deliver a 15 minute presentation about the internals of their Java tool covering the issues: i -) the overall tool structure, ii -) the subset studied in detail, iii -) the UML diagrams developed, iv -) any patterns identified, v-) extensions or refactorings planned or completed, vi -) planned or completed testing related to extensions or refactorings.

Results and Discussion of the Method

The authors point out that the use of open source software helped achieve the goals of the course. The response rate to the given survey at the end of the course was good (149 out of 264) because the survey was administered in the tutorial classes. The survey asked about how the team was performing, reactions to the team's Java tool, and course organization. The questions about the Java tool were ("My Java tool is"):

1. easy to understand;
2. giving me real-world software experience;
3. helping me understand software design and construction;
4. helping me improve my programming skills;
5. helping me improve my knowledge of Java;
6. an enjoyable way to learn this subject matter.

The graphical representation of the course evaluation can be seen in the paper [12]. The general conclusion of the authors about the course is that students appreciate the benefits of this approach to studying software design and testing. Nevertheless, at this point in the course, the authors were not convinced that this method was easy or enjoyable. More information about this method can be found the paper [12].

9.4 Teaching Undergraduate Software Design in a Liberal Arts Environment Using RoboCup

Now we would like to introduce an achieved method of the teaching of software design given in the paper titled "Teaching Undergraduate Software Design in a Liberal Arts environment Using RoboCup" by T.Huang and F. Swenton, [18]. In this paper the curriculum, client software, and benefits of an undergraduate software design course based on the RoboCup software simulation environment. This course utilized the RoboCup to teach software design to undergraduate students in a liberal arts environment. Most of the following notes are taken from the paper [18]. In this paper authors present a software design course, which is tailored to undergraduate computer science students within a liberal arts environment, based on the RoboCup soccer simulation platform. Some factors make this course suitable for use in undergraduate curriculum. Firstly, RoboCup is an research and education platform with clear objectives and reliable, readily installed server software. Secondly, an easy objective and a set of simple communication protocols make the systems easy for instructors and students to learn. Thirdly, the platform allows discussion of a variety of computer science topics such as real-time programming, multi-agent systems, and artificial intelligence. Fourthly, some

publicly available client software options make it possible to customize courses for a range of time constraints and student backgrounds.

Now, here we will simply try to introduce the goals, features, and pedagogical benefits of the RoboCup platform surveying the paper and the comprehensive details of RoboCup platform can be found in [17]. RoboCup is an international initiative whose main goal is to encourage research in artificial intelligence and robotics by providing a standard environment in which a wide range of technologies and algorithmic approaches can be explored and evaluated. As the name suggests, the standard problem is a soccer match played by robots. International competitions take place regularly in a software simulation league as well as several physical robot leagues with different size and sensor restrictions.

We are learning from the article that: In the RoboCup software simulation league, two teams of up to 11 simulated autonomous robots play soccer on a virtual field using simplified but fairly realistic models for physics, sensors, communication, and player fatigue. For any given game, a single soccer server process runs the simulation, generates all sensor information, and communicates with all players. Each player runs as a separate client process that receives sensor information from the server at fixed intervals and generates real-time actions. These clients can run on the same computer as the server or on separate computers. All communication among players takes place via the server using a predefined communications protocol. The structure of the simulation environment provides a challenging domain for research in many areas, including multi-agent, uncertain reasoning, realtime decision-making, machine learning, sensor fusion, and software design and management.

Now we will try to explain the role of the RoboCup in education: The RoboCup initiative is to provide a platform for project-oriented educational experiences. The authors claim that with a well-established client-server protocol and a substantial body of publicly available client source code, the RoboCup simulation league provides a rich environment within which undergraduate students can learn about larger-scale software design and analysis, software tools and environments, and teamwork. Furthermore, the well-defined objective and the competitive framework encourage students to develop evaluation methodologies and to measure progress carefully. The platform itself has been evaluated in the context of its pedagogical utility, though not specifically as the basis for a software design course, see [25] for details.

The main goals of the course were to provide computer science students opportunities to design and develop larger-scale software projects and to work in a programming team. In the paper it is said that on the theoretical side, the RoboCup environment encouraged students to address basic issues in artificial intelligence, synchronization and decision-making under uncertainty, and coordination among autonomous clients. During the semester, four assignments were given and each assignment involved modifying behavior a RoboCup client program. Detailed information about assignments can be obtained from the main paper [18].

Now we will try to explain the client program, and client considerations. It is described in the paper that: Krislet [26], a Java-based client, provides an interface to the RoboCup server. In the course the Krislet was chosen to use as a starting point and to build upon it for the students, on the principle that the students would be better served by extending a

basic client.

From the paper we are obtaining the knowledge that in order to teach students some basic principles of client control and to provide simple examples of their implementation, the authors modified Krislet by adding several utility and client control features. In addition to providing an immediate entry point for the development of Robocup clients, the code illustrates several basic concepts and methods such as multi-threading and sharing of dynamic data, local and global coordinates, and push-down automata, inviting the students to explore and refine these aspects of the code to suit their client's needs. The outline and some features of the NK client and their importance to the course can be obtained from the main paper [18].

Student Responses

At the close of the term, the course evaluations were given to the students for course assessment. Over two-thirds of the students agreed with the statement, "Building RoboCup players has helped improve my programming skills substantially," and nearly %90 agreed with the statement, "Building our RoboCup players has helped me to improve my ability to communicate and work with others on a software design project. "Every student in the course agreed with the statement, "If I had it to do all over again, I would still choose to take the RoboCup course this winter term. "In reaction to the question, "What did you most enjoy about this course?" the written responses fell into several categories: i -) competition (9 responses), ii -) teamwork (4 responses), iii -) introduction to new concepts in computer science (4 responses), iv -) seeing their clients' performance (4 responses). In response to the question, "What was the most challenging aspect of building your RoboCup teams?" the students identified: i -) teams/cooperation, both of students and of players (7 responses), ii -) implementation of strategic ideas (3 responses) iii -) reasoning under uncertainty (2 responses)

Conclusions

In the paper it is told us that this course is the first course to utilize RoboCup to teach software design to undergraduate students in a liberal arts environment. The authors believe that the assignments and NK client software provide a convenient starting point for instructors wishing to utilize the RoboCup platform. According to the course evaluations, we are getting the idea that most students agreed that the course helped them improve both their software development abilities and their communication as well as teamwork skills. More information about this method can be obtained from the paper [18].

9.5 Teaching Software Engineering Using Lightweight Analysis

Now we will try to introduce a teaching method of Software engineering rather than a teaching method of software design. The proposed approach in the paper has not been yet applied and used in a class but the author plans first to apply in a pilot course and then he hopes to extend this application.

The goal of this project is to explore a new pedagogical approach to teaching software engineering centered on the close integration of analysis tools with instruction in programming methodologies. The author claims that instead of learning methodologies as abstract ideas, students will benefit from applying analysis tools that embody methodologies to large, realistic programs. Most of the following notes are taken from the paper [20]. According to the author, the presented approach in this paper will allow for realistic experience with industrial scale programs, and enable direct application of theory and methodology to practical programming.

In this paper, the focus is on the use of lightweight analysis tools that offer clear and immediate benefits with minimal initial costs. These tools include LCLint [13], [14], [15], [8], a lightweight static analysis tool that exploits annotations added to the program source code; the Extended Static Checker for Java (ESC/Java) [8], a static analysis tool that incorporates an automatic theorem prover; and Daikon [9], a tool for automatically determining likely program invariants. Examples of topics where these tools can be used for pedagogical benefit including information hiding, invariants, memory management and security.

Objectives and Description of the Method

The author of the paper makes the following observation in the beginning of the article: Students often regard the methods and theories taught in software engineering courses as abstract, academic concepts. Without experiencing their practical impact on realistic programs, students rarely develop a deep understanding or appreciation of important ideas in software engineering.

Although Edsger Dijkstra [7] and David Gries [23] proposed elegant approaches to teaching programming that closely integrate proof techniques, they were not accepted as successful. Since there is a big gap between formal methods and practical programming experience of students, the previous attempts have met limited success. The author believes that the lightweight analysis tools are a promising way to shrink this gap. Although lightweight analysis tools require significant effort to use traditional formal methods, students can easily and effectively use lightweight analysis tools on real programs.

The author concentrates on lightweight analysis tools since he claims that they are readily accessible and provide clear and early benefits for limited effort and using lightweight analysis tools, students will be able to work with and modify industrial programs. Now, let us describe the analysis tools that were intended to use.

Analysis Tools

The author points out the fact that the range of selected tools, LCLint, ESC/Java, Daikon, provides a sampling of the design space of analysis tools where efficiency, effort required, soundness and completeness are often conflicting goals. Detailed description of these tools can be seen in [20]. Pedagogical Uses of Analysis tools can be listed as follows: Information Hiding, Invariants, Security Vulnerabilities, Extensible Checking, and a detailed treatment to each of these uses can be seen in the main paper [20].

Evaluation Plan

The author asks the following evaluation question: can lightweight analysis tools be used to improve teaching of software engineering? He says that Software engineering is not yet at the point as a discipline where there are clear and objective metrics for measuring a student's ability. Again, we are learning from the paper that some of the subjective criteria to evaluate the success of the pilot course include: Do students gain a better understanding of abstract concepts by using analysis tools? Do students become better software engineers because of their experience with analysis tools? Are students able to efficiently manage and manipulate larger programs than with previous techniques? Do students develop original checking rules and use them effectively? The author uses surveys to assess the effectiveness of the approach which can be learnt better and in detail from the paper [20]. The author notes that he has a good track record of supporting LCLint use in teaching and industry, and believes that continued efforts to support use of lightweight analysis tools in education will contribute to both educational and research goals. More information about this teaching method can be obtained from the main paper [20].

9.6 Teaching Software Engineering Through Simulation

This paper introduces plans for SimSE, a detailed, graphical, fully interactive educational software engineering simulation environment that teaches the software process in a practical manner without the time and scope constraints of an actual class project. This project is still not completed but ,once completed, the author claims that, this tool will enable students to form a concrete understanding of the software process by allowing its users to explore different approaches to managing the software process and giving them insight into the complex cause and effect relationships underlying the process.

The author of the paper believe that software engineering education will gain a new method of teaching that is aimed at effectively introducing students to the software process as experienced in real-world software engineering projects. Furthermore, the author believes that industrial organizations will also be able to leverage this environment: by using SimSE with models that reflect their organization and processes therein, new employees can be more quickly and successfully trained.

From the paper we are learning the fact that: This research project is based on the hypothesis that a game-like, educational software engineering simulation environment is a solution to the problem of teaching the software engineering process [16]. It is told us in the paper that simulation should not replace existing educational techniques, but rather, serve a complementary role. In particular, lectures are still required to introduce the topics to be simulated and class projects are still required to demonstrate and reinforce some of the lessons learned in the lectures and simulations.

In the paper the author explains to us that Simulation/adventure games, such as The Sims [1] and SimCity [2], provide a tremendous source of experience and technology that can successfully be adapted to illustrate the software process.

It is said to us in the paper that the basic architecture of SimSE environment contains three main components: (i) a generic simulation engine, (ii) a graphical user interface, and (iii) simulation models. A detailed discussion of each of these components can be obtained from the main paper [28].

Course Evaluation

The author says to us that SimSE will be used in an introductory software engineering class to understand if its use helps students in achieving a better understanding of the software process. The author says that in particular, he will be using three different techniques to evaluate the effectiveness of SimSE. First, the students will be presented with surveys, both during the class and afterward. Second, he, the author, will compare the grades of one session of the course, in which students will be introduced to the simulation materials, to the grades of another session, in which students will not be introduced to those materials. Finally, the author will track the grades these students earn in subsequent software engineering classes and do the same comparison.

Now we would like to give syllabus and mention aims and learning outcomes of software design-I and II undergraduate courses [11] and [24] in order to have some ideas about what the other people are studying in software design course. First the syllabus and aims of the software design-I course can be listed as follows:

1. to provide skills needed in object-oriented program development and software design;
2. to teach the basic strategies for modeling, analyzing and designing software projects;
3. to teach the foundations of software development processes;
4. to develop logic-based specifications of software systems.

At the end of the course it is expected that students would have acquired knowledge of:

1. Object-oriented programming, classes, interfaces, use of inheritance and polymorphism and the role of software architectures;

2. Foundations of software development processes and the role of design in the software development life-cycle;
3. Systems modeling, analysis and design across both architectural and behavioral specifications;
4. Modeling and development methodology;
5. Principles and techniques for the engineering of large software projects;
6. Fundamental principles of formal specifications, including state, operation and class schemas.

In the similar way, upon completion of the course students would have obtained the skills of:

1. Decomposing problems and designing software architectures;
2. Producing static and behavioral models of software programs;
3. Applying software design methodologies;
4. Developing formal specifications from informal requirements of software systems;
5. Implementing software models in a structured and efficient way.

The aim of the software design-II course is basically to teach the principles of Human Computer Interaction and the Design of Interactive Systems, including Desktop Windowing systems and the Web and the syllabus of this course can be listed as follows:

1. Overview of Human Computer Interaction
2. The PACT Framework for Designing Interactive Systems
3. Human Cognition
4. Input/Output Technologies and Interaction Styles
5. The Design Process
6. Accessibility, Usability and Engagement
7. Principles of Good Design
8. User Requirements Analysis
9. Conceptual and Physical Design
10. Envisionment and Prototyping
11. Evaluating Interactive Systems Design
12. Designing for the Web

9.7 Conclusion and Discussion

In the literature there are some other methods about teaching software design, some of which can be listed as follows: i -) A teaching laboratory and course programs for embedded software design, [19]; ii -) Teaching software design tools via design patterns, [29]; iii -) Immersive visual Modeling: Potential use of virtual reality in teaching software design, [27]. We can easily conclude that although there is not a traditional way to teach software design, the standard concepts such as verification, modularization, testing, design patterns, modeling of software design are studied in almost every method about teaching software design. We are observing in this survey that in the almost every method, main tools given to the students are object oriented programming languages, Unified Modeling Language (UML), and students have to know at least one programming language and preferably it is an object oriented one.

9.8 Exam Questions

1. In the first introduced method in this survey, according to the author of the method, what are the rational and benefits of the integrating testing and design concepts?
2. Pick two introduced method in this survey arbitrarily, and describe and compare the evaluation tests to measure the understanding of students in those methods?
3. Do you prefer to study this course in a very specific methods such as RoboCup, SimSE, and maybe using open source tools? (Personally, I do not!)

Bibliography

- [1] Electronic Arts. Electronic Arts. The Sims. 2000.
- [2] Electronic Arts. Electronic Arts. SimCity 3000. 1998.
- [3] Boris Beizer. *Software Testing and Quality Assurance*. Int. Thompson Computer Press, 1996.
- [4] Robert V. Bider. *Testing Object-Oriented Systems-Models, Patterns and Tools*. Addison and Wesley, 1999.
- [5] David Carrington. Teaching Software Design and Testing. In *28th Annual Frontiers in Education Conference*, pages 547–550, 1998.
- [6] David Carrington. Time monitoring for students. In *28th Annual Frontiers in Education Conference*, pages 8–13, 1998.
- [7] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

- [8] et al. D.L. Detlefs. Extended Static Checking. Technical Report 159, Carnegie Mellon University, 1998.
- [9] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Springer-Verlag, 2000.
- [10] A. Hunt et. al. Software Archaeology. In *IEEE Software*, volume 19, pages 20–22, March 2002.
- [11] Arosha Bandara et al. Software Engineering - Design I. <http://www.doc.ic.ac.uk/teaching/coursedetails/220>.
- [12] David Carrington et. al. Teaching Software Design with Open source software. In *33th ASEE/IEEE Frontiers in Education Conference*, November 2003.
- [13] David Evans et al. LCLint: A Tool for Using Specifications to Check Code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.
- [14] David Evans et al. Static Detection of Dynamic Memory Errors. In *SIGPLAN Conference on Programming Language Design and Implementation.*, May 1996.
- [15] David Larochelle et al. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *10th USENIX Security Symposium.*, August 2001.
- [16] J. M. Randel et al. *The Effectiveness of Games for Educational Purposes: A Review of Recent Research. Simulation and Gaming*, volume 29(3). Springer-Verlag, 1992.
- [17] M. Chen et al. RoboCup Soccer Server Users Manual, 2002. <http://sserver.sourceforge.net/docs/manual.pdf>.
- [18] Timothy Huang et. al. Teaching Undergraduate Software Design in a Liberal Arts environment Using RoboCup. In *IT/CSE 03*, June 2003.
- [19] Yu-Lun Huang et al. A Teaching Laboratory and Course Programs For Embedded Software Design. In *IT/CSE 03*, pages 1–5, March 2005.
- [20] David Evans. Teaching Software Engineering Using Lightweight Analysis. June 2001.
- [21] Stephen Frezza. Integrating Testing and Design Methods for Undergraduates: Teaching Software Testing in the context of Software design. In *32rd ASEE/IEEE Frontiers in Education Conference*, November 2002.
- [22] Mike Godfrey. Teaching Design, Ph.D professional seminar, 25 February 1997. <http://www.cs.cornell.edu/html/Courses/Spring-97/CS706/godfrey.html>.
- [23] David Gries. *The Science of Programming*. Springer-Verlag, 1981.

- [24] Frank Kriwaczek. Software Engineering - Design II. <http://www.doc.ic.ac.uk/teaching/coursedetails/222>.
- [25] J. Kummeneje. RoboCup as a means to research, education, and dissemination, licentiate of Philosophy thesis, 2001. <http://www.dsv.su.se/johank/publications/2001/licentiathesis.pdf>.
- [26] K. Langner. Krislet RoboCup client. <http://www.ida.liu.se/frehe/RoboCup/Libs/Sources/krislet-O.l.tar.gz>.
- [27] Bruce J. Neubauer. *Immersive Visual Modeling: Potential Use of Virtual Reality in Teaching Software Design*. Addison and Wesley, 2003.
- [28] Emily Oh. Teaching Software Engineering through Simulation. In *Workshop on Education and Training (WET)*, Santa Barbara, CA, July 2001.
- [29] Yonglei Tao. Teaching Software Tools via Design patterns. In *IT/CSE 03*, December 2000.

Chapter 10

Lutfi Azhari: Software Documentation Environments

People who are involved with programming activities know how difficult it becomes to maintain (large/complex) codes. Programs that you have written yourself are difficult enough to understand when you come back to them later. Things are more difficult when the program has been written by someone else or a big team, and has been modified so many times that the existing documentation no longer accurately reflects the program code and structure. This is a nightmare still awaiting those who are involved with program maintenance, comprehension, audit, or analysis. Tools that automate the process of code analysis and documentation help to overcome this problem painlessly.

Software documentation can mean so many things. *Code* documentation is what most programmers mean when using the term software documentation. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. This documentation is usually embedded within the source code itself so it is readily accessible to anyone who may be traversing it.

Often, tools such as Doxygen [6] , Javadoc [10] or DOC++ [1] can be used to auto generate the code documents; that is they extract the comments from the source code and create reference manuals in such forms as text or HTML files. Code documents are often organized into a reference guide style, allowing a programmer to quickly look up an arbitrary function or class.

Software documentation goes far beyond comments that describe the source code. Unlike code documentation, *design* documents tend to take a much more broad view. Rather than describe how things are used, this type of documentation focuses more on the *why*. It explains the rationale behind organizing a data structure or a class in a particular way, points out patterns, and even goes as far as to outline ideas for ways to improve it later on. None of this is appropriate for code documents, but it is important for design.

In this paper, I survey a set of code documentation tools. An overview of the tool is presented along with examples that show how to use them. Moreover, I scratch the surface of literate programming and documentation of the design and architecture of software systems.

```
/**
 * PrintArgs prints a list of command line arguments.
 * @see java.lang.Object
 * @author Lutfi Azhari
 */
public class PrintArgs {
/**
 * main is the entry point to printArgs
 * @since 1.0
 * @version 1
 * @param args an array of printArgs command line arguments
 */
    public static void main (String [] args)
    {
        for (int i = 0; i < args.length; i++)
            System.out.println (args [i]);
    }
}
```

Figure 10.1: PrintArgs.java

10.1 Javadoc

Javadoc is an SDK tool that parses the declarations and documentation comments in one or more Java source files. It also produces an associated set of HTML files that describe public and protected classes, inner classes, interfaces, methods (including constructor methods), and fields.

Javadoc works in concert with the Java compiler [15]. Javadoc calls a portion of the compiler to compile only declarations; the compiler ignores member implementations such as method code bodies. With the Java compiler, Javadoc creates an internal representation of classes that includes class hierarchies and other relationships. That representation, along with user-supplied documentation, forms the basis of Javadoc-generated HTML files that comprise the documentation.

How to Write Doc Comments for the Javadoc Tool

The best way to get comfortable with Javadoc is to play with an example. See Fig 10.1 for the `PrintArgs` application.

Documentation comments begin with `/**` and end with `*/`. Each comment provides descriptive information on a class member, followed by a tag section that completes the comment by specifying various commands. The tag section begins with an `@` character and continues to the `*/`. As a result, no descriptive information can follow the tag section.

Briefly, a tag is a Javadoc command that provides special processing. Each tag begins with an @ character followed by a keyword. Examples include:

- `@see` provides a see-also reference
- `@author` identifies the source code's author. The `-author` Javadoc option is needed to process the `@author` tag
- `@return` specifies the return value of a method
- `@param` specifies the parameters expected by a method

Doclets

Javadoc contains many other features. For example, it uses small documentation programs called *doclets* [14] to customize the output and format of generated documentation. The default doclet produces HTML documentation, but it's possible to create custom doclets for generating documentation in formats like XML or RTF. (Javadoc's `-doclet` option identifies a custom doclet.)

Writing a customized doclet is simple. To do so, we write the Java program that constitutes the doclet. The program should import `com.sun.javadoc.*` in order to use the doclet API. The entry point of the program is a class with a `public static boolean start` method that takes a `RootDoc` as a parameter. After compiling the doclet, we run the Javadoc tool using the `-doclet myDoclet` option to produce the output specified by the doclet, where `myDoclet` is the fully-qualified name of the starting class mentioned above.

Conclusion

Javadoc is an extensible tool used for code documentation. It extracts comments from the java source files, and produces an associated set of HTML files that describe classes, methods and fields. Javadoc's extensibility comes from creating custom doclets to produce documentation output in different formats like XML, RTF, as well as HTML.

As the vocabulary of Javadoc is not heavily typed, and there is only a limited number of tags that the programmer need to know, it is easy to learn how to effectively use the tool. This makes Javadoc simple enough to actually get used while coding.

10.2 Doxygen

Doxygen is a documentation system for C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors) and to some extent PHP and C#. It automatically parses source and header files in a directory and hence generates function lists and class/UML diagrams. Furthermore, comments in the appropriate format are parsed and used for the documentation.

```

/**
 * Class documentation appears here
 */
class someClass{
    ///variable documentation
    int var;
    /**Documenting Method 1 */
    void Method1();
    ///Documenting Method 2
    void Method2();
};

```

Figure 10.2: Comment style in Doxygen

It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LATEX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it easy to keep the documentation consistent with the source code.

You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. You can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, class hierarchies, and collaboration diagrams, which are all generated automatically.

Documenting in Doxygen

Like Javadoc, Doxygen automatically generates documentation by analyzing the comments the programmer has put in his/her code. Doxygen derives documentation from comments of two basic forms, `/** */` and `///` (see Fig 10.2). Doxygen documents whatever immediately follows the doxygen comment.

In some cases it may be desirable to put documentation after an item, in which case you can use the commenting form `/**< */` and `///<` (Fig 10.3).

There are a variety of commands that can be used within these special comment blocks. Doxygen commands are preceded by either a `\` or an `@` (either is fine). Some of the more common ones are `param`, `return`, `date`, `author`, `file`.

Special commands for method documentation can be used as follows:

```

/**
 * Takes an integer and squares it
 * \param a The integer to be squared
 * @return The integer squared*/
int square(int a);

```

```
/**  
 * Class documentation appears here  
 */  
class someClass{  
    int var;    ///< variable documentation after  
    void Method1(); /**< Documenting Method1 after */  
    void Method2(); ///< Documenting Method2 after  
};
```

Figure 10.3: Trailing comments in Doxygen

Conclusion

Like Javadoc, Doxygen can be used to generate documentation by extracting comments embedded in source code. Several output formats like HTML, RTF, PDF are supported. The simplicity of comment format makes Doxygen an excellent candidate when selecting a documentation tool. Unlike Javadoc, UML class diagrams, class hierarchy diagrams and inheritance diagrams can be produced automatically from undocumented source files.

10.3 DOC++

DOC++ [1] is a documentation system for C, C++ and Java. It generates TeX output for high quality hard copies and HTML output for online browsing of the documentation. The documentation is extracted directly from the C/C++ headers and source files or Java class files.

As C++ and Java are object-oriented languages, class hierarchies are an important aspect of documentation that need to be supported. The best way to read such a hierarchy is by looking at a picture of it. Indeed, DOC++ automatically draws a picture for each class derivation hierarchy or shows it with a Java applet in the HTML output (see Fig 10.4).

Writing Comments in DOC++

Similar to Doxygen, documentation comments are of the following format:

- `/** ... */`
- `///
...`

Such comments are referred to as *DOC++ comments*. Each DOC++ comment generates a manual entry for the *next* declaration in the source code. Trailing comments can be used too.

Every DOC++ comment defines a *manual entry*. Manual entries are structured into various *fields*. Some of them are automatically filled in by DOC++ while the others may be

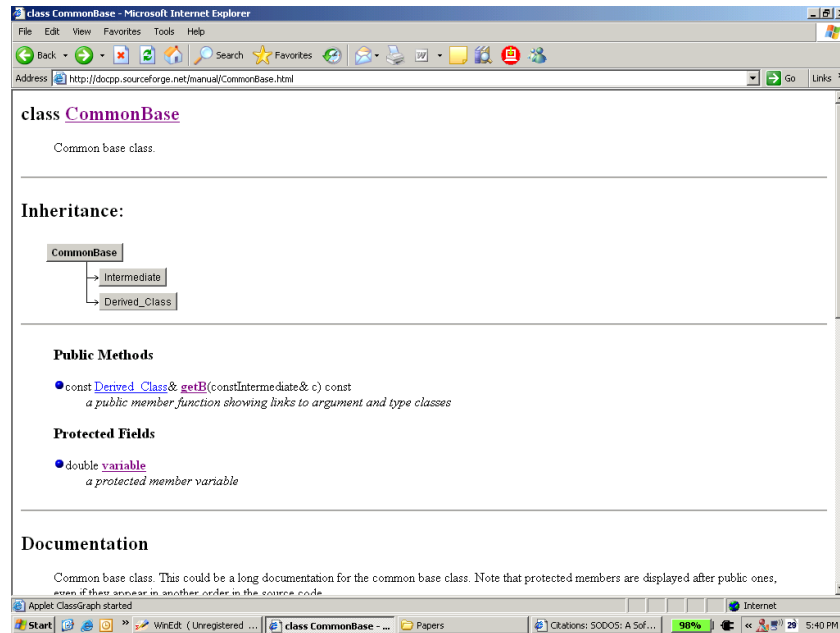


Figure 10.4: Snapshot of an HTML documentation page generated by DOC++

specified by the documentation writer. Among the various fields are, `@doc`, `@param`, `@memo`, `@exception` ...

DOC++ automatically imposes a hierarchical structure to the manual entries for classes, structs, unions, enums and interfaces, in that it organizes members of such as *subentries*.

Additionally DOC++ provides means for manually creating subentries to a manual entry. This is done via *documentation scopes*. A documentation scope is defined using a pair of brackets:

```
//@{
...
//@}
```

just like variable scopes in C, C++ or Java. Instead of “`//@`” and “`//@`” one can also use “`/*@*/`” and “`/*@*/`”. All the manual entries within a documentation scope are organized as subentries of the manual entry preceding the opening bracket of the scope (see Fig. 10.5).

In addition to this, Java allows the programmer to organize classes hierarchically by means of “packages”. Packages are directly represented in the manual entry hierarchy generated by DOC++. When a DOC++ comment is found before a ‘package’ statement, the documentation is added to the package’s manual entry. This functionality as well as documentation scopes are extensions to the features of JavaDoc [21].

```
/**@name Parameters */
//@{
/// the first parameter
double a;
/// a second parameter
int b;
//@}
```

- **parameters**
 - **double a**
the first parameter
 - **double b**
a second parameter

Figure 10.5: Subentries for a manual entry as generated by DOC++

Similar to Java’s packages, C++ comes with the “namespace” concept. The idea is to group various class, functions, etc. declarations into different universes. DOC++ deals with namespaces in the same way it does with packages.

There is one more special type of comments for DOC++, namely “//@Include: <files>” and “/*@Include: <files>*/”. When any of such comments is parsed, DOC++ will read the specified files in the order they are given. Also wildcards using “*” are allowed. It is good practice to use one input file only and include all documented files using such comments, especially when explicit manual entries are used for structuring the documentation.

10.4 SODOS

SODOS [9] [8] (Software Documentation Support) is a computerized environment which supports the definition and manipulation of documents used in developing software. The SODOS environment differs from others in that it is based on a database management system (DBMS) and an object-based model of the Software Life Cycle (SLC). All of the documents used during the life-cycle of a software project are stored in a project database. The documents are developed using a predefined document structure and a set of document

relationships. The document structure consists of the chapter, section, paragraph and figure hierarchy, and the document relationships consist of mappings between components of a single document or several documents.

Viewing the development of software as an information management problem, the solution used here is to define all information entering the environment to be part of the structured database. The various documents generated during the SLC are the principal objects that are stored in the database. The classification of objects and the inheritance rules followed in the Class hierarchy of Smalltalk-80 [7] are ideal for defining the structure and the operations on the objects defined in the model of the SLC environment.

Using SODOS on a Software Development Project

The framework provided by SODOS is available from the start of the project through specification and implementation and finally to the maintenance and evolution of the completed software product.

At the beginning of a software project, a document administrator would be responsible for defining a set of software document standards and software development methodologies. Each document type is subsequently defined by him, in SODOS, using a database schema that describes the document structure, internal relationships, keywords, and related documents. After the document types have been defined, the project database model is then created and loaded based on the set of document types and methodologies chosen for the project. The requirements, design and implementation documents are entered by the software developer within SODOS. SODOS supports this entry by acting as both an editor and by reflecting the form of each document type. Moreover, it supports the developer as he highlights keywords and relationships within the documents. As each document is updated, the document components and the relationships among components are defined and entered into the database. Each component consists of text for either requirements, design modules, code elements or test procedures. Each relationship consists of a link between the component being defined and another component either in the same document or in another document.

The final product of the software development project is a database containing the Requirements Documents, the Design Documents, the Implementation Notebook, and the testing specifications, plans and procedure documents. The relationships among the components in each document are also stored in the database, based on the SLC information model representation of the documents. The database can then be used during maintenance and evolution of the software system to more easily test the application or change the requirements, the design specifications or the implementation.

During the maintenance or evolution phase, all of the requirements, design, implementation and testing documents are available to the maintenance personnel. Changes in requirements and the corresponding changes in design and implementation can be traced using the relationships stored in the database. Queries may be issued against the database to determine design modules, code or tests associated with the requirement which has changed. The maintenance personnel then use the SODOS interface to update the Requirement Document,

the Design Documents, the Implementation Notebooks and the testing documents. These changes create new versions of the database which are then used for subsequent changes.

Conclusion

SODOS is an attempt to support Software Life Cycle document production through the application of database technology. SODOS requires the system/software engineer to identify keywords or key elements as he creates his documents. Making these identifications requires only marginal extra effort due to the user-friendly interface. The resulting database contains all documents, code, test data and relationships both within and between documents. These relationships may be used to determine consistency within a document, structural completeness of a document and traceability across documents.

10.5 SLEUTH

The SLEUTH [16](Software Literacy Enhancing Usefulness To Humans) system provides a mechanism to link a collection of documents using typed hypertext links. It consists of both an authoring and a viewing environment. It also provides the facility to perform keyword searches on the text of the documentation. It applies information retrieval techniques to software documentation in order to provide a mechanism that allows a variety of users to find the answers to questions about software documentation quickly and consistently. Specifically, given a collection of software documents, a user should be able to find the answer to a specific question or a broad array of information on a general topic with equal ease.

SLEUTH utilizes Adobe FrameMaker¹ as the basis for both the authoring and viewing environments. FrameMaker provides a WYSIWYG (What You See Is What You Get) editing environment and supports both hypertext and cross referencing. It also provides basic navigational features and provides a toolkit for customization. Many of the basic features necessary for document creation and editing are provided, allowing effort to be concentrated on more specialized features. FrameMaker can be used to produce effective hardcopy versions of the documents because it is a document preparation system. In addition to the basis provided by FrameMaker, the major components of the SLEUTH system are:

- a search engine which allows full text searching on the documents in the collection. The search engine allows a user to locate information on a specific topic if it is included in the set of documents.
- the interface to that search engine.
- configurable hypertext and cross-reference filter generators,
- and the facility to produce a directory-structured index for the system and library source code.

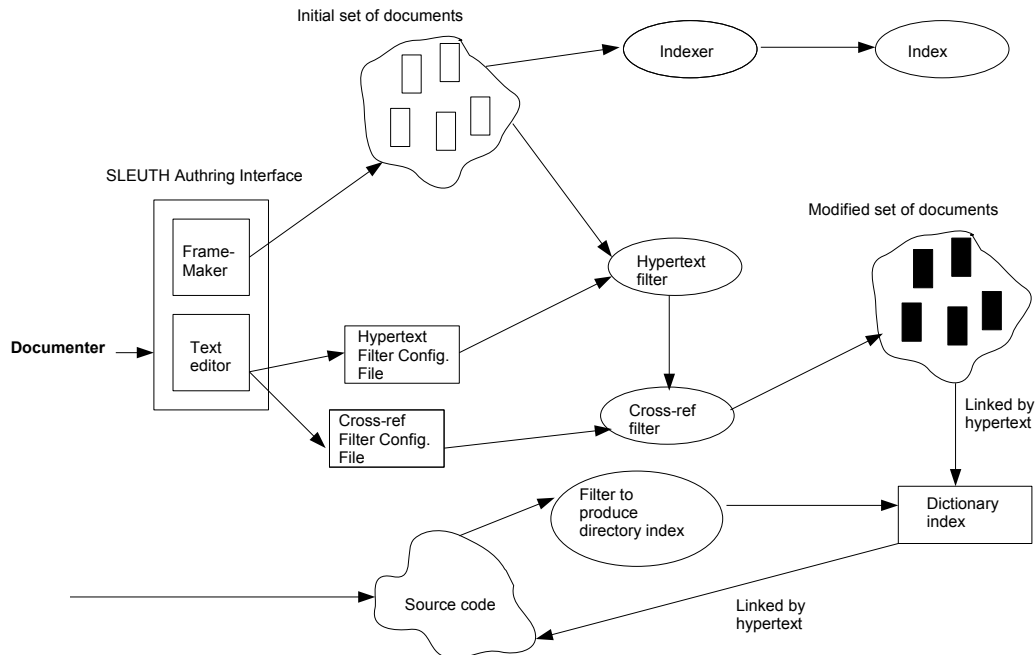


Figure 10.6: SLEUTH as seen by the documenter

SLEUTH as seen by the documenter (see Fig. 10.6 [16]) consists of the FrameMaker editing environment, configurable filter generators, a search engine and the facility to produce a directory-structured index for the source code. When the initial set of documents has been completed and the configuration files have been updated, the custom hypertext and cross-reference filters are generated for the user. The initial set of documents is then passed through these filters to produce the modified set of documents. The text contained in the initial set of documents is also indexed for use by the search engine. The indexing process does not modify the documents. Finally, a directory-structured index is produced for the source code. This index provides links into the source code, but the indexing process does not modify the source code. The modified set of documents, directory index and source code will be visible to the users.

The Authoring Interface The interface presented to the documenter is the standard FrameMaker WYSIWYG editing environment. The documenter uses FrameMaker to compose the initial set of documents and any associated figures and tables. The SLEUTH system provides a document template which defines paragraph types, the formats to differentiate the typed hypertext links and other document formatting information.

¹<http://www.adobe.com/products/framemaker>

The Search Engine The SLEUTH system currently utilizes a WAIS (Wide Area Information Server) [12] search engine. WAIS is intended for distributed information retrieval and based on a clientserver model of computation. The indexer is the only portion of the search engine which concerns the author. The indexer creates a table of the document and paragraph locations of terms in the documentation text. Terms such as "a", "an", "the", etc. are not indexed. The documentation should be re-indexed whenever changes are released to the users.

The Filters While composing these documents, the author uses a text editor to configure the filter generators for the hypertext and cross reference filters. The author provides a list of terms that will become hypertext links (to glossary entries, appendices, other sections, source codes and figures) and a list that will become cross-references. The filters produce document transformation as in Fig. 10.7. In the segment that has been modified, underlined terms designate hypertext links. The page numbers are cross references inserted to facilitate the use of the documents in hard copy form.

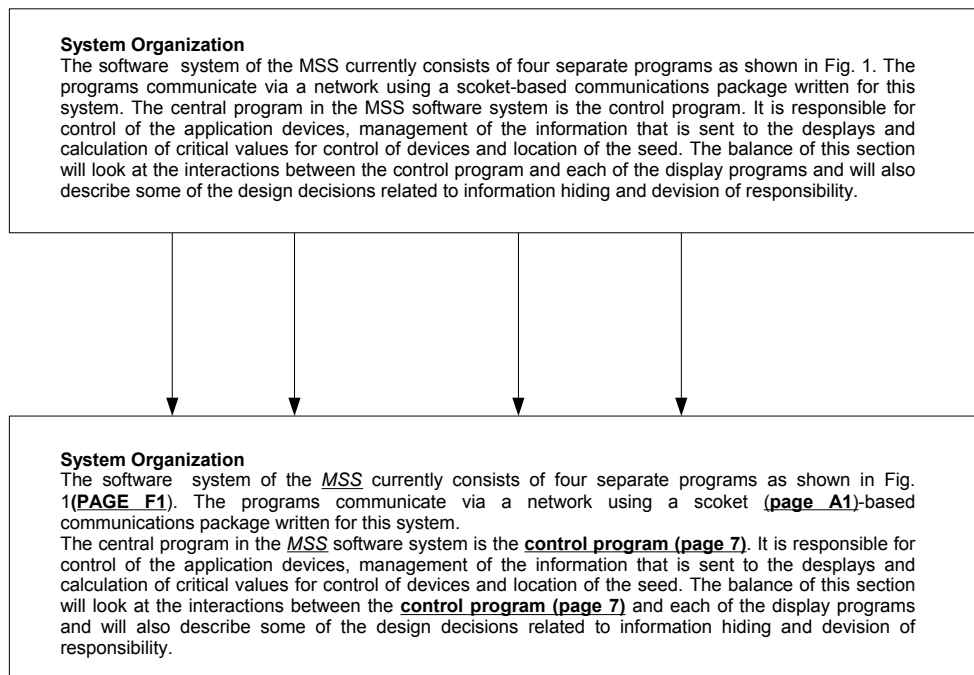


Figure 10.7: Document transformation using the SLEUTH filters

Conclusion

SLEUTH consists of an authoring and viewing environment and provides both a mechanism to link a collection of documents using typed hypertext links and the facility to perform keyword searches on the text of the documentation. These techniques can improve the usefulness of software documentation by addressing the needs of a wide variety of users and by providing these users with a number of ways to uncover the answers to their questions.

10.6 Variorum: A multimedia-Based Program Documentation System

Conventional software documentation systems are mostly based on textual descriptions that explain or annotate the program's source code. Because these systems rely primarily on texts, it is difficult for program authors to describe the overall algorithm structure and implementation techniques used, especially those that require visual presentation.

From a software maintainer's standpoint, the easiest way to understand a program is to sit side by side with the program's original author and go over the program line by line, and explain what the program does and why it was designed the way it is. In practice, this is rarely the case for the obvious reasons.

Variorum [5] is a software documentation system that is designed to duplicate this ideal scenario by providing a multimedia recording and editing tool for program authors to explain how the programs work, exactly in the way they would have done if they were asked to work with software maintainers. It uses multimedia technologies like audio, digital drawing pen as well as text to help program authors record various design decisions and program annotations. Specifically, program authors can verbally describe the program's design and implementation, and graphically illustrate the algorithms or program structures by drawing figures on digital tablets.

Authoring and Playback Interface

To annotate a program segment, users first define the *scope* of the annotation by highlighting the code segment. *Variorum* allows nested scoping, so one can further annotate a code segment within another code segment that has already been annotated. This is particularly useful to, for example, give an overview of what a procedure does and then to detail the implementation of the tricks used in each block of the procedure. When a code segment is annotated, the text of the program segment is shown in a different color to distinguish it from other un-annotated segments.

After the scope is defined, *Variorum* presents a separate window initialized to the program segment highlighted. Users can then verbally explain, with the help of digital pen sketches, the algorithmic and implementation details of the program segment. *Variorum* transparently records the speech inputs and the digital pen strokes, and stores them in a way that guarantees synchronized playback of these two streams. Users can still type in

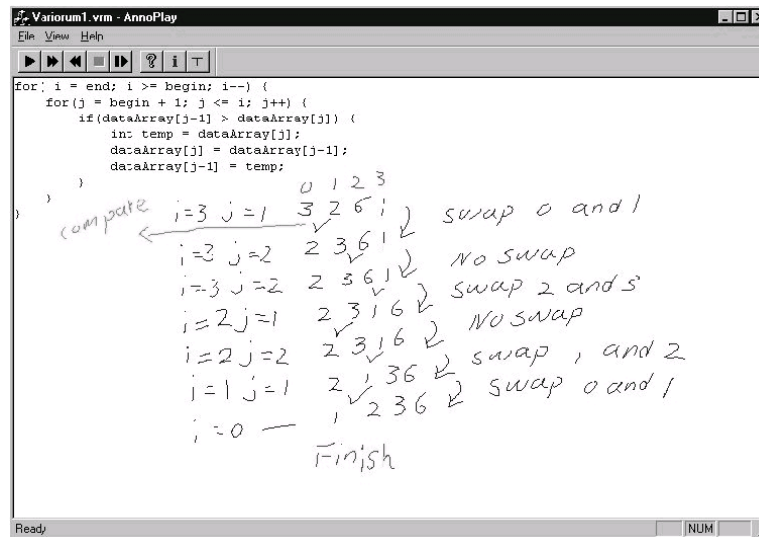


Figure 10.8: Snapshot of an annotation session using Variorum for the bubble sort program

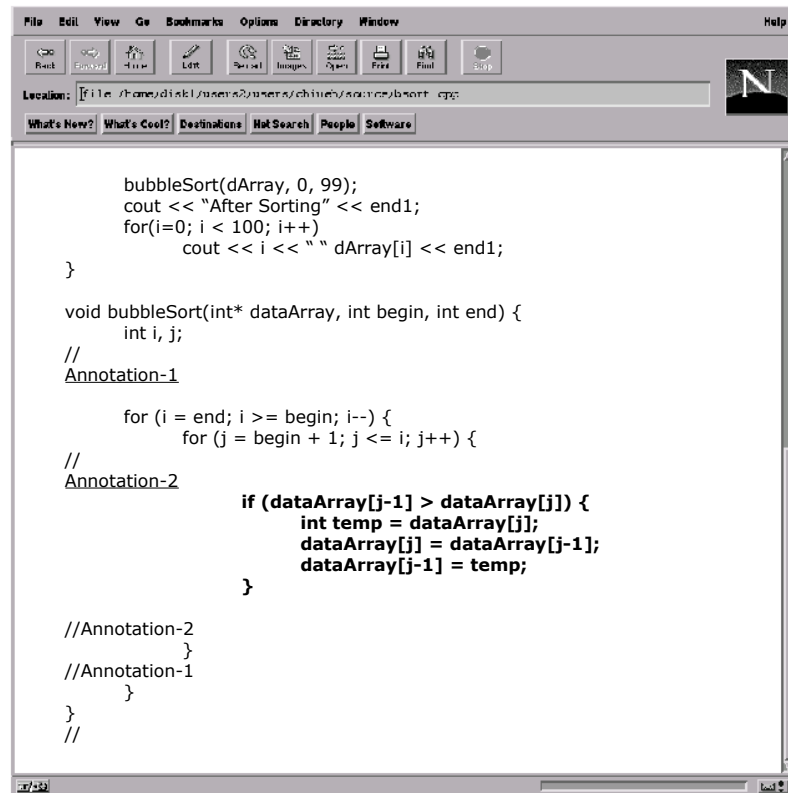
traditional textual annotations as well. Figure 10.8 shows an example screen snapshot of an annotation session for the "bubble sort" program.

When source programs are modified, the corresponding annotations should be modified accordingly to reflect the changes. Variorum does not attempt to solve completely the well-known problem of out-dated documentation with respect to the source code. Instead, Variorum supports only timestamp-based change notification to alert program authors of potential inconsistency, and an annotation editing capability to update existing annotations in place.

Program maintainers can then access the source code files and annotations using a web browser. An example snapshot is shown in Figure 10.9. Different colors are used to denote different nesting levels. By clicking on the underlined annotation links on the program text, users can access the corresponding annotations. Fast forward/rewind and pause/resume are provided to help navigating through the annotation sessions

Conclusion

From a cognitive science point of view, understanding a program is essentially a reverse engineering process - recovering the logical design decisions from the source code [5]. A multimedia approach to software documentation greatly improves the interactivity and flexibility of the process. It also enhances the quality of the resulting documentation as visual



The image shows a web browser window with a menu bar (File, Edit, View, Go, Bookmarks, Options, Directory, Window, Help) and a toolbar with icons for Back, Forward, Home, Stop, Reload, Print, and others. The address bar shows the file path: file:///home/disk1/users/2/users/chinib/source/bubble_sort.cpp. Below the address bar are navigation links: What's New?, What's Cool?, Destinations, Hot Search, People, and Software. The main content area displays the following C++ code with annotations:

```

bubbleSort(dArray, 0, 99);
cout << "After Sorting" << endl;
for(i=0; i < 100; i++)
    cout << i << " " dArray[i] << endl;
}

void bubbleSort(int* dataArray, int begin, int end) {
    int i, j;
    // Annotation-1
    for (i = end; i >= begin; i--) {
        for (j = begin + 1; j <= i; j++) {
            // Annotation-2
            if (dataArray[j-1] > dataArray[j]) {
                int temp = dataArray[j];
                dataArray[j] = dataArray[j-1];
                dataArray[j-1] = temp;
            }
            //Annotation-2
        }
        //Annotation-1
    }
}
//

```

Figure 10.9: An example snapshot of the web layout of the bubble sort annotated program segment

presentations are proven to be most effective for program understanding [19].

10.7 Literate Programming

”Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *humans* what we want the computer to do.” (Donald E. Knuth, 1984)

In essence, this is the purpose of literate programming. Such an environment reverses the notion of including documentation in the form of comments within the code, to one where the code is embedded within a program’s description. In doing so, literate programming facilitates the development and presentation of computer programs that more closely follow the way they are put together and implemented. This, in turn, leads to programs that are easier to debug and maintain [11].

When literate programming, one specifies the program description and the program code in a single source file in the order best suited to human understanding. The program code can be extracted and assembled into a form understandable for the compiler or interpreter by a process called ‘tangling’. Documentation is produced by a process of ‘weaving’ the description and code into a form ready to be typeset (most often by TEX or LATEX), see Fig 10.10 [13].

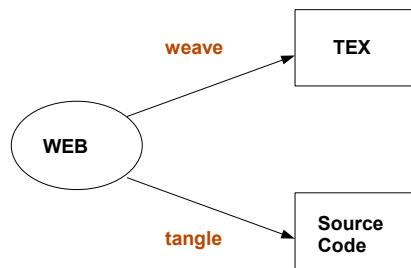


Figure 10.10: The dual usage of WEB system

Many different tools have been created for literate programming over the years and most of the more popular are based, either directly or conceptually, on the WEB system created by D. E. Knuth.

In terms of documentation and explanation, the ability to describe components as they

come into play in the design of the program –rather than in the order they must occur for the compiler or interpreter– is a vast improvement over traditional commented code. In addition to the benefits of improved code and easier maintenance, literate programs can also serve well as excellent teaching tools.

10.8 Design and Architecture Documentation

Documenting the design decisions that were undertaken during the various development phases of the system, as well as the architecture adapted are as important sides of software documentation as is code documentation.

To document the design of the system, one should start with an overview of the system in hand. Provide a general description of the software system including its functionality and matters related to the overall system and its design.

Any assumptions or dependencies regarding the software and its use should be documented. These may concern such issues as related software or hardware, operating systems, end-user characteristics and possible and/or probable changes in functionality.

Also, it should be taken into consideration to describe any design decisions and/or strategies that affect the overall organization of the system and its higher-level structures. These strategies should provide insight into the key abstractions and mechanisms used in the system architecture [3]. The reasoning employed for each decision and/or strategy (possibly referring to previously stated design goals and principles) should also be mentioned here. Such decisions might concern (but are not limited to) things like the use of a particular type of product (programming language, database, library, etc. ...), future plans for extending or enhancing the software, and hardware and/or software interface paradigms.

A high-level overview of how the functionality and responsibilities of the system were partitioned and then assigned to subsystems or components should also be documented [20]. The main purpose here is to gain a general understanding of how and why the system was decomposed, and how the individual parts work together to provide the desired functionality.

The top-most level should describe the major responsibilities that the software must undertake and the various roles that the system (or portions of the system) must play. How the system was broken down into its components/subsystems is important to document as well. In doing so, one gains a clearer image of the system structure and how that relates to its intended functionality. Another significant architectural aspect of the system that should be documented is how the higher-level components collaborate with each other in order to achieve the required results. And by including the rationale for choosing this particular decomposition of the system, it becomes easier to extend the software system to attend to new required functionalities.

Architecture documentation is both prescriptive and descriptive [2]. That is, it prescribes what should be true by placing constraints on decisions that are about to be made, and it describes what is true by recounting decisions that already have been made. However, the best architectural documentation for performance analysis may well differ from the best documentation for system integrators. Both of these will differ from the documentation

that a new hire receives. The documentation planning and review process must support all relevant needs.

10.9 How much documentation is enough?

Quite a lot, certainly more than most programmers, analysts, or program designers are willing to do. The first rule of managing software development is ruthless enforcement of documentation requirements [4] [17].

Why so much documentation? In part, this is because each designer must communicate with other designers, with his management and possibly with the customer. A verbal record is too intangible to provide an adequate basis for a management decision. During the early phase of software development the documentation is the specification and is the design. If the documentation is bad the design is bad. If the documentation does not yet exist there is as yet no design, only people thinking and talking about the design which is of some value, but not much.

In addition to achieving software systems that better meet the specifications and that are easier to maintain, good documentation has real monetary value [17]. During the testing phase, with good documentation the manager can concentrate personnel on the mistakes in the program. With good documentation the manager can use operation-oriented personnel to operate the program and to do a better job, cheaper. Without good documentation the software must be operated by those who built it. Generally these people are relatively disinterested in operations and do not do as effective a job as operations-oriented personnel. It should be pointed out that traditionally in an operational situation, if there is some hangup the software is always blamed first. In these situations, the software documentation must speak clearly to fix the blame. Finally when system improvements are in order, good documentation permits effective redesign and updating. If documentation does not exist or only poor documentation is available, generally the entire existing framework of operating software must be junked, even for relatively modest changes.

10.10 Summary

Documentation is the castor oil of programming. Managers think it is good for programmers, and programmers hate it! [18]. When considering a source code documentation tool, it is useful to examine different aspects like usability, target media, produced document structure, comment extraction capabilities, languages supported and inline formatting in order to select the best tool for the job.

When documenting a software system, one should consider more than just code documentation that explain the API's, algorithms and data structures used. The rationale behind the design of the system should also be documented, as well as the reasons for structuring the system in a particular way. Without such documents the maintenance and modifiability of the system become a nightmare, and sometimes even impossible.

10.11 Exam Questions

1. What characteristics should a documentation tool have to make it a good candidate for documenting a software system?
2. What are the major differences between DOC++, Variorum and SODOS?
3. What is design documentation? And how is it different from code documentation?

Bibliography

- [1] Dragos Acostachioaie. Doc++. open source - open systems -open science. In *Proceeding of the 5th International Conference on Development and Application Systems*, 2000.
- [2] F. Bachmann, L. Bass, P. Clements, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting software architectures: Organization of documentation package, 2001.
- [3] Felix Bachmann, Len Bass, Paul Clements, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. The SEI Series in Software Engineering. Addison Wesley Professional, 1st edition, 2002.
- [4] Lionel C. Briand. Software documentation:how much is enough? In *Proceedings of the Seventh European Conference On Software Maintenance And Reengineering*.
- [5] Tzi cker Chiueh, Wei Wu, and Lap-Chung Lam. Variorum: A multimedia-based program documentation system. In *IEEE International Conference on Multimedia and Expo (I)*, pages 155–158, 2000.
- [6] Van Heesch D. *Doxygen manual*, 2003.
- [7] A. Goldberg. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.
- [8] Ellis Horowitz and Ronald Williamson. Sodos a software documentation support environment: its use. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 8–14, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.
- [9] Ellis Horowitz and Ronald C. Williamson. Sodos: a software documentation support environment-its definition. *IEEE Trans. Softw. Eng.*, 12(8):849–859, 1986.
- [10] Sun Microsystems. JavaDoc. <http://java.sun.com/j2se/javadoc/>.

- [11] Andrew Johnson and Brad Johnson. Literate programming using noweb. *Linux J.*, 1997(42es):1, 1997.
- [12] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers,. *Online Magazine*, pages 56–60, September 1991.
- [13] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.
- [14] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM Press.
- [15] Donald M. Leslie. Using javadoc and xml to produce api reference documentation. In *SIGDOC '02: Proceedings of the 20th annual international conference on Computer documentation*, pages 104–109, New York, NY, USA, 2002. ACM Press.
- [16] Allison L. Powell, James C. French, and John C. Knight. A systematic approach to creating and maintaining software documentation. In *SAC '96: Proceedings of the 1996 ACM symposium on Applied Computing*, pages 201–208, New York, NY, USA, 1996. ACM Press.
- [17] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.
- [18] Andreas Ruping. *Agile documentation : a pattern guide to producing lightweight documents for software projects*. Hoboken, NJ : Wiley, 2003.
- [19] K.; Muller H. A. Storey, M.-A.D.; Wong. How do program understanding tools affect how programmers understand programs?, 1997. Proceedings of the Fourth Working Conference on Reverse Engineering.
- [20] Scott R. Tilley, Hausi A. Muller, and Mehmet A. Orgun. Documenting software systems with views. In *SIGDOC '92: Proceedings of the 10th annual international conference on Systems documentation*, pages 211–219, New York, NY, USA, 1992. ACM Press.
- [21] R. Wunderling and M. Zockler. Doc++: A documentation system for c/c++ and java.

Chapter 11

Ming Yu Zhao: Object-Oriented Literate Programming

Object-oriented literate programming is what the name implies: implementing object-oriented programs with Literate Programming approach. By combining these two approaches, we can obtain the advantages coming from both of them and avoid or reduce their shortcomings at the same time. In this paper we, first, talk about object-oriented design and Literate Programming where both their advantages and limitations are analyzed, respectively. Then, a way to combine them together is introduced. Finally, we present a sizeable example, a banking system, which is implemented under the concept of object-oriented literate programming.

11.1 Introduction

As a computer programming paradigm, object-oriented programming is a revolutionary approach of thinking about the process of decomposing problems and developing programming solutions, totally unlike anything that has come before. It views a program as a collection of loosely connected objects. Each object is responsible for specific tasks and it is by the interaction of objects that computation proceeds. Designing an object-oriented program is like organizing a community of individuals. Each member of the community is given certain responsibilities. The achievement of the goals for the community as a whole comes about through the work of each member, and the interactions of members with each other. By reducing the interdependency among software components, object-oriented programming permits the development of reusable software systems(see [3], [8]). Object-oriented programming has become the dominant programming paradigm in the past few years. Characterized by compatibility, reusability and continuity, object-oriented programming significantly improved the software development process. Because for the traditional design, it is impossible to combine functions if the data structures they access are not taken into consideration and difficult to build reusable components if they embody functions alone and ignore the data part. In addition, focusing on requirements definition and the first operational system re-

lease, the traditional design method hardly satisfies continuity in a long-term concern. In addition, it is easier to understand, because there is direct mapping from the real-world to the object-oriented program. However, the results obtained by the use of such methodology are not always entirely satisfactory. Most development programmers do not like to write documentation, but the documentation is, certainly, extremely important for those maintenance programmers to understand the software system they are responsible for. It is well known that understanding programs is one of the most time-consuming activities in software maintenance. Therefore, complete and up-to-date documentation plays a key role in minimizing software costs and further improvements are required to achieve high quality programs. Literate Programming introduced by Donald Knuth is an approach to improve program understanding by regarding programs as works of literature. One can obtain human readable documentation and computer source code from a single unified program description. It seems that Literate Programming can provide an answer to the problem of the software maintenance. Object-oriented design is the first stage of object-oriented programming, so in order to find a proper way to combine object-oriented programming with Literate Programming, we have to examine object-oriented design first(see [6]).

11.2 Object-Oriented Design

There are many object-oriented design techniques, such as Responsibility-Driven Design coined by Rebecca Wirfs-Brock, Use Case Analysis introduced by Jacobson, Object-Modeling Technique developed by Rumbaugh, as well as Design Patterns. They share a lot of commonalities. That is, they all strive for loose coupling and strong cohesion, hide as much information as possible, such as data, implementation, class, and even design, apply the “once and only once” rule, and use intention-revealing names. Additionally, they have same objectives, which are to give a practical approach to produce high quality object-oriented software and to provide the knowledge and experience necessary to avoid the most common risks associated with building production systems. The difference between them is only their perspective to the problem(see [4], [2]). I prefer Responsibility-Driven Design because it is among the simplest to explain and suits Literate Programming.

Responsibility-Driven Design is a method for deriving a software design in terms of collaborating objects, by asking what responsibilities must be fulfilled to meet the requirements, and assigning them to the appropriate objects. Responsibility implies a degree of independence or noninterference. That is, when Object A gives a request to Object B to do a specific action, it is not necessary to supervise how the request would be serviced. By responsibility, Responsibility-Driven Design realizes the concepts of information hiding and modularity. And it is followed by reusability, one of the major benefits of object-oriented programming. Responsibility-Driven Design is driven by an emphasis on behavior at all levels of development. It is reasonable, because the behavior of a system is usually understood long before any other aspects(see [3]).

How do we apply this technique in object-oriented programming? First of all, we have to refine the specification. Because initial specifications are almost always ambiguous and

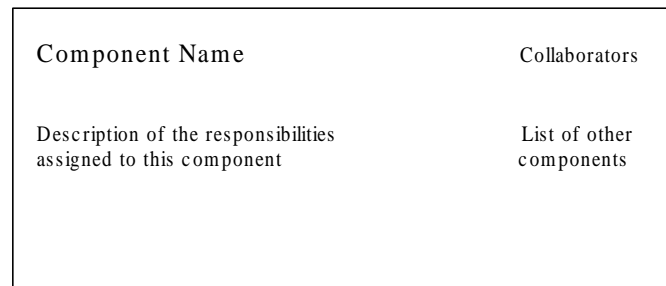


Figure 11.1: Component, Responsibility, Collaborator.

unclear on anything except the most general points. This step can bring us a better handle on the “look and feel” of the eventual product and make sure if it is in agreement with the original conception of the client.

Then we decompose the whole problem into activities by asking what/who questions. First, we identify what activity needs to be performed next. This is immediately followed by answering the question of who performs this action. In this manner, designing a software system is much like organizing a collection of people, such as a community. Any activity that is to be performed must be assigned as a responsibility to a certain component, which is simply an abstract entity that can perform tasks and may ultimately be turned into a function, a structure or class, or a collection of other components. Therefore, a component must have a small, well-defined set of responsibilities and should interact with other components to the minimal extent possible. During this process, CRC(Component, Responsibility, Collaborator) cards are often employed to represent components(See Figure 1.1). Written on the face of the card is the name of the software component, the responsibilities of the component, and the names of other components with which the component must interact. An advantage of CRC cards is that they are widely available, inexpensive, and flexible. This encourages experimentation, since alternative designs can be tried, explored, or abandoned with little investment. The physical separation of the cards encourages an intuitive understanding of the importance of the logical separation of the various components, helping to emphasize the cohesion and coupling.

So far, we should have decided that all the activities can be adequately handled by several components. Therefore, a communication diagram and an interaction diagram can be used to illustrate the static relationships between components and their dynamic interactions during the execution time, respectively. See Figure 1.2 and Figure 1.3,

Next, we need to describe the information held by these components. In Responsibility-Driven Design, most components consist of a combination of behavior and state, where the behavior is the set of actions the component can perform and the states represent all the information held within the component at a given point of time. Of course, it is not necessary that all components maintain state information. The term class is used to describe a set of objects with similar behavior. An individual representative of a class is known as an

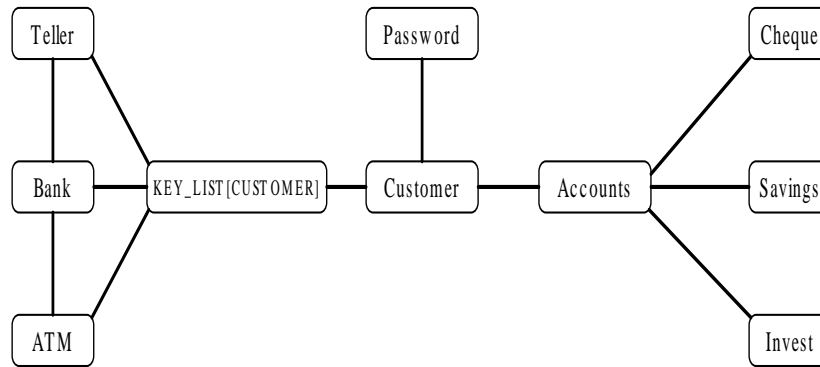


Figure 11.2: Communication between the components in the Banking System.

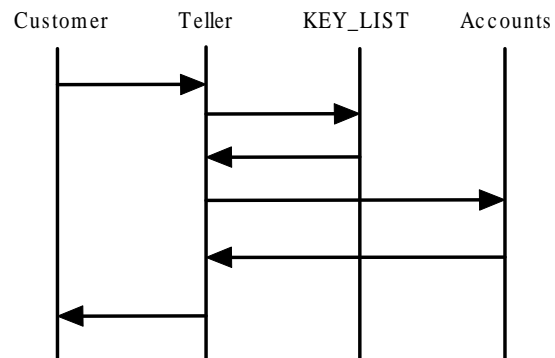


Figure 11.3: An example interaction diagram.

instance. All instances of a class will respond to the same instructions and perform in a similar manner. State, on the other hand, is a property of an individual.

Cohesion is the degree to which the responsibilities of a single component form a meaningful tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, because connections between software components inhibit ease of development, modification, or reuse.

After that, we need to refine the components. A component with only one behavior and no internal state may be made into a function. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto routine names. Along with the names, the types of any arguments to be passed to the function are identified. Next, the information maintained within the component itself should be described. All information must be accounted for.

Now, our focus should move to a software system implementation from the description of a component. The major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfill the assigned responsibilities. It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process.

Once the design of each software subsystem is finished, the next step is to implement each component's desired behavior. At this point, each responsibility or behavior should be characterized by a short description. The task at this step is to implement the desired activities in a computer language.

Once software subsystems have been individually designed and tested, they can be integrated into the final product. Starting from a simple base, elements are slowly added to the system and tested. The application is finally complete when all components have been added. During integration, it is very like that new errors can be discovered. If so, some of the components have to be modified, and then individual test is needed before an attempt to reintegrate these refurbished ones into the system.

Finally, we come to the stage of maintenance and evolution. It is unavoidable that errors can be discovered in the delivered product. These must be corrected, either in updates or corrections to existing release or in subsequent releases. Requirement or hardware may change. So this stage can not be ignored.

11.3 Literate Programming

We have already seen how Responsibility-Driven Design works, then let us talk about Literate Programming. When writing programs, development programmers should not concentrate on how to instruct the computer what to do, but instead, they should try to write programs in a way that other people can understand them more easily. This is the main idea of Literate Programming introduced by Donald Knuth in the early 80's (see [5], [1], [9]) and it seems to

put a dent in the software crisis mentioned above. In fact, it is because that the combination of two languages, formatting language and programming language, is much more expressive than either single language by itself. In Literate Programming, programmers analyze and design software at a high level, just like an author conceives a story for his or her reader. Hence all the implementations, algorithms, as well as variables could be clearly explained and justified, which is very important for both development programmers themselves and maintenance programmers, and further ensures the good quality of the software system.

The first Literate Programming system is WEB, which was published by Donald Knuth for his TeX typesetting system. The main idea behind WEB is that if an experienced programmer wants to provide the best documentation of his or her softwares, two kinds of techniques are necessary at the same time: TeX, a formatting language, and Pascal, a programming language. The structure of a software program may be thought of a “web” that is made up of many interconnected pieces. Programmer is responsible to explain each individual part of the web and how it relates to its neighbors only and computer is responsible for the laborious work of arranging the parts of the program in an order required by the compiler. Thus, we can obtain software system as well as its consistent document simultaneity.

Actually, in the area of Literate Programming, all source code goes into one file, from which “weave” will generate a section headings, a table of contents, a proper index for identifiers, an alphabetized listing of the code section names, and other things designated by the author using the web control code and “tangle” will produce the program code for computer according to the relation tag set by programmer. The author is meant to follow a narrative order in presenting the parts of the program. As Knuth says “but always it is an order that makes sense on expository grounds” and further, “there’s no need to be hung up on the question of top-down versus bottom-up—since a programmer can now view a large program as a web, to be explored in a psychologically correct order...” (see [5]). Therefore, in most cases, there will be a number of ways of ordering that will work. I believe that Literate Programming can be adapted to object-oriented programming.

11.4 Complement and Conflict

We have introduced both object-oriented programming and Literate Programming approach. It is not hard to see that they complement each other. Because, on the one hand, object-oriented programming can obtain some advantages from the use of the Literate Programming approach, such as readability and maintainability. Readability, because 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 constrains that have led to certain intricacies in the implementation. Presenting this discussion in book form allows programmers to break it up into discrete sections. The result will automatically be more readable as the author’s intentions will be laid out in much more detail. Maintainability, because 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 changed. On the other hand, Literate Programming can become more popular and be enhanced, because of the merits of object-oriented programming. Popularity, because object-oriented programming has become exceedingly popular in the past few years. Software producers rush to release object-oriented versions of their products. Countless books and special issues of academic and trade journals have appeared on the subject. If Literate Programming can provide the ability of object-oriented programming, then it will naturally attract more and more attention of users. Enhancement, because object-oriented programming can share reusability, one of its major benefits, with Literate Programming.

However, in order to combine them together, we have to resolve their incompatibilities. The main problem comes from the design structure. While an object-oriented analysis and design is supposed to go seamlessly into an object-oriented implementation, typically the initial design is presented as a huge diagram that contains no information for its rationale, in a literate analysis and design approach we generally build up the model gradually. For example, at the beginning of Responsibility-Driven Design, programmers decompose the whole problem into components according to the responsibilities, where components are composed of few empty behaviors and then refined step by step. In fact, factoring is one of the advantages of Literate Programming, hence one can break up any big block into its constituent parts. So we can take several succeeded chunks, which share the same name, as a component of Responsibility-Driven Design and refine the empty chunks later.

Another problem is that most classic Literate Programming approaches are characterized by language-independence. For instance, such as WEB depends on Pascal and MatlabWEB depends on Matlab. So we must find out a language-independent Literate Programming tool, noweb is what we want.

Noweb's simplicity comes from a simple model of files, which are marked up using a simple syntax. A noweb file is a sequence of chunks and a chunk may contain code or documentation. Chunks may appear in any order. Each code chunk has a name and begins with `<<chunk name>>=` on a line by itself. Each documentation chunk begins with a line that starts with an `@` symbol followed by a space or newline. Chunks are terminated implicitly by the beginning of another chunk or by the end of the file. If the first line of the file does not mark the beginning of a chunk, noweb assumes it is the first line of a documentation chunk. Actually, the noweb manual is only three pages; an additional page explains how to customize its LaTeX output. It is more important that noweb is language-independent. In other word, programmer, who wants to develop software with literate analysis and design approach, is freed from certain programming language.

As Figure 1.2 shows, noweb uses its `notangle` and `noweave` tools to retrieve code and documentation, respectively. When `notangle` is given a noweb file, it writes the program on standard output and when `noweave` is given a noweb file, it produces, on the standard output, TeX source for typeset documentation. A more detailed introduction of noweb can be found in [7] and [3]. For all reasons given above, we choose noweb as another one of main tools in our example.

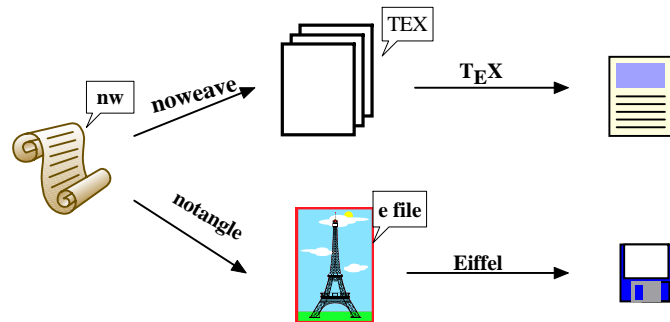


Figure 11.4: Dual usage of a noweb file.

11.5 An Example: A Banking System

We take a banking system as our example, which is specified as follow:

1. A banking system has many customers. Each customer may have a savings account, a cheque account, and an investment account, one account of each type.
2. The bank offers access to cheque and savings accounts through an interactive menu like that seen in an automatic teller machine (ATM); an investment account cannot be accessed through the ATM.
3. Savings account accrues daily interest, paid on the current balance and its interest rate is 4.5% a year. A cheque account gets no interest. An investment account is created with an initial balance of at least \$1000, and accrues daily interest for a period of 3, 6, or 12 months. A 3-month investment account has an annual interest rate of 5.5%, a 6-month account has a 6.0% rate, and a 12-month account 6.5%. When the account matures at the end of the period, the total amount is transferred into the customer's cheque account.
4. A positive amount may be deposited in an account or withdrawn from an account. A withdrawal from a savings account decrements the balance by that amount; there are no charges or penalties for a savings account. A successful withdrawal from a cheque account costs 50 cents. An unsuccessful withdrawal from a cheque account costs \$5. The balance of an account cannot be negative.
5. The bank system runs for an extended period. At the start of each day, a bank teller creates new customers, and new accounts for existing customers. Each customer has

a unique integer key; successive integers are used for each new customer. Customers and accounts are never deleted from the bank. The ATM then runs all day, handling multiple customers. To use the ATM, a customer enters his or her unique key (this simulates putting a card into the ATM) and his or her password, chooses an account, then chooses commands from the menu. Menu commands are read and executed until the customer finishes; the ATM then waits for the next customer. A special key of 666 exits the ATM system for the day. Interest is then added to all savings and investment accounts. Entry of the special key value of 999 into the ATM shuts down the whole system. The bank data is stored to file when the system shuts down, and is retrieved from file when the system starts up again.

6. A customer is allowed three attempts to login to the ATM by entering a valid password. If no correct password is entered after three attempts, then the ATM system rejects the login attempt and asks for a new customer identifier. If the password is correct, then the customer is shown a menu of account choices, and the system reads and executes the choices. Any number of transactions may be made; processing on the account continues until the customer chooses to exit that account. Multiple accounts may be chosen and used within a single ATM session.
7. The ATM menu choices (upper or lower case) are
 - D Deposit
 - W Withdraw up to the total amount in the account
 - B Show the balance
 - Q Quit the system
 - H Help: Show the menu choices

Literature of the Banking System

Refinement

1. Customers can not withdraw money from his or her investment account.
2. Customers can not deposit money to his or her investment account.
3. A Customer is allowed three attempts to login to the ATM by a valid password continuously.

Components and Behavior

To identify all the components of the Banking System, a good idea is imagining an execution of a working system and picking up all the responsibilities.

According to the specification, the Banking System runs for an extended period and at the start of each day, bank data is retrieved from a file when the system starts up, so the responsibilities for making a bank, running the whole system and retrieving bank data are assigned to a component called the **bank**:

198a $\langle bank\ 198a \rangle \equiv$ 204a \triangleright
 $\langle makeBank\ 208a \rangle$
 $\langle runBank\ 208b \rangle$
 $\langle retrieveBankData\ 209a \rangle$

The teller then creates new customers, so the responsibilities for making a teller, running the teller, promoting and creating new customers are assigned to a component called the **teller**:

198b $\langle teller\ 198b \rangle \equiv$ 199e \triangleright
 $\langle makeTeller\ 209c \rangle$
 $\langle runTeller\ 209d \rangle$
 $\langle newCustomers\ 210a \rangle$
 $\langle askForMoreCustomers\ 210b \rangle$

In order to add a new customer, the teller has to send a request to a component called the **customer**. In other word, the job of creating customer should be assigned to the component **customer**:

198c $\langle customer\ 198c \rangle \equiv$ 199b \triangleright
 $\langle makeCustomer\ 211a \rangle$

Every customer is a person, so to create a new customer, we need create a record for the person first. The responsibility for making a new personal record is assigned to a component called the **person**:

198d $\langle person\ 198d \rangle \equiv$ 198e \triangleright
 $\langle makePerson\ 211b \rangle$

Making a new personal record is nothing but getting the personal information, including name, gender and address. These responsibilities concerning new person records are assigned to the component **person**:

198e $\langle person\ 198d \rangle + \equiv$ \triangleleft 198d 198f \triangleright
 $\langle getName\ 211c \rangle$
 $\langle getGender\ 212a \rangle$
 $\langle getAddress\ 213a \rangle$

It is because there are only two possibilities (male or female) for the gender of a person that the responsibilities for promoting and verifying the input of the gender are assigned to the component **person**:

198f $\langle person\ 198d \rangle + \equiv$ \triangleleft 198e 199a \triangleright
 $\langle readGender\ 212b \rangle$
 $\langle isGoodGender\ 212c \rangle$

Showing the personal information of a customer is necessary, so the responsibility for that is assigned to the component **person**:

199a $\langle person\ 198d \rangle + \equiv$ ◁198f
 $\langle showPersonalInfo\ 213b \rangle$

Each customer has a unique ID; successive integer are used for each new customer, so the responsibility for setting ID for a new customer is assigned to the component **customer**:

199b $\langle customer\ 198c \rangle + \equiv$ ◁198c 199c▷
 $\langle setCustomerID\ 213c \rangle$

Similarly, the responsibility to show the customer information(including the personal information, account information and customer ID) should be assigned to the component **customer**:

199c $\langle customer\ 198c \rangle + \equiv$ ◁199b 199g▷
 $\langle showCustomer\ 214b \rangle$
 $\langle showCustomerID\ 214a \rangle$

Every customer has his or her own password, so the responsibility for making a password for a customer is assigned to a component called **password**:

199d $\langle password\ 199d \rangle \equiv$ 202b▷
 $\langle makePassword\ 214c \rangle$

After that, the teller creates new accounts for exiting customers, so the responsibilities for promoting and creating new accounts are assigned to the component **teller**, too:

199e $\langle teller\ 198b \rangle + \equiv$ ◁198b
 $\langle newAccounts\ 215a \rangle$
 $\langle askForMoreAccounts\ 215b \rangle$
 $\langle noMore\ 215c \rangle$

But the responsibility for finding a certain customer should be assigned to a component called the **customerManager**:

199f $\langle customerManager\ 199f \rangle \equiv$
 $\langle findCustomer\ 216a \rangle$

And the responsibility for matching the customer ID should be assigned to the component **customer**:

199g $\langle customer\ 198c \rangle + \equiv$ ◁199c 199h▷
 $\langle matchID\ 216b \rangle$

In order to open a new account for a certain customer, the teller has to send a request to the component **customer**:

199h $\langle customer\ 198c \rangle + \equiv$ ◁199g 202a▷
 $\langle addAccount\ 216c \rangle$

Each customer may have a savings account, a cheque account, and an investment account, so it is reasonable that the responsibilities for making accounts and opening a certain account for a certain customer are assigned to a component called **accounts**:

200a $\langle \textit{accounts} \ 200a \rangle \equiv$ 200b \triangleright
 $\langle \textit{makeAccounts} \ 217a \rangle$
 $\langle \textit{openAccounts} \ 217b \rangle$

The responsibility for showing all the accounts of a certain customer is assigned to the component **accounts**:

200b $\langle \textit{accounts} \ 200a \rangle + \equiv$ $\triangleleft 200a \ 202c \triangleright$
 $\langle \textit{showAccounts} \ 217c \rangle$

One account of each type, so the responsibilities for making, setting, matching and showing account type to a component call **account**:

200c $\langle \textit{account} \ 200c \rangle \equiv$ 200d \triangleright
 $\langle \textit{makeAccount} \ 218a \rangle$
 $\langle \textit{setAccountType} \ 218b \rangle$
 $\langle \textit{showAccountType} \ 218c \rangle$
 $\langle \textit{matchAccountType} \ 218d \rangle$

Every account cannot has a negative balance, so the responsibilities for getting and showing account balance to the component **account**:

200d $\langle \textit{account} \ 200c \rangle + \equiv$ $\triangleleft 200c \ 200e \triangleright$
 $\langle \textit{getAccountBalance} \ 218e \rangle$
 $\langle \textit{showAccountBalance} \ 219a \rangle$

Showing the account information of a customer is necessary, so the responsibility for that is assigned to the component **account**:

200e $\langle \textit{account} \ 200c \rangle + \equiv$ $\triangleleft 200d \triangleright$
 $\langle \textit{showAccountInfo} \ 219b \rangle$

The responsibility for making a cheque account, setting cheque account type is assigned to a component called the **cheque**:

200f $\langle \textit{cheque} \ 200f \rangle \equiv$ 203d \triangleright
 $\langle \textit{makeCheque} \ 219c \rangle$
 $\langle \textit{setChequeType} \ 219d \rangle$

The responsibilities for making a savings account, setting savings account type, and set the interest rate are assigned to a component called the **savings**:

200g $\langle \textit{savings} \ 200g \rangle \equiv$ 203c \triangleright
 $\langle \textit{setSavingsType} \ 219e \rangle$
 $\langle \textit{setSavingsRate} \ 220a \rangle$

The system should have the ability to show the investment account information to its owner, so the responsibilities for showing period, showing the elapsed time, and showing the whole investment account are assigned to the component **investment**:

```
201a  <investment 201a>≡ 201b>
      <showInvestment 220b>
      <showPeriod 220c>
      <showElapsed 220d>
```

To create an investment account, the customer need to provide an initial balance and to choose a period, so the responsibilities for making an investment account, setting investment account type, getting a initial balance and setting a period are assigned to a component called the **investment**:

```
201b  <investment 201a>+≡ <201a 201c>
      <makeInvestment 220e>
      <setInvestType 221a>
      <getInitBalance 221b>
      <getPeriod 221c>
      <setInvestRate 221d>
```

The system should have the ability to show the investment account information to its owner, so the responsibilities for showing period, showing the elapsed time, and showing the whole investment account are assigned to the component **investment**:

```
201c  <investment 201a>+≡ <201b 204d>
      <showInvestment 220b>
      <showPeriod 220c>
      <showElapsed 220d>
```

Both savings account and investment account have to deal with the matters concerning interest, it is reasonable to separate the concept of interest from those two accounts, so the responsibilities for making an interest account, setting interest rate are assigned to a component called the **interest**:

```
201d  <interest 201d>≡ 204e>
      <makeInterest 222d>
      <setInterestRate 222e>
```

Next, the ATM runs all day, serving multiple customers, so the responsibilities for making a ATM, running and serving customers are assigned to a component called the **ATM**:

```
201e  <ATM 201e>≡ 203g>
      <makeATM 223a>
      <runATM 223b>
      <serveCustomer 223c>
```

To use the ATM, a customer need to enter his or her unique ID as well as password, so the responsibility for logging in is assigned to the component **customer**:

202a $\langle customer\ 198c \rangle + \equiv$ $\langle 199h \rangle$
 $\langle login\ 224a \rangle$

In order to log in, the customer has to send a request to the component **password**, so the responsibilities for logging in and validating the password are assigned to the component **password**:

202b $\langle password\ 199d \rangle + \equiv$ $\langle 199d \rangle$
 $\langle loginByPassword\ 224b \rangle$
 $\langle isPasswordValid\ 224c \rangle$

After a customer logs in successfully, he or she can use all accounts he or she has, so the responsibilities for using accounts and using menu are assigned to the component **accounts**:

202c $\langle accounts\ 200a \rangle + \equiv$ $\langle 200b\ 202d \rangle$
 $\langle useAccounts\ 225a \rangle$
 $\langle useMenu\ 225b \rangle$

In order to use his or her accounts properly, the customer should input a valid account type, so the responsibilities for getting a valid type, reading a key, verifying the key, concerning the key to the account type and the existence of a certain account are assigned to the component **accounts**:

202d $\langle accounts\ 200a \rangle + \equiv$ $\langle 202c\ 202e \rangle$
 $\langle findType\ 226a \rangle$
 $\langle getValidCommand\ 226c \rangle$
 $\langle getValidType\ 226d \rangle$
 $\langle readAKey\ 227a \rangle$
 $\langle isExist\ 227b \rangle$

During the process of interacting with customer, many promoting behaviors are needed, such as and promoting to ask for more

202e $\langle accounts\ 200a \rangle + \equiv$ $\langle 202d\ 204b \rangle$
 $\langle askForKey\ 227c \rangle$
 $\langle askForMore\ 227d \rangle$
 $\langle waitForCommand\ 227e \rangle$
 $\langle isValidCommand\ 228a \rangle$

And after the customer chooses either savings account or cheque account, he or she can choose the commands from the menu. The menu commands are read and executed until the customer quits the service. So the responsibilities for making and showing the menu, getting and doing the choices are assigned to a component called the **menu**:

```
203a  <menu 203a>≡ 203b>
      <menuMenu 228b>
      <showMenu 228c>
      <getChoice 228d>
      <doChoice 229a>
```

According to the specification, the menu have five choices, so the responsibilities for doing deposit, and doing withdraw are assigned to the component **menu**:

```
203b  <menu 203a>+≡ <203a 203f>
      <doWithdraw 229c>
      <doDeposit 229b>
      <doShowBalance 225c>
```

Different type accounts have different rule to deposit and withdraw, so the responsibilities for depositing money to accounts and withdrawing money from accounts are assigned to the component **savings** and **cheque**, respectively:

```
203c  <savings 200g>+≡ <200g
      <deposit 230a>
      <withdrawSavings 225d>
```

```
203d  <cheque 200f>+≡ <200f 203e>
      <withdrawCheque 230b>
      <deposit 230a>
```

There is a penalty for a withdrawing money from cheque account, so the responsibility for penalising is assigned to the component **cheque**:

```
203e  <cheque 200f>+≡ <203d
      <penalise 230c>
```

In the process of getting menu choice, it is very likely that reading in a wrong choice, so the responsibility for reading choices and verifying choices are assigned to the component **menu**:

```
203f  <menu 203a>+≡ <203b
      <readChoice 230d>
      <isChoiceValid 226b>
```

Entry of the special key of 666 or 999 into the ATM exits the ATM or shuts down the whole system, respectively, so the responsibilities for exiting the ATM system and shutting down the whole system are assigned to the component **ATM**, too:

```
203g  <ATM 201e>+≡ <201e
      <finishATM 231c>
      <finishBank 231b>
```

Interest is then added to all savings and investment accounts, so the the responsibility for dealing with the ending work is assigned to the component **bank**:

204a $\langle bank\ 198a \rangle + \equiv$ $\langle 198a\ 204f \rangle$
 $\langle endDay\ 231d \rangle$

In order to modify all concerning account information, component **bank** needs to send a request to component **accounts**, so all the detail actions should be executed within component **accounts**:

204b $\langle accounts\ 200a \rangle + \equiv$ $\langle 202e\ 204c \rangle$
 $\langle endDayModify\ 232a \rangle$

At this time, when the investment account matures at the end of the period, the total amount is transfered into the customer's cheque account, so the responsibility for transferring money between accounts is assigned to the component **accounts**:

204c $\langle accounts\ 200a \rangle + \equiv$ $\langle 204b \rangle$
 $\langle transfer2Cheque\ 232b \rangle$

Otherwise, the system needs to increase the day counter of all the immature investment accounts, so this responsibilities for verifying the period and adding interest are assigned to the component **investment**:

204d $\langle investment\ 201a \rangle + \equiv$ $\langle 201c \rangle$
 $\langle isMature\ 232c \rangle$

And the system needs to add interest to both savings accounts and investment accounts, so the responsibilities for calculating and adding interest are assigned to the component **interest**:

204e $\langle interest\ 201d \rangle + \equiv$ $\langle 201d \rangle$
 $\langle calculateInterest\ 232d \rangle$
 $\langle addInterest\ 232e \rangle$

At last, when the system shuts down, the bank data is stored to a file, so the responsibility for storing bank data is assigned to the component **bank**:

204f $\langle bank\ 198a \rangle + \equiv$ $\langle 204a \rangle$
 $\langle storeBankData\ 233 \rangle$

State Information

So far, all the activities can be adequately handled by these fourteen components. What we need to do is to describe the information held by these components.

For simplicity, we assume there are only one teller and one ATM in the Banking System, so these two attributes are assigned to the component **bank**:

204g $\langle bank\ state\ 204g \rangle \equiv$ $\langle 205a \rangle$
 $\langle aTeller\ 206d \rangle$
 $\langle aATM\ 206e \rangle$

The Banking System may have many customers, so the attribute `patrons` is assigned to the component `bank`:

205a $\langle bank\ state\ 204g \rangle + \equiv$ $\langle 204g\ 205b \rangle$
 $\langle aPatrons\ 206f \rangle$

In order to store the bank data, so the Banking System should remember the file name and have a repository:

205b $\langle bank\ state\ 204g \rangle + \equiv$ $\langle 205a \rangle$
 $\langle aFile\ 206g \rangle$
 $\langle aRepository\ 206h \rangle$

The component `teller` and the component `ATM` should have the same customer information as the component `bank`:

205c $\langle teller\ state\ 205c \rangle \equiv$
 $\langle aPatrons\ 206f \rangle$

205d $\langle ATM\ state\ 205d \rangle \equiv$
 $\langle aPatrons\ 206f \rangle$

Every person has his or her own name, gender and address, so the personal information should record these information:

205e $\langle person\ state\ 205e \rangle \equiv$
 $\langle aName\ 206i \rangle$
 $\langle aGender\ 206j \rangle$
 $\langle aAddress\ 206k \rangle$

Every customer has his or her own password, ID and accounts, so the component `customer` should have these three attributes:

205f $\langle customer\ state\ 205f \rangle \equiv$
 $\langle aPassword\ 206l \rangle$
 $\langle aID\ 207a \rangle$
 $\langle aAccount\ 207b \rangle$

Every customer may have a savings account, a cheque account and a investment account, so the responsibility to remember the account info of a certain customer is assigned to component `accounts`:

205g $\langle accounts\ state\ 205g \rangle \equiv$
 $\langle aSavings\ 207c \rangle$
 $\langle aCheque\ 207d \rangle$
 $\langle aInvestment\ 207e \rangle$

An account has a balance and a type, so these two attributes are assigned to component `account`:

205h $\langle account\ state\ 205h \rangle \equiv$
 $\langle aBalance\ 207f \rangle$
 $\langle aType\ 207g \rangle$

Component `password` should have attribute to remember the password and the max of attempt times:

206a $\langle password\ state\ 206a \rangle \equiv$
 $\langle aPassword\ 206l \rangle$
 $\langle aMaxAttempts\ 207i \rangle$

Component `investment` should indicate the minimal initial balance and remember the period and elapsed days:

206b $\langle investment\ state\ 206b \rangle \equiv$
 $\langle aMinimalBalance\ 207j \rangle$
 $\langle aPeriod\ 207k \rangle$
 $\langle aDayCounter\ 207l \rangle$

Every interest account must indicate the rate of interest:

206c $\langle interest\ state\ 206c \rangle \equiv$
 $\langle aInterestRate\ 207m \rangle$

Designing the Representation

206d $\langle aTeller\ 206d \rangle \equiv$ (204g)
`teller: TELLER`

206e $\langle aATM\ 206e \rangle \equiv$ (204g)
`atm: ATM`

We choose structure list to store all the data of customers:

206f $\langle aPatrons\ 206f \rangle \equiv$ (205)
`patrons: KEY_LIST[CUSTOMER]`

206g $\langle aFile\ 206g \rangle \equiv$ (205b)
`repository_name: STRING is "patrons.dat"`

206h $\langle aRepository\ 206h \rangle \equiv$ (205b)
`xml_repository: XML_REPOSITORY[KEY_LIST[CUSTOMER]]`

206i $\langle aName\ 206i \rangle \equiv$ (205e)
`name: STRING`

206j $\langle aGender\ 206j \rangle \equiv$ (205e)
`gender: CHARACTER`

206k $\langle aAddress\ 206k \rangle \equiv$ (205e)
`address: STRING`

206l $\langle aPassword\ 206l \rangle \equiv$ (205f 206a) 207h▷
`password: PASSWORD`

207a	$\langle aID\ 207a \rangle \equiv$ id: INTEGER	(205f)
207b	$\langle aAccount\ 207b \rangle \equiv$ account: ACCOUNTS	(205f)
207c	$\langle aSavings\ 207c \rangle \equiv$ savings: SAVINGS	(205g)
207d	$\langle aCheque\ 207d \rangle \equiv$ cheque: CHEQUE	(205g)
207e	$\langle aInvestment\ 207e \rangle \equiv$ invest: INVEST	(205g)
207f	$\langle aBalance\ 207f \rangle \equiv$ balance: REAL	(205h)
207g	$\langle aType\ 207g \rangle \equiv$ id: CHARACTER	(205h)
207h	$\langle aPassword\ 206l \rangle + \equiv$ password: STRING	(205f 206a) \triangleleft 206l
207i	$\langle aMaxAttempts\ 207i \rangle \equiv$ max_tries: INTEGER is 3	(206a)
207j	$\langle aMinimalBalance\ 207j \rangle \equiv$ minimum: REAL is 1000.0	(206b)
207k	$\langle aPeriod\ 207k \rangle \equiv$ period: INTEGER	(206b)
207l	$\langle aDayCounter\ 207l \rangle \equiv$ days: INTEGER	(206b)
207m	$\langle aInterestRate\ 207m \rangle \equiv$ rate: REAL	(206c)

Implementation of Component

The daily cycle of the bank has three parts, to make or retrieve the list of customers, to make and run the ATM and TELLER, and to store the list of customers at last. In order to fulfil this job, the `make` routine calls every other routine in the class:

```
208a  <makeBank 208a>≡ (198a)
      make is
        do
          retrieve
          create teller.make(patrons)
          create atm.make(patrons)
          run
          store
          io.put_string("%N%NExit banking system%N")
        end --make
```

Each day, the main task of bank is to run the teller then the ATM, at last modifies the account information for every customer. The precondition is that both `atm` and `teller` could not be `Void`.

```
208b  <runBank 208b>≡ (198a)
      run is
        require
          atm /= Void
          teller /= Void
        do
          from
          until atm.system_finished
          loop
            teller.run
            atm.run
            end_day
          end
        end -- run
```


The `retrieve` routine makes or retrieves the list of customers, so after this routine is called, the list of customers can not be void any more.

```
209a <retrieveBankData 209a>≡ (198a)
  retrieve is
    do
      create xml_repository.from_file(repository_name)
      patrons := xml_repository.at("patrons")
      <create list 209b>
    ensure
      patrons /= Void
    end -- retrieve
```

If there is no data in the file, we create a empty list of customes.

```
209b <create list 209b>≡ (209a)
  if patrons = Void then
    create patrons.make
  end
```

Component `teller` is one of suppliers for component `bank`, so `teller` has to provide routines to its client. First, `teller` should accept the list of customers, when it is made. But first of all, we must make sure that the list passed by is not `Void`. If the parameter is not void, `make` can assign this list of customers to the attribute `patrons` of itself. So we know that after this operation, attribute `patrons` of `teller` should not be void, so

```
209c <makeTeller 209c>≡ (198b)
  make(customers: KEY_LIST[CUSTOMER]) is
    require
      customers /= Void
    do
      patrons := customers
    ensure
      patrons /= Void
    end -- make
```

Another supplier routine is `run`, which is called by component `bank` to add new customers and new accounts into the banking system:

```
209d <runTeller 209d>≡ (198b)
  run is
    do
      io.put_string("%NAdd new customers and new accounts")
      new_customers
      new_accounts
    end --run
```

The main task of the teller is to create new customers and new accounts. Similarly, we encapsulate the details of these operations into component `customer` and component `account`. In order to add a new customer, we first need to make sure that the list is not void. And then the teller could apply a new id, and after that add this new customer at the end of the list. As soon as we finish this operation, the number of customer can not be decreased. Then, the `new_customers` becomes:

210a $\langle newCustomers\ 210a \rangle \equiv$ (198b)

```

new_customers is
  require
    patrons /= Void
  local patron: CUSTOMER
  do
    from ask_for_more_customers
    until no_more
    loop
      create patron.make
      patron.set_id(patrons.count + 1)
      patron.add_account
      patrons.add_last(patron)
      ask_for_more_customers
    end
  ensure
    patrons.count >= old patrons.count
  end -- new_customers

```

where `ask_for_more_customers` routine is very simple, since it just prompts the user for more customers and then read in a char, 'Y' or 'y' denotes yes and 'N' or 'n' denotes no:

210b $\langle askForMoreCustomers\ 210b \rangle \equiv$ (198b)

```

ask_for_more_customers is
  do
    io.put_string("%NAny customers to add(Y/N)?")
    io.read_character
    io.flush
  end -- ask_for_more_customers

```

We know that component `teller` takes the charge of creating customers, so component `customer` has to provide `make` routine for `teller` to do so with the specified ID:

```
211a <makeCustomer 211a>≡ (198c)
  feature{TELLER}
  make is
  do
    make_person
    create password.make
    create account.make
  end -- make
```

The `make` routine, which sets the personal details, is one of supplier routines of the component `person`,

```
211b <makePerson 211b>≡ (198d)
  make is
  do
    io.put_string("%NEnter the personal details%N")
    get_name
    get_address
    get_gender
  end -- make
```

Whenever the behaviors, such as `getName`, `getGender` and `getAddress` are called, the corresponding attributes should not be `Void` any more. So `getName` should be:

```
211c <getName 211c>≡ (198e)
  get_name is
  do
    io.put_string("%TName: ")
    io.read_line
    create name.make(io.last_string.count)
    name.copy(io.last_string)
  ensure
    name /= Void
  end -- get_name
```

After reading in the name from the user, the system need also a valid gender:

```
212a  <getGender 212a>≡ (198e)
      get_gender is
        do
          from read_gender
          until good_gender
          loop
            io.put_string("Valid codes are M, m, F, or f. Try again%N")
            read_gender
          end
          gender := io.last_character
        ensure
          gender.to_string /= Void
        end -- get_gender
```

where read_gender routine prompts the user for the gender:

```
212b  <readGender 212b>≡ (198f)
      read_gender is
        do
          io.put_string("%TGender(M/F): ")
          io.read_character
          io.flush
        end -- read_gender
```

and good_gender routine determines if the entered is a valid gender code:

```
212c  <isGoodGender 212c>≡ (198f)
      good_gender: BOOLEAN is
        do
          inspect io.last_character.to_upper
          when 'M', 'F' then
            Result := True
          else
            Result := False
          end
        end -- good_gender
```

After obtaining a valid gender, the system need also the address of the user:

```
213a  <getAddress 213a>≡ (198e)
      get_address is
        do
          io.put_string("%TAddress: ")
          io.read_line
          create address.make(io.last_string.count)
          address.copy(io.last_string)
        ensure
          address /= Void
        end -- get_address
```

Before `showPersonalInfo` is called, we have to make sure that all of those three attribute could not be void:

```
213b  <showPersonalInfo 213b>≡ (199a)
      show is
        require
          name /= Void
          gender.to_string /= Void
          address /= Void
        do
          io.put_string("%N")
          inspect gender
          when 'M' then io.put_string("%NMr.")
          when 'F' then io.put_string("%NMs.")
          end
          io.put_string(name)
          io.put_string(" lives at ")
          io.put_string(address)
        end
```

`setCustomerID` routine just set a specified id for the new customer:

```
213c  <setCustomerID 213c>≡ (199b)
      set_id(key: INTEGER) is
        require
          key >= 0
        do
          id := key
        ensure
          id = key
        end -- set_id
```

```

214a  <showCustomerID 214a>≡ (199c)
      feature{NONE}
      show_id is
      do
          io.put_string("%NThe customer id is ")
          io.put_integer(id)
      end --show_id

```

Another request may come from component `teller` is to show the information of the customer, so we need:

```

214b  <showCustomer 214b>≡ (199c)
      feature{TELLER}
      show is
      require
          account /= Void
      do
          show_person
          show_id
          account.show
      end -- show

```

```

214c  <makePassword 214c>≡ (199d)
      make is
      do
          io.put_string("%TPassword: ")
          io.read_word
          io.flush
          password := io.last_string
      ensure
          password /= Void
      end -- make

```

The `new_accounts` routine takes the charge of adding any new accounts for existing customers, and the same as `new_customers`, we need make sure that `patrons` is not `Void`. And `new_accounts` routine should be:

```
215a  <newAccounts 215a>≡ (199e)
      new_accounts is
        require
          patrons /= Void
        do
          from ask_for_more_accounts
          until no_more
          loop
            io.put_string("Input your ID")
            io.flush
            io.read_int
            if patrons.find(io.last_integer) then
              patrons.item(io.last_integer).add_account
            else
              io.put_string("%NThat is not a valid userID")
            end
            ask_for_more_accounts
          end
        end --new_accounts
```

The `ask_for_more_accounts` routine prompts the user to create more accounts:

```
215b  <askForMoreAccounts 215b>≡ (199e)
      ask_for_more_accounts is
        do
          io.put_string("%NAny accounts to add(Y/N)?")
          io.read_character
          io.flush
        end -- ask_for_more_accounts
```

`noMore` routine is simpler, since it just check it the user type in 'N' or 'n' and then return a `BOOLEAN` accordingly.

```
215c  <noMore 215c>≡ (199e)
      feature{NONE}
      no_more: BOOLEAN is
        do
          Result := io.last_character.to_upper = 'N'
        end -- no_more
```

216a $\langle findCustomer\ 216a \rangle \equiv$ (199f)

```
feature{ATM, TELLER}
find(key: INTEGER): BOOLEAN is
  require
    key >= 0
  local
    i: INTEGER
  do
    from i := lower
    until i > upper or else item(i).match(key)
    loop i := i + 1
    end
    Result := i <= upper
  end -- find
```

216b $\langle matchID\ 216b \rangle \equiv$ (199g)

```
match(key: INTEGER): BOOLEAN is
  do
    Result := key = id
  end -- match
```

Component teller also open new accounts for the customer:

216c $\langle addAccount\ 216c \rangle \equiv$ (199h)

```
feature{TELLER}
add_account is
  require
    account /= Void
  do
    account.make
  end
```



```

217a  <makeAccounts 217a>≡ (200a)
      make is
        local key: CHARACTER
        do
          from
          until io.last_character.to_upper = 'N'
          loop
            get_key
            key := io.last_character.to_upper
            if exists(find(key)) then
              io.put_string("%N%TCustomer has that type. Try again")
            else
              open(key)
            end
            ask_for_more
          end
        end -- make

```

If the customer does not have the specified kind of account, teller create one for him. But for an investment account, ensure there is a cheque account:

```

217b  <openAccounts 217b>≡ (200a)
      open(key: CHARACTER) is
        require
          key.to_upper = 'S' or key.to_upper = 'C' or key.to_upper = 'I'
        do
          if (key.to_upper = 'S') then create savings.make
          elseif (key.to_upper = 'C') then create cheque.make
          elseif (key.to_upper = 'I') then
            create invest.make
            if not exists(cheque) then create cheque.make_zero end
          end
        end
      end -- open

```

When customers want to see their accounts, they can use `show` routine:

```

217c  <showAccounts 217c>≡ (200b)
      show is
        do
          io.put_string("%TThe accounts are:")
          if exists(savings) then savings.show end
          if exists(cheque) then cheque.show end
          if exists(invest) then invest.show end
        end -- show

```

```
218a  <makeAccount 218a>≡ (200c)
      make is
      do
        set_id
        get_balance
      end -- make
```

```
218b  <setAccountType 218b>≡ (200c)
      set_id is
      deferred end --set_id
```

```
218c  <showAccountType 218c>≡ (200c)
      show_id is
      require
        id.to_string /= Void
      do
        io.put_string("%N%TAccount type is ")
        io.put_character(id)
      end --show_id
```

The match routine determines if the specified key match the account id.

```
218d  <matchAccountType 218d>≡ (200c)
      match(key: CHARACTER): BOOLEAN is
      require
        key.to_string /= Void
      do
        Result := key = id
      ensure
        Result = (key = id)
      end --match
```

```
218e  <getAccountBalance 218e>≡ (200d)
      get_balance is
      do
        io.put_string("%TEnter initial account balance: ")
        io.read_real
        balance := io.last_real
      ensure
        balance = io.last_real
      end -- get_balance
```

```

219a  <showAccountBalance 219a>≡ (200d)
      show_balance is
      do
        io.put_string("%N%TThe balance is $")
        io.put_real(balance)
      end --show_balance

```

```

219b  <showAccountInfo 219b>≡ (200e)
      show is
      do
        show_id
        show_balance
      end -- show

```

Because when the investment account matures at the end of the period, the total amount is transferred into the customer's cheque account, we have to define a empty cheque account for those customer who has investment account.

```

219c  <makeCheque 219c>≡ (200f)
      make_zero is
      do
        set_id
      end --make_zero

```

```

219d  <setChequeType 219d>≡ (200f)
      set_id is
      do
        id := 'C'
      ensure
        id = 'C'
      end -- set_id

```

```

219e  <setSavingsType 219e>≡ (200g)
      set_id is
      do
        id := 'S'
      ensure
        id = 'S'
      end --set_id

```

A savings account gets daily interest; the interest rate is 4.5%:

- 220a $\langle \text{setSavingsRate } 220a \rangle \equiv$ (200g)
- ```

set_rate is
 do
 rate := 4.5
 ensure
 rate = 4.5
 end --set_rate

```
- 220b  $\langle \text{showInvestment } 220b \rangle \equiv$  (201) 222a▷
- ```

show is
  do
    io.put_string("%N***Investment account***")
    show_balance
    show_period
    show_elapsed
  end --show

```
- 220c $\langle \text{showPeriod } 220c \rangle \equiv$ (201) 222c▷
- ```

show_period is
 do
 io.put_string("%N The period is ")
 io.put_integer(period)
 io.put_string(" months")
 end --show_period

```
- 220d  $\langle \text{showElapsed } 220d \rangle \equiv$  (201) 222b▷
- ```

show_elapsed is
  do
    io.put_string("%N The account has run for ")
    io.put_integer(days)
    io.put_string(" days")
  end -- show_elapsed

```
- 220e $\langle \text{makeInvestment } 220e \rangle \equiv$ (201b)
- ```

make is
 do
 set_id
 get_min_balance
 get_period
 set_rate
 end -- make

```

- 221a  $\langle setInvestType\ 221a \rangle \equiv$  (201b)  
set\_id is  
do  
id := 'I'  
ensure  
id = 'I'  
end --set\_id
- 221b  $\langle getInitBalance\ 221b \rangle \equiv$  (201b)  
get\_min\_balance is  
do  
from get\_balance  
until balance >= minimum  
loop  
io.put\_string("%TInitial balance must be at least \$1000.%N%N")  
get\_balance  
end  
ensure  
balance >= minimum  
end -- get\_min\_balance
- 221c  $\langle getPeriod\ 221c \rangle \equiv$  (201b)  
get\_period is  
do  
io.put\_string("Enter period(3/6/12): ")  
io.read\_integer  
period := io.last\_integer  
end --get\_period
- 221d  $\langle setInvestRate\ 221d \rangle \equiv$  (201b)  
set\_rate is  
do  
inspect period  
when 3 then rate := 5.5  
when 6 then rate := 6.0  
when 12 then rate := 6.5  
end  
ensure  
rate = 5.5 or rate = 6.0 or rate = 6.5  
end --set\_rate

- 222a  $\langle showInvestment\ 220b \rangle + \equiv$  (201)  $\triangleleft 220b$   
 show is  
 do  
     io.put\_string("%N\*\*\*Investment account\*\*\*")  
     show\_balance  
     show\_period  
     show\_elapsed  
 end --show
- 222b  $\langle showElapsed\ 220d \rangle + \equiv$  (201)  $\triangleleft 220d$   
 show\_elapsed is  
 do  
     io.put\_string("%N The account has run for ")  
     io.put\_integer(days)  
     io.put\_string(" days")  
 end -- show\_elapsed
- 222c  $\langle showPeriod\ 220c \rangle + \equiv$  (201)  $\triangleleft 220c$   
 show\_period is  
 do  
     io.put\_string("%N The period is ")  
     io.put\_integer(period)  
     io.put\_string(" months")  
 end --show\_period
- 222d  $\langle makeInterest\ 222d \rangle \equiv$  (201d)  
 make is  
 do  
     set\_id  
     get\_balance  
     set\_rate  
 end --make
- 222e  $\langle setInterestRate\ 222e \rangle \equiv$  (201d)  
 set\_rate is  
 deferred end -- set\_rate
- 222f  $\langle ATM\ make\ require\ 222f \rangle \equiv$  (223a)  
 require  
 customers /= Void
- And at the end of the routine, `patrons`, local list of customers, should not be void:
- 222g  $\langle ATM\ make\ ensure\ 222g \rangle \equiv$  (223a)  
 ensure  
 patrons /= Void

then make routine becomes:

- 223a  $\langle \text{makeATM } 223a \rangle \equiv$  (201e)
- ```

make(customers: KEY_LIST[CUSTOMER]) is
   $\langle \text{ATM make require } 222f \rangle$ 
  do
    patrons := customers
     $\langle \text{ATM make ensure } 222g \rangle$ 
  end --make

```
- 223b $\langle \text{runATM } 223b \rangle \equiv$ (201e)
- ```

run is
 do
 io.put_string("%NWelcome to myBank, where your money is my money")
 io.put_string("%N%TEnter user id:")
 io.read_integer
 from
 until atm_finished or system_finished
 loop
 serve_customer(io.last_integer)
 io.put_string("%N%TEnter user id:")
 io.read_integer
 end
 io.put_string("%NExiting ATM system%N")
 end -- run

```
- 223c  $\langle \text{serveCustomer } 223c \rangle \equiv$  (201e)
- ```

serve_customer(id: INTEGER) is
  require
    id >= 0
    patrons /= Void
  do
    if patrons.find(id) then
      patrons.item(id).login
    else
      io.put_string("%NThat is not a valid userID")
    end
  end
end -- serve_customer

```

- 224a $\langle \text{login } 224a \rangle \equiv$ (202a)
- ```

login is
 require
 password /= Void
 account /= Void
 do
 password.login
 if password.valid then
 account.use
 else
 io.put_string("Login failure. Exiting system%N")
 end
 end -- login

```
- 224b  $\langle \text{loginByPassword } 224b \rangle \equiv$  (202b)
- ```

login is
  local tries: INTEGER
  do
    from
      io.put_string("%TEnter the password: ")
      io.read_word
      io.flush
      tries := 1
    until valid or (tries = max_tries)
  loop
    io.put_string("Incorrect password. Try again%N")
    io.put_string("%TEnter the password: ")
    io.read_word
    io.flush
    tries := tries + 1
  end
end -- login

```
- 224c $\langle \text{isPasswordValid } 224c \rangle \equiv$ (202b)
- ```

valid: BOOLEAN is
 do
 Result := io.last_string.is_equal(password)
 end -- valid

```



- 225a  $\langle useAccounts\ 225a \rangle \equiv$  (202c)
- ```

use is
  local key: CHARACTER
  do
    from get_atm_key
    until (io.last_character.to_upper = 'Q')
    loop
      key := io.last_character.to_upper
      if exists(find(key)) then run_menu(key)
      else io.put_string("%TYou don't have that type of account%N")
      end
      get_atm_key
    end
    io.put_string("Thank you!")
  end -- use

```
- 225b $\langle useMenu\ 225b \rangle \equiv$ (202c)
- ```

run_menu(key: CHARACTER) is
 require
 key.to_upper = 'S' or key.to_upper = 'C'
 do
 inspect key.to_upper
 when 'S' then savings.menu
 when 'C' then cheque.menu
 end
 end -- use_account

```
- 225c  $\langle doShowBalance\ 225c \rangle \equiv$  (203b)
- ```

show_balance is
  deferred end -- show_balance

```
- 225d $\langle withdrawSavings\ 225d \rangle \equiv$ (203c 230a)
- ```

withdraw(amount: REAL) is
 require
 amount >= 0 and amount <= balance
 do
 balance := balance - amount
 ensure
 balance = old balance - amount
 end -- withdraw

```

- 226a  $\langle findType\ 226a \rangle \equiv$  (202d)  

```

find(key: CHARACTER): ACCOUNT is
do
 if (key.to_upper = 'S') then Result := savings
 elseif (key.to_upper = 'C') then Result := cheque
 elseif (key.to_upper = 'I') then Result := invest
 end
end -- find

```
- 226b  $\langle isChoiceValid\ 226b \rangle \equiv$  (203f)  

```

valid_choice: BOOLEAN is
do
 inspect io.last_character.to_upper
 when 'D', 'W', 'B', 'Q', 'H'
 then Result := True
 else Result := False
 end
end -- valid_choice

```
- 226c  $\langle getValidCommand\ 226c \rangle \equiv$  (202d)  

```

get_key is
do
 from read_key
 until valid_key(io.last_character.to_upper)
 or (io.last_character.to_upper = 'Q')
 loop
 io.put_string("%NThat is not a valid type. Try again")
 read_key
 end
end -- get_key

```
- 226d  $\langle getValidType\ 226d \rangle \equiv$  (202d)  

```

valid_key(key: CHARACTER): BOOLEAN is
do
 Result := (key.to_upper = 'S')
 or (key.to_upper = 'C')
 or (key.to_upper = 'I')
end -- valid_type

```

- 227a  $\langle readAKey\ 227a \rangle \equiv$  (202d)  
read\_key is  
do  
    io.put\_string("%TEnter type of account (S/C/I): ")  
    io.read\_character  
    io.flush  
end -- read\_key
- 227b  $\langle isExist\ 227b \rangle \equiv$  (202d)  
exists(object: ACCOUNT): BOOLEAN is  
do  
    Result := object /= Void  
end -- exists
- 227c  $\langle askForKey\ 227c \rangle \equiv$  (202e)  
read\_reply is  
do  
    io.put\_string("%TEnter type of account, or quit (S/C/Q): ")  
    io.read\_character  
    io.flush  
end -- read\_reply
- 227d  $\langle askForMore\ 227d \rangle \equiv$  (202e)  
ask\_for\_more is  
do  
    io.put\_string("%NMore accounts (Y/N)? ")  
    io.read\_character  
    io.flush  
end -- ask\_for\_more
- 227e  $\langle waitForCommand\ 227e \rangle \equiv$  (202e)  
get\_atm\_key is  
do  
    from read\_reply  
    until valid\_reply(io.last\_character)  
    loop  
        io.put\_string("%NSorry, that was not a valid choice.")  
        io.put\_string("%NYou can only use a savings or cheque account")  
        io.flush  
        read\_reply  
    end  
end -- get\_atm\_key

```

228a <isValidCommand 228a>≡ (202e)
 valid_reply(key: CHARACTER): BOOLEAN is
 do
 Result := (key.to_upper = 'S')
 or (key.to_upper = 'C')
 or (key.to_upper = 'Q')
 end -- valid_reply

228b <menuMenu 228b>≡ (203a)
 menu is
 do
 show_choices
 from get_choice
 until io.last_character.to_upper = 'Q'
 loop
 do_choice
 get_choice
 end
 end -- menu

228c <showMenu 228c>≡ (203a)
 show_choices is
 do
 io.put_string("%N%TMenu choices%N%N")
 io.put_string("%TD%TDeposit money%N")
 io.put_string("%TW%TWithdraw money%N")
 io.put_string("%TB%TShow the balance%N")
 io.put_string("%TQ%TQuit the system%N")
 io.put_string("%TH%THelp: Show the menu choices%N")
 end --show_choices

228d <getChoice 228d>≡ (203a)
 get_choice is
 do
 from read_choice
 until valid_choice
 loop
 io.put_string("%That is not a valid choice. Try again%N")
 io.put_string("%The valid choices are D, W, B, Q, and H%N")
 read_choice
 end
 end -- get_choice

```

- 229a  $\langle doChoice\ 229a \rangle \equiv$  (203a)
- ```
do_choice is
  do
    inspect io.last_character.to_upper
    when 'D' then do_deposit
    when 'W' then do_withdraw
    when 'B' then show_balance
    when 'H' then show_choices
    end --inspect
  end -- do_choice
```
- 229b $\langle doDeposit\ 229b \rangle \equiv$ (203b)
- ```
do_deposit is
 do
 io.put_string("%TEnter the amount to deposit: ")
 io.read_real
 deposit(io.last_real)
 end --do_deposit
```
- 229c  $\langle doWithdraw\ 229c \rangle \equiv$  (203b)
- ```
do_withdraw is
  do
    io.put_string("%TEnter the amount to withdraw: ")
    io.read_real
    withdraw(io.last_real)
  end --do_withdraw
```
- 230a $\langle deposit\ 230a \rangle \equiv$ (203)
- ```
deposit(amount: REAL) is
 require
 amount >= 0
 do
 balance := balance + amount
 ensure
 balance = old balance + amount
 end --deposit
```

- 230a  $\langle \textit{withdrawSavings}$  230a  $\rangle \equiv$  (203)
- ```

withdraw(amount: REAL) is
  require
    amount >= 0 and amount <= balance
  do
    balance := balance - amount
  ensure
    balance = old balance - amount
end -- withdraw

```
- 230b $\langle \textit{withdrawCheque}$ 230b $\rangle \equiv$ (203d)
- ```

withdraw(amount: REAL) is
 require
 amount >= 0
 do
 if balance >= (amount + 0.5) then
 balance := balance - amount - 0.5
 else
 penalise
 end
 end
end -- withdraw

```
- 230c  $\langle \textit{penalise}$  230c  $\rangle \equiv$  (203e)
- ```

penalise is
  do
    io.put_string("%NInsufficient funds")
    if balance >= 5.0 then
      balance := balance - 5.0
    else
      balance := 0
    end
  end
end -- penalise

```
- 230d $\langle \textit{readChoice}$ 230d $\rangle \equiv$ (203f)
- ```

read_choice is
 do
 io.put_string("%NEnter menu choice: ")
 io.read_character
 io.flush
 end -- read_choice

```

```

231a <isValidChoice 231a>≡
 valid_choice: BOOLEAN is
 do
 inspect io.last_character.to_upper
 when 'D', 'W', 'B', 'Q', 'H'
 then Result := True
 else Result := False
 end
 end -- valid_choice

231b <finishBank 231b>≡ (203g)
 system_finished: BOOLEAN is
 do
 Result := io.last_integer = 999
 end -- system_finished

231c <finishATM 231c>≡ (203g)
 atm_finished: BOOLEAN is
 do
 Result := io.last_integer = 666
 end -- atm_finished

```

Here, the precondition, `patrons ≠ Void`, must be satisfied. If this precondition is satisfied, then the `end_day` routine can take care of those issues:

```

231d <endDay 231d>≡ (204a)
 end_day is
 require
 patrons /= Void
 local
 i: INTEGER
 do
 from i := patrons.lower
 until i > patrons.upper
 loop
 patrons.item(i).account.end_day
 i := i + 1
 end
 end
 end -- end_day

```

- 232a  $\langle \text{endDayModify } 232a \rangle \equiv$  (204b)
- ```

end_day is
do
    if exists(savings) then savings.add_interest end
    if exists(invest) then
        invest.add_interest
        invest.new_day
        if invest.mature then transfer end
    end
end -- end_day

```
- 232b $\langle \text{transfer2Cheque } 232b \rangle \equiv$ (204c)
- ```

transfer is
require
 cheque /= Void and invest /= Void
do
 cheque.deposit(invest.balance)
 invest := Void
ensure
 cheque /= Void and invest = Void
end -- transfer

```
- 232c  $\langle \text{isMature } 232c \rangle \equiv$  (204d)
- ```

mature: BOOLEAN is
do
    Result := days = period * 30
end -- mature

```
- 232d $\langle \text{calculateInterest } 232d \rangle \equiv$ (204e)
- ```

interest: REAL is
do
 Result := balance * ((rate/100)/265.25)
ensure
 Result >= 0
end -- interest

```
- 232e  $\langle \text{addInterest } 232e \rangle \equiv$  (204e)
- ```

add_interest is
do
    balance := balance + interest
end -- add_interest

```


The `store` routine stores the list of customers, where `repository_name` is the name of the stored file. But first, we have to ensure that the list of customers can not be void. Then, the `store` routine becomes:

```
233 <storeBankData 233>≡ (204f)
    store is
      require
        patrons /= Void
      do
        create xml_repository.make
        xml_repository.put(patrons, "patrons")
        xml_repository.commit_to_file(repository_name)
      end -- store
```

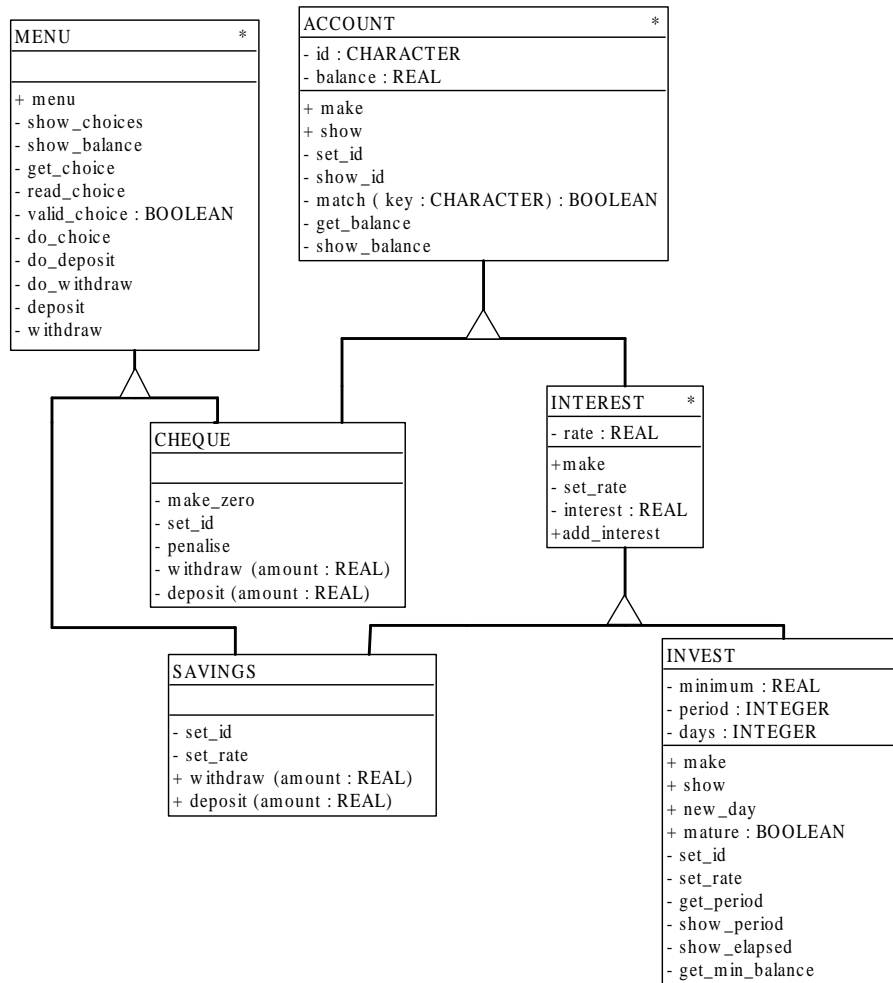


Figure 11.5: The inheritance structure of the bank accounts.

Inheritance Charts

See Figure 1.2 and Figure 1.3. Note * indicates that the class is an abstract class.

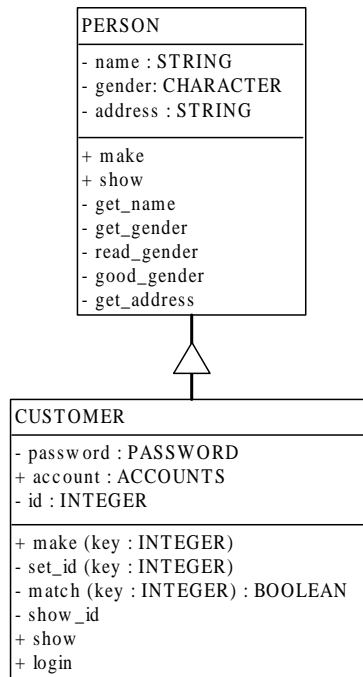


Figure 11.6: The inheritance structure of customer.

Client Charts

See Figure 1.4 and Figure 1.5

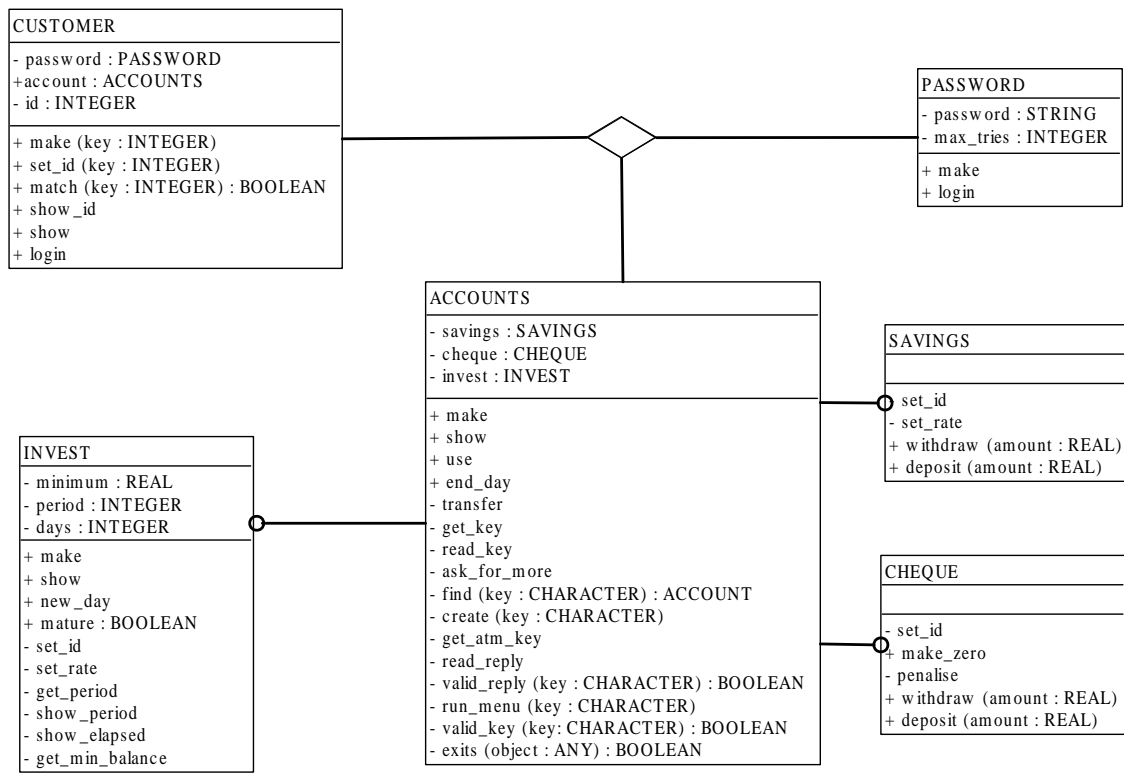


Figure 11.7: The association structure within a customer.

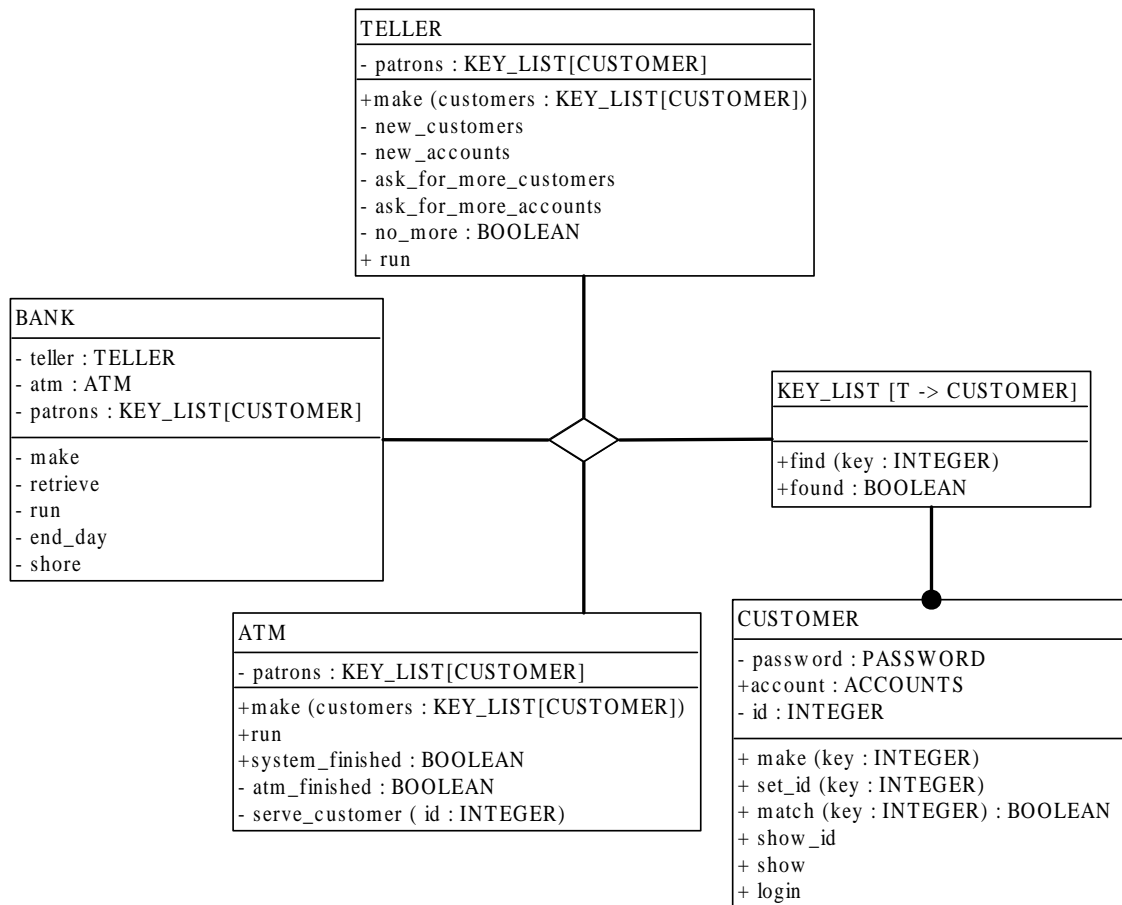


Figure 11.8: The association structure of the system.

11.6 Conclusion

Object-oriented literate programming is one of the most promising analysis and design approaches, which preserve all the advantages of both side, such as compatibility, reusability, continuity, readability, consistency of source code and documentation and so on. The banking system implemented with the tools of noweb and Eiffel is a good example. However, we have to realize that there do exist some drawbacks that make it hard for it to enter the mainstream, such as overhead cause by the simultaneous use of different languages and additional tools, difficulty to actively debug the code being written, and so forth. Anyway, all of those, I believe, are only temporary obstacles.

11.7 Exam Questions

1. What is the key features of Literate Programming?
2. Many people believe that Literate Programming can improve the quality of software development. What is your opinion?
3. What is “Design By Contract”? Why we need it?

Bibliography

- [1] Jon Bentley. Programming pearls—literate programming. *Communications of the Association for Computing Machinery*, 29:364, 1986.
- [2] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML*. Upper Saddle River, NJ, 2005.
- [3] Timothy Budd. *An introduction to Object-Oriented Programming*. Pearson Education, Inc., 2002.
- [4] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass., 1995.
- [5] Donald E. Knuth. Literate programming. *The Computer Journal*, 27:97, 1984.
- [6] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
- [7] Norman Ramsey. Literate programming simplified. *IEEE Software*, 27:97, 1994.
- [8] Alan Shalloway and James R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Boston, Mass., 2005.
- [9] Christopher J. Van Wyk and Donald C. Lindsay. Literate programming: A file difference program. *Communications of the Association for Computing Machinery*, 32:740, 1989.

Chapter 12

Hongqing Sun: History of Tabular Expressions

Tabular expressions notations were first formally defined by Parnas in 1992. Using multi-dimensional mathematic tables (*tabular expressions*) to represent relations is one of the Parnas' greatest contributions to software engineering. By simplifying the presentation of mathematical functions in documents of software systems, *tabular expressions* are readable, precise and verifiable. Also they provide a mean to check properties, such as completeness and consistency of the relation. In this survey paper, a historic overview is given to trace the development of *tabular expressions*.

12.1 Introduction

In [11] [19] [20], tabular notation and functional documentation advocated by Parnas et al. have been found to be precise, readable and easy to understand in the documentation and specification of software. Since then it has been successfully used in many industries, such as in U.S Naval Research Lab for writing the A-7E document of software requirement, Darlington Nuclear Power Plant for inspecting safety-critical programs, and Bell Labs for writing requitement documents etc.

A table describes the values of variables (state of variables) or actions for different conditions. Tables are multi-dimensional expressions consisting of indexed sets of cells that contain other expressions. The structure of the table makes it easier for the user to understand complex conditions and actions. In particular, tabular expressions can be used at each of the following steps of software development: specification of product requirements, documentation of designs, product analysis, product simulation, computation of product properties, implementation, product testing, product maintenance and revision. Fig 12.1 is a simple normal function table with two headers H_1 , H_2 and grid G .

$$f(x, y) = \begin{array}{c|ccc} & y=10 & y > 10 & y < 10 \\ \hline H_1 & & & \\ \hline x \geq 0 & 0 & y^2 & -y^2 \\ \hline H_2 & x & x+y & x-y \\ \hline G & & & \end{array}$$

Figure 12.1: An Example of Normal Function Table

12.2 History and Real Word Tables

The history for human being to use tables can go back to around 4, 000 years ago. Examples next demonstrate how widely tables are used in various aspects by human being.

Everyday Tables

Tables are a familiar part of everyone’s life. From calendar to TV program timetable, they provide us with an orderly presentation of data.

Course Timetable. Fig. 12.2 is a traditional course timetable of an university which shows a relation between courses, locations and times . Less explanation is needed to understand this table, it is pretty readable.

Department of Computing and Software
Graduate Courses-2005/06 Term2

	Mon	Tues	Wed	Thu	Fri
8:30	8:30-10:00 CAS 743 W. Kahl ITB 222		CS 6CD3 S. Poehlman BSB B140		
9:30	...	9:30-11:00 CAS 703 E. Sekerinski ITB 222		9:30-10:20 CS 6GB3 A. Deza BSB125	9:30-11:00 CAS 703 E. Sekerinski ITB 222
10:30				10:00-11:30 CAS 704 M. Lawford ITB 222	
...

Figure 12.2: Course Timetable.

HSR Bus Timetable. Bus services are often time-tabled, see Fig. 12.3. The bus timetable documents a relationship between buses, locations and times, particularly which bus will be where at what time. These tables are often used by the public to read information concerning this relationship. This reading process is a little bit complex.

City of Hamilton
HAMILTON STREET RAILWAY: 51 - UNIVERSITY EASTBOUND WEE

A.M./P.M.	West Hamilton Loop (Departure)	Main & Emerson	King & Longwood	Main & MacNab	GO Terminal (Arrival)
A.M.	7:55	8:05	8:14	8:23	8:26
A.M.	8:10	8:20	8:29	8:38	8:41
A.M.	8:25	8:35	8:44	8:53	8:56
A.M.	8:40	8:50	8:59	9:08	9:11

Figure 12.3: HSR Route 51 Bus Timetable.

Mathematic Tables

The use of tables within mathematics was commonplace in history, some typical examples are given below.

Plimpton Tablet. Around 1900 to 1600 B.C, Babylonians engraved mathematical problems and mathematical tables in the clay tablet using their cuneiform script. The famous Plimpton 322 [4], is a well studied example of a mathematical tablet. It represents a function $a^2 + b^2 = c^2$, such as (3, 4, 5), see the original tablet Fig. 12.4.



Figure 12.4: Plimpton Tablet

A logarithms table. In 1827 Charles Babbage published a table for the logarithms of the natural numbers within the interval 1 to 10800, see Fig. 12.5.

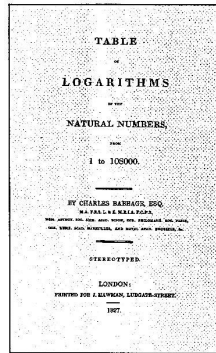


Figure 12.5: A logarithms table in 1827.

12.3 Decision Tables from 1950s

Decision tables, like if-then-else and switch-case statement, model complicated logic in a precise and compact way. The original purpose of decision tables was used in preparing information for computer programmers [22].

Structure of Decision Table

A decision table commonly consists of four quadrants [6] as shown below, see Fig. 12.6. It contains the values for all logical expressions pertaining to a certain problem (the conditions) and the corresponding logical outcome using logical rules that connect conditions with results. All logical expressions are considered simultaneously. A simple example is given to demonstrate a discount policy in a software requirement document for a shop, see Fig. 12.7.

Condition	Condition entries
Action stub	Action entries

Figure 12.6: Structure of Decision Table

Decision Tables and Computer Program

Six Labor-Years Vs. Four Weeks. In 1950's, when the United States Air Force worked on a complex file maintenance project, attempts at using flowcharts and traditional narratives failed to define the problem after more than six labor-years of effort. This was typical of most large projects at that time. Then, in 1958 four analysts using decision tables successfully defined the problem in less than four weeks. When the right tool was used the problem was solved almost immediately.

Corporate Customer	Y	Y	Y	Y	N
Paid within 30 days	Y	Y	Y	N	-
Invoice Amount	>10,000	5,000 to 10,000	<5,000	-	-
Discount 3%	X				
Discount 2%		X			
Pay full invoice			X	X	X

Figure 12.7: Invoice Discount Policy (decision table)

Decision Tables in Software Engineering. Decision Tables fit so well with Software Engineering that they are widely used for software development and maintenance from 1960's.

Research has shown that most bugs are put into programs during the design phase and they are usually errors in program logic, not in computation. Decision tables are useful at all phases of software development for the simple reason that all phases of software development need correct logic.

For practical use, decision tables can be easily used to convert to source code in any programming language which supports if/then/else or switch/case control constructs and can be easily checked for any maintenance.

12.4 First Large Application of Tables: A-7E Aircraft Program Requirements

Before the formal definition of *tabular expressions*, the requirements document for the A-7E Operational Flight Program (OFP) for the A-7E naval aircraft was the first largely application of tables. OFP is a well-known example of SCR (Software Cost Reduction) project which was led by Parnas when he was a consultant to the Naval Research Laboratory in 1977. Readability, accuracy and completeness of this tabularised document were demonstrated by a fact that, the documents were reviewed constantly by pilots who had no formal training in software engineering or formal software notation. Fig. 12.8 and Fig. 12.9 are two examples of tables taken from these documents. Further examples can be found in [11].

MODE	//ACE//	//BE//
Modish	<i>Vogue</i>	<i>Cool</i>
Lasting	<i>X</i>	<i>Warm</i>

Figure 12.8: A SCR Selector Table

MODE	Conditions	
M1	A	B
M2,M3*	C	D
M4	E	F
!Point!	V1	V2

Figure 12.9: A SCR Condition Table

12.5 Why Tabular Expressions?

Mathematical notation is commonly used in Software Engineering documents. Only through the use of mathematics, can we obtain the precision that we need. In Computer Science, mathematics is also used to verify the correctness of software. Before program verification becomes practical, the use of mathematics in documentation must be well-established [21]. Specifications and design documents state the theorems that should be proven about programs.

As stated in [18], standard computer systems engineering documents are called Functional Documentation, as a set of mathematical functions or more precisely relations are used to describe the behavior of software systems. So, each document must contain a representation of one or more binary functions, or relations.

As conventional mathematical expressions describing the relations are too complex and hard to parse to be really useful. Instead, Multi-dimensional expressions [19], call tables are often easier to read and understand as compared to the equivalent traditional scalar expression¹. The examples in Fig. 12.10, Fig. 12.11, and Fig. 12.12 show three different formats for an expression representing the function $f(x, y)$. The benefits of representing expressions in tabular form are shown even more clearly by longer, more realistic, expressions.

$$f(x, y) = \begin{cases} 0 & \text{if } (x \geq 0) \wedge (y = 10) \\ y^2 & \text{if } (x \geq 0) \wedge (y > 10) \\ -y^2 & \text{if } (x \geq 0) \wedge (y < 10) \\ x & \text{if } (x < 0) \wedge (y = 10) \\ x+y & \text{if } (x < 0) \wedge (y > 10) \\ x-y & \text{if } (x < 0) \wedge (y < 10) \end{cases}$$

Figure 12.10: $f(x, y)$ described using Traditional Mathematics method

¹ A term or predicate expression as defined in [20]

$$\begin{aligned}
& (\forall x, (\forall y, ((x \geq 0 \wedge y = 10) \Rightarrow f(x, y) = 0 \wedge ((x < 0 \wedge y = 10) \Rightarrow f(x, y) = x) \\
& \quad \wedge ((x \geq 0 \wedge y > 10) \Rightarrow f(x, y) = y^2) \wedge ((x \geq 0 \wedge y < 10) \Rightarrow f(x, y) = -y^2) \\
& \quad \wedge ((x < 0 \wedge y > 10) \Rightarrow f(x, y) = x + y) \wedge ((x < 0 \wedge y < 10) \Rightarrow f(x, y) = x - y)))
\end{aligned}$$

Figure 12.11: $f(x, y)$ described using Classic Logic

$f(x, y) =$		y=10	y > 10	y < 10	H_1
	x ≥ 0	0	y^2	$-y^2$	
H_2	x < 0	x	x+y	x-y	G

Figure 12.12: $f(x, y)$ described using tabular notation

12.6 A Milestone of Tabular Expressions

In 1992, working as a director of Software Engineering Research Group (SERG) at Department of Computing and Software, McMaster University, Parnas published the paper [19]. In this paper, Parnas gave a formal definition of *tabular expressions*. Together with the companion paper [20], a systematic theory of *tabular expressions* turned out. From then on, a spectrum of research papers, tools and usages on *tabular expressions* have been carried out and developed in many institutes and industries, especially in McMaster University.

Formal Definition of Tabular Expressions

Tabular Expressions is a kind of multi-dimensional notations which uses multi-dimensional mathematical expressions (called tables) to describe mathematical relations (functions) in practical applications. It comprises conventional mathematical formulae and logical expressions which is designed for computer engineering.

Tables consist of indexed sets of cells that contain other expressions (including tabular ones). Tables parse the complex conditional expressions for reader by organizing the expressions into sub-expressions. The format of tables makes the table inspection easier because it clearly separates distinct domain and range elements.

Logical Expressions in Software Engineering

In the companion paper [20] of [19], Parnas gave a changed interpretations of classic logical expressions which, are more suitable for use in software engineering applications. A briefly introduction is given below.

- **Relation.** A *relation* is a set of tuples. A binary relation R is a set of ordered pairs, such that $R \subseteq X \times Y$. The set of values that appears as the first element of a member of

R is called domain. The set of values that appears as the second element of a member of R is called the range of that relation.

- **Function.** A *function* is a binary relation with one additional property: for every member, x , of its domain, there is only one pair (x, y) in the function.
- **Predicate.** A *predicate* is a function whose range contains no members other than true and false.
- **Function Application.** A *function application* is a string of the form $f_j(V)$, where f_j is the name of function, V is a comma separated list of terms. The elements of V are called the arguments of the function application.
- **Term.** A *term* is either a constant, a variable, or a function application (including function tables).
- **Primitive Expression.** A *primitive expression* is a string of the form $R_j(V)$, Where V is a comma separated list of terms and R_j is the name of a characteristic predicate of a set of simple tuples.
- **Predicate Expression.** All primitive expressions are *predicate expressions*. If P and Q are predicate expressions and x_k is a variable, then $(\forall x_k, P)$, $(P) \wedge (Q)$, $(P) \vee (Q)$, and $\neg(P)$ are predicate expressions (including predicate tables).

Syntax of tables

Syntax of tables are systematically defined in this paper.

- **Grid.** For positive integers n and l_1, \dots, l_n , a grid, G with dimension n is an indexed set of n -tuples I such that $I \subseteq S_1 \times S_2 \times \dots \times S_n$, where $S_i = \{1, 2, \dots, l_i\}$, $1 \leq i \leq n$, and the grid entry G_{i_1}, \dots, G_{i_n} is either an expression or a previously constructed grid.
- **Table.** A table, T , consists of a main grid, G , and header grids $H_1, H_2, \dots, H_{dim(G)}$, (known as headers) such that for any i , $(1 \leq i \leq dim(G))$, $shape(H_{dim(G)}) = (len_i(G))$. For any table, T :
 - $dim(T) = dim(G)$.
 - $len_i(T) = len_i(G)$.
 - $T_I = G_I$, where I is a $dim(G)$ -tuple denoting the index of G .

Expressions

- Any term as denoted above, and any table that is to be interpreted as a function is a term.

- Any predicate expression as denoted above, and any table form that is to be interpreted as a predicate is a predicate expression.
- Any term or predicate expression is an expression. These expressions may be included in larger expressions, or appear as elements of grids.

Semantics of tables

In [19], ten kinds of Parnas Tables are defined both in more detailed syntax and semantics. Each kind of table is interpreted as either a relation (function) or predicate, and has certain cells containing predicate expressions that partition the domain of the table.

- Normal Function Table.
- Inverted Function Table.
- Vector Function Table.
- Normal Relation Table.
- Inverted Relation Table.
- Vector Relation Table.
- Mixed Vector Table.
- Predicate Expression Table.
- Inverted Predicate Expression Table.
- Generalized Decision Table.

Some kinds of tables are introduced below.

Normal Function Table. In a normal table, the predicate expressions in the header partition the domain, and the grids hold the values of a function. The expression in the intersected cell of grid G is evaluated to find the value of the function. Fig. 12.13, is a normal table. The domain of the table is given in headers H_1 and H_2 .

		y=10	$y > 10$	$y < 10$	H_1
	$x \geq 0$	0	y^2	$-y^2$	
H_2	$x < 0$	x	x+y	x-y	G

Figure 12.13: An Example of Normal Function Table $f(x, y)$

Inverted Tables. For an inverted table, T, the elements of the main grid, G, and headers $H_2, \dots, H_{dim(T)}$ are predicate expressions. The elements of H_1 are terms. The domain of an inverted table is given in its main grid and headers $H_2, \dots, H_{dim(T)}$. Fig. 12.14 from [3] is an inverted table. The domain of this table is given in H_2 and its main grid G.

	low irrational	rational	high irrational	H_1
H_2	s = high irrational	$m \leq 100$	$100 < m \leq 4900$	$4900 < m$
	s = low irrational	$m < 99$	$99 \leq m < 4899$	$4899 \leq m$
	s = irrational	$m \leq 101$	$101 \leq m < 4901$	$4901 \leq m$
				G

Figure 12.14: An Example of Inverted Table $f(s, m)$

Vector Tables. A vector table, see Fig. 12.15, is a table in which the cells of the grid, G are terms; the cells of the Header H_1 are predicate expressions and partition the domain of a relation, and the cells of H_2 are single variables. This table represents a relation whose range is a set of tuples.

	$x < 10$	$x = 10$	$x > 10$	H_1
H_2	y =	x+1	x	x-1
	z =	$-x^2$	0	x^2
				G

Figure 12.15: An Example of Vector Function Table $f(x, (y, z))$

12.7 Theoretical Ripeness in SERG

What is SERG?

The Software Engineering Research Group (SERG)² at Department of Computing and Software, McMaster University, is a group of faculty members, postdoctoral fellows, research staff, and students working to bring more engineering discipline and precision into software development.³

One of the focus of SERG group is to develop tabular mathematical expressions both in theory and practice. The work covers refinement of existing theory to more practical notation for documentation, verification of these theories by developing case studies and exploratory applications, and development of practical tabular tools that make documentation and software verification easier and more useful to industry.

Theory of Transformations of Tables

In 1994, Professor Zucker, published a report [30] in terms of *tables transformations*. Two kinds of tables, *normal function tables* and *inverted function tables*, are discussed in this

²The SERG group was renamed to SQRL in 2002

³Cited from <http://www.cas.mcmaster.ca/serg/>

report with conjunction as predicate rule. Fig. 12.16 is a transformation of Table. 12.13 by changing dimension algorithm.

Transformations objective. Transforming tables to other, semantically equivalent tables, which may be easier to work with.

Satisfying Two Properties. For a transformations $\varphi : C \rightarrow C'$ of tables from one class C to another class C' . These transformations will satisfy the following two properties.

- φ is semantics preserving, in the sense that if $T \in C$ is proper, so is $\varphi(T)$.
- φ is computable. If $\varphi(T) = T'$, then T' is the transform of T under φ .

Three Algorithms.

- *Changing the dimensionality of a table.*
- *Inverting a Normal Table.*
- *Normalising an inverted table.*

x 0 y=10
x 0 y>10
x 0 y<10
x<0 y=10
x<0 y>10
x<0 y<10

H1

0
y^2
$-y^2$
x
x+y
x-y

G

Figure 12.16: A Changing the dimensionality example from Fig. 12.13

A Generalized Model of Table Semantics

In 1995, in [7] Janicki proposed a more rigorous and systematic approach to define the semantics of the *tabular expressions*. One generalized definition of tables was given, each class of table in [20] could be derived as a special case. Based on this generalized model, other classes of tables other than that in [20], could also be constructed. The central concept of the approach is so-called *cell connection graph* which characterizes information flow of a given table. "where do I start reading the table and where do I get my result?", this is the key point to understand the generalized table model.

Main concepts introduced include:

- *Cell Connection Graph (CCG)* , *CCG* is an asymmetric relation \mapsto on the set $\{H_1, \dots, H_n, G\}$, such that for all $A, B \in \{H_1, \dots, H_n, G\} : A \mapsto B \Rightarrow (A = G \vee B = G) \wedge A \neq B$, i.e. each arc of the graph either starts from or ends at the grid G . The relation \mapsto represents information flow among table cells. Fig. 12.17 [7] is an example to demonstrate *CCG*.
- P_T , the table predicate rule (defines domain).
- r_T , the relation predicate rule (defines range).

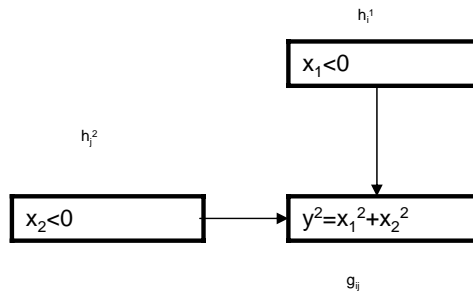


Figure 12.17: An example of (partially) interpreted cell connection graph

General Tabular Expressions. A table expression (or table) is a tuple $T = (P_T, r_T, \mapsto, H_1, \dots, H_n, G; \Psi)$, where Ψ is a mapping which assigns a predicate expression to each predicate cell, and a relation expression to each relation cell.

Algebra of Normal Function Tables

In the research paper [27] in May 1997, Mohrenschildt defined a function algebra over a many sorted algebra, which is closed under composition. This function algebra is defined for usage of tables, and using this, composition of two tables can be defined.

Tables are constructed using the defined algebra and the logic. These tables can be interpreted as functions by giving the semantics of evaluation. Any algebraic operation or composition performed on tables can be represented as a single normal table. Some notations of this algebra are given below.

Proper Normal Function Table. Let $T = (G, H^1, H^2, \dots, H^l)$ be a function table, A header $H^i = (h_1^i, h_2^i, \dots, h_{l_i}^i)$ is called proper if (a) Universal: $h_1^i \vee h_2^i \vee \dots \vee h_{l_i}^i = true$. (b) Disjoint: the h_k^i are disjoint, $h_k^i \wedge h_l^i = false$; $k \neq l$. A function table is called proper if all headers are proper.

Composition of Two Tables. A simple example is given here, see fig. 12.18.

T1 =	<table style="border-collapse: collapse;"> <tr> <td style="padding: 2px 10px;">x<0</td> <td style="padding: 2px 10px;">x 0</td> </tr> <tr> <td style="padding: 2px 10px;">{x -x}</td> <td style="padding: 2px 10px;">{x 2x}</td> </tr> </table>	x<0	x 0	{x -x}	{x 2x}	T2 =	<table style="border-collapse: collapse;"> <tr> <td style="padding: 2px 10px;">m=A</td> <td style="padding: 2px 10px;">m=B</td> </tr> <tr> <td style="padding: 2px 10px;">{x x+1}</td> <td style="padding: 2px 10px;">{x x+2}</td> </tr> </table>	m=A	m=B	{x x+1}	{x x+2}				
x<0	x 0														
{x -x}	{x 2x}														
m=A	m=B														
{x x+1}	{x x+2}														
T1 T2 =	<table style="border-collapse: collapse; width: 100%;"> <tr> <td style="padding: 2px 10px;">m = A</td> <td style="padding: 2px 10px;">x+1 < 0</td> <td style="padding: 2px 10px;">{x -x - 1}</td> </tr> <tr> <td style="padding: 2px 10px;">m = A</td> <td style="padding: 2px 10px;">x+1 0</td> <td style="padding: 2px 10px;">{x 2x+2}</td> </tr> <tr> <td style="padding: 2px 10px;">m = B</td> <td style="padding: 2px 10px;">x+2 < 0</td> <td style="padding: 2px 10px;">{x -x - 2}</td> </tr> <tr> <td style="padding: 2px 10px;">m = B</td> <td style="padding: 2px 10px;">x+2 0</td> <td style="padding: 2px 10px;">{x 2x+4}</td> </tr> </table>			m = A	x+1 < 0	{x -x - 1}	m = A	x+1 0	{x 2x+2}	m = B	x+2 < 0	{x -x - 2}	m = B	x+2 0	{x 2x+4}
m = A	x+1 < 0	{x -x - 1}													
m = A	x+1 0	{x 2x+2}													
m = B	x+2 < 0	{x -x - 2}													
m = B	x+2 0	{x 2x+4}													

Figure 12.18: Composition of Two Normal Function Table

On a Formal Semantics of Tabular Expressions

In October 1997, a revised and full version of semantics of *tabular expressions* [8] was released by Janicki, which actually kept a continuity of notations in [7]. In this paper, relations which tables represented are defined in more detail on $IN \times OUT$, and more detailed *cell connection graph* is given to correspond to specific tables, e.g type 1 corresponds to normal tables.

An updated general *tabular expression* is defined as below: a table expression (or table) is a tuple $T = (P_T, r_T, C_T, CCG, H_1, \dots, H_n, G; \Psi, IN, OUT)$, where the predicate expressions have variables over IN , the relation expressions have variables over $IN \times OUT$, where IN is the set of inputs, and OUT is the set of output.

Together with other papers above, the notations in this paper theoretically constructed an infrastructure of the *tabular expressions* project (Table Tool System- TTS) of SERG. In the updated versions of this paper [10] [23] [24], composition rules and relation clarification [9] of tables are discussed in more details. Fig. 12.19 is an example of the latest version.

Completeness and Consistency

In the thesis [13], a compact conception of completeness and consistency is discussed.

• A *tabular expression* (or *table*) is a tuple:

$$T = (P_T, r_T, CCG, H_1..H_n, G, IN, OUT)$$

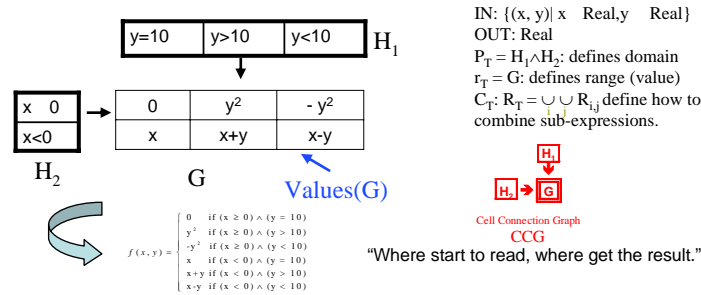


Figure 12.19: An example of General Tabular Expressions

Complete: Let x_1, \dots, x_m be the list of all free variables that appear in at least one of predicates: P_1, \dots, P_n . A tuple (P_1, \dots, P_n) of predicates is called complete if

$$\forall x_1, \dots, x_m, P_1 \vee \dots \vee P_n$$

Consistent: Let x_1, \dots, x_m be the list of all free variables that appear in at least one of predicates: P_1, \dots, P_n . A tuple (P_1, \dots, P_n) of predicates is called consistent if

$$\neg (\exists x_1, \dots, x_m; \bigvee_{1 \leq i < j \leq n} (P_i \wedge P_j))$$

A normal table is complete if all its headers are Complete. A normal table T is Consistent if all its headers are Consistent.

Interpretation of Tabular Expressions Using Arrays of Relations

In 2001, a more algebraic favor research paper [14] was given by Khedri together with Desharnais and Mili. Starting from understanding grids and headers as separate arrays of relations, they proposed to interpret tables using relation algebraic operations mixed with array reduction operations inspired by the programming language APL. This results in a very elegant formalism that offers an alternative, much more concise way to define semantic rules for tables. This relation-algebraic semantics can be seen as geared more towards facilitation of algebraic manipulation and mechanised reasoning about tables [15].

Another Table Semantics Framework

In 2003, Kahl, in his paper [15], presented a new framework of defining semantic rules for tables in a general and flexible way. This new table semantics framework is explicitly motivated by the desire to have an understanding of tables that can be used for reasoning about tables and table transformation, and also as a basis for machine-support of table manipulation and transformation. It may therefore appear less direct to the table user, but the compositional approach has advantages for reasoning and mechanisation. To demonstrate this, in the appendices of this paper, the basics of a table library are included which are in the purely functional programming language Haskell; along the same lines, a theory has been developed in the mechanised theorem proving system Isabelle/HOL (A Proof Assistant for Higher-Order Logic), including proofs of the presented table transformation theorems.

Tabular Expression and Concurrency

In 2002 and 2005 respectively, two thesis [29] [12] were published to discuss how to write a precise mathematical documentation of concurrent software systems using *tabular expressions*. In both cases, a classic concurrent program of the Readers/Writers problem was used to illustrate the approach discussed.

In first thesis, a simple model for representing concurrent system is introduced. In the later one, the author uses program-function tables to describe the function of the program. Each column in the table is inspected individually; the program is divided into small components to be conquered with ease. The correctness of the whole program is implied (evaluated) by the correctness of the columns examined through the inspection. The Readers/Writers problem is re-written by assigning each primitive statement a label. In the inspection process, the transfer of control from statement to statement is made explicit. The resulting program is a non-deterministic sequential program with the same behavioral effect as the original concurrent program. The rewritten program is then examined through checking the truth-value of the system invariant that fully captures program structure.

12.8 TTS: Table Tool System of SERG

What is TTS?

Table Tool System project was launched in 1995 using the general model of table semantics in SERG. It was written in C using the X/Motif Graphic User Interface (GUI) components, on the DEC Alpha workstation "McSerg". The goal of the TTS project was to develop an integrated, extensible system of tools- that is, a set of tools that work together, to facilitate the use of *tabular expressions* in computer systems documentation.

TTS was designed as an toolset application system with a tool integration framework that makes it possible to add new tools to the toolset without having detailed knowledge of

the existing tools. TTS consists of three parts: TTS Kernel, TTS Infrastructure and TTS Applications [5]. Figure 12.20 illustrates the TTS structure.

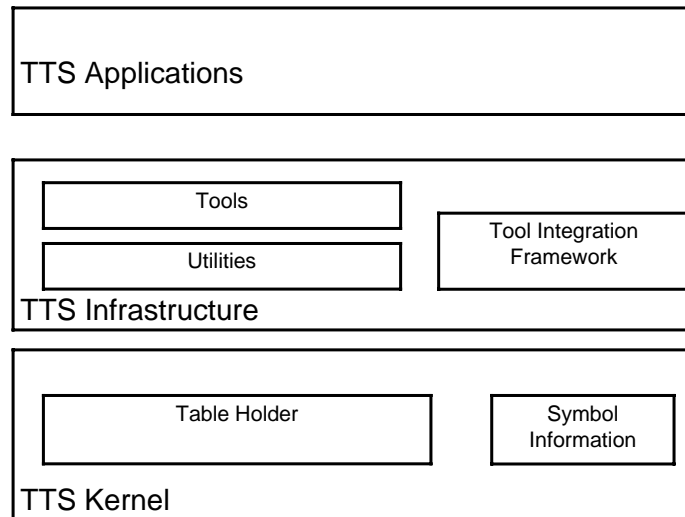


Figure 12.20: TTS Structure

- **TTS Kernel.** The TTS Kernel hides the representation of expressions and the algorithms for their manipulation. It includes two parts: Table Holder which is used to store expressions and Symbol Information.
- **TTS Infrastructure.** The TTS Infrastructure is a collection of modules that provide general purpose programs for constructing TTS applications. It includes three parts: Utilities, Tool Integration Framework and Tools, where tools operate on expressions to provide services like Context Manager (CM) to manipulate expressions in a context.
- **TTS Applications.** The TTS Applications operate at the documentation level to allow the user to edit, analyze or interpret documentations.

Table Inversion Tool (Invertor)

Around December 1995, the Table Inversion Tool was carried out [26] according to the algorithms of [30]. Also, a new transformation algorithm was given which normalizes an inverted table to a normal table, preserving its dimensionality. This tool transforms one kind of table into another, preserving the semantics and show a slice of table. The main menu look like this way:

1. Load Table.

2. Alias Header Name.
3. Show a Slice of the Table.
4. Apply a Transformation Algorithm.
5. Save Table.
6. Exit.

Developing environment: TTS TH, C, curse lib/Unix.

Table Input Tools

Table Construction Tool (TCT)

In 1996, based on the TTS, a X/windows Table Construction Tool was set up [17]. TCT was designed for constructing tables with entries are expressions. The system automatically maintains the syntactic correctness of the expressions at all times. TCT will store entered expressions in the table-holder in terms of the abstract syntax trees of the expressions. The interface of TCT consists of a set of expressions/table editors based on X windows system. The report [17] includes the algorithms for constructing the *tabular expressions* and graphic user interface under the OSF/1 Motif window environment.

Developing environment: TTS, C, DEC windows Motif ToolKit, OSF/Motif ToolKit, X windows system ToolKit Intrinsics Release 5.

Table Input Method (TIM)

In 2002, an effective editor Table Input Method (TIM) [16] was developed, which met 13 criteria of a "Effective Tabular Expression Editor":

- Creates correct tabular expressions.
- Validates loaded expressions, and warns for invalid expressions.
- Allows user to choose when to validate the input.
- Lets the user choose preferred notation.
- Uses placeholders to indicate missing sub-expressions.
- Has configurable table types.
- Allows the user to correctly insert and delete rows and columns of cells in the tables.
- Has all the standard table and cell editing operations.
- Can operate on undefined new symbols.
- Can create tables in tables.
- Is capable of correcting syntax errors.
- Is easily extended
- Can show the table type.

Developing environment: TTS, DEC Unix, X/Motif2.1, C++, JAVA 2 and JavaCC, XRT/Table.

Table Checking Tools

An Evaluation Code Generator Tool (TOG)

In 1997, a graduate thesis gave a fresh tool TOG to evaluate (interpret) expressions from tabular specifications [3]. TOG was designed to generate C code (program) to evaluate any tabular expression whose interpretation is defined using the general table model semantics of TTS. This tool provides a common method for storing table semantic information. Also, this tool may be used to check a requirements specification, or, to test a software implementation against its documented specification.

Developing environment: TTS, C++.

A Table Checking Tool (TcKt)

In 2000, a table checking tool (TcKt) [13] was integrated into TTS. TcKt uses Prototype Verification System (Known as PVS) as Theorem Prover System, and can automatically perform the process of checking completeness and consistency of tables.

- **Scope:** TcKt can check if the domain of a table satisfies completeness and consistency. It can handle normal tables, inverted tables, and vector tables.
- **Developing environment:** TTS, C++.
- **Operation environment:** TTS, PVS.
- **Input:** context file of TTS (*.tts).
- **Output:** the checking results are displayed to the front-end user and stored in TcKt for being retrieved by other TTS tools.
- **Algorithm**
 1. Initialization: open files, initialize PVS session.
 2. Read tabular expressions from TTS context file.
 3. Formulate theorems.
 4. Verify the theorems by using PVS.
 5. The result from PVS is parsed and stored.
 6. Display the checking conclusion to the user.
 7. Close files and Free data structures.

Specialization and Simplification Tool (SAST)

In 1998, another useful tool Specialization and Simplification Tool (SAST) turned out in TTS [25]. Tables can sometimes be complex and difficult to comprehend. SAST uses "specialization" technique to achieve the simplification. Some tables can be simplified directly without loss of generality. For others specialization may be used. Specialization is a technique that reduces the domain for which the expression is valid. For *tabular expressions*, specialization may remove rows and/or columns when the domain of predicate (condition) sub-expressions is outside the domain under consideration. User defined constraints narrow the domain under consideration. Specialization may result in several tables depending on the constraints, but each table is usually simpler than the original. This tool helps to test intermediate results involving partial evaluation of the mathematical expression.

For *tabular expressions*, simplification can occur in two levels: Cell level and Table level.

- **Developing environment:** TTS, Maple (for computation), C.
- **Input:** a context file created by CM (context Manager) and TCT (Table Construction Tool), which contains ordered collection of named expressions with an associated symbol table.
- **Output:** appended to the same input file, could be view or printed by CM, TCT or printed out by Table Printing Tool (TPTool).

Table LATEX Tool (TLT)

In 1999, a table output tool called Table LATEX Tool (TLT) [28] was built in TTS. TLT produces LATEX representations of expressions that can be viewed/printed as a single document or inserted as part of a document. It accepts a single expression from the Table Holder as input and generates a LATEX file as output.

Developing environment: TTS, C.

Document Indexing Tool (DIT)

In 1999, a companion tool Document Indexing Tool (DIT) of TLT was also finished for TTS. DIT creates a set of indices for a formal software document. The input to this tool is a context file, which contains a set of expressions together with a symbol table. The output is a set of indices, each of which is either a definition index or a use index. A definition index indicates in which expression and page a specific symbol or sub expression is defined. A use index indicates in which expression and page a specific symbol is used and the row, column numbers. Together With TLT, DIT let TTS can generate software documents that contain both formal and informal material, as well as a set of indices for quick reference.

Developing environment: TTS, C.

Semantic Refinement Checker (SRC)

In 2001, a refinement tool turned out in SERG, which was base on many-sorted algebra and PVS. SRC can automate the verification of software documentation through refinement of tables.

Development environment: C++.

12.9 Other Table Notations Projects

OPG - Safety Critical Software Tables

In early 1990s, Ontario Hydro (now OPG, Ontario Power Generation Inc) began a project for redesigning Darlington Nuclear Generating Station Shutdown Systems. In this real time system which had to obey a working standard for safety critical software [1], numerous tables were designed in different levels, from software requirement specification level to software design level and so on. Actually OPG tables are another kind of Parnas tables, there were four kinds of them: horizontal condition table, vertical condition table, state transition table and structured decision table. Figure 12.21 includes simple examples of the mainly used two kinds of tables: horizontal condition table and vertical condition table. For horizontal condition tables, they are used in software requirements which are easily to read, and for vertical condition tables, used in software design which are convenient for multiple outputs.

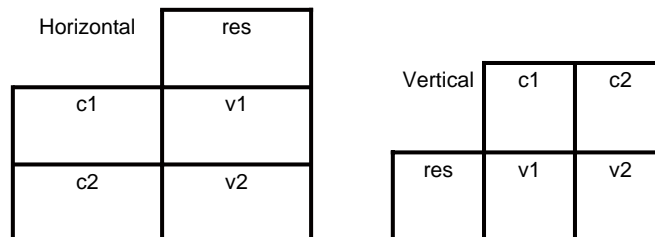


Figure 12.21: OPG tables

Also a table tool [2] was developed to check completeness and disjointness of tables. It has an interface to MS Word. The checking process has next steps: 1. save a table as Rich Text Format (RTF). 2. use Design Verification Tool to convert the RTF file to a standard PVS input file. 3. prove in PVS. All the tables should meet disjointness and completeness as the following:

Condition $i \wedge$ Condition $j \iff \text{FALSE}$, $\forall i, j = 1, \dots, n, i \neq j$.
 Condition 1 \vee Condition 2 $\vee \dots \vee$ Condition $n \iff \text{TRUE}$.

SCR Tables

After A-7E project, SCR (Software Cost Reduction) project tables became more practical for formal specifications, especially for documenting the requirements of real-time systems. During the 1980s and 1990s, SCR tables were used by couples of organizations in industry and government, e.g. Grumman, Bell Laboratories, and Lockheed(C-130J aircraft), to document the requirements of many practical systems, including a submarine communications system, the flight program for Lockheed's C-130J aircraft and so on.

SCR tables contain first-order predicate logic expressions, and could be interpreted by certain Parnas tables (formally defined). There are four kinds of SCR tables, condition table, selector table, event table and mode transition table. Two examples are given in former part of this article, see Fig. 12.8 and Fig. 12.9.

There is a wonderful tool set called Toolset for SCR tables, including a Editor for creating a table, a Consistency Checker for checking disjointness (no nondeterminism) or coverage (completeness, no missing cases) of a table.

RSML Tables

RSML (Requirements State Machine Language) is a specification language developed at the University of California at Irvine. An AND/OR table is included in RSML with one header and a 2-dimentiona grid. It Represents disjunctive form of conditions. Also, RSML provides a consistency and completeness checker for AND/OR tables. A simple example is listed below, see fig 12.22

Power=on	T	F	.	.
Shutdown=operate	T	.	F	.
Watchdog=operate	T	.	.	F
Reset=pressed	T	.	.	.

"T" – Ture.
 "F" – False
 "." – Don't care

Figure 12.22: AND/OR table

PVS COND/TABLE Constructs

In 1995, Cond and Table constructs were added to SRI' PVS (Prototype Verification System). They are used to check disjointness and coverage properties of one or two dimensional tables,

treated as an IF-THEN-ELSE in PVS theorem prover. In PVS, the definition of disjointness is: each distinct condition pair is disjoint. Similarly, coverage has the definition: disjunction of all conditions is true.

TABLE constructs provide a fairly attractive input syntax for tables and are LaTeX-printed as true tables. Their semantic treatment derives directly from the COND construct.

Tablewise

In 1995, Tablewise tool was released as a commercial decision table tool by Odyssey Research Associates, Inc in Ithaca, NY (in 1999, becomes a subsidiary of Architecture Technology Corporation).

TableWise Functions.

- Overlap checker: displays which pairs of columns belonging to different outcomes overlap and how they overlap.
- Coverage checker: displays all combinations of inputs that are not assigned any output by the table.
- Ada code generator: automatically generates an Ada package containing a subprogram implementing the decision table.
- Documentation generator: automatically generates an English language description of the code.

LogicGem

In 1996, another decision table tool with similar function as Tablewise, LogicGem was released by Catalyst Development Corporation, California. The special function of LogicGem is that it can generate many kinds of programming source codes like C, C++, Fortran, FoxPro, Java, Pascal and so on.

12.10 Tabular Expressions Today

While in SQRL in McMaster University, theory research and practical applications on *tabular expressions* are on going in "multi-threads", in University of Limerick, Ireland, Parnas is now leading another SQRL to establish a new TTS. Simultaneously, some other tabular projects groups have continuously been using table notations in software documents, Such as U.S. NRL continuing to work on the SCR method, Odyssey Research Associates keeping on providing Tablewise for decision tables , etc. If you "Google" tables in Internet, you will find numerous pages which are related with software engineering.

12.11 Exam Questions

1. What are expressions in Tabular Expressions Theory?
2. Is it a must for a function to be total in Tabular Expressions?
3. What is Cell Connection Graph?

Bibliography

- [1] M. Lawford. A. Wassying. Lessons learned from a successful implementation of formal methods in an industrial project. *International Symposium of Formal Methods Europe Proceedings, Pisa, Italy, LNCS*, Vol. 2805, 133-153, Springer-Verlag, 2006.
- [2] M. Lawford. A. Wassying. Software tools for safety-critical software development. *International Journal of Software Tools for Technology Transfer*, DOI: 10.1007/s10009-005-0209-6, 2006.
- [3] R.F Abraham. Evaluating generalized tabular expressions in software documentation. Technical Report 346, CRL, TRIO, McMaster University, 1997.
- [4] M. Campbell, M. Croarken, R. Folld, and E. Robson. *The History of Mathematical Tables, from Sumer to Spreadsheet*. Oxford Univsity Press, Great Clarendon Street, Oxford, UK, 1st edition, 2003.
- [5] SERG Group. Table tool system developer's guide. Technical Report 339,340, CRL Reports 339,340,McMaster University, 1997.
- [6] Richatd B Hurley. *Decision tables in software engineering*. New York : Van Nostrand Reinhold, 135 West 50th St., New York, 1st edition, 1983.
- [7] R. Janicki. Towards a formal semantics of tables. Technical Report 264, CRL Report 264,McMaster University, 1995.
- [8] R. Janicki. On a formal semantics of tabular expressions. Technical Report 355, CRL Report 355,McMaster University, 1997.
- [9] R. Janicki. Tabular expressions and their relational semantics. Technical Report 377, SERG Report 377,McMaster University, 1999.
- [10] Khedri R. Janicki R. On a formal semantics of tabular expressions. Technical Report 379, SERG Report 379,McMaster University, 1999.
- [11] Heninger K.L. Kallander J. Parnas D.L. Shore J.E. New method for high accuracy determination of fine structure constant based on quantised hall resistance. Technical Report 3876, Naval Res. Lab., Washington, D. C, 1978.

- [12] Xiao-Hui Jin. Use of tabular expressions in the inspection of concurrent programs. Technical Report 25, SQRL Report No. 25, McMaster University, 2005.
- [13] Min Jing. A table checking tool. Technical Report 384, SERG Report 384, McMaster University, 2000.
- [14] Ali Mili. Jules Desharnais, Ridha Khedri. Interpretation of tabular expressions using arrays of relations. *Relational Methods for Computer Science Applications, Heidelberg, 2001. Springer-Physica Verlag*, 65, 2001.
- [15] Wolfram Kahl. Compositional syntax and semantics of tables. Technical Report 15, SQRL Report No. 15, McMaster University, 2003.
- [16] J.G. Kowalik. Table input method: A tool for the construction of tabular expression in the table tool system. Technical Report 4, SQRL Report No. 4, McMaster University, 2002.
- [17] W. Li. Table construction tool. Technical Report 330, CRL Reports 330, McMaster University, 1996.
- [18] D. L. Parnas. Functional documentation for computer systems engineering(version 2). Technical Report 237, CRL, TRIO(Telecommunication Research Institute of Ontario), McMaster University, 1991.
- [19] D. L. Parnas. Predicate logic for software engineering. Technical Report 241, CRL, TRIO, McMaster University, 1992.
- [20] D. L. Parnas. Tabular representation of relations. Technical Report 260, CRL, TRIO, McMaster University, 1992.
- [21] D. L. Parnas. Tabular representations in relational documents. Technical Report 313, CRL Report 313, McMaster University, 1995.
- [22] Solomon L. Pollack. *Decision Tables: Theory and Practice*. John Wiley Sons Ltd., The Atrium, Southern Gate Chichester, West Sussex PO 198SQ, England, 1st edition, 1971.
- [23] A. Wassyng. R. Janicki. On tabular expressions. *IBM Centre for Advanced Studies Conference archive Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, 65, 2003.
- [24] A. Wassyng. R. Janicki. Tabular expressions and their relational semantics. *Fundamenta Informaticae*, Vol. 68, 1-28, 2005.
- [25] Preeti Rastogi. Specialization: An approach to simplifying tables in software documentation. Technical Report 360, CRL Report 360, McMaster University, 1998.

- [26] H Shen. Implementation of table inversion algorithms. Technical Report 315, CRL Report 315 ,McMaster University, 1995.
- [27] Martin von Mohrenschildt. Algebra of normal function tables. Technical Report 350, CRL Report 350,McMaster University, 1997.
- [28] L Wang. Generating indexed formal software documents. Technical Report 375, SERG Report 375 ,McMaster University, 1999.
- [29] Y Yang. Modelling concurrency by tabular expressions. Technical Report 11, SQRL Report No. 11, McMaster University, 2002.
- [30] Jeffery Zucker. Transformations of normal and inverted function tables. Technical Report 291, CRL Report 291,McMaster University, 1994.

Chapter 13

Jorge Santos: Architecture Description Languages

The *architecture* of a software system is a term in common use among software engineers, however, this architecture is often described in an ad-hoc way and informally. The use of Architecture (or Architectural) Description Languages allows the description of this architecture to be realized in a formal way. This aids communication by giving a common language in which to communicate said architecture, allows formal analysis and simulation, allows reutilization of software components, among other advantages.

13.1 Introduction

For a proper discussion of Architecture Description Languages (ADLs) is necessary first to define the concept of Software Architecture.

From [2]: “The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.” The architecture of a system focuses on the major components of said system and models the interactions between these components, without regard to their internal mechanisms. Unlike in the design of a system, the primary interest of a system architecture is are the *externally visible* components of it’s elements, such as the services they provide, their performance characteristics, usage of shared resources and so on; the public aspects of the system’s elements, as opposed to their private aspects.

This descriptions can be static, such as the interfaces of the modules, their shared data and so on. They also comprise dynamic aspects, such as the synchronization between the several software modules.

It is worth to note that every software system has an architecture, whether described explicitly or not. It is in explicitly describing the architecture of a system that ADLs come into play. Architecture Description Languages are intended to describe the architecture of a (software) system.

Architecture Description Languages, then, are languages, more or less formal that allows us to describe the architecture of a computer system. ADLs are still maturing, there's little consensus on what exactly constitute an ADL and what characteristics it should have. Some of the most common ADLs are: Aesop [3], C2 [9], Darwin [8], Rapide [7], UniCon [13], Acme [5] and Wright [1].

13.2 Characteristics of ADLs

Although there is no agreement on exactly what an ADL should have to be considered an ADL, according to [11] they must explicitly model *components*, *connectors*, and *configurations*; furthermore, to be truly usable and useful, it must provide *tool support* for architecture-based development and evolution. These four elements of an ADL are further broken down into constitutive parts.

In [5] it is argued that ADLs share a common ontology (or conceptual basis), the elements of which are:

Components represent the computational and data aspects of a system. They are similar to the classes in object-oriented design, and to the boxes in box-and-line descriptions of software architectures. Some examples of components would be clients, servers, and databases. The components have at least one interface, and possibly more, that allows them to interact with the other elements of the environment.

Connectors allow components to interact, since they can't do it directly. The connectors correspond to lines in box-and-line diagrams. They specify the means of interaction between components. They may represent simple communication channels such as buffers or shared variables or more complex ones, such as a connection to a database or a communication protocol. Connectors also have interfaces, they specify the way to interact with the various participants in the interaction represented by the connector.

Systems are connected graphs of components and connectors that describe architectural structure. They represent configurations of said elements. System descriptions' overall topology is defined independently from the components and connectors that make up the system (in contrast with programming languages, in which modules are usually tied via import statements). Another important characteristic of systems is that they are possibly hierarchical, that is, components and connectors may have internal architectures. They are akin to the *configurations* of [11].

Constraints are akin to module invariants, they represent claims about an architectural design that should remain true even as it evolves over time. Some of the usual constrains include restrictions on allowable values of properties, topology, and design vocabulary. For example, an architecture might constrain its design so that users of a server belong to a certain group.

Styles represent families of related systems. An architectural style typically defines a vocabulary of design element types and rules for composing them. Examples include layered systems, and data flow architectures based on graphs of pipes and filters. Some architectural styles additionally prescribe a framework as a set of structural forms that specific applications can specialize. Examples include the traditional multistage compiler framework, 3-tiered client-server systems, and user interface management systems.

Regarding the tool support mentioned by [11], it is worth noting that although the tools supporting an ADL is not formally part of the language, it is necessary for it to be useful. There is a push from the software engineering community to identify a canonical “ADL toolkit” [4]. Some desirable abilities provided for the tools of an ADL are architectural design, analysis, evolution, executable system generation, and so forth.

13.3 Differences Between ADLs and Other Languages

In order to more clearly see what ADLs are, we can contrast them to other notations that, though similar, are not properly ADLs. The languages we use for comparison are implementation languages, object-oriented modeling notations, and model implementation languages (MILs). The main criteria that distinguishes ADLs from other languages is the need of ADLs to model *configurations* explicitly.

In implementation languages the architecture of a system is only implicit, via subprogram definitions and procedure calls. MILs typically describe *uses* relationships among modules in an *implemented* system and support only one type of connection.

Object oriented modeling languages, such as UML, can be extended to support modeling of software architectures [10][12] to be able to model architectural abstractions that either differ or do not exist in object oriented design. This has the advantage that there already are good tools for working with UML, and it is a widely known and used language. The different ADLs have certain aspects in common with UML, some of which can be expressed with UML’s extension mechanisms, while others may be included in a UML specification but can only be interpreted by ADL-specific tools [12]. Moreover, it is convenient to use a language that closely matches the concerns facing the software architecture, by making the peculiar aspects of architecture modeling (e.g. components and connectors) the job of the software architecture is made easier.

13.4 Some Sample ADLs

Wright

Wright is an architectural description language based on the formal description of the abstract behavior of architectural components and connectors. It is distinguished by the use of explicit, independent connector types as interaction patterns, the ability to describe the

```

Configuration SimpleSimulation
  Component TerrainModel(map : Function)
    Port ProvideMap = [Interaction Protocol]
    Computation = [provide terrain data]
  Component = VehicleModel
    Port Environment = [Interaction Protocol]
    Computation = [compute vehicle movement]
  Connector UpdateValues(nsims : 1..)
    Role Model_1..nsims = [Interaction Protocol]
    Glue = [Data travels from one Model to another]
  Instances
    Hamilton : TerrainModel([map of Hamilton])
    Bus : VehicleModel
    C : UpdateValues(2)
  Attachments
    Hamilton.ProvideMap, Bus.Environment as C.Model
End SimpleSimulation

```

Figure 13.1: A sample system

abstract behavior of components using a CSP-like notation, the characterization of styles using predicates over system instances, and a collection of static checks to determine the consistency and completeness of an architectural specification. Because the semantics of Wright specifications are formally defined, an architecture characterized in Wright provides a sound basis for reasoning about the properties of the system or style described.

To give a view overview of Wright we now illustrate its main ideas via a simple example system. The system will simulate a bus driving through Hamilton. It will have two components, one for simulating the bus and its movements and another simulating the places through which it drives through. The two components communicate by transmitting updates of the values of objects' attributes. Figure 13.1 shows the outline of how this would look like in Wright. Of note in this description is the explicit specification of components and connectors, as well as the delineation of instances and their attachments. In our example, the terrain model `Hamilton` is accessed by the vehicle `Bus` component will interact with its environment via the `Environment` port.

A connector represents an interaction among a collection of components. For example, a pipe represents sequential communication between two filters, while a RPC connector represents one component requesting a service of another. A Wright description of a connector consists of a set of *roles* and *glue*. A connector, in our example, `C`, acts as a source of data and recipient of data, one for each model it coordinates. The connector glue defines how the roles will interact with each other.

The parts of a Wright description (port, role, computation and glue) are described using

a variant of CSP [6]. For example, the Model role of UpdateValues might be defined by:

$$\begin{aligned} \text{Role Model} &= \overline{\text{update!x}} \rightarrow \text{Model} \\ &\quad \sqcap \overline{\text{request}} \rightarrow \text{newValue?y} \rightarrow \text{Model} \\ &\quad \sqcap \S \end{aligned}$$

This defines a participant in an interaction that repeatedly either provides an updated value ($\overline{\text{update!x}}$) or request a new value ($\overline{\text{request}}$). If it requires a new value, it will be provided one (newValue?y). It may also choose to terminate successfully at any time (\S).

One immediate benefit of describing architectural designs with Wright, obtained from making the meaning of an architectural description precise, is that it facilitates the communication of ideas from the architect to the other interested parties. For example, the **Model** role defines exactly what actions a component may or may not take if it is to participate in an **UpdateValues** interaction. Furthermore it provides a basis for analyzing architectures. The description of connectors in Wright can be used to determine whether the connector satisfies certain critical properties, such as internal consistency of the protocol and whether the roles are sufficiently constrained to ensure proper behavior by participants. In considering the **UpdateValues** connector above, for example, we notice that, as it is described, if both **Bus** and **Hamilton** were to choose to request a value before providing an update, a conflict would occur. Both expect a value and there is no value available. In addition to analyzing connectors, components can be analyzed to determine, for example, whether they conform to their interface specifications.

Wright further structures the description of an architectural configuration by distinguishing between component or connector types and specific instances of them in the configuration. In the example, **UpdateValues** is a connector type: it is defined by a set of potential participants, the **Models**, and constrains how they may behave, via the **Glue**. **C** is an instance of this type: the two participants of which are **Hamilton** and **Bus**, which are associated with the protocol in the attachments.

Since the global system behavior is derived from the architectural structure and behavior descriptions of types, Wright provides a means of extending the type-level guarantees to system instances. At the configuration level, Wright provides checks to confirm that a given component port properly fulfills the obligations of any role to which is attached. If the appropriate constraints are met, then any analyses at the type level automatically apply to instances.

In addition to describing and analyzing system configurations, Wright permits the designer to describe and analyze entire families of systems, or architectural styles. by formalizing a style, the architect is able to leverage analysis across many systems and thus reduce the effort to produce individual systems.

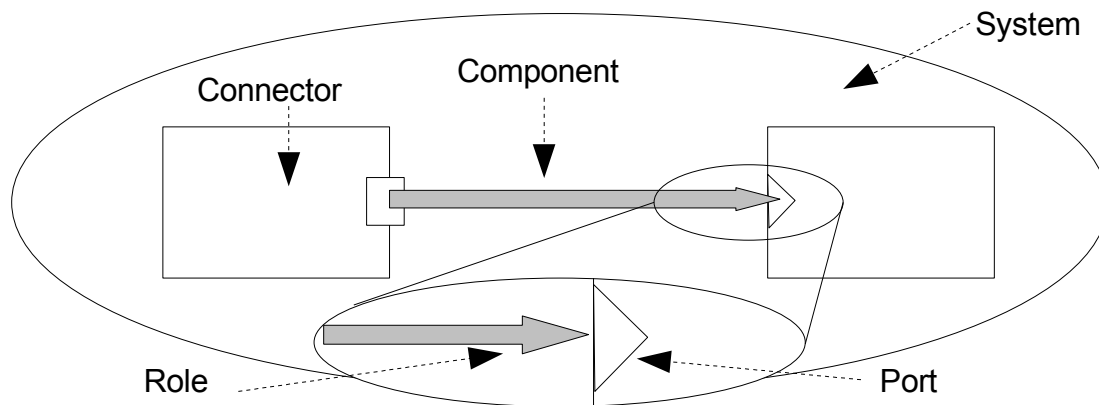


Figure 13.2: Elements of an Acme Description

Acme

Acme was originally conceived as a language to interchange architectural representations between various implementations of the different ADLs, however it has evolved into an ADL on its own. The creators of Acme call it a *second generation* ADL [5], meaning that it has been built on the experience of other ADLs, and has been built with the intention of providing the basics of ADLs with a simple syntax.

Nowadays the Acme language and toolkit provide three different capabilities:

Architectural interchange as it was its original goal, Acme provides a generic interchange format for architectural design, thus allowing architects using Acme-compatible tools a wider access to analysis and design tools.

Extensible foundation for new architecture design and analysis tools. Acme provides a solid, extensible foundation and infrastructure that allows tool builders to avoid needlessly rebuilding standard tooling infrastructure.

Architecture description by itself. Although not appropriate for all applications Acme can serve to describe relatively simple software architectures.

To illustrate the characteristics of Acme we will go through a small example: a simple architecture in which a *client* component is declared to have a single *send-request* port, and the server has a single *receive-request* port is shown in Figure 13.3. The connector has two roles designated *caller* and *callee*. The topology (configuration) of the system is defined by listing a set of *attachments* that bind component ports to connector roles, and a graphical representation can be seen in Figure 13.2.

```

System simple_cs = {
  Component client = { Port sendRequest }
  Component server = { Port receiveRequest }
  Connector rpc = { Roles {caller, callee} }
  Attachments : [
    client.sendRequest to rpc.caller ;
    server.receiveRequest to rpc.callee }
}

```

Figure 13.3: Simple Client-Server System in Acme.

Acme allows any component or connector to be represented by one or more detailed, lower level descriptions, called *representations*, in order to support hierarchical descriptions of architectures. The ability to associate multiple representations with a design element allows Acme to encode multiple views of architectural entities. Representations of a component are illustrated in Figure 13.4.

Rapide

Rapide is an ADL focusing on large-scale, distributed systems. It allows the definition and execution of models of system architectures. The result of executing a Rapide model is a set of events that occurred during the execution together with *causal* and *timing* relationships between events. These sets of events together with their causal histories form a *poset* (a partially ordered set) [14].

Rapide 1.0 is structured as a set of languages consisting of the Types, Patterns, Architecture, Constraint, and Executable Module languages; called the Rapide *language framework*.

Rapide implements an interface connection architecture model. It provides tools to express the functionality offered by an interface, the functionality required by other modules/interfaces and the connections between interfaces. Rapide also allows for expressing the requirements/constraints an interface behavior has to exhibit.

The main elements that define a Rapide interface are:

Actions represent a “one-way” message to be sent or received by the interface. They are asynchronous from the point of view of sender and receiver.

Functions represent a typed request/replay pair with synchronous interaction between the involved interfaces.

Behavior An interface behavior can be expressed in three ways, either attaching an implementation module to the interface, defining an architecture that implements the interface, or describing its behavior by means of reactive rules that specify the reaction of the interface to events offered to it.

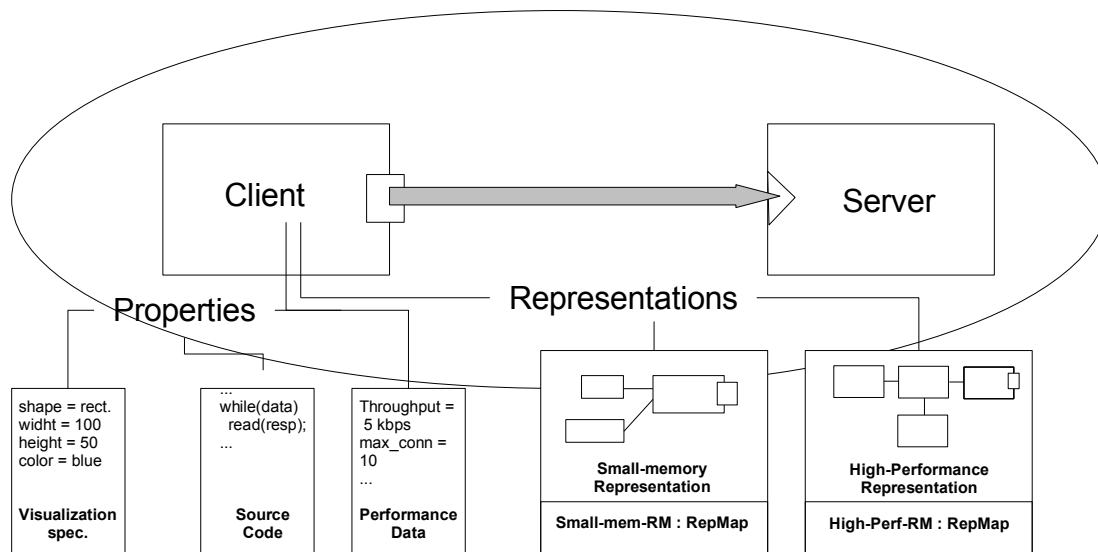


Figure 13.4: Representations and Properties of a Component

Actions and functions may be grouped in services to aid in their reusability.

Interfaces are assembled into an architecture by using connections. Connections, as is usual in ADLs are dynamic entities, Rapide allows connection behavior to be specified in terms of the relationships that the events going into a connection and the ones coming out have.

The semantics of a connection in Rapide is such that when the triggering event is present (expressed in the left hand side of the connections), the connection triggers and produces the event specified in the right hand side of the connection. Three types of connections are supported by Rapide: Basic connections ($A \text{ to } B$), pipe connections ($A \Rightarrow B$), and agent connections ($A \parallel > B$).

Basic connections are identity connections, events A and B are the same.

Pipe connections behave as a single thread control when producing B events, regardless of the concurrency behavior of the triggering A events.

Agent connections behave as if each B event is generated by a different thread of control, and thus its produced B events are not related to each other.

The main distinguishing characteristic of Rapide is its model of computation based on Partially Ordered Sets of Events (posets). Each event represents the occurrence of an activity within a program at a particular level of detail. Events are generated by communication

between two components of the system via actions and functions. Actions generate a single event, while functions generate two events; one corresponding to the function invocation and another to the function return. Events in Rapide are typed, that means that they are characterized by the number, order and type of their arguments.

Events can be ordered both by time and causality. Each of these criteria yield partial orders on the event set of a computation.

Time order is specified with respect to local clocks, since in Rapide there is no required global clock. Events are ordered with respect to the clocks that apply to it, so times can only be compared with times obtained by clocks in its scope. This fact of multiple clocks and the relationship of events to clocks imply that events that are referred to different, non related clocks are NOT ordered with respect to each other.

Causal order represents the generator/generated events. Both interfaces, via their behavior and connections may generate dependent events. Two events A and B are dependent (A precedes B) if:

1. A and B are generated by the same process or
2. A process is triggered by A and then generated B or
3. A process generated A and then assigns to a variable *v*, another process reads *v* and then generates B or
4. A triggers a connection which generates B or
5. A precedes C which precedes B (transitive closure).

By introducing this concept of order and causality, Rapide enables the programmer to explicitly visualize and analyze the execution of the system. In a more sophisticated use of this facility, a system behavior may be expressed as constraints on how events can relate to each other.

UniCon

UniCon is an architectural description language whose focus is on supporting the variety of architectural parts and styles found in the real world and on constructing systems from their architecture descriptions.

As in other ADLs, there are *components*, where data or computation are located, and *connectors*, which are used to connect components. The components export *players*, which serve as input or output points. These players connect to connector's *roles*, and thus communicate with other components. Both components and connectors have a *specification part* and an *implementation part*.

Components are specified by an *interface*, which describes three things: its computational commitments, constraints on its usage, and performance and behavior guarantees. It contains three types of information:

Component type is similar to the type of an object in an object oriented language. The type of a component captures the semantics of its behavior, the kind of functionality it implements, its performance characteristics, and its expectations of the style of interaction with other components.

Properties are attribute-value pairs that specify additional information about a component as a whole, such as assertions or constraints.

Player definitions are the way in which a component interact with other components, via connectors.

Connectors are specified by a *protocol*, which defines the kind of communication possible among a collection of of components and provides guarantees about those interactions. It contains three types of information:

Connector type expresses the designer's intentions about the general class of interactions to be mediated by the connector.

Properties are attribute-value pairs that specify additional information about a component as a whole, such as assertions or constraints, just as in components.

Role definitions give the requirements and responsibilities for the players in a connection. They are the elements to which components' players associate in a system.

Component implementations can be *primitive* or *composite*.

A primitive implementation is a pointer to a a source document external to the UniCon language that contains the implementation. For example, it could be an object file or a C language source code file.

A composite implementation is a description of other components and connectors defined with UniCon. It contains three types of information:

Pieces are the specific component and connector *instances* used to create the configuration description.

Configuration information is a description of the way in which components are hooked together to form a *configuration*.

Abstraction information is a description of how the players in the component interface are implemented by players in the component instances of the composite implementation.

13.5 Concluding Remarks

Architecture Description Languages, if used consistently, can be an useful tool in the development of large systems. By having a formal description of the architecture of the system, is possible to communicate clearly the design of the system to interested parties, as well as analyze the system before building it.

Of the four ADLs reviewed in this report, Acme seems to be the more mature one. This is probably due to the fact that it is based on older ADLs and was originally meant to be an interchange language for several different tools, thus it encompasses the common elements of other, older, ADLs. Nevertheless it was not meant to be an ADL by itself, instead aiming at becoming a basis for the development of other ADLs, so it is not as powerful as could be required for complex projects.

Wright, by allowing the detailed description of components and connectors, allows detailed analysis of components and connectors, allowing, for example, to determine if components conform to their interface specifications.

Unicon allows the construction of systems from their architectural descriptions, this may have the advantage of simplifying the mapping of the architectural model to the implementation of the system, but has the disadvantage of constraining said implementation.

Rapide focuses on modeling and simulation of the dynamic behavior described by an architecture. It also has code generation and has a strong notion of event-based communication.

ADLs are still not as well developed as, for example, programming languages or Object Oriented Design Languages (such as UML). There are several of them and they differ in several areas, such as area of application. Having a generally applicable and widely available ADL with good tools would go a long way in establishing the use of these tools in more projects.

13.6 Exam Questions

1. What is the architecture of a software system?
2. What is a difference between a Component and a Connector?
3. The architecture of a software system can be expressed both in an ADL and a programming language. What are the differences in the expression of this architecture in both cases?

Bibliography

- [1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

- [2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, second edition, 2003.
- [3] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, 1994.
- [4] D. Garlan, J. Ockerbloom, and D. Wile. EDC architecture and generation cluster (<http://www.cs.cmu.edu/~spok/adl/index.html>), 1998.
- [5] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
- [6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [7] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [8] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, 1995.
- [9] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on The Foundations of Software Engineering (FSE4)*, pages 24–32, 1996.
- [10] N. Medvidovic and D. S. Rosenblum. Assessing the suitability of a standard design method for modeling software architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 161–182, 1999.
- [11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.
- [12] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 209–218, 1998.
- [13] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. In *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.
- [14] Rapide Design Team. Draft guide to the rapide 1.0 language reference manuals.

Chapter 14

Dan Zingaro: On the Practice of B-ing Earley

This paper aims to formalize a popular technique for recognizing whether a sentence belongs to a given context-free grammar, with the B Method. To facilitate this, we begin with an abstract description of a recognizer for context-free languages. In successive refinements, we make steps which lead us from this abstract description, to Earley’s algorithm. Earley’s algorithm is certainly elegant in its operation, but it is unclear why it is correct. It is the goal of this paper to illuminate the workings of the algorithm through the refinement steps, to obtain a more clear picture of its operation. Earley’s algorithm was chosen because it runs in worst-case cubic time. While faster algorithms exist, they require complex Boolean matrix multiplication methods and so $O(n^3)$ appears to be the best practical bound [6].

14.1 Languages and Recognizers

Familiarity with standard Formal Language Theory concepts is assumed—the following are only the details most relevant to the current work, and mirror those used by Earley [3]. A grammar G is a quadruple: (T, N, P, S) . T is the set of terminal symbols, N is the set of nonterminal symbols, P is the set of productions, and S is a nonterminal designated as the “start symbol”. T and N are assumed to be disjoint. Productions in P are (A, B) pairs, where A and B are made entirely of terminals or nonterminals, and where the length of A is at least 1. In context-free grammars, the length of A is exactly 1, and it is this subclass of grammars which is handled by Earley’s algorithm.

We say that sequence χ is directly derivable from π , written $\pi \Rightarrow \chi$, if we can substitute the symbols corresponding to the left side of some production P in π , with the corresponding right-side of production P , and arrive at χ . That is, if $\pi = \mu\sigma\nu$, and $\chi = \mu\tau\nu$, $\sigma \rightarrow \tau$ is a production, and σ, τ, μ, ν are all sequences, then $\pi \Rightarrow \chi$.

Of course, knowing if two sequences are “directly derivable” is not entirely interesting on its own. However, a related problem, determining if one sequence is *derivable* in zero or more steps from another, is the very basis for what it means to act as a *recognizer*.

For χ to be derivable from π , then, we require sequences $\alpha_0, \alpha_1, \dots, \alpha_k (k > 0)$, such that $\pi = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_k$. It is thus evident that *Directly Derivable*, as defined above, is transitive, and *Derivable* is in fact the transitive closure of *Directly Derivable*. It is these elegant mathematical descriptions which must be used when reasoning with the B method.

A *Sentential Form* is a string which is derivable from the start (or root) symbol of the grammar. Since nonterminals are only present to properly construct a grammar, we are usually interested in sentential forms which are composed of only terminal symbols. These are called *sentences*, and so the definition of a recognizer can also be stated as determining whether or not a given sentence belongs to the grammar, or if the sentence is derivable from the root nonterminal of the grammar.

14.2 General Context-Free Recognizer Machine

We begin with a context-free recognizer based on the definitions of *directly derivable* and *derivable* given above. Recall that modelling with the B method begins with an abstract machine, which contains a specification of a software system, or a part thereof [7]. We use the syntax of B4free [2]. Being the first step in the development, our recognizer will act as this abstract machine, which will undergo refinements in order to arrive at Earley's recognizer.

It is first necessary to outline the data types, such as rules and grammars, that will be used by this machine and its refinements. The most straightforward way to represent rules appears to be via a double, where the first component is a token, and the second component is a sequence of tokens. We can represent a grammar by a 4-tuple, whose components are the same as described above. However, not all such 4-tuples are acceptable. First, it is necessary that the terminal and nonterminal sets be disjoint. Second, since we are only dealing with context-free grammars, the rules in the grammar must obey the required format of having a left-hand side consisting of only one nonterminal; this is handled in line 11. The third restriction on grammars, imposed on line 12, requires that the start symbol be the left-hand side of just one production. This is advocated by Earley in his original paper [3], because it simplifies the execution of the algorithm. The proof that all grammars can be transformed into grammars where the start symbol is involved in only one production is obvious. Let an arbitrary grammar have a start symbol of t , which is necessarily the left-hand side of more than one production. Simply introduce a fresh non-terminal s , make this the start symbol, and add the production $s \rightarrow t$.

Several other useful operations on grammars are also added at this point. Since the *derivable* relations require knowledge of whether or not a certain production is in the grammar, a function *gProd* is added in lines 14-15, which returns the productions component of the grammar tuple. Functions for returning the other components of the grammar are also added for symmetry (and because they will probably be used later). Finally, for simplicity, we add a function *gElements* for returning the union of the grammar's terminals and nonterminals.

We can now define *directlyDerivable* starting at line 25, a function from grammars to relations. The relation returned by the function is between elements of the supplied grammar, which makes use of the *gElements* function just defined. Recall from the definition of *directly*

derivable that $\sigma \rightarrow \tau$ must be a member of the grammar's productions. This restricts σ to be one nonterminal, and so it is defined as a token, rather than a sequence of tokens, in line 28.

The function *derivable* can then be easily defined, starting at line 32. It is again a function from grammars to relations, where elements xx and yy are in the relation if they are in the closure of *directlyDerivable*(gg).

The final step is to actually provide a means whereby a user can determine if a sentence is in a given grammar. To facilitate this, an operation *isSentence* is defined, taking a grammar and a sentence, and returning *TRUE* or *FALSE*, depending on whether or not the root production of the grammar can derive the supplied sentence. This completes the description of the machine, which is next given in its entirety.

Listing 14.1: Context-free Recognizer

```

1 MACHINE recm
2 SETS TOKENS
3 CONSTANTS grammars, rules, gProd, gElements, gNT,
4   gTT, gRoot, directlyDerivable, derivable, seqTokens
5
6 PROPERTIES
7 seqTokens = seq (TOKENS)  $\wedge$ 
8 rules = TOKENS  $\times$  seqTokens  $\wedge$ 
9 grammars = {gg | ( $\exists p, n, t, s$ ).
10   ( $p \subseteq rules \wedge n \subseteq TOKENS \wedge t \subseteq TOKENS \wedge s \in TOKENS \wedge n \cap t = \{\}$ )  $\wedge$ 
11   ( $\forall a, b \bullet a \in TOKENS \wedge b \in seq(TOKENS) \wedge (a, b) \in p \implies a \in n$ )  $\wedge$ 
12   card(rules{s}) = 1  $\wedge$ 
13   gg = (p, n, t, s) }  $\wedge$ 
14 gProd = ( $\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge$ 
15    $t \subseteq TOKENS \wedge s \in TOKENS \wedge (p, n, t, s) \in grammars \mid p$ )  $\wedge$ 
16 gNT = ( $\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge$ 
17    $t \subseteq TOKENS \wedge s \in TOKENS \wedge (p, n, t, s) \in grammars \mid n$ )  $\wedge$ 
18 gTT = ( $\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge$ 
19    $t \subseteq TOKENS \wedge s \in TOKENS \wedge (p, n, t, s) \in grammars \mid t$ )  $\wedge$ 
20 gRoot = ( $\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge$ 
21    $t \subseteq TOKENS \wedge s \in TOKENS \wedge (p, n, t, s) \in grammars \mid s$ )  $\wedge$ 
22 gElements = ( $\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge$ 
23    $t \subseteq TOKENS \wedge s \in TOKENS \wedge (p, n, t, s) \in grammars \mid n \cup t$ )  $\wedge$ 
24
25 directlyDerivable = ( $\lambda gg \bullet gg \in grammars \mid$ 
26   {xx, yy | xx  $\in seq(gElements(gg)) \wedge yy \in seq(gElements(gg)) \wedge$ 
27   ( $\exists mu, sigma, nu, tau$ ).
28   ( $mu \in seq(gElements(gg)) \wedge sigma \in gElements(gg) \wedge$ 
29    $nu \in seq(gElements(gg)) \wedge tau \in seq(gElements(gg)) \wedge$ 
```

```

30    $xx = mu \wedge (\sigma \rightarrow nu) \wedge yy = mu \wedge \tau \wedge nu \wedge \sigma \mapsto \tau \in gProd(gg)) \wedge$ 
31
32    $derivable = (\lambda gg \bullet gg \in grammars \mid$ 
33      $\{xx, yy \mid xx \mapsto yy \in closure(directlyDerivable(gg))\})$ 
34
35   OPERATIONS
36    $ans \leftarrow isSentence(gg, sentence) =$ 
37   PRE  $gg \in grammars \wedge sentence \in seq(gElements(gg))$  THEN
38      $ans := bool([gRoot(gg)] \mapsto sentence \in derivable(gg))$ 
39   END
40 END

```

14.3 State Sets

There are several ways to describe the operation of Earley's algorithm. Earley himself [3] gives two such methods: one is informal, using "dot notation", and the other is more formal, using loops, arrays, and linked lists. Jones [5] gives an enlightening account of Earley's algorithm in a more mathematical framework, and his ideas are the basis for much of what follows in the current work. Below, we describe the main ideas of the algorithm, as they relate to the concept of the state set. By using state sets, we can arrive at another, more concrete, representation of a recognizer; the only thing we will not have is a systematic way to generate these state sets. This is returned to in the next refinement.

The algorithm works conceptually in $n + 1$ phases, where n is the length of the input string. These phases are all associated with their own set of data, referred to as a *state set*. These state sets are composed of so-called Earley items, which, taken in unison, maintain the entire state of the parser (that is, what has been parsed so far and what can potentially be parsed next).

Earley items are always described as being tuples. But, while Earley recommends 4-tuples, most of his epigones (for instance, see [1]) favor eliminating the fourth component—a lookahead string—and using only triples; we shall eliminate the lookahead as well.

The components of the tuples are typically integers, and require that the productions in the grammar be ordered so that they can be referred to by unique integers. Jones [5] instead defines the triples to consist of a rule and two integers, so that no order is required on the productions. Following Earley's notation, production D_p is of the form $C_{p_1}, C_{p_2}, \dots, C_{p_{p'}}$. If we designate triples as (p, j, f) , then p is a rule of the grammar, $0 \leq j \leq p'$ (j represents the token in the grammar that was just parsed), and $1 \leq f \leq n + 1$ (f is the state in which parsing this production began). A *final* item is one which has been completely parsed. In such a case, $j = p'$, since the last token parsed was the last token in the production.

The algorithm begins with the item $(RootRule, 0, 1)$ in state set 1. If set i is not related via any items to set $i + 1$, then the input is not accepted by the algorithm. Alternatively, if, at the end of the algorithm, s_{n+1} is the set consisting of just the triple $(RootRule, 1, 1)$, the

input is accepted. The reason why this criterion is synonymous with acceptance is outlined next.

14.4 Refinement 1 – Intuition Behind State Sets

It is certainly not obvious how using state sets can facilitate the recognition of a sentence in a grammar. However, after elaborating on a point made by Jones [5], it is more clear. Consider an item (r, j, f) belonging to state set i . Recall that the f component represents the state set where the production r began to be parsed. Keeping in mind that a state set corresponds with the current position in the input string, f represents the fact that the first $f - 1$ characters of the input string, followed by the nonterminal on the left-hand side of rule r , can be generated.

Next, we turn to the j component, which says that we have parsed j tokens of the right-hand side of rule r . Since we are in state set i , and only $f - 1$ terminals in the input string have been recognized, it must be the case that the first j tokens can generate the string consisting of the terminals from f to $i - 1$ in the input sentence. We will see that these conditions are represented in the abstract refinement given in the next section.

Now it makes sense why $\langle \text{RootRule}, 1, 1 \rangle$ in state set $n + 1$ means that the sentence has been successfully parsed. Since f is 1, and this equals the length of the root production, it says exactly that the right side of the root production can generate the input sentence.

14.5 Refinement 1 – Refinement Machine

We are now at a point to refine our original context-free recognizer machine. It is the first refinement to rely on state sets, so numerous accessory functions must be introduced. The conditions for constructing state sets are also complicated, and are split into their own functions. This is all described presently.

It will be necessary to refer to the left-side (nonterminal) and right-side (sequence of tokens) of rules, and so two functions (lines 7 and 8) allow these components to be easily extracted from the composite rule. States are then defined as triples, consisting of one rule and two natural numbers. A state set is then conveniently defined as the power set of states in line 10.

We can then define *stateObj* (line 11), which will be the “object” containing all state sets; it is therefore represented as a sequence. It will prove convenient to refer to individual state sets in this object, and so a function *extractSet* is defined in line 12. It takes a state object, and a NAT, and returns the associated state set.

The root (or start) rule plays a prominent role in this refinement, and so to avoid a messy syntax for referring to that rule, the function *gRootRule* is introduced (line 13). It takes a grammar as input, and returns a double representing the root production. It relies on previously defined functions *gProd* and *gRoot* to do this.

Let us ignore the *cond1*, *cond2*, *cond3* functions for the moment, and continue with *validObj* in line 29. It takes a grammar and an input sentence, and returns valid state objects for the pair. By *valid*, I mean that the state sets have length $n + 1$, and satisfy certain conditions that make it a consistent set.

The first necessary condition is that some base element exists in the first state set. We impose this by ensuring that state set 1 of the state object, contains the rule whose first component is the root rule, whose second component is 0, and whose third component is 1 (see line 31).

Next, the consistency constraints must be imposed, and this is done with two universal quantifications, simplified by shuffling the bulk of the code into separate conditions. First, we check, in line 34, to make sure that, if a state (r, j, f) is present, then r is really a rule of the grammar, and that f is between 1 and i . The reason for the latter is because, if $f > i$, then the state is saying that more than $i - 1$ terminals of the sentence can be recognized by having only looked at $i - 1$ such terminals, which is impossible¹. It is also necessary that j be between 1 and the length of the right side of rule r .

Recall the stipulation on state sets that the first $f - 1$ characters of the input sentence, followed by the left-side of r , can be generated by the grammar. This is housed by *cond1*, defined in line 16, and imposed in line 35. *validObj* thus ensures that it is true for all state sets.

If $j > 0$, then we have another condition to verify; namely, that the first j tokens on the right side of r can generate the sequence $x_f, x_{f+1}, \dots, x_{i-1}$, where x is the input string. This is handled in line 20, where operations to extract pieces of sequences are used to construct the *derivable* condition. *cond2* is imposed in line 36.

There is one more necessary condition on state sets, relating a state set i to future state sets $m > i$. As explained by Jones [5], if (r, j, f) is in state set i , and the $j + 1$ th element on the right-side of r can produce x_i, \dots, x_{m-1} , then $(r, j + 1, f)$ must exist in state set m . This follows directly from the definition of state sets, and without it, state set m would not contain all required elements. The condition is verified by forcing the condition at line 24 to be true via line 39.

We can then define *acceptObj*, to consist of the state-set objects which, given a grammar and sentence, represent the situation where the sentence is accepted by the grammar. Accepting state objects must of course be valid, and exhibit the additional property that they contain the required state in the final state set.

The refinement of the *isSentence* operation returns *TRUE* if there is at least one such valid state object, and *FALSE* otherwise. The reason why this represents a valid refinement will be returned to in the next section. For now, the complete refinement is presented.

Listing 14.2: State Set Refinement

```

1 REFINEMENT recr
2 REFINES recm
3 CONSTANTS states, stateSet, stateObj, extractSet, validObj, gRootRule,

```

¹Earley may have developed a landmark parser, but he was certainly no prognosticator

4 $lsRule, rsRule, cond1, cond2, cond3, acceptObj$

5

6 **PROPERTIES**

7 $lsRule = (\lambda ls, rs \bullet ls \in TOKENS \wedge rs \in seqTokens \mid ls) \wedge$

8 $rsRule = (\lambda ls, rs \bullet ls \in TOKENS \wedge rs \in seqTokens \mid rs) \wedge$

9 $states = rules \times \mathbb{N} \times \mathbb{N} \wedge$

10 $stateSet = \mathbb{P}(states) \wedge$

11 $stateObj = seq(stateSet) \wedge$

12 $extractSet = (\lambda ss, nn \bullet ss \in stateObj \wedge nn \in \mathbb{N} \wedge nn \geq 1 \wedge nn \leq size(ss) \mid ss(nn)) \wedge$

13 $gRootRule = (\lambda p, n, t, s \bullet p \subseteq rules \wedge n \subseteq TOKENS \wedge t \subseteq TOKENS \wedge$

14 $s \in TOKENS \wedge (p, n, t, s) \in grammars \mid s \mapsto p(s)) \wedge$

15

16 $cond1 = (\lambda gg, ss, ff, rr \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \wedge ff \in \mathbb{N} \wedge rr \in$
 $rules \mid$

17 $bool((\exists alpha \bullet alpha \in seq(gElements(gg)) \wedge$

18 $[gRoot(gg)] \mapsto (((ss \uparrow ff-1) \leftarrow lsRule(rr) \frown alpha) \in derivable(gg)))) \wedge$

19

20 $cond2 = (\lambda gg, ss, ff, rr, jj, ii \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \wedge ff \in$
 $\mathbb{N} \wedge$

21 $rr \in rules \wedge jj \in \mathbb{N} \wedge ii \in \mathbb{N} \mid$

22 $bool(rsRule(rr) \uparrow jj \mapsto ((ss \uparrow ii-1) \downarrow ff-1) \in derivable(gg))) \wedge$

23

24 $cond3 = (\lambda gg, qq, ss, rr, jj, ff, ii \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \wedge$

25 $rr \in rules \wedge jj \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge qq \in stateObj \wedge ii \in \mathbb{N} \mid$

26 $bool((\forall m \bullet m \in \mathbb{N} \wedge ii+1 \leq m \wedge m \leq size(ss) \wedge$

27 $[rsRule(rr)(jj+1)] \mapsto ((ss \downarrow m-1) \uparrow ii-1) \in derivable(gg) \implies (rr, jj+1, ff) \in extractSet(qq, m)))$

28

29 $validObj = (\lambda gg, ss \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \mid$

30 $\{qq \mid qq \in stateObj \wedge size(qq) = size(ss) + 1 \wedge$

31 $(gRootRule(gg), 0, 1) \in extractSet(qq, 1) \wedge$

32 $(\forall i \bullet i \in 1..size(qq) \implies$

33 $(\forall r, j, f \bullet r \in rules \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge ((r, j, f) \in states \wedge (r, j, f) \in extractSet(qq, i)) \implies$

34 $(r \in gProd(gg) \wedge 1 \leq f \wedge f \leq i \wedge j \leq size(rsRule(r)) \wedge$

35 $cond1(gg, ss, f, r) = TRUE \wedge$

36 $j > 0 \implies (cond2(gg, ss, f, r, j, i) = TRUE)))) \wedge$

37 $(\forall i \bullet i \in 1..(size(qq)-1) \implies$

38 $(\forall r, j, f \bullet r \in rules \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge ((r, j, f) \in states \wedge (r, j, f) \in extractSet(qq, i)) \implies$

39 $(cond3(gg, qq, ss, r, j, f, i) = TRUE)))) \wedge$

40

41 $acceptObj = (\lambda gg, ss \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \mid$

42 $\{xx \mid xx \in validObj(gg, ss) \wedge (gRootRule(gg), size(rsRule(gRootRule(gg))), 1) \in xx(size(ss) +$

43

44 **INVARIANT**

45 $(\forall gg, ss \bullet gg \in grammars \wedge ss \in seq(gElements(gg)) \implies$
 46 $(card(acceptObj(gg, ss)) \geq 1 \iff ([gRoot(gg)] \mapsto ss \in derivable(gg))))$

47
48 **OPERATIONS**

49 $ans \longleftarrow isSentence(gg, sentence) =$

50 **PRE** $gg \in grammars \wedge sentence \in seq(gElements(gg))$ **THEN**

51 $ans := bool(card(acceptObj(gg, sentence)) \geq 1)$

52 **END**

53 **END**

14.6 Refinement 1 – Linking Invariant

To ensure that the relationship between an abstract machine and its refinements is valid, we require the presence of a linking invariant, relating aspects of the previous and current refinement. The refinement machine just presented includes a refinement invariant which must still be machine-proven, but for which it is not difficult to understand the intuition. If there is at least one element in *acceptObj*, then there must be at least one collection of state sets which represents the case where the sentence belongs to the grammar. This corresponds exactly to saying that the root can derive the sentence in zero or more steps, according to the definition of *derivable* given in the abstract machine. Conversely, consider all cases where the root can derive some sentence. This means that the elements of the right-hand side of the root production, on their own, can derive the entirety of the sentence, and this corresponds exactly to those state objects which accept their input. So there must be at least one member of *acceptObj* in this case, completing the argument for the implication.

14.7 Refinement 2 – Earley’s Algorithm Revisited

The next refinement involves formalizing Earley’s algorithm using the B method. Jones [5] provides many of the ideas used—the main complication arises in trying to model his closure relations, described in due time.

Earley’s algorithm seeks to systematically build the state sets from the previous refinement. It involves the repeated application of three functions: predictor, scanner, and completer. We review them for completeness, and assume that they are operating on a state set (r, j, f) in state set i .

- Predictor: adds, to state set i , items $(q, 0, i)$, where $D_q = C_r(j + 1)$
- Completer: if operating on a final state, then for all items of the form (q, l, g) , found in state set f , if $C_q(l + 1) = D_r$, then add $(q, l + 1, g)$ to state set i
- Scanner: If $C_r(j + 1) = x_{i+1}$, add $(r, j + 1, f)$ to state set $i + 1$

State set 1 is created by inserting the rule $(rootRule, 0, 1)$, and then applying the closure of the predictor to it. If we have a state set i , then set $i + 1$ can be constructed partially by applying the scanner to set i , and then repeatedly applying the predictor and completer to set $i + 1$ until no new items are added. Jones makes an optimization on the last state set, in that it is unnecessary to run the predictor on it; we drop this optimization in favor of simplicity.

14.8 Refinement 2 – Refinement Machine

We can now present the refinement machine for Earley’s algorithm. We begin with the predictor. Jones defined the predictor as a function which takes, among other things, one state set, and returns a set of state sets. It was more straightforward to change this so that the predictor is a function, which returns a relation between state sets. One would then use the relational image of the returned relation, to obtain the items that the predictor would relate to a set of items (the cardinality of this “set” may of course be 1, and it is when operating on the base element).

In my predictor function (and other functions below), double-letter identifiers (like jj) represent components of the new state, and single letter identifiers (like j) represent corresponding components of the old state. It should then be easy to see how the predictor, starting on line 6, works. Firstly, it adds only states where $jj = 0$ (nothing has been parsed), and $ff = ii$ (they are created from the current state set). These states can only result from non-final states in the old state set, and must also have the property that their left-hand side equals the $j + 1st$ token on the right; this last part is covered by line 11.

We then immediately use the relational image in the *predictOn* function starting at line 13, to obtain the results of predicting on a given item. Since this must be done “recursively”, we first take the closure of the prediction, and then restrict ourselves to the subset of interest.

We similarly define the scanner and completer operations according to their definitions. Scanner additionally takes the input sentence as a parameter, since it must match the next input character against the next symbol in Earley items. Completer, in contrast, takes the state-set object as a parameter, since it must look back to previous state sets. We then add a *completeOn* function, which works like *predictOn*, applying the closure of the completer to a specific state. The scanner has no such closure function: applying the scanner to states only adds states to the next state set, and so taking the closure would prove futile.

Throughout the algorithm, after applying the scanner to a state set, we want to repeatedly call the completer and predictor, until nothing new is added. In other words, we are looking for the closure of the union of the predictor and completer, and this is what we get with the *closePc* function, defined at line 34. Note that this differs from Earley’s original description of the algorithm, in which he stated that items should be scanned linearly, and the appropriate function should be applied to the current item to add more items to the end of the list. As explained in [4], this order is too restrictive, and it is conceptually simpler if the closure of the predictor and completer are taken first, and then the scanner is applied—in other words, it is safe to save the scanning until the end.

It is then possible to proceed as we did with the last refinement. That is, we define a *valid* object and an *accepting* object. This time, valid objects consist of those where the first state set consists of the objects from the closure of the predictor, and the remainder of the state sets are equal to the result of the scanner applied to the previous set, and the closure of the completer and predictor on the current set. The refinement machine is presented below.

Listing 14.3: Earley Refinement

```

1 REFINEMENT recrr
2 REFINES recr
3 CONSTANTS predictor, predictOn, closePC, scanner, completer, completeOn, validEarley, acceptEarley
4
5 PROPERTIES
6 predictor = ( $\lambda ii, ss, gg \bullet ii \in \mathbb{N} \wedge ss \in seqTokens \wedge gg \in grammars \mid$ 
7  $\{pp, qq \mid pp \in stateSet \wedge qq \in stateSet \wedge$ 
8  $qq = \{zz \mid zz \in states \wedge$ 
9  $(\exists rr, jj, ff \bullet rr \in rules \wedge rr \in gProd(gg) \wedge jj = 1 \wedge ff = ii \wedge zz = (rr, jj, ff) \wedge$ 
10  $(\exists r, j, f \bullet r \in rules \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge (r, j, f) \in pp \wedge j \neq size(rsRule(r))$ 
11  $\wedge lsRule(rr) = rsRule(r)(j+1))\}\}) \wedge$ 
12
13 predictOn = ( $\lambda ii, ss, gg, rjf \bullet ii \in \mathbb{N} \wedge ss \in seqTokens \wedge gg \in grammars \wedge rjf \in stateSet \mid$ 
14  $closure(predictor(ii, ss, gg)) [\{rjf\}] \wedge$ 
15
16 scanner = ( $\lambda ii, ss, gg \bullet ii \in \mathbb{N} \wedge ss \in seqTokens \wedge gg \in grammars \mid$ 
17  $\{pp, qq \mid pp \in stateSet \wedge qq \in stateSet \wedge$ 
18  $qq = \{zz \mid zz \in states \wedge$ 
19  $(\exists rr, jj, ff \bullet rr \in rules \wedge rr \in gProd(gg) \wedge jj \in \mathbb{N} \wedge ff \in \mathbb{N} \wedge zz = (rr, jj, ff)$ 
20  $\wedge (\exists r, j, f \bullet r \in rules \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge (r, j, f) \in pp \wedge r = rr \wedge f = ff \wedge$ 
21  $jj = j + 1 \wedge j \neq size(rsRule(r)) \wedge rsRule(r)(j+1) \in gTT(gg) \wedge ss(j+1) = rsRule(r)(j+1))\}\}) \wedge$ 
22
23 completer = ( $\lambda stObj, ii, ss, gg \bullet stObj \in stateObj \wedge ii \in \mathbb{N} \wedge ss \in seqTokens \wedge gg \in grammars \mid$ 
24  $\{pp, qq \mid pp \in stateSet \wedge qq \in stateSet \wedge qq = \{zz \mid zz \in states \wedge$ 
25  $(\exists rr, jj, ff \bullet rr \in rules \wedge rr \in gProd(gg) \wedge jj \in \mathbb{N} \wedge jj \neq size(rsRule(rr)) \wedge$ 
26  $ff \in \mathbb{N} \wedge zz = (rr, jj + 1, ff)$ 
27  $\wedge (\exists r, j, f \bullet r \in rules \wedge j \in \mathbb{N} \wedge f \in \mathbb{N} \wedge j = size(rsRule(r)) \wedge$ 
28  $(rr, jj, ff) \in extractSet(stObj, f) \wedge$ 
29  $rsRule(rr)(jj+1) = lsRule(r) \wedge (r, j, f) \in pp)\}\}) \wedge$ 
30
31 completeOn = ( $\lambda stObj, ii, ss, gg, rjf \bullet stObj \in stateObj \wedge ii \in \mathbb{N} \wedge$ 
32  $ss \in seqTokens \wedge gg \in grammars \wedge rjf \in states \mid closure(completer(stObj, ii, ss, gg)) [\{\{rjf\}\}] \wedge$ 
33
34 closePC = ( $\lambda stObj, ii, ss, gg, rjf \bullet stObj \in stateObj \wedge ii \in \mathbb{N} \wedge ss \in seqTokens \wedge gg \in$ 
35  $grammars \wedge$ 

```

```

35  $rjf \in stateSet \mid closure (predictor (ii, ss, gg) \cup completer (stObj, ii, ss, gg)) [\{ rjf \}]) \wedge$ 
36
37  $validEarley = (\lambda gg, ss \bullet gg \in grammars \wedge ss \in seq (gElements (gg)) \mid$ 
38    $\{qq \mid qq \in stateObj \wedge size (qq) = size (ss) + 1 \wedge$ 
39    $\{extractSet (qq, 1)\} = predictOn (1, ss, gg, \{gRootRule (gg)\} \times \{0\} \times \{1\}) \wedge$ 
40    $(\forall ii \bullet ii \in 2..size (qq) \implies$ 
41    $(\exists tempSet \bullet tempSet \in stateSet \wedge (\{tempSet\} = scanner (ii, ss, gg) [\{extractSet (qq, ii-1)\}]) \wedge$ 
42    $\{extractSet (qq, ii)\} = closePC (qq, ii, ss, gg, tempSet))) \wedge$ 
43
44  $acceptEarley = (\lambda gg, ss \bullet gg \in grammars \wedge ss \in seq (gElements (gg)) \mid$ 
45    $\{xx \mid xx \in validEarley (gg, ss) \wedge (gRootRule (gg), size (rsRule (gRootRule (gg))), 1) \in$ 
46    $xx(size (ss) + 1)\}$ 
47 INVARIANT
48  $(\forall gg, ss \bullet gg \in grammars \wedge ss \in seq (gElements (gg)) \implies$ 
49    $(card(acceptEarley (gg, ss)) \geq 1 \iff (card (acceptObj (gg, ss)) \geq 1)))$ 
50
51 OPERATIONS
52  $ans \leftarrow isSentence (gg, sentence) =$ 
53   PRE  $gg \in grammars \wedge sentence \in seq (gElements (gg))$  THEN
54      $ans := bool (card (acceptEarley (gg, sentence)) \geq 1)$ 
55   END
56 END

```

14.9 Refinement 2 – Linking Invariant

The given refinement contains a linking invariant similar to the last one, this time relating accepting Earley sets to accepting sets from the previous refinement. By transitivity (if this was proven to be correct), accepting Earley sets should be equivalent to a sentence being derivable from the root production. To verify this requires a proof that Earley's algorithm does indeed construct state sets according to the claim in the previous refinement.

14.10 Next Steps

It is still necessary to machine-prove my refinements, which will (hopefully) provide proof that Jones' [5] ideas, and my own, are correct. It is then necessary to perform the last step of the B refinement process, to arrive at an implementation of Earley's algorithm.

14.11 Exam Questions

- In the complete proof of correctness, why would it be only necessary to show that Earley's recognizer computes state sets correctly, and *not* to prove that it can recognize sentences belonging to any context-free grammar?
- Typical implementations of Earley's algorithm use lists of Earley items belonging to elements of an array of state sets. The lists are scanned linearly, and new Earley items are produced at the end, if they did not already exist in the list. How could this situation be represented in the B method (I.E. which data structures, control flow, etc.)?
- Explain the purpose of the *predictOn* function. Why can't *predict* be called directly?

Bibliography

- [1] John Aycock and Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45:620–630, 2002.
- [2] ClearSy. <http://www.b4free.com>.
- [3] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13:94–102, 1970.
- [4] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. On line context free language recognition in less than cubic time(extended abstract). In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 112–120, 1976.
- [5] Cliff Jones. Formal development of correct algorithms: an example based on earley's recogniser. *SIGPLAN Notices*, 7:150–169, 1972.
- [6] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.
- [7] Steve Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave, 2001.

Chapter 15

Marwan Abdeen: Use Cases, Scenarios, Sequence Diagrams and Message Sequence Charts

Within the software life cycle, increasing attention has been paid to the stages of specification and design since the quality of all following stages essentially depend on them. Hence, this survey paper starts with the discussion of four significant design notations, describes their evolution and concludes with a conceptual similarities, differences and description.

15.1 Use Cases

Starting with the Unified Modeling Language(UML) 1.1 in 1997, Use Cases, as behavioural diagrams describing the system functionality went with the evolution of UML. Although describing the system functionality from a high-level view continued to be the main objective of Use Cases, some of the representations of key Use Case elements have changed. The change in the stereotypes of the main two relationships in Use Case; <<include>> and <<extend>> described in further detail next, can be considered as the main significant evolution of Use Case during the last decade.

Use Case Definition: A Use Case organizes a cohesive set of requirements around a single (named) system capability but does not imply anything about internal implementation. It represents functionality or services provided by a system to its users.

A Use Case is considered as a description of the possible sequences of interactions between the system and its external actors, related to a particular goal. The Use Case collects together all the Scenarios related to that goal of that primary actor, including both those in which the goal is achieved, and those in which the goal must be abandoned.

A Use Case model [15] describes:

- the system to be constructed,

- the actors (described in more details later) - representing a role played by a person or other entity which interacts with the system.
- the Use Cases - Families of usage Scenarios of the application, grouped into coherent cases of functionality. Hence, it is a generalization of the Scenarios of use of the system. In a more technical description, a Scenario is an instance of a Use Case.

The complete set of Use Cases specifies all possible ways in which the system can be used, without revealing how this is to be implemented by the system. Since Use Cases do not deal with technicalities inside the system but focus on how the system is viewed from the outside, they are most useful in discussions with end-users to make sure that there is an agreement on the requirements on the system, on its delimitation etc [8].

Actors: An actor can be defined as any object outside the scope of the system that interacts with the system. It represents a user of the system; Actors represent roles. One person could have the role of various actors. An actor could be anything having behaviour, it might be a person, a company or an organization, a computer program or a computer system, hardware or software or both. It doesn't have to be a human in specific, other systems or even partner applications could be considered as actors as well.

The word "Role" is more correct to describe the actor as the word "Actor" seems to imply a particular single entity, when it really means a group of entities. Ideally one would speak of "Actor Type" or "Role", to indicate a category of similar entities. However, "Actor" is the word the industry has accepted, and it works quite adequately.

Relationships: The main two major relationships used in Use Cases are <<include>> and <<extend>>. In UML 1.1 and as a response to the claim that 80% of the software developers did not understand the difference between the <<uses>> and <<extends>> relationships, researchers started attempting to explain the meaning of these relationships but unfortunately, statements show to what extent even the experts disagree about Use Cases.

In UML 1.3 and later, <<include>> and <<extend>> are the main two relationships which have quietly replaced old relationships called <<uses>> and <<extends>> after adoption by the Object Management Group(OMG) [1]. In UML 2.0, the most immediately noticeable change is that the new relationships are defined as stereotypes of dependency instead of stereotypes of generalization. This is also indicated in the style of arrow used. Where UML 1.1 adopted the generalization arrow for use-case relationships, nowadays UML 2.0 employs the dependency arrow. A dependency is defined abstractly as: any relationship between a dependent entity and a master entity, such that changes to the master would have consequential effects on the dependant [10].

There are also other kinds of relationships that are applicable to Use Cases, such as Associations and Constraints.

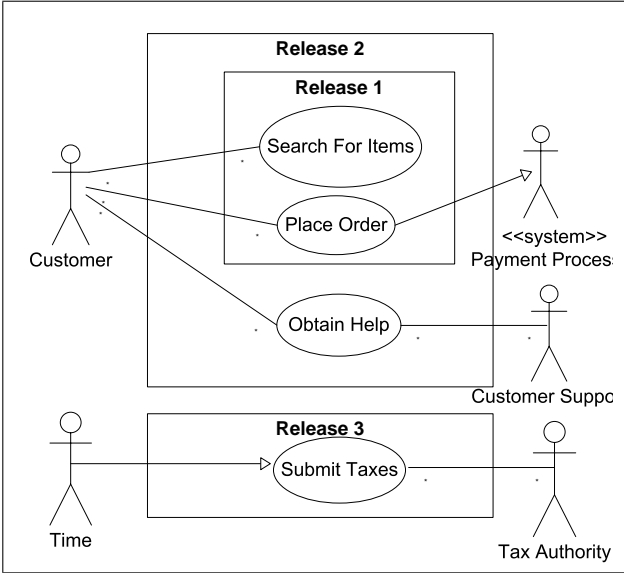


Figure 15.1: Online Shopping [16]

Description: Figure 1.1 demonstrates the idea of system actors, as they represent entities other than humans. Whereas, Figure 1.2 demonstrates the inheritance between two actors, it shows the way <<extend>> relationship is used when a Use Case may be invoked across several Use Case steps, also, the way <<include>> is used when invocation of a Use Case is accurately known. A generalization of a Use Case takes place when a single condition results significantly in new business logic, (Generalization is read as 'Is Like') . The line connecting the actors to the Use Case means that, that actor interacts in some interesting way with the system as the system executes that Use Case. The directed line indicates the information flow; single direction from the actor to the Use Case. Inheritance between actors is possible in a Use Case as described; An international student is a student.

One can tell and from the above figures that Use Cases capture the functional requirements by describing the systems behaviour as a black border box and the reaction to the stimulus environment. In more details, a Use Case defines the systems environment and how the system reacts to input. Use Cases are suited to describing the behaviour of any system which purposefully interacts with its external environment, and can be written at any time. The in common property which Use Case has is that they show how the system responds to the outside world, the responsibilities and behaviour of the system, without revealing how the internal parts are constructed.

At the same time, we want to write as little as possible, as clearly as possible, showing the ways in which the system reacts to various situations. Use Cases can have additional textual descriptions. A complete textual description of a Use Case includes:

1. Use Case Name

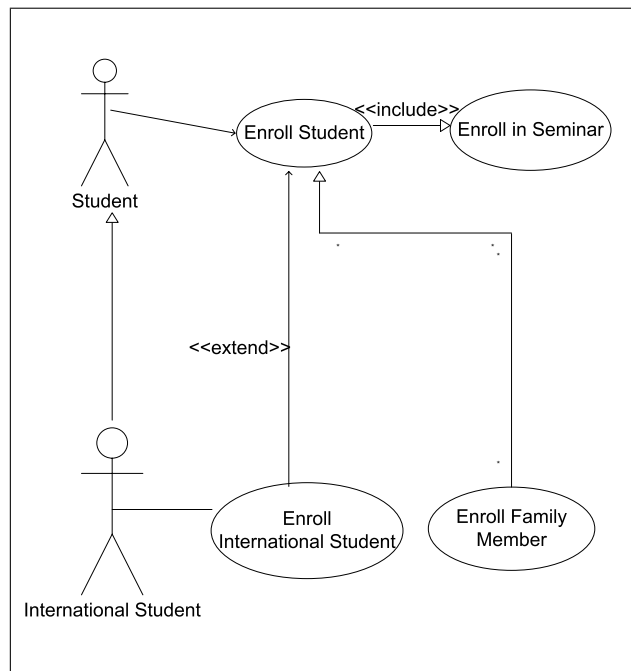


Figure 15.2: University Student Enrollment [16]

2. Summary
3. Start of the Use Case (trigger event)
4. End of the Use Case (termination event)
5. Involved Actors
6. Interaction between the Use Case and the actors
7. Exchange of Information
8. Chronology and Origin of Information
9. Repetitions of Behaviour
10. Optional Situation
11. Capacity
12. Exceptions.

Including all/some of the additional textual description will be helpful depending on the system and the Use Case simplicity.

Types of Use Cases

In general and as a guideline for better Use Cases identification, Use Cases can be described with different types[16], these are;

- **Data Maintenance Use Cases**
Typical Create, Read, Update, Delete Use Cases
- **Request Approval Use Cases**
Use Cases for routing requests through a series of approval stages.
- **Data Analysis Use Cases**
Uses Cases for analyzing transactions on entities that are manipulated by other Use Cases.
- **Payment Use Cases**
Use Cases for making payments. The complexity comes when there are multiple payment methods. Some of them can be immediate, whereas other may take time to clear. Also some may involve a number of business rules which have to be configurable.
- **Loyalty Program Use Cases**
Use Cases allow customers to accrue credits when using a service, and these credits can be used as a payment method or as rewards for corporate gifts.

Evaluating Completeness and Details Level

When describing systems with Uses Cases, crucial questions are to be asked; 'How can I tell that the described Use Cases have covered my system? Is there any indication for that? To what level of details should I go?

As there is no magic tool to test the system's Use Cases against the system requirements, the following points can be considered as good indicators [5]:

- **Use Case Specifications.**
The basic flow and the alternative flows are clear and are understandable
- **Business Use Case Specifications.**
The specifications are very clear when the Use Case is invoked within the business process. Also, the flow of events in the Use Case has been verified against the business processes.
- **Business Entities.**
All the business entities which will be manipulated by the Use Case have been detailed, and their attributes and subcategorization have been defined.
- **Business Rules.**
The business rules required to support the Use Case are clear.

- Supplementary Specifications.

It is clear how supplementary specifications affect the Use Case flow of events.

Use Cases and Testing

One significance of producing tests from specifications is that the tests can be created earlier in the development process, and be ready for execution before the program is finished. Additionally, when the tests are generated, the test engineer will often find inconsistencies and ambiguities in the specifications, allowing the specifications to be improved before the program is written [20].

IBM Research has developed Use Case Based Testing (UCBT), which is a technique for generating test cases and recommended configurations for system level testing. In this approach, testers build a test model based on the standard UML notions of Use Cases, actors, and the relationships between these elements. The Use Cases are enhanced with additional information, including the inputs from actors, the outputs to the actors, and how the Use Case affects the state of the system.

Moreover, newly developed algorithms use this model to generate a test suite which provides a specified level of coverage of each Use Case. The generation algorithm also performs minimization to reduce the number of test cases required to cover the system to the specified level. These features form a powerful basis for model-based test case generation.

UCBT addresses phases where the tester starts with testing the Use Cases individually, then he/she will be interested in looking at combinations of the Use Cases. 'System Test' addresses the situation when all required functionality for the system is present, and the tester seeks to ensure the proper functioning of the system as a whole. An important component of System Test is also ensuring that the system can handle customer-like Scenarios and workloads. After that, 'Solution Test' addresses the situation in which several complete systems are combined to provide complex functionality through some process which involves the systems. These processes can be captured, modeled, and tested using UCBT. Finally, the Use Cases can be connected using flows that describe a sequence of Use Case that are performed to accomplish some goal.

In this approach, testers build a test model based on the standard UML notions of Use Cases, actors, and the relationships between these elements. The Use Cases are enhanced with additional information, including the inputs from actors, the outputs to the actors, and how the Use Case affects the state of the system

Use Cases in Real-Time Systems

A new trend has appeared as a trial to implement Use Cases in Real-Time and Embedded Systems, known as Real-Time UML [7]. In this approach, Timeliness requirements may be addressed by first determining the end-to-end performance requirement of an event-response action sequence. These are normally determined during Use Cases. As an example, a deadline might exist from an external perspective: e.g. "when an actor sends a command,

the system shall respond within 10 ms \pm 2 ms.”

Use Cases view the system from a black-box perspective and serve as a means of capturing external requirements, such as overall performance and response times. BUT, once the ”box is opened” and classes are identified, a Sequence Diagram shows the objects (and operations) involved in handling the request and controlling the output response. Each operation in the sequence is given a portion of that budget such that the sum of all execution times for each operation in the sequence, including potential blocking, is less than or equal to the overall performance budget.

These budgets may be captured as *constraints* associated with the Use Cases or captured graphically on Sequence Diagrams using *timing constraint expressions*. In Figure 1.3, what does ”Monitor System Health” for the elevator system exactly mean? This Use Case internally contains more details about what it means. Therefore, UML has two primary means to provide ”detailing for the requirements”; namely, by example or by specification.

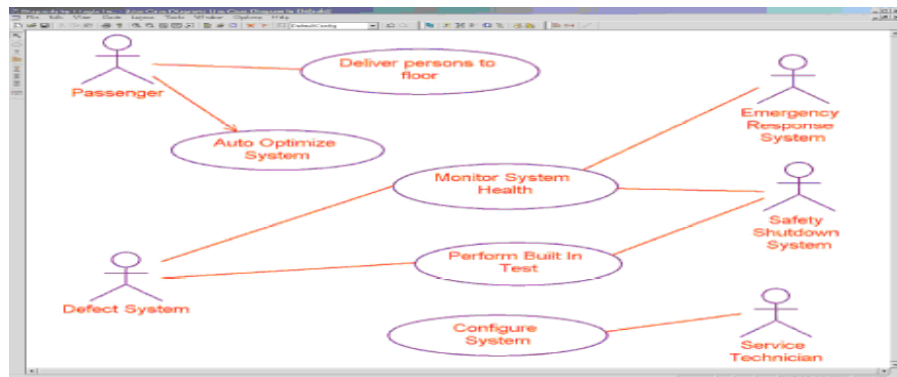


Figure 15.3: Actors and Use Cases for Elevators System [7]

In the UML, a constraint is a user-defined well-formedness rule that applies to one or more model elements. Development engineers have *approached the problems of modeling the real-time aspects through applying constraints to specify crucial and critical specifications*. The recently adopted UML Profile for Schedulability, Performance, and Time Specification is an attempt to codify the standard ways that have been used now for several years of applying the UML to the problems of real-time and embedded systems. The profile really adds no new capabilities to the UML, but does standardize how certain aspects can be represented.

Use Cases And Formalization Attempts

Use Cases are informal, why? UML description is ”semi-formal”, i.e. parts of it are specified with well-defined languages, while others have been described informally in English.

The abstract syntax of the different language constructs in UML is specified with the graphical notation of Class diagrams in UML itself, while the well-formedness rules of UML

are given in OCL, an object-oriented constraint language. BUT, ordinary English is chosen for describing the semantics of UML. This makes the structure of the language rigorous whereas the semantics of the language is still quite informal!

Hence, a technique other than the OMG project worked for the specification of the semantics of UML constructs. The operational semantics are given using an object-oriented specification language named ODAL, which has been formalized using the π calculus. ODAL is a simple, strongly typed language with a familiar syntax. It is used as the specification language in a framework for formal specification of modelling languages.

The syntax of some of the ODAL constructs is described in figure 1.4.

ClassDef	::	CLASS ClassName [SUPERCLASS ClassName] [VARIABLES VarDef*] [METHODS MethDef*]
VarDef	::	VarName : Type
Type	=	ClassName BOOLEAN Type*
MethDef	::	MethodName([VarDef*]) : Type ([VarDef*]) Expr
Expr	=	ExprSeq AssignExpr CondExpr MsgExpr IterExpr FindExpr ...
ExprSeq	::	Expr; Expr
AssignExpr	::	VarName := Expr
CondExpr	::	IF Expr _b THEN Expr _t [ELSE Expr _r]
MsgExpr	::	Expr MethodName ([Expr _i *])
IterExpr	::	FOREACH VarName IN Expr _s DO Expr _b
FindExpr	::	FIND VarName IN Expr _s SUCHTHAT Expr _b

Figure 15.4: Syntax of some of the ODAL constructs [11]

It is possible to describe a Use Case more formally by means of a set of Operations and Methods. Since a Use Case is a kind of Classifier, it has a collection of Operations and a collection of methods describing its behaviour.

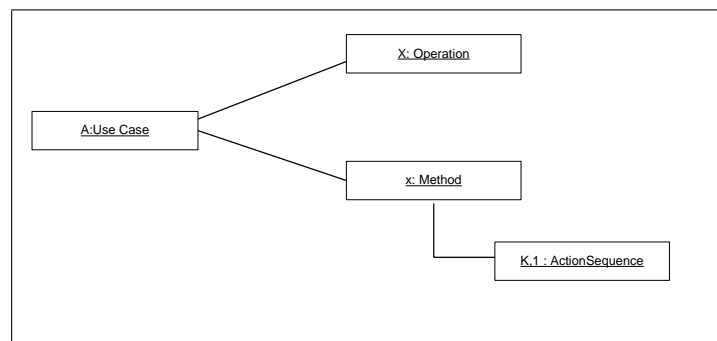


Figure 15.5: Use Case Description [17]

The Operations of a Use Case describe what message instances an instance of that Use Case may receive, while the methods describe what sequences of actions are performed by instances of the Use Case. A complete sequence of the Use Case consists of one or several

Operations Methods, performed in a pre-defined order. As described in Figure 1.5, a simple Use Case(A) is described with one Operation X and a Method with the same name which realizes the Operation. The sequence of actions of the method consists of two actions: k and l. To specify the order between the methods of a Use Case, a technique with a state variable is used. *Each method has a value representing the state in which the method may be invoked, and each of the action sequences within a method is ended by giving the state variable its new value, thus specifying which methods are the successors of that sequence.*

```

CLASS UseCase
SUPERCLASS Classifier
METHODS
allowedAssociati ()
  c := TRUE;
  FOREACH r IN relationships DO
    IF r isKindOf (Association) THEN
      FOREACH n IN r getAllOppositeNames (SELF) DO
        c := c AND SELF owner ≠ r lookupType (n)
owner 0;
C

consistent ()
  SUPER consistent () AND SELF allowedAssociations () AND
  SELF contents () = NULL

```

Figure 15.6: Formal Use Case [17]

Drawbacks, Confusion and Deficiencies

Internal communication inside the system, conflicts between Use-Case instances and concurrency between Use Cases can't be modeled. Although Use Cases are supposed to be independent of any formal design, the conceptual structures fostered by Use Case development mislead developers about design [19]. Here are some problems which lead to missed logical dependencies in systems analysis.

1. Arbitrary goto and comefrom jumps in the flow of control

There is no clear description about the transfer of the internal focus of control within Use Cases. Through the <<extend>> relationship, an extension ultimately returns control to using case, whether or not the extension eventually executes. (comefrom and goto). These vague extraordinary flows of control speak for themselves. You may compare the kind of jumping out of blocks and breaking into blocks required by the UML Use Case model with the kinds of programming style that were judged "harmful" at the beginning of the block-structured programming era [6]. Use cases conceal an extraordinary complexity in the flow of control, which programmers must completely deconstruct, if they are to avoid disastrous logical program structures.

2. Semantics are inadequate to model insertion, exceptions and alternatives

Use case diagrams are dangerously ambiguous; developers have to rely on intuitions about the labeling of the cases to establish the intended logic, in disregard of the official semantics. Developers commonly use <<extend>> to indicate exceptions where the dependency semantics of <<extend>> does not support exceptions. The extension of 'Abort' operation to 'Withdrawal' through 'Abort <<extends>> Withdrawal', is an example of that. Moreover, developers commonly use <<extend>> to indicate alternative. The extension of 'PayDirect' operation to 'Sign for Order' is intended as an alternative.

Modelling these control variants properly would require three different definitions of <<extend>> altogether in terms of considering the flow graphs in the cases of dependency, exception or alternative.

3. The missed long-range logical dependencies

In Use Cases, modelling promotes a highly localized perspective which obscures the true business logic of a system. Even simple examples exhibit unpleasant mutual interactions between extensions and base cases. Sometimes, nodes described in other forms wouldn't qualify as Use Cases, since they have a larger granularity than accepted and may in general be quite abstract in nature.

Recall: The emphasis on a single observable result is a deliberate constraint which seeks to ensure a minimum and maximum granularity for a Use Case, which can be neither an incomplete sequence (offering no benefit), nor multiple sequences (achieving several benefits) [13].

15.2 Scenarios

A Scenario is a common way to detail a Use Case by providing a set of examples about the Use Case behaviour. Each Scenario captures a specific interaction of the use case and details it. By providing a set of these Scenarios, each Scenario provides typical or exceptional cases of the use of the system capability.

Advantages in Brief

As Scenarios are primarily useful for adding detail to an outline requirements description, developing a set of Scenarios for end-users is in itself a useful exercise for elicitation. Scenarios can be identified by initial discussions with stakeholders who interact with the system. Moreover and as experience shows, analysts think about Scenarios as a medium to understand an already-built system, by asking how the system responds (component by component) to a particular input or operational situation.

Considering the system through all the thought-off Scenarios and from different perspectives (viewpoints) can be considered as an implementation to concept of *Separation of Concerns* for better maintainability at the requirement level, design level and development

level. Moreover, the process of choosing Scenarios for analysis forces designers to consider the future usage of the system and the changes to the system.

Scenarios capture three different kinds of requirements: messages or operations performed by the system, protocols of interaction between the system and its actors; and constraints on messages, operations, or protocols.

A good thing about Scenarios modeling is that it describes how the system interacts with the actors in its environment in an implementation-free way.

Limitations

One downside of using Scenarios is that there is an infinite set of Scenarios; it may not always be obvious which Scenarios ought to be modeled. Another downside which was considered before UML 2.0, was that, Scenarios provide only a means to model positive requirements, i.e. requirements that could be stated as "The system shall" but no way to model negative requirements, i.e. requirements that would be stated "The system shall not". UML 2.0 and through Interactions(Sequence Diagrams)has this advantage over Message Sequence Charts(MSC2000) as described in the last section of this paper.

15.3 Sequence Diagrams

A Sequence Diagram is a diagram which depicts the interactions between classes, instances or even actors in the form of method calls and call returns. These diagrams are defined by the Unified Modeling Language (UML) notation. It shows the processes that execute in sequence among objects. Despite its first appearance in 1997, UML 1.0 did have some simple sequence diagrams similar to those found in Message Sequence Charts (MSC-92). UML went through small revisions leading to UML 1.5 in 2003. Still over the last three to four years a major revision of UML has taken place leading to UML 2.0 which became an available technology from OMG in 2004. In UML 2.0 also Sequence Diagrams (or Interactions) have been thoroughly revisited and revised to overcome the problematic in UML 1.X because of its lack to language support in terms of reusing and combining sequences which were offered in MSC.

Key Language Elements

Boxes: The boxes across the top of Figure 1.7 represent classifiers or their instances, they are typically Use Cases, objects, classes, or actors. Because messages can be sent to both objects and classes, objects respond to messages through the invocation of an operation and classes do so through the invocation of static operations, it makes sense to include both on Sequence Diagrams. Objects have labels in the standard UML format, objectName: ClassName. When not specifying the object name, then this object is considered and known as Anonymous. Object labels are underlined, classes and actors are not.

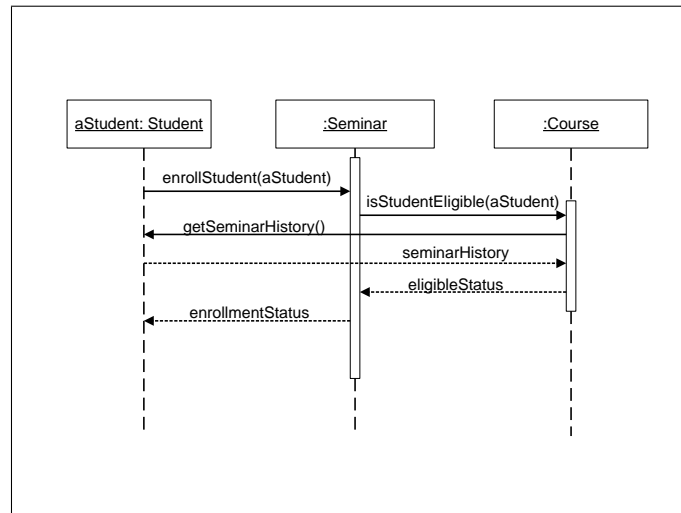


Figure 15.7: Student Seminar enrolment Scenario [4]

Lines, Dashed Lines and Activation Boxes: Forward Lines describe the direction of the message (from/to). Dashed lines hanging from the boxes are called object lifelines, representing the life span of the object during the Scenario being modeled. The long, thin vertical boxes on the lifelines are activation boxes, also called method-invocation boxes, which indicate processing is being performed by the target object/class to fulfill a message.

Object Creation and Destruction: For the instance creation and destruction, an (X) at the bottom of an activation box, is a UML convention to indicate an object has been removed from memory. As a response to garbage collection in programming languages, it's commonly not to worry about modeling destruction.

Messages: Messages are indicated on UML Sequence Diagrams as labeled arrows. In UML, when the source and target of a message is an object or a class, the label is the signature of that method invoked in response to the message. However, if either the source or target is an actor, then the message is labeled with brief text describing the information being communicated.

Return Values: Return values are optionally indicated using a dashed arrow with a label indicating the return value. A good style is not to indicate the return values when it is obvious what is being returned as Sequence Diagrams get complicated fairly quickly.

Notes: Notes are basically free-form text which can be placed on any UML diagram, to provide a header for the diagram, indicating its title and identifier or even to provide explanation. Notes are depicted as a piece of paper with the top-right corner folded over.

Also, notes are used to indicate future work that needs to be done, either during analysis or design.

Conditional Decision []: When it comes to decisions, conditions can be expressed in two bracket '[]', whereas the complement of the condition can be described using [else] notation

Loops: Loops can be expressed in two ways. One way is to show a frame with the label loop and a constraint indicating what is being looped through, such as (for each objectName). Another approach is to simply precede a message that will be invoked several times with an asterisk. It is quite useful to insert textual descriptions of what's happening along the left side of the Sequence Diagram. This involves lining up each text block with the appropriate message within the diagram. This helps in understanding the diagram (at the cost of some extra work).

Sequence Diagram and Concurrency

Sequence diagrams are also valuable for concurrent processes. As an example in Figure 1.8, when a transaction is created, it creates a Transaction Coordinator to coordinate the checking of the transaction. This coordinator creates a number (in this case, two) of Transaction Checker objects, each of which is responsible for a particular check. This process would make it easy to add different checking processes because each checker is *called asynchronously and proceeds in parallel*. When a Transaction Checker completes, it notifies the Transaction Coordinator. The coordinator looks to see if all the checkers called back. If not, the coordinator does nothing. If they have, and all of them are successful, as in this case, then the coordinator notifies the Transaction that all is well.

Figure 1.8 introduces a number of new elements to Sequence Diagrams. First, *activations*, which appear explicitly when a method is active because it is either executing or waiting for a subroutine to return. Many designers use activations especially in concurrent situations. The *half-arrowheads* indicate an asynchronous message. An *asynchronous* message does not block the caller, so it can carry on with its own processing.

Sequence Diagrams in Real-Time Systems

In Real-Time, Scenarios are modeled primarily with sequence diagrams. Sequence Diagrams describe the Use Case, its preconditions and postconditions, and keeps a running dialog explaining what's going on as the Scenario unfolds. Unfortunately and for the lift passenger system described in [9], one can easily imagine the many variants showing what happens when passengers want to go up, when the same request occurs several times, when the cable breaks, and so on. Each of these different Scenarios would be captured in a different Sequence Diagram, resulting in probably a dozen or so Scenarios to elaborate what we mean by "Deliver Persons to Floor" Use Case!

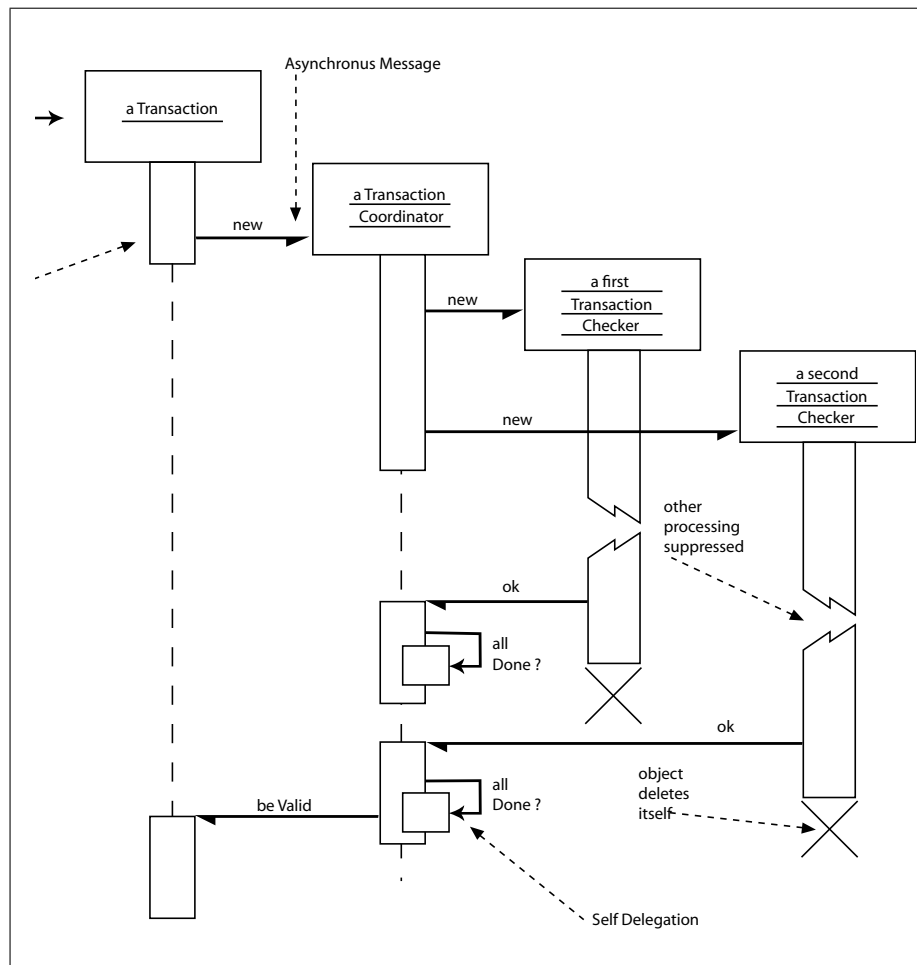


Figure 15.8: Concurrent Processes and Activations [18]

Sequence Diagrams and Testing

Sequence Diagrams provide a picture of what testers must test and validate. A tester going through a Use Case at the GUI level can see what is going on underneath the covers [3]. Because Sequence Diagrams are time-based, if the system generates an error at any point, the tester can correlate that with the underlying code. He/She never sees the code as he/she doesn't have to. UML is handling that layer of abstraction between the low-level code and the high-level GUI. This allows the tester to annotate the defect which he/she submits and reference the Sequence Diagram for better communication at the team level.

On the other hand, consuming horizontal space, readability issues when it comes to large Sequence Diagrams and dependency on the designer in case of badly expressed Sequence Diagrams are of the major drawbacks of Sequence Diagrams.

15.4 Message Sequence Charts

As a language for the description and specification of the interactions between system components, Message Sequence Chart (MSC) has appeared in particular telecommunication switching systems. However, Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation of real-time systems. MSC can be viewed as a special trace language which mainly concentrates on message interchange by communicating entities and their environment.

MSCs have been used for a long time within international standardization bodies and within industry, following different conventions under various names such as Arrow Diagrams, Extended Sequence Charts, Information Flow Diagrams and Message Flow Diagrams. These MSC variants mainly differ with respect to syntax and terminology. There are only minor semantic differences. The International Telecommunications Union (ITU), previously known as CCITT, developed the MSC language definition Z.120, which was approved by the CCITT members at that time in May 1992. In 1996, MSC-96 was introduced. MSC-96 was backward compatible with MSC-92 and added a number of features which made MSC-96 more versatile and powerful than MSC-92. The concepts of timers, instance creation and instance stop in MSC-92 had not changed in MSC-96. MSC96 included two syntactical forms, MSC/GR as a graphical and MSC/PR as a pure textual representation. Despite the concepts of condition and submsc of MSC-92 had been slightly modified, the interpretation of old diagrams weren't drastically different when interpreted as MSC-96 diagrams. The new features included: MSC references, gates and High-level Message Sequence Chart (HMSC). In 2000 [2], MSC-2000 was introduced with improved structural concepts and object orientation, more time observations and time constraints and better support for synchronizing communication than MSC-96. MSC-2000 will be maintained for more on quality of service and improvements on grammar and meta-grammar.

MSC-2000 and as MSC-96 also did, included two syntactical forms, MSC/PR as a pure textual and MSC/GR as a graphical representation. An example of the MSC/GR and the corresponding MSC/PR representation for a connection MSC is shown in Figure 1.9.

The MSC/PR contained in Z.120 lists message sending and receiving events in association with an instance. A better readable notation was suggested; a new event oriented textual representation was elaborated where events are listed in form of a possible execution trace and not ordered with respect to instances. Figure 1.10 demonstrates the event oriented textual syntax description.

MSC Language Elements

Basic Elements:

a. Instances and Messages

Instances and Messages describing the communication events, are the most fundamental language constructs of MSCs. In the graphical representation, instances are shown by

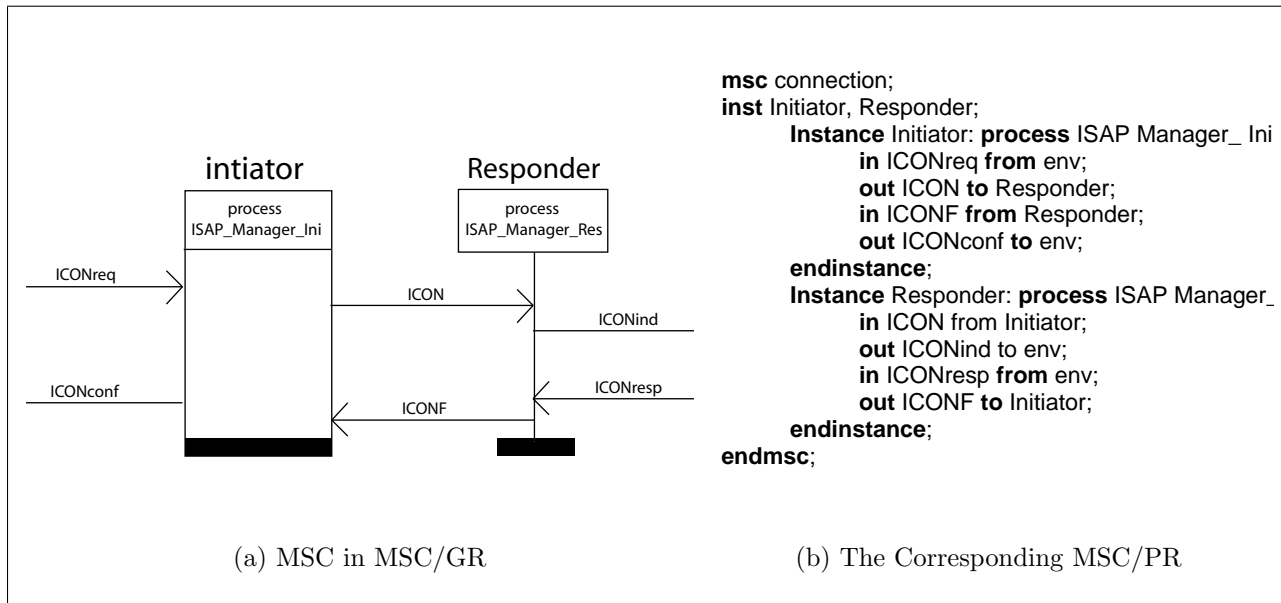


Figure 15.9: MSC in MSC/GR and in the Corresponding MSC/PR [18]

```

msc connection;
inst Initiator, Responder;
Initiator: instancehead process ISAP Manager Ini;
Responder: instancehead process ISAP Manager Resp;
Initiator: in ICONreq from env;
Initiator: out ICON to Responder;
Responder: in ICON from Initiator;
Responder: out ICONind to env;
Responder: in ICONresp from env;
Responder: out ICONCONF to Initiator;
Initiator: in ICONCONF from Responder;
Initiator: out ICONconf to env;
Initiator: endinstance;
Responder: endinstance;
endmsc;

```

Figure 15.10: Event-Oriented textual description [18]

vertical lines whereas message flow is represented by arrows. The head of the message arrow denotes the event message consumption, the opposite end denotes message sending. In addition to the message name, message parameters in parentheses may be assigned to a message.

b. System Environment

The system environment is graphically represented by the frame which forms the boundary of an MSC diagram.

c. Actions

An action describes an internal activity of an instance is graphically represented by a rectangle containing text.

Figure 1.11 demonstrates instances, messages, system environment and actions of an MSC

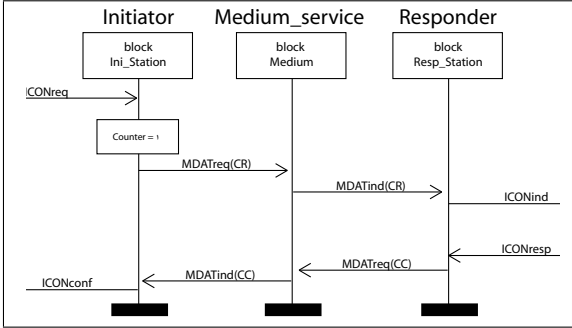


Figure 15.11: Instances, Messages, System Environment and Actions [18]

d. Timers

Timer handling in MSCs encloses the setting of a timer and a subsequent time-out (timer expiration) or the setting of a timer and a subsequent timer reset (time supervision). The setting of a timer is represented by an hour-glass. The reset symbol is presented by a cross (X). Time-out is described by an arrow which is connected to the hour-glass symbol. An optional timer description containing name and duration may be associated with each timer symbol.

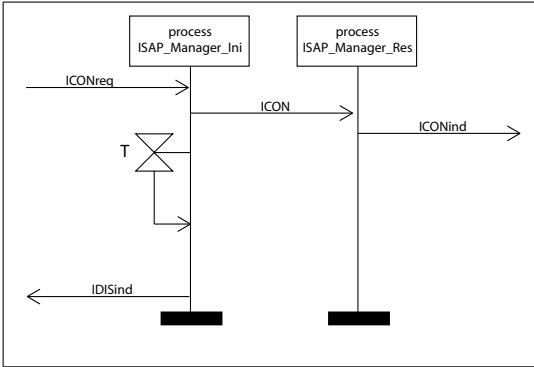


Figure 15.12: Time-out [18]

e. Instances Creation and Termination

Due to the fact that most communication systems are dynamic systems where instances appear and disappear during system run-time, MSC supports such features. The create

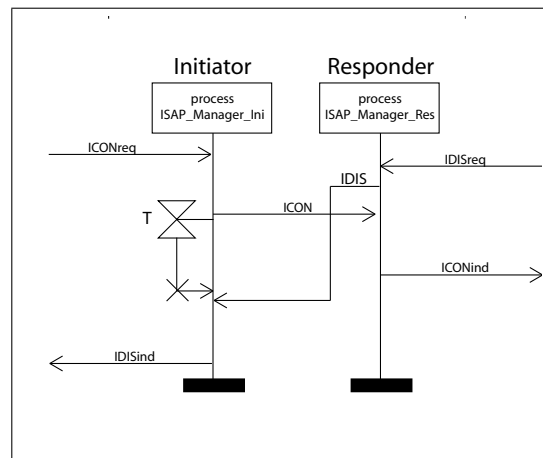


Figure 15.13: Timer Reset [18]

symbol is a dashed arrow which may be associated with textual parameters. A create arrow originates from a parent instance and points at the instance head of the child instance. The termination of an instance is graphically represented by a stop symbol in form of a cross at the end of the instance axis as described below

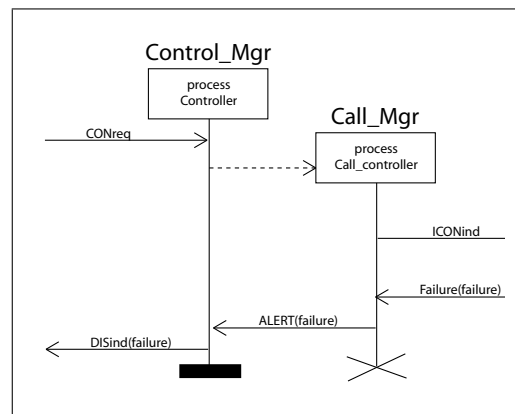


Figure 15.14: Instance Creation and Termination [18]

f. Conditions

A condition either describes a global system state referring to all instances contained in the MSC(global condition), or a state referring to a subset of instances (non-global condition). If it refers to one instance only, then it is a local condition.

Structural Elements: Structural language elements of MSCs include all constructs which can be used to specify more general MSCs or to refine MSCs

a. Coregion

Along an MSC instance, a total ordering of message events is assumed. To cope with this, coregion is introduced. A coregion is graphically represented by a dashed section of an MSC instance. Within a coregion, the specified communication events are not ordered as described in Figure 1.15.

b. Submsc

Since MSCs can be rather complex, there is a need for a refinement of one instance by a set of instances defined in another MSC. An MSC instance can be refined by another MSC, which is then called submsc. By means of the keyword *decomposed*, a submsc with the same name is attached to the refined instance. The submsc represents a decomposition of this instance without affecting its observable behaviour.

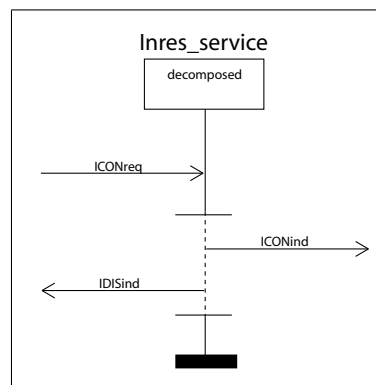


Figure 15.15: msc Structural.Concepts with coregion [18]

Composition of MSC

Since one MSC only describes a partial system behaviour, it is beneficial to be able to combine a number of simple MSCs. To determine possible combinations, global and non-global conditions may be used employing certain rules. An example of composition is described through Figures 1.17, 1.18 and 1.19.

Object-Oriented MSC

In practice, a combination of pure composition techniques and object oriented techniques has proven to be most powerful. As MSC documents define instance kinds, they are suited

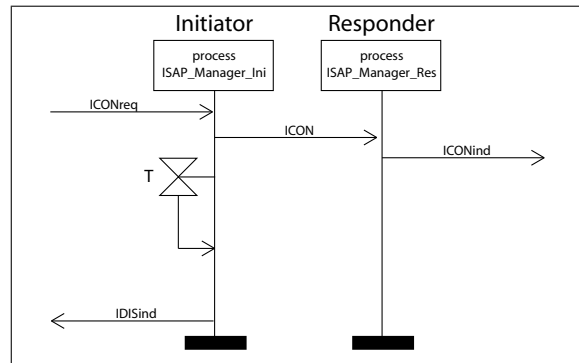


Figure 15.16: submsc Inres_service: Refinement for Figure 1.15 [18]

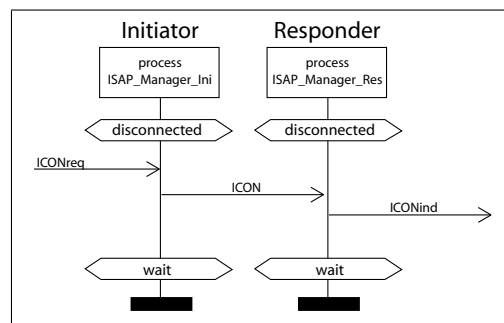


Figure 15.17: request MSC [18]

for object orientation. Inheritance of instance kinds means inheriting all contained instances and MSCs. Virtual MSCs means that MSCs may be redefined in specialized instance kinds. The idea behind virtuality is that virtual types enclosed in the general type may get a new definition in the specialization. Virtual means that it is possible to adapt an MSC to special configurations or situations by redefining virtual MSC parts as described in the Figure 1.21.

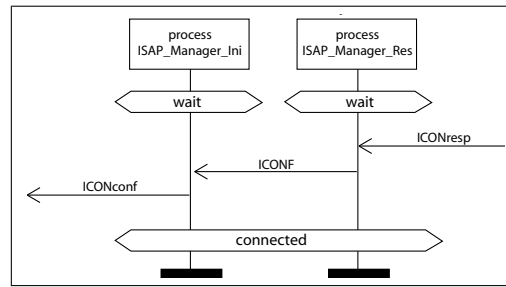


Figure 15.18: confirm MSC [18]

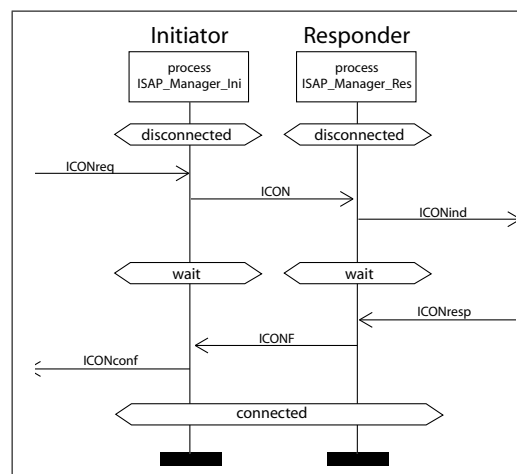


Figure 15.19: Composition of 1.17 and 1.18 [18]

15.5 Sequence Diagrams Versus Message Sequence Charts

Messages, considered as the key language element in both MSC and UML for Scenarios description, through the representation of their sequence.

In UML 2.0, Interactions come in several graphic forms. The most expressive form is the *Sequence Diagram* and every concept of Interactions can be expressed in Sequence Diagrams.

Table 1.1 and as a comparison table, intended to give overview of the different central concepts in the field of Interactions/MSC such that, those only familiar with one of the languages can see what terms have been used in the other language.

As one of the main major diagrams in UML 2.0, communication diagram gives an overview of how simple communication goes between the lifelines. It is overloaded on a composite structure diagram where the messages are numbered and shown on the connectors (communication lines). In UML 1.x, communication diagram was called a collaboration diagram, but the term "collaboration" was inadequately overloaded. Nowadays, UML 2.0 "collaboration" is a term for a special kind of classifier; a kind of generic class concept.

HMSC in MSC2000 and in its graphical form focuses on showing the larger picture where

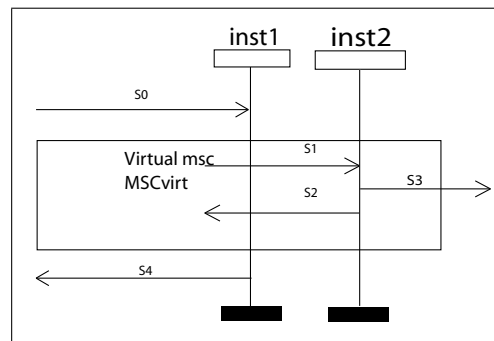


Figure 15.20: msc MSCwithVirt [18]

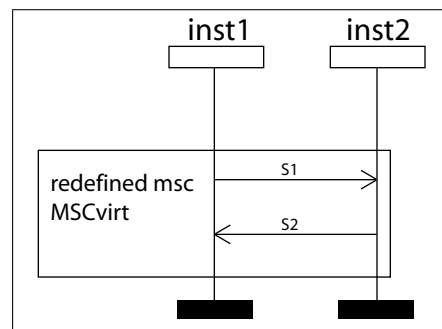


Figure 15.21: msc MSCinheritance inherits MSCwithVirt with redefined MSCvirt [18]

the general control flow is the most significant, has the interaction *Overview Diagram* as a counterpart in UML 2.0 and activity diagrams in UML 1.x. The concept of MSC *reference* and interaction *Occurrence* of UML 2.0 are almost identical. In its basic form they can both be understood by substituting the referred diagram into where the reference was. MSC also features "reference expressions" where the text of the MSC reference can designate an expression like an inline expression. Whereas, UML does not have a direct counterpart of this feature.

For the messages and while UML has a tradition of using interactions to describe the control flow of a sequential program, MSC considers a set of entirely concurrent instances. However, in both cases the semantics is given by the traces of the events leading from the initiation of the operation call (MSC: method call) to the reception of the reply. UML 2.0 has added a few notations. *neg* and *assert* are introduced to make Sequence Diagrams more suitable to express requirements that are more absolute. The *neg* operator defines those *traces that should not occur*, and the *assert* defines the traces that should (mandatory) occur at a given point in the Scenario.

For the formality, UML concepts may even exist without syntax, which means that it is up to the tool how to present the concept to the user and this will often result in values given in a dialogue box. Whereas, In MSC on the other hand, every concept has concrete

Table 15.1: MSC-2000 Versus UML 2 Terminologies [12]

MSC-2000	UML 2.0
Event	EventOccurrence
MSC Document	Class (or Collaboration)
HMSC	Interaction Overview Diagram
Instance	Lifeline
Message	Message
Method call	Operation call
Method (area)	ExecutionOccurrence
Action	ExecutionOccurrence
Suspension (area)	No direct counterpart
Gate	Gate
No direct counterpart	Interaction fragment
Inline expression	Combined Fragment
Coregion	Coregion
MSC reference	Interaction Occurrence
Decomposition	PartDecomposition
General Ordering	General Ordering
Condition (global state)	Continuation
Condition (predicate)	Interaction Constraint
Relative time	Duration
Absolute time	Time
Time measurement	TimeObservationAction, DurationObservationAction
Timer	No counterpart

syntax. The practical difference between UML and MSC regarding formality is not as big as the SDL/MSC community likes to pretend [12]. The determining interpretations come in both languages from reading the informal specification and adjusting it to the situation where it is going to be applied. It is still the case that MSCs / Interactions are used more for illustration and discussion than formal requirements specification and verification [12].

Historically, in the mid-1990s there was a lot of exploratory academic work on MSC, while recently academics in general are turning more towards UML for the same opportunistic reasons as does the industry. Even though the semantics of MSC may be slightly more formally defined through the early work on MSC-96 and the precision of the Z.120 [14] standard, than the UML 2.0, the average user will not notice [12].

In conclusion, MSC-2000 and UML 2.0 Interactions are very similar, which is exactly

what was expected and intended. Proper tool support and how those tools integrate with SDL and UML 2.0 respectively, will determine which language will survive. There is also the possibility that the languages will co-exist and cross-pollinate. From a pure market view, it seems probable that the next couple of years will choose a winner. If the UML community can reach real code generation and machine supported verification, their market position will make them the winners. On the other hand if UML users are unable to reach the level of automatic support that is commonplace in the SDL community, UML will vanish [12].

15.6 Exam Questions

1. What indicators can tell the Completeness/"Good Enough" level of details of System Use Cases?
2. How could Scenarios be considered as an implementation of the concept of separation of concerns?
3. Summarize the key milestones in the evolution of both UML and SDL/MSD.

Bibliography

- [1] <http://www.omg.org>.
- [2] <http://www.sdl-forum.org/MSD2000present/index.htm>.
- [3] Ed Adams and Sam Guckenheimer. *Achieving Quality by Design*. IBM Laboratory, <http://www-128.ibm.com/developerworks/rational/library/4663.html>, 2001.
- [4] Scott W. Ambler. *The Object Primer: Agile Model Driven Development with UML 2*. Addison-Wesley, 3rd edition, 2004.
- [5] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [6] Edsger W. Dijkstra. Letters to the editor: goto statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.
- [7] BP Douglass. Real-time uml. In E.-R. Olderog W. Damm, editor, *Lecture Notes in Computer Science*, volume 2469 / 2002, pages 53 – 70. Springer Berlin / Heidelberg, 1998.
- [8] Hans-Erik Eriksson and Magnus Penker. *Business Modeling with UML*. John Wiley and Sons, 2000.
- [9] Martin Fowler and Kendall Scott. *UML Distilled : Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

- [10] J. Rumbaugh G. Booch and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 2nd edition, 2005.
- [11] Overgaard Gunnar. A formal approach to relationships in the unified modeling language. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universitaet Muenchen, TUM-I9803, April 1997.
- [12] Oystein Haugen. Comparing uml 2.0 interactions and msc-2000. In Alan W. Williams Daniel Amyot, editor, *Lecture Notes in Computer Science*, volume 3319 / 2005, page p.65. Springer Berlin / Heidelberg, 2005.
- [13] M. Griss I. Jacobson and P. Jonsson. *Software Reuse: Architecture, Process and Organisation for Business Success*. Addison-Wesley, 1997.
- [14] ITU-T. Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, International Telecommunication Union – Standardization Sector, Genève, 2000.
- [15] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [16] Pan-Wei Ng. Adopting use cases. rational software, rational edge, http://www-128.ibm.com/developerworks/rational/library/content/rationale/may03/m_ng.pdf. 2003.
- [17] Gunnar Overgaard and Karin Palmkvist. A formal approach to use cases and their relationships. In *UML98: Selected papers from the First International Workshop on The Unified Modeling Language (UML98)*, pages 406–418, London, UK, 1999. Springer-Verlag.
- [18] Ekkart Rudolph, Peter Graubmann, and Jens Grabowski. Tutorial on message sequence charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
- [19] Anthony Simons. 37 things that don't work in object-oriented modelling with UML. In Haim Kilov and Bernhard Rumpe, editors, *Proceedings Second ECOOP Workshop on Precise Behavioral Semantics (with an Emphasis on OO Business Specifications)*, pages 209–232. Technische Universität München, TUM-I9813, 1998.
- [20] Clay Williams and Amit Paradkar. *Efficient Regression Testing of Multi-Panel Systems*, page 158. IEEE Computer Society, Washington, DC, USA, 1999.

Chapter 16

Jie Gui: Models for Configuration Management

Roger Pressman, in his book, *Software Engineering: A Practitioner's Approach*, says that software configuration management (SCM) is a "set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made." In a nutshell, SCM is a methodology to control and manage a software development project. In this report, B method [6] is used to illustrate the basic models inside SCM revision control and examples of implementations from several typical softwares.

16.1 Introduction

SCM is the control of the evolution of complex systems [7]. More specifically, it is the discipline that help the developers to keep evolving software projects under manageable control, that contributes to improved software quality.

Most SCM tools are based on a tiny core of concepts and mechanisms. Among them, the unaged key function is the *Revision Control*, which is the heart of any SCM system Without this foundation [3], diffing, auditable version forking and multiple branching of development and release bits would be unmanageable, let alone tracking any changes made among team members. Because source code is such a key role in the software development, choosing the wrong solution for the job can be a disastrous. This issue will be mainly talked about with the models inside it later in this report. With each model given, specific implementations from different SCM tools will be compared.

The whole implementation for an ideal SCM tool that fulfill the need of the practical software development cannot be quite easily achieved due to its complexity, thus it is more feasible for the users to choose a suitable tool rather than writing the tool himself [1]. In this

report, following typical SCM tools will be mentioned, CVS (Concurrent Version System)¹ and Subversion² are the most popular implementations among the choices of free licences while ClearCase³, Perforce⁴ and VOODOO⁵ are exchanging the high-end performance with the outgoing cost for commercial licences.

- **Concurrent Versions System(CVS)** is also known as the Concurrent Versioning System, implements a version control system: it keeps track of all work and all changes in a set of files, typically the implementation of a software project, and allows several (potentially widely separated) developers to collaborate. CVS has become popular in the open-source world. CVS is released under the GNU General Public License⁶.
- **Subversion** is an open source system for revision control, sometimes known as svn from the name of its command line interface. Subversion is designed specifically to be a modern replacement for CVS and shares a number of the same key developers.
- **ClearCase** is a commercial software tool for revision control of source code and other software development assets. It originally derived from a product of Apollo Computers⁷: DSEE (Domain Software Engineering Environment), which was ported to Unix and further developed by Atria Software after Hewlett-Packard⁸ bought Apollo. Atria later merged with Pure Software to form PureAtria. That firm merged with Rational Software, which was purchased by IBM. IBM continues to develop and market ClearCase. ClearCase forms the base of version control for many large and medium sized businesses and can handle projects with hundreds or thousands of developers, but the price is quite steep for smaller companies.
- **Perforce** is a commercial Revision Control (RC) system. It is developed by Perforce Software, Inc⁹. and was founded in 1995 by Christopher Seiwald. The Perforce system is based on a client/server model with the server managing the collection of source versions in one or more depots. The server software runs on the Unix, Mac OS X¹⁰, or Microsoft Windows¹¹ operating systems.
- **VOODOO** is also commercial version control system for software developers. VOODOO Server is uncompromisingly designed for Mac OS X. It provides main functions of a SCM system plus an improved revision control mechanism.

¹<http://www.nongnu.org/cvs/>

²<http://subversion.tigris.org/>

³Trademark of IBM Inc. <http://www-306.ibm.com/software/awdtools/clearcase/>

⁴Trademark of Perforce Inc. <http://www.perforce.com/>

⁵Trademark of *Uni Software Plus Inc.*. <http://www.unisoft.co.at/products/voodoooserver/>

⁶<http://www.gnu.org/licenses/licenses.htm>

⁷Apollo was acquired by Hewlett-Packard in 1989 for US \$476 million, and gradually closed down over the period 1990-1997.

⁸<http://www.hp.com>

⁹<http://www.perforce.com>

¹⁰Product of Apple Computers.(<http://www.apple.com>)

¹¹Product of Microsoft Corp.(<http://www.microsoft.com>)

All the models given in this report will be in B-Method [6]. The B-Method is a collection of mathematically based techniques for the specification, design and implementation of software components. Systems are modeled as a collection of interdependent *Abstract Machines*, for which an object-based approach is employed at all stages of development.

16.2 Models for SCM

In this section, models inside SCM system will be discussed within one abstract model *BasicWorkCycle*, mainly focus on basic operations that implemented by most of the version control systems.

Key Concepts of Revision Control

Among all the elements inside a SCM system, following basic concepts are considered most fundamental.¹²

Repository The repository is much like an ordinary file server, except that it keeps tracking of every single change ever made to the files and directories. This enables the users to recover previous versions of data, or track back the history of how the data being changed. In this regard, many people think of a version control system as a sort of *time machine* [10].

At some level, the ability for various people to modify and manage the same set of data from their respective locations fosters collaboration. Progress can occur more quickly without a single conduit through which all modifications must occur. Because the work is versified, it need not to fear that quality is the trade-off for losing that conduit – if some incorrect change is made to the data, just undo that change [2]. Figure 16.1 illustrates a simple example for repository.

Normally, no direct access to any of the files in the repository is permitted [5]. Instead, specified commands is used to get the copy of the files into a *working directory*, and then work on that copy. When a set of changes have been applied to the working copies, *checkin* (or *commit*) operation is used to upload them back into the *repository*. The repository then contains the changes which been just made, as well as recording exactly what is changed, when it is changed, and other such information [2]. Note that the repository is not a subdirectory of the working directory, or vice versa; they should be in separate locations. In this report, the repository and local working directory are connected in a server-client model by default.

The basic operation models used towards the repository mentioned in later chapters discuss can be classified into the following categories:

¹²Since this report is mainly focusing on the operation models of SCM system, readers are assumed to have the professional knowledge of how the system functions. Therefore, the fundamentals of SCM will be introduced briefly. The system discussed here are free of security concerns, such as user authentication, the files element mentioned are by default referred to ASCII files.

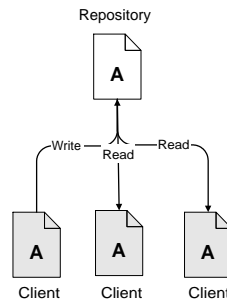


Figure 16.1: The simple repository model for a file system.

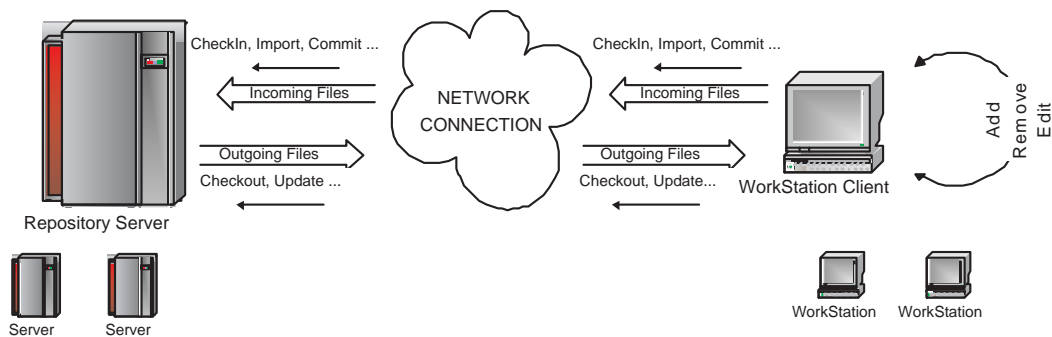


Figure 16.2: The work flow chart of the three types of operations.

- Create repository initially.
checkout
- Update working copy.
update
- Make changes.
add
delete
- Commit changes from working copy to repository.
commit
- Import new project files to the repository.
import

In the following Figure 16.2, a brief work flow chart is given to illustrate the above operations between the repository server and workstation client.

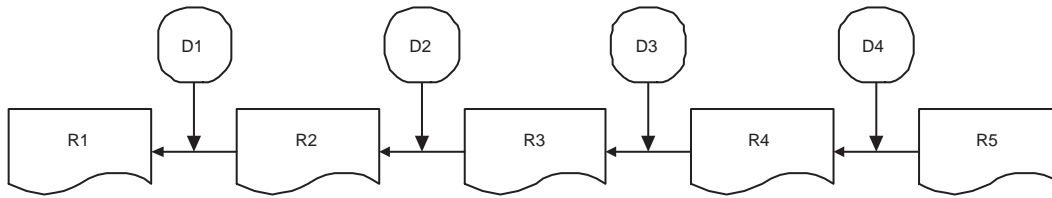


Figure 16.3: Example of how the delta compression work in the revision control.

Delta compression Delta compression is a way of storing data in the form of differences between sequential data rather than complete files. Delta compression frequently applied particularly where archival histories of changes are required, thus the most typical example is the SCM revision control. Almost every tool uses this technique to efficiently record the modifications in different versions of a single ASCII file.

The differences are recorded in discrete files called *deltas* or *diffs*. Because changes are often small (only 2% total size on average), delta compression greatly reduces data redundancy. Collections of unique deltas are substantially more space-efficient than their non-compressed equivalents. From Figure 16.3, a possible example of the use of the compression is given. In this way, the only file will be stored to server will be latest submitted file, together with the set of generated delta files corresponding to the revisions in history, thus any previous version of the same file can be reproduced with the sequential combination of the delta files and the latest full-text file, as the formula below indicates.

$$R1 = ((R5 \oplus D4) \oplus D3) \oplus D2 \oplus D1$$

The Basic Work Cycle

After the brief introduction to the concepts of revision control, a basic idea on how the system works can be found from the following simple work cycle(Figure 16.4).

Consider the situation where a developer needs to make a change to one source file. The following steps are of the most simple and common.

- Checkout the file
- Edit the working file as needed
- Checkin the file

All the abstract models mentioned in the following section are actually carried out in the same type of work cycle in different cases. In the following section, these operation models will be formalized with the models using *B-Method*.

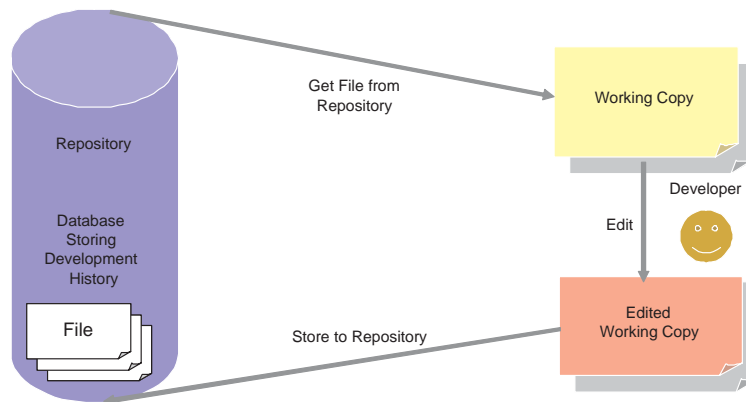


Figure 16.4: The basic checkout-edit-checkin cycle

The Uniform Models

Consider the above operations mentioned, they can be mainly modeled into two types, by the different communication directions between client and server. The basic work cycle as mentioned in last section illustrate the idea about what happens in real software development, before the development can be carried out on the files of the project, the users have to *checkout* the files needed from the remote repository, after the modification is finished, to share the changes with other users in the whole development, they all will then synchronize the changed file with the remote repository server, that is *checkin*. Then the same cycle happens on other users consequently or even simultaneously.

The machine *BasicWorkCycle* models the checkout, checkin and some other methods in different situations, including: *add*, *delete*, *import*, *commit*, *branch*, *merge*.

MACHINE

BasicWorkCycle

ABSTRACT CONSTANTS

InitRev, *InitBRev*

PROPERTIES

$InitRev \in \mathbb{Z}$;

$InitBRev \in \mathbb{Z}$;

SETS

DirSet; *TagSet*; *PathSet* *FileText*

VARIABLES

RevFiles, *WorkFiles*, *revTag*, *revRev*, *revPath*,

tagRev, *Maxrev*, *Baserev*, *workRev* *PathSet*

revtext, *revname*, *workname*, *worktext*

INVARIANT

$Baserev \in \mathbb{Z} \wedge Maxrev \in \mathbb{Z}$

$\wedge RevFiles \subset DirSet \wedge WorkFiles \subset DirSet$

$$\begin{aligned}
&\wedge revTag \in TagSet \mapsto DirSet \\
&\wedge revPathSet \subset PathSet \\
&\wedge revRev \in \mathbb{Z} \rightarrow DirSet \\
&\wedge revPath \in revPathSet \rightarrow DirSet \\
&\wedge tagRev \in TagSet \mapsto \mathbb{Z} \\
&\wedge workRev \in \mathbb{Z} \rightarrow DirSet \\
&revtext \in FileText \rightarrow DirSet \\
&\wedge revname \in FileText \rightarrow DirSet \\
&\wedge worktext \in FileText \rightarrow DirSet \\
&\wedge workname \in FileText \rightarrow DirSet
\end{aligned}$$

Since the repository stores information in the form of a *file system tree*, the abstract set used here to model this is *DirSet*, *WrokFiles* and *RevFiles* are defined as subsets of *DirSet*. *TagSet* and *PathSet* are basically standing for the set of tag and path strings that are related with specific files and directories in the repository. *Maxrev* gives the value of the latest revision number, while *Baserev* stands for the revision number of local working copy.

The full injection relation *revTag* corresponds the specified tag string to certain files and directories. *revRev* has the surjection relation on the domain of all the revision numbers, the domain of surjection relation *revPath* contains all the information of legal paths in the repository and *tagRev* maps the full injection relation from tag strings to revision numbers. Relation *revPathSet* denotes the subset of *TagSet*, all available repository path information are kept in this subset.

The abstract constant *InitRev* indicates the initial revision number which will be given to the uploaded files. Besides those identical relations defined in *checkOut*, *revtext*, *worktext*, *workname* and *worktext* are introduced, such that any single file can be found modified or not on the local client side, when the *commit* procedure being carried out. All operations with communications to server will all cause an increment on the number in the database of the repository.

Imagine an array of revision numbers, starting at 0, stretching from left to right. Each revision number has a filesystem tree hanging below it, and each tree is a *snapshot* of the way the repository looked after a commit.

With these elements and relations, the system is able to carry out some basic operations.

INITIALISATION

$$\begin{aligned}
&RevFiles := \emptyset \parallel \\
&WorkFiles := \emptyset \parallel \\
&revTag := \emptyset \parallel \\
&revRev := \emptyset \parallel \\
&revPath := \emptyset \parallel \\
&tagRev := \emptyset \parallel \\
&Baserev := \emptyset \parallel
\end{aligned}$$

```

revPathSet :=  $\emptyset$  ||
workRev :=  $\emptyset$  ||
Maxrev :=  $\emptyset$  ||
revname :=  $\emptyset$  ||
revtext :=  $\emptyset$  ||
workname :=  $\emptyset$  ||
worktext :=  $\emptyset$ 

```

In the **INITIALISATION** part, each element and relation is initialized with \emptyset , so that they reflect the very initial state of the workplace in local system.

OPERATIONS

checkoutRev(*revi*) \triangleq

PRE

revi $\in \mathbb{Z} \wedge \text{RevFiles} \neq \emptyset$

THEN

Baserev := *revi* ||

WorkFiles := *WorkFiles* \cup {*revRev*(*revi*)} ||

workRev(*Baserev*) := *revRev*(*revi*)

END;

checkoutPath(*path*) \triangleq

PRE

path $\in \text{revPathSet} \wedge \text{RevFiles} \neq \emptyset$

THEN

Baserev := *Maxrev* ||

ANY *x*, *y* **WHERE** *x* $\in \text{RevFiles} \wedge y \in \text{RevFiles}$

$\wedge x \in (\text{RevFiles} \cap \{\text{revPath}(\text{Path})\}) \wedge y = x$

THEN *WorkFiles* := *WorkFiles* \cup {*y*}

END||

ANY *x* **WHERE** *x* $\in \text{revPath}(\text{path})$

THEN

workRev(*Baserev*) := *x*

END

END;

checkoutAll \triangleq

PRE

RevFiles $\neq \emptyset \wedge \text{Maxrev} \notin \emptyset$

THEN

Baserev := *Maxrev* ||

```

ANY  $x, y$  WHERE  $x \in RevFiles \wedge y \in RevFiles$ 
     $\wedge x \in (RevFiles \cap \{revRev(Maxrev)\}) \wedge y = x$ 
THEN  $WorkFiles := WorkFiles \cup \{y\}$ 
END||
ANY  $x$  WHERE  $x \in revRev(Maxrev)$ 
THEN
     $workRev(Baserev) := x$ 
END
END;

```

The above given operations are essentially of the same category with different parameters, hence can be optimized into one complex operation. However, since the related parameter types are not so suitable to be grouped together, the command *checkout* are modeled in different specific situations with corresponding parameters, explicitly.

The operation *checkoutRev(revi)* carries out the checkout procedure with the given parameter. The input revision number is related with the files and directories inside the repository with a full injection relation which is modeled as *revRev* in the given machine. In this way, the selected files on the server can be transferred to the local filesystem at the client side. Following the same routine, *checkoutPath(path)* are modeled according to the different parameters given.

The *checkoutAll* operation models the *checkout* operation with default argument assumed, that to check out the files with latest version from the repository server to user client. The value for the latest repository side version is kept in the *Maxver* variable.

```

 $update \triangleq$ 
PRE
     $RevFiles \neq \emptyset \wedge Baserev \notin \emptyset$ 
THEN
    IF  $Baserev \neq Maxrev$  THEN
        IF  $\exists (x, y).(x \in RevFiles \wedge y \in RevFiles$ 
             $\wedge x \in (RevFiles \cap \{revRev(Maxrev)\}) \wedge y = x)$ 
        THEN
            IF  $\exists(x).(x \in DirSet \wedge x = workRev(Baserev))$ 
            THEN
                 $WorkFiles := (WorkFiles - (WorkFiles \cap$ 
                     $workRev(Baserev))) \cup \{y\}$ 
            ELSE
                 $WorkFiles := WorkFiles \cup \{y\}$ 
            END||
             $Baserev := Maxrev$  ||
        ANY  $x$  WHERE  $x \in \{revRev(Maxrev)\}$ 

```

```

    THEN
      workRev(Baserev) := x
    END
  END||
  IF  $\exists (x, y).(x \in WorkFiles \wedge y \in RevFiles$ 
     $\wedge y \in (RevFiles \cap \{revRev(Maxrev)\}) \wedge y = x)$ 
  THEN
    WorkFiles := WorkFiles  $\cup$ ; {x} ||
    WorkFiles := WorkFiles - {workRev(Baserev)} ||
    Baserev := Maxrev ||
    workRev(Baserev) := x
  END||
  IF  $\exists (x, y).(x \in WorkFiles \wedge y \in RevFiles$ 
     $\wedge y \notin (RevFiles \cap \{revRev(Maxrev)\}) \wedge y = x$ 
     $\wedge x \in (WorkFiles \cap \{workRev(Baserev)\}))$ 
  THEN
    WorkFiles := WorkFiles - {x} ||
    Baserev := Maxrev ||
  END
END
END
END
END

```

The *update* operation models the situation that when files are updated at the repository with a newer revision. This contains the three specific situations,

- new files are added to the repository;
- original files are modified and changes have been committed;
- the pervious version of file is removed from the repository;

For the first case, the whole procedure is executed to retrieve the new files to the client. When the files in the previous version are existing in the local working directory, they will be replaced with the new files. Otherwise, this will be more like a default checkout operation. In the second case, the latest version of all files qualified files will be transferred to client; and those have been removed permanently in the repository will be cleaned up from the client working directory.

Import(newFiles, path) \triangleq

PRE

$newFiles \in WorkFiles \wedge newFiles \notin RevFiles \wedge path \notin PathSet$

```

THEN
  RevFiles := RevFiles  $\cup$  {newFiles} ||
  revPathSet := revPathSet  $\cup$  {path} ||
  revRev(InitRev) := newFiles ||
  revPath(path) := newFiles ||
  Baserev := InitRev ||
  workRev(Baserev) := newFiles
END;

```

Imports method modeled in this machine is slightly different from other operations mentioned below, since it provide the function that could DIRECTLY transfer the desired files to the repository with an initial revision assigned, regardless where the files exist in the local file system while other operations are basically based upon the prior modifications in the local working space that corresponds to the repository on the server. The final transfer to the repository will be uniformly implemented by the *commit* operation.

```

Add(files)  $\triangleq$ 
PRE
  files  $\in$  DirSet  $\wedge$  files  $\notin$  WorkFiles  $\wedge$  files  $\notin$  RevFiles
THEN
  WorkFiles := WorkFiles  $\cup$  {files}
END;

```

```

Delete(files)  $\triangleq$ 
PRE
  files  $\in$  WorkFiles
THEN
  WorkFiles := WorkFiles - {files}
END;

```

The operations above are the two of most typical local modification operations, which are also called **scheduled operations**, indicating that they will not affect the structure of the repository on the server unless combined with the **commit** command. When called, *add* method will put any desired files into the working copy repository in the local system of client side, once the *commit* operation is executed thereafter, the modification will be detected and processed to the repository server. Similarly, *delete* try to remove the desired files from the local repository accompanied by the server updated using *commit* command.

```

commitA(files, path)  $\triangleq$ 

```

```

PRE
   $WorkFiles \notin \emptyset \wedge RevFiles \neq \emptyset$ 
   $\wedge files \in DirSet \wedge path \in PathSet$ 
THEN
  IF  $\exists(files).(files \in WorkFiles \wedge files \notin RevFiles)$ 
  THEN
    ANY  $y$  WHERE  $y \in DirSet \wedge y = files$ 
       $RevFiles := RevFiles \cap \{y\} \parallel$ 
       $revRev(Maxrev) := y \parallel$ 
    END  $\parallel$ 
     $Maxrev := Maxrev + 1 \parallel$ 
     $Baserev := Maxrev \parallel$ 
    IF  $\exists(path).(path \in PathSet \wedge workPath(path) = files)$ 
    THEN
       $revPathSet := revPathSet \cup \{path\}$ 
    END
  END
END;

```

$commitD(files, path) \triangleq$

```

PRE
   $WorkFiles \neq \emptyset \wedge RevFiles \neq \emptyset$ 
   $\wedge files \in DirSet \wedge path \in PathSet$ 
THEN
  IF  $\exists(files).(files \in RevFiles \wedge files \notin WorkFiles)$ 
  THEN
     $RevFiles := RevFiles - \{files\} \parallel$ 
     $Maxrev := Maxrev + 1 \parallel$ 
     $Baserev := Maxrev \parallel$ 
     $revRev := revRev \triangleright \{files\} \parallel$ 
     $revTag := revTag \triangleright \{files\} \parallel$ 
     $revPath := revPath \triangleright \{files\} \parallel$ 
    IF  $\exists(path).(path \in PathSet \wedge revPath(path) = files)$ 
    THEN
       $revPathSet := revPathSet - \{path\}$ 
    END
  END
END;

```

In the operations above, *commitA* and *commitD* models the two different operations that could be combined with one or more *add*, *delete* command combination. Specific forms may vary from one implementation to another, whether these two operation models will be

implemented into single complex one.

```

commitM(files, rfiles, path)  $\triangleq$ 
PRE
  files  $\in$  WorkFiles  $\wedge$  path  $\in$  PathSet  $\wedge$  files  $\neq$  RevFiles
   $\wedge$  rfiles  $\in$  RevFiles  $\wedge$  rfiles  $\neq$  WorkFiles
   $\wedge$  Maxrev = Baserev  $\wedge$  workRev(Baserev) = files
THEN
  IF  $\exists$ (name, text revi).(name  $\in$  FileText  $\wedge$  files = workname(name)  $\wedge$ 
    rfiles = revname(name)  $\wedge$  text  $\in$  FileText  $\wedge$  revi  $\in$  RevSet
     $\wedge$  files = worktext(text)  $\wedge$  rfiles  $\neq$  revtext(text))
  THEN
    Maxrev := Maxrev + 1 ||
    Baserev := Maxrev ||
    ANY x WHERE x  $\in$  DirSet  $\wedge$  x = files
    THEN
      RevFiles := (RevFiles - {rfiles})  $\cup$  {x} ||
      revRev(Maxrev) := x
    END||
    ANY y WHERE y  $\in$  TagSet  $\wedge$  revTag(y) = rfiles
    THEN
      revTag(y) := files
    END||
    IF revPath(path) = rfiles  $\wedge$  files = workPath(path)
    THEN
      revPath(path) := files
    END
  END
END

```

The *commitM* gives the model of the operation can be applied to any other local repository modification commands besides *add* and *delete* (especially the direct modification to the contents of the file) updated to the server repository, increasing the corresponding latest revision number by 1. The preconditions for this operation assumes consistency between the target file in repository and the working copy on the client side. In the main process of this operation, if result of the content difference comparison of the two files gives true result(which indicates that difference exist), the file in the server repository will be replaced by the newly modified file from the local working copy on the client side.

In practise, when the *commit/ update* command is executed, the SCM system should be able to tell which of the following four states a working file is in:

- *Unchanged, and current*

The file is unchanged in the working directory, and no changes to that file have been committed to the repository since its working revision. An commit of the file will do nothing, and an update of the file will do nothing.

- *Locally changed, and current*

The file has been changed in the working directory, and no changes to that file have been committed to the repository since its base revision. There are local changes that have not been committed to the repository, thus a commit of the file will succeed in publishing changes, and a update of the file will do nothing.

- *Unchanged, and out-of-date*

The file has not been changed in the working directory, but it has been changed in the repository. The file should eventually be updated, to make it current with the public revision. A commit of the file will do nothing, and a update of the file will fold the latest changes into working copy.

- *Locally changed, and out-of-date*

The file has been changed both in the working directory, and in the repository. A commit of the file will fail with an *out-of-date* error. The file should be updated first; a update command will attempt to merge the public changes with the local changes. If the tool itself can't complete the merge in a plausible way automatically, it leaves it to the user to resolve the conflict.

$makeTag(tag) \triangleq$

PRE

$tag \in TagSet \wedge WorkFiles \neq \emptyset \wedge RevFiles \neq \emptyset$

THEN

IF $revRev(Baserev) \notin \emptyset$

THEN

$tagRev(tag) := Baserev$

END

END

END

For the need of make a snapshot of the project that is still under development or make milestone release, the programmers have to give some or all of the files in the repository in the current revision a special tag, including a special message explains rationale of this very revision number.

$branch(files) \triangleq$

PRE


```

    files ∈ WorkFiles ∧ files ≠ RevFiles
    ∧ workRev(revi) ≠ files
THEN
    IF ∃(name, text, revi).(name ∈ FileText ∧ files = workname(name) ∧
    rfiles = revname(name) ∧ text ∈ FileText ∧ revi ∈ RevSet
    ∧ files = worktext(text) ∧ rfiles ≠ revtext(text))
    THEN
        Baserev := Baserev ∪ InitBRev ||
        RevFiles := ( RevFiles - {rfiles} ) ∪ {x} ||
        revRev(Baserev) := x ||
        revTag(y) := files
    END
END

```

In the process of software development, users will need to make *branches* [4] from the *main* revision tree, especially to maintain the files used in release versions of the software, or used as bug-fix purposes. A branch is a point often item from which on several revisions can exist simultaneously [15]. In the model above, when qualified, the revision related with the selected files will be generated using the latest revision number from the main branch concatenated with the initial revision number of a new branch. Thus the newly submitted files will exist on a branch revision rather than the main, connected to its parent revision number on the main branch.

```

merge(fromFile, toFile) ≜
PRE
    fromFile ∈ WorkFiles ∧ toFile ∈ WorkFiles
    ∧ fromFile ∉ RevFiles ∧ toFile ∉ RevFiles
    ∧ workRev(Baserev) ≠ fromFile ∧ workRev(Baserev) ≠ toFile
THEN
    IF ∃(deltaSetA, deltaSetB).(deltaSetA ⊂ DirSet ∧ deltaSetB ⊂ DirSet
    ∧ {toFile} ∪ deltaSetB = {fromFile} ∪ deltaSetA ∧ deltaSetB ∩ deltaSetA = ∅)
    THEN
        toFile := toFile ∪ deltaSetA ∪ deltaSetB
    END
END

```

The *merge* operation described above models one of the cases when solving the conflicts occur during parallel development in real projects. This is consider to be the ideal situation that files can be merged without manual process, given that the conflicting files do not have intersections in their delta files respectively. The following example decries how the conflict could happen.

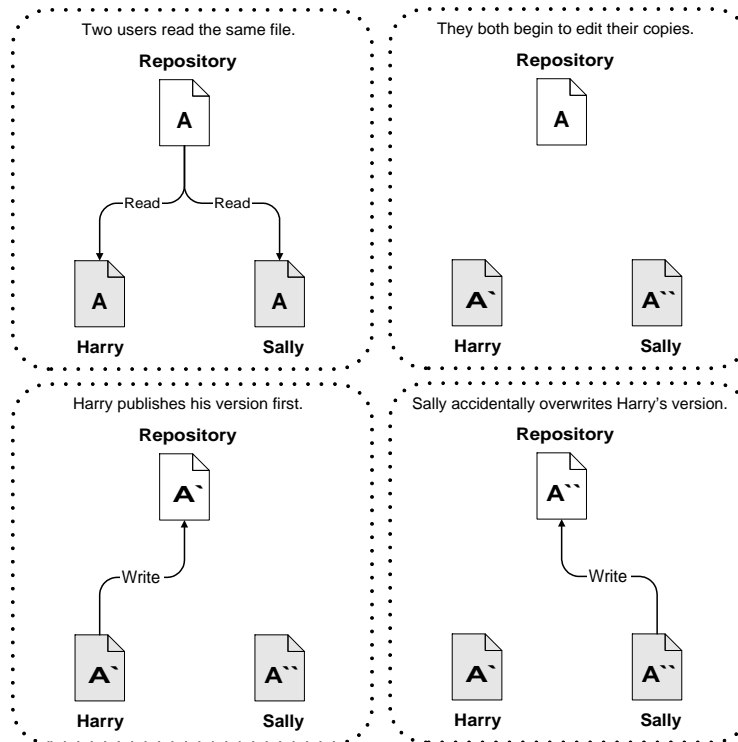


Figure 16.5: The problem to avoid

In the figure 16.5, suppose there exist two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost-or at least missing from the latest version of the file-and probably by accident. This is definitely a situation must be avoided!

In the following section, these abstract models mentioned above will be discussed with the implementation in real practice, coping with many detail questions.

16.3 Implementations of SCM Tools

With those modeled operations in last section, a brief look is given at several software implementations, getting a more specific idea of how the models of software configuration work in practice.

The Checkout/Checkin Model

The first two tools touched, CVS and Subversion are actually belongs to the basic Checkout/Checkin model in SCM. Such SCM systems consist of two relatively independent tools: a repository tool, and a build tool. While the build tool usually is provided by the operating system, the repository tool stores versions of files and provides mechanisms for controlling the creation of new versions.

In this system, the user works with a repository and with the file system. Files are versioned and stored in the repository. Creation of new versions is controlled by the repository tool. Files are, however, not directly accessible in the repository. Users have to retrieve, i.e., check out, a version of a file into the file system in order to access its content. Files can be retrieved for read access, e.g., for the user to examine a design document or for the compiler to access a file that has been included by another file. Files can also be retrieved for write access. In that case, concurrency control mechanisms of the repository coordinate multiple retrieval for modification. Modified files can be stored back into the repository, i.e., check in, resulting in a new version of the file.

CVS (Concurrent Versioning System), as the most widely used software configuration tool in industry, provides most the of of the basic operations that needed in project revision control. CVS started out as a bunch of shell scripts written by *Dick Grune*, posted to the newsgroup `comp.sources.unix` in the volume 6 release of July, 1986. While no actual code from these shell scripts is present in the current version of CVS much of the CVS conflict resolution algorithms come from them [9].

The repository in CVS is implemented to fit a typical `server/client` style. With the directories are history files for each file under version control. The name of the history file is the name of the corresponding file with `,v` appended to the end. There are also several important control files and directories stored in the working directory to keep important information required by CVS, such as `Root`, `Repository`, `Entries`, `Entries.log`, `Tag`, `Base`, `Baserev`, `Entries.Backup`, `Entries.Static`, `Notify`, `Notify.tmp`, `Baserev.tmp`. Information saved in those special files provide all the necessary data for CVS itself to work properly with its revision control function, therefore the operations talked in the models section can be carried out.

Client/server CVS enables disparate developers to function as a single team. The version history is stored on a single central server and the client machines have a copy of all the files that the developers are working on. Therefore, the network between the client and the server must be up to perform CVS operations (such as checkins or updates) but need not be up to edit or manipulate the current versions of the files. Clients can perform all the same operations which are available locally.

Since CVS's built for basic version control functionality, it is one of the systems that look closely to the original models given in last section, it simply but practically maintains a history of all changes made to each directory tree it manages, operating on entire directory trees, not just single files. It supports branching in the revision tree that allows several lines

of development to occur in parallel and providing mechanisms for merging branches back together when desired. CVS has Unreserved Checkouts allowing more than one developer to work on the same files at the same time.

However, CVS is now showing its age through a number of awkward limitations: changes are tracked per-file instead of per-change, commits aren't atomic, renaming files and directories is awkward, and its branching limitations mean that either to better faithfully tag things or there'll be trouble later.

One of the obvious issues is that CVS offers no support for atomic submit neither changelist, in this case, it will be some kind of disaster when transfer failures occur during multiple changes are submitted together, making the half submitted changes unmanageable. Meanwhile, CVS do not has versioning control on directories, resulting impossibilities for refactoring operations such as rename, move, since the history of these file operations must be trackable after they cannot be identified with the original filename or pathname. Moreover, as none SCM tool permits permanent delete in repository, for each time files are /em deleted for the sake of manual rename and move, the abandoned old files produced will pile inside the repository, taking up much unnecessary disk space. These problems led the main CVS developers to start over and create Subversion.

The following gives the list of basic commands that implements the checkout/checkin model.

- **cv`s checkout`**

Create or update a working directory containing copies of the source files specified by modules. Checkout must be executed before using most of the other `cvs` commands, since most of them operate on the local working directory.

- **cv`s update`**

After running checkout to create the private copy of source from the common repository, other developers will continue changing the central source. From time to time, when it is convenient in the development process, the update command can be used within the working directory to reconcile the work with any revisions applied to the source repository since the last checkout or update.

- **cv`s import`**

Use import to incorporate an entire source distribution from an outside source (e.g., a source vendor) into the local source repository directory.

- **cv`s add`**

This commands adds new files to the existing working directory. Before any commands which operate on sandbox files can be used, they must be added to the list of `cvs` controlled files using this command.

- **cv`s remove`**

Remove a file from the working directory, marking the file as *'dead'* which comes into effect after the next commit.

- `cvs commit`

Use `commit` when it is necessary to incorporate changes from working source files into the source repository.

There are also several common CVS operations that are frequently used in revision control, such as `rename`, `history browse`, these are all well illustrated in CVS reference material with detail examples. Though CVS has many short comings compared with other advanced SCM tools and is being slowly replaced by *Subversion*, it is still one of the widely used revision control system for software development.

Subversion In early 2000, CollabNet, Inc.¹³ began seeking developers to write a replacement for CVS. CollabNet offers a collaboration software suite called *CollabNet Enterprise Edition* (CEE) of which one component is version control. Although CEE used CVS as its initial version control system, CVS's limitations were obvious from the beginning, and CollabNet knew it would eventually have to find something better. Unfortunately, CVS had become the *de facto* standard in the open source world largely because there wasn't anything better, at least not under a free license. So CollabNet determined to write a new version control system from scratch, retaining the basic ideas of CVS, but without the bugs and misfeatures.

The original design team settled on some simple goals. They didn't want to break new ground in version control methodology, they just wanted to fix CVS. They decided that Subversion would match CVS's features, and preserve the same development model, but not duplicate CVS's most obvious flaws. Although it did not need to be a drop-in replacement for CVS, it should be similar enough that any CVS user could make the switch with little effort.

On one end is a Subversion repository that holds all of versioned data. On the other end is Subversion client program, which manages local reflections of portions of that versioned data (called *working copies*). Between these extremes are multiple routes through various *Repository Access*(RA) layers. Some of these routes go across computer networks and through network servers which then access the repository. Others bypass the network altogether and access the repository directly.

Like CVS is control schema, Subversion also keeps important repository information and temp files in each folder of the repository, using a hidden directory with name `.svn` [14]. With the control information, the basic operation commands can be carried out.

CVS only tracks the history of individual files, but Subversion implements a *virtual* versioned filesystem that tracks changes to whole directory trees over time. Files and directories are both versioned with a unique single number generated each time a set of changes submitted to the server. That allows the user to simply change the name of desire file as they wish as well as the location in repository, without having such embarrassing problem mention above for CVS. The total repository tree shares only one revision number, which is quite different from other SCM tools. After getting used to it, users will find it much helpful

¹³<http://www.collab.net>

to determine and identify any tiny change to the whole project with such a related unique revision number. This has been proved to have much more advantages than the traditional revision style, especially the whole develop process is carried out with a number of developers rather than one or two people.

A collection of modifications either goes into the repository completely, or not at all. This allows developers to construct and commit changes as logical chunks, and prevents problems that can occur when only a portion of a set of changes is successfully sent to the repository. The cost of branching and tagging need not be proportional to the project size. Subversion creates branches and tags by simply copying the project, using a mechanism similar to a hard-link. Thus these operations take only a very small, constant amount of time. This great and unique technique gives much more improvement on the *advanced* operation, yielding quick response, high performance for large software systems, even much easier in maintainable in the long run.

Besides those commands mentioned above, Subversion also provides many advanced operation commands offering useful services. In the future of software development, Subversion will gradually replace the role of CVS with its improved performance.

Summary The checkout/checkin model focuses on versioning of product components. The operational concepts of checkout, checkin, branch, and merge are low-level primitives for which users have to develop usage conventions to better address their SCM support needs. Examples are conventions for the use of branches, for maintenance of configuration information, and for supporting scopes of visibility of changes. SCM systems primarily supporting this model focus on managing the repository. Support of users in their work areas is limited to component branch locking in the repository as a means of coordinating modifications. In practice, users of such SCM systems have evolved conventions whose patterns emulate some of the concepts found in the other SCM systems.

The Composition Model

The composition model, a natural outgrowth of the checkout/checkin model, relies on the notions of repository and work area, as well as concurrency control through component locking, while it builds on the properties of a component version graph.

A configuration in this model consists of a system model and version selection rules. A system model lists all the components that make up a system. Version selection rules indicate which version is to be chosen for each component to make up a configuration. The selection rules are applied to a system model, selecting a component version, i.e., binding a component to a version. The tools introduced below are typical examples of this model.

ClearCase is the market leader and provides change management functionality in addition to the standard version control. ClearCase is part of a suite of products from Rational¹⁴

¹⁴Commercial product of IBM Corp.

that implement Rationals' *best practices* software development methodology, Unified Change Management.

Unlike the previous two mainly focus on the basic functions on revision control, ClearCase is designed to provide complete solution for the whole process of software development. It manages multiple variants of evolving software systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies, hence fulfills the criteria of a comprehensive SCM system. Therefore, the architecture implementation of ClearCase is far more sophisticated than the simple repository structure.

In the heart of ClearCase is a permanent, secure data repository. It contains data that is shared by all users: this includes current and historical versions of source files, along with derived objects built from the sources by compilers, linkers, and so on. In addition, the repository stores detailed *accounting* data on the development process itself: who created a particular version (and when, and why), what versions of sources went into a particular build, and other relevant information.

ClearCase manages all software development objects: any kind of file, and directories and links, as well. Versions of text files are stored efficiently as deltas, much like CVS or Subversion versions.

ClearCase development data is organized into any number of versioned object bases (VOBs) [13]. Each VOB provides permanent storage for all the historical versions of all the source objects in a particular directory tree. As seen through a ClearCase view, a VOB seems to be a standard directory tree – the *right* versions of the development objects appear, and all other versions are hidden. A version-controlled object in a VOB is called an *element*; its versions are organized into a *version tree* structure, with *branches* and *subbranches*. Users access the ClearCase data repository through *views*. A view is a software development work environment that is similar to – but greatly improves on – a traditional *development sandbox*. Each view can easily be configured to access just the right source data from the central repository. A view is an isolated *virtual workspace*, which provides dynamic access to the entire data repository. The view accesses the appropriate data automatically and transparently.

The magic functions of the views brings a significant difference to the *repository management* of ClearCase. There is no need to copy the versions required for a particular project to a view; instead, the correct versions are accessed dynamically. A particular version of each element is selected according to user-specified rules in the view's *config spec* (*configuration specification*): a file element appears to be an ordinary file; a directory element appears to be an ordinary directory.

The overall effect of automatic version selection is transparency: the version-control system becomes invisible, so that a VOB appears to be a standard directory tree. The following figure 16.6 gives a vivid illustration of all the concepts mentioned above.

Since the operation commands towards the working views are essentially the same with those mentioned and explained in CVS and Subversion, significant differences with the checkout/checkin commands will be introduced. The first will be the ordinary sequence of checkout

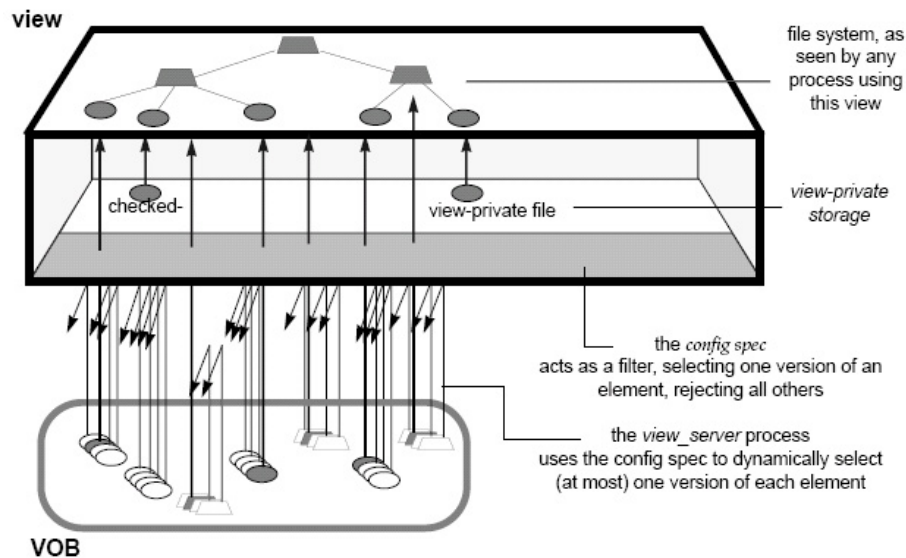


Figure 16.6: Version Selection by a View

and checkin [12].

- In the *steady-state*, an element is read-only – users can neither modify it or remove it.
- To modify an element, a user establishes a view context, then enters a checkout command. This seems simply to change the element from read-only to read-write; in actuality, it makes an editable copy – a checked-out version.
- The user revises the checked-out version using any available system-supplied or third-party tools.
- The user enters a checkin command. This creates a new, permanent version of the element, which then reverts to the *steady-state* of being read-only.

ClearCase also has a mechanism called *reserved/unreserved checkout* to solve the potential conflicts caused by multi-users modifying the same file with identical base revision number. Performing a checkout of a branch does not necessarily guarantee the user the right to perform a subsequent checkin. Many users can checkout the same branch, as long as the users are in different views. At most one of these can be a reserved checkout, which guarantees the user's right to checkin a new version. An unreserved checkout affords no such guarantee. If several users have unreserved checkouts on the same branch in different views, the first user to perform a checkin *wins* – another user must perform a merge if he wishes to save his checked-out version.

ClearCase as a comprehensive SCM system with quite a history, covers almost many important aspects inside SCM applications, which is far more than what CVS and Subversion

in the simple model can give. Especially its rule-based file control and reserved checkout with a true physical distributed server architecture. However, it is not a free licence tool, hence a very important disadvantage of ClearCase is the price, it is too expensive.

Perforce is a popular tool in the academic community, perhaps because the company provides the tool for free to open source efforts Perforce is known for its simple architecture and unique branching model that promotes outwards instead of inwards towards the trees' trunk.

Perforce is based on a client/server architecture. The Perforce server manages the master file repository, or depot. There can be more than one depot per server. The depots contain every revision of every file under Perforce control. Perforce organizes files in depots into directory trees, like a large hard drive. Files in a depot are referred to as depot files or versioned files. The server maintains a database to track change logs, user permissions, and which users have which files checked out at any time. The information stored in this database is referred to as metadata.

Similar with ClearCase's view concept, to control where the depot files appear under client workspace root, *mapping* [16] the files and directories on the Perforce server to corresponding areas of the local hard drive is a must. These mappings constitute the client workspace view. Client workspace views:

- Determine which files in the depot can appear in a client workspace.
- Map files in the depot to files in the client workspace.

Client workspace views consist of one or more lines, or mappings. Each line in the workspace view has two sides: a *depot side* that designates a subset of files within the depot and a *client side* that controls where the files specified on the depot side are located under the client workspace root.

Another specific feature provides by Perforce that yielding faster performance is the concept of *changelist*. Before a file in the workspace can be edited, it must be firstly opened in a *changelist*. A changelist consists of a list of files, their revision numbers, the changes have been made to the files, and a description.

Changelists serve two purposes:

- to organize work into logical units by grouping related changes to files together
- to guarantee the integrity of work by ensuring that related changes to files are checked in together

Like mentioned in previous tools, Perforce also gives strong support in atomic commit and grouped changes. If the user is working on a change to some software that requires changes to three files, open all three files in one changelist. Perforce changelists are atomic change transactions; if a changelist affects three files, then the changes for all three files are committed to the depot, or none of the changes are. Even if the network connection between

Perforce client program and the Perforce server is interrupted during changelist submission, the entire submit fails.

The three-way merge process for resolving file conflicts helps the user to resolve conflicting changes to text files. Because changelists are atomic, the user must resolve every file in a changelist before the submit can succeed. These situations can be resolved in one of three ways:

- **Automatically:** In many cases, whether to accept the changes that are the current (that is, the target revisions in our current user's client workspace) or others (that is, the source revisions in the depot).
- **Accept merged:** Sometimes, there are changes made to the files that are others and ours do not conflict. In these cases, Perforce merges the two files and provides the user with an option to accept the merged result. Such a resolve is referred to as a safe automatic resolve with merging.
- **Manual merge:** Finally, there may be cases where the same lines in others and ours have been changed. Such lines are said to conflict. When changes conflict, Perforce resolves as many differences as possible and produces a merged file containing conflict markers for manual resolution. The user must either edit the merged file manually before submitting it, or accept the merged file with the conflict markers included, and fix the conflict in a subsequent changelist.

There are some special features brought in Perforce that enable the whole system to give more efficient performance, such as the mapping mechanism, changelist and three-way merge. Perforce also has a ClearCase-like distributed server architecture. However, it still has certain obvious defects that make its position located between CVS, Subversion towards ClearCase, especially not providing directory versioning which is used quite often in daily work. Besides, as a commercial software, the expensive licence is another significant factor to be considered.

Summary The composition model operates on configurations by composing aggregates from components and selecting appropriate versions of each. The system structure is captured in a system model. The system model provides the link between configuration support, system build tools, and language systems. This link permits the SCM system supporting the composition model to include management of derived objects and checking of interfaces between components as well as between aggregates, i.e., subsystems. Selection rules provide guidelines for the SCM system to perform version selections. This allows the developer to express selection of alternatives in a natural way. Support for evolution must be expressed in terms of composition with changing selection rules. Support for change migration is limited to the capabilities of change merging at the component level and record keeping by the developer.

The Object-Oriented Model

After focusing the last two major model that have been widely used in the major tools in SCM, another new interesting type is the Object-Oriented version model, particularly brought by VOODOO.

VOODOO

VOODOO Server is a version control system for software developers using Metrowerks CodeWarrior¹⁵ under Mac OS 9 or Mac OS X. VOODOO Server and the corresponding VOODOO clients (e.g., the CodeWarrior VCS plugin and the VOODOO Admin application) are designed to offer reliable and robust version control features while minimizing the administrative overhead that usually accompanies version control.

VOODOO Server is recommended for individual programmers and large developer groups using the Metrowerks CodeWarrior IDE.

Unlike most other version control systems available for Macintosh computers, VOODOO Server and the corresponding clients offer a state-of-the-art client/server architecture [11].

In contrast to most existing approaches, object-oriented version management emphasizes the entire project over individual files. It organizes versions of the whole project, rather than versions of individual files. Variants and revisions are considered orthogonal to each other and an inheritance relationship between variants facilitates project-wide branches and instant access to the required versions without the need for a complicated version access mechanism like multi-digit version numbers or tags.

This small example (figure 16.7) shows that the number and order of branching influences the naming of the different versions of the project's files. The shapes of the trees of file A and B are different. Since the tree structure is used for naming individual software objects, the developer must know different structures in order to retrieve a certain revision of a given variant. E.g., if the user wants to retrieve the most recent versions of files A and B for building the English/Windows variant of the local system, he would have to look for version 1.3.1.2 of file A and version 2.4 of file B. This means that even he's looking for the same variant of the files, the user has to use differently structured version numbers. Since each new branch adds two more digits to the version number, he's faced with version numbers like 1.2.4.2.3.6.2.5.3.4 if there are only four distinctive marks within project.

It be much nicer if the developer would not have to care about all these weird long numbers and just ask the version control tool to retrieve the "latest versions of files A and B for the English/Windows variant". The remainder of this report will demonstrate how the user can do that.

Instead of managing branches of single files only and mixing up variants and revisions, orthogonal version organization emphasizes the whole project and considers variants to be orthogonal to revisions. As a result, all files' versions can be imagined as a three-dimensional

¹⁵Product of FreeScale SemiConductors Inc.
<http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=012726>

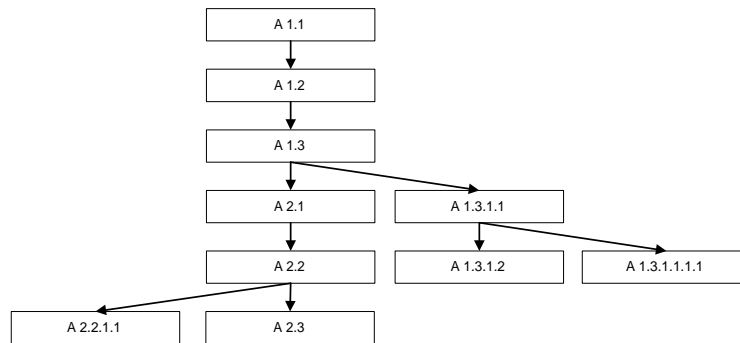


Figure 16.7: The project's version tree in original models

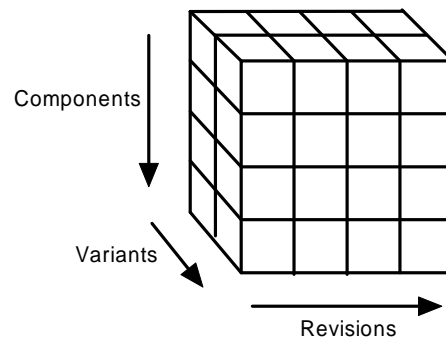


Figure 16.8: Three-dimensional Repository

space whose three dimensions are file, variant, and revision (see Figure 16.8).

It is obvious that this leads to a clear and uniform version organization and every single version can be accessed by just specifying a particular value for each of the three attributes, e.g., file path, variant name and revision id. It is no longer necessary to use long and confusing version numbers to uniquely identify a particular version.

The concept can be explained by means of a small example. First, start a new project with two files and only one variant named main. Suppose the repository contains four versions of FileA and two versions of FileB in the meanwhile. The black circles denote the most recent revisions; historic revisions are shown in gray.

Consider the following simple and concrete example, if define a variant bugfix that branches from main right after t2, bugfix "inherits" the complete history of its base variant from the base date t2 back to the beginning of main. This means that as soon as the variant bugfix defined - even if a version is never checked in into it - particular versions of FileA and FileB can be retrieved from the new variant. Actually, when track back the bugfix variant path to find out which versions are available in variant bugfix, it is obvious to find that versions 1 and 2 of FileA can be retrieved (with v2 being the most recent version available in

variant bugfix) as well as version 1 of FileB. The more recent versions (3 and 4 of FileA and version 2 of FileB) do not belong to bugfix; they can be accessed only via the main variant. Now, if a new revision is checked into variant bugfix, this will overwrite the inherited main version, and from now on newer versions of the files will be kept in the separate branch. Assume that to check in new revisions of FileA for each variant and a new version of FileB for variant main only. For variant main, versions 1 to 5 of FileA and versions 1 to 3 of FileB are accessible (with version 5 of FileA and version 3 of FileB being the most recent versions). For variant bugfix versions 1, 2, and 6 of FileA and version 1 of FileB are accessible (with version 6 of FileA and version 1 of FileB being the most recent versions).

Summary As in the mentioned example of VOODOO, the user don't have to remember multi-digit version numbers in order to distinguish between variants. To retrieve a particular version from the repository, the user can simply specify three attributes: the file name (e.g., FileA), the variant name (e.g., bugfix) and the revision date (e.g., t4 or the special date "now"). Object selection works the same way for all files within the project, and all three attributes are intuitive and therefore easy to manage. This example illustrates the key idea of object-oriented versioning model.

16.4 Concluding Remarks

The discussions in this report on the use of the SCM model in different software process scenarios have shown that the SCM support of a particular system matches certain scenarios well, while others can be supported with limitations by following certain usage conventions. A single SCM system may have difficulties meeting all needs throughout the software process. Besides these tools mentioned in this report, there obviously exist many other requirement-related SCM tools, like [8].

Much of the current research in support for configuration management tends to focus on a particular model and the refinement of its concepts. Given this state of support for software configuration management, two concluding observations can be made. First, Selection on SCM tool for practical use is not only based on the performance of the SCM tool itself, the specific requirements should also be considered. There is not a perfect solution for every software project. Second, there is a need for a unified SCM model that provides a framework for configuration management support. This unified model should be a multi-paradigm model that supports several SCM concepts cooperating in harmony.

16.5 Exam Question

1. Explain why the branching operation in Subversion is cheap and fast.
2. What is the mechanism helps the user using ClearCase to select files from VOB database?
3. Explain why object-oriented model gives more efficient revision control?

Bibliography

- [1] Wayne A. Babich. *Software Configuration Management- Coordination for Team Productivity*. Addison-Wesley Publishing Company, 1st edition, 1986.
- [2] C. Michael Pilato Ben Collins-Sussman, Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media., Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA., 1st edition, June 2004.
- [3] NATIONAL COMPUTERSECURITY CENTER. A guide to understanding configuration management in trusted systems. *NCSC-TG-006-88*, S-228,590, March 1988.
- [4] Darrel Strom Chuck Walrad. The importance of branching models in scm. *IEEE Computer*, 2002.
- [5] Stanley G. Siegel Edward H. Bersoff, Vilas D. Henderson. *Software Configuration Management: An Investment in Product Integrity*. Prentice Hall, Inc., Englewood Cliffs, N.J 07632, 1st edition, 1980.
- [6] Kaisa Sere Emil Sekerenski. *Program Development by Refinement -Case Studies Using the B Method*. Springer-Verlag London Limited, Department of Mathematical and Computing Science, University of Surrey, Guildford, Surrey GU2 5 XH, UK, 1st edition, 1999.
- [7] Jacky Estublier. *Software Configuration Management:A Roadmap*. Dassault Systemes / LSR, Grenoble University, Bat C, BP 53, 38041 Grenoble 9 France.
- [8] Juliana Silva da Cunha Fabio Q. B. da Silva. An nfs configuration management system and its underlying object-oriented model. *Proceedings of the Twelfth Systems Administration Conference (LISA'98)*, December 1998.
- [9] <http://www.cvshome.org>. Concurrent versions system 2.5.03.2260(cvsnt). March 2006.
- [10] Alexis Leon. *Software Configuration Handbook*. ARTECH HOUSE INC., 685 Canton Street Norwood, MA 02062, 2nd edition, 2005.
- [11] Uni Software Plus. Voodoo server tutorial and command reference. 2005.
- [12] John Posner. *Casevision./clearcase user's guide*. 1994.
- [13] John Posner and Jeffery Block. *Casevision./clearcase concepts guide*. 1994.
- [14] Garrett Rooney. *Practical Subversion*. APRESS, 233 Spring Street 6th Floor New York, NY 10013, United States., 1st edition, 2005.
- [15] Emil Sekerenski. *Computer Science 703 Software Design: Custom Courseware*, volume 2 of *Custom Courseware*. McMaster Titles BookStore, 1280 Main Street West, Hamilton, Ontario, Canada., 1st edition, January 2006.

- [16] Perforce Software. Introducing perforce. December 2005.