A Case Study of Software Reengineering

by Harry I. Hornreich

A Case Study of Software Reengineering

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science

Harry I. Hornreich

Submitted to the Senate of the Technion - Israel Institute of Technology
Adar 5758 Haifa February 1998

The work described herein was supervised by Prof. Daniel M. Berry under the auspices of the Computer Science committee.

I wish to thank the Technion for the scholarship it has given me during this thesis

This research is dedicated to my late father Prof. Richard Hornreich

I wish to thank my wife for her great support during this thesis

Contents

Abstract **1**

1 Introduction **3**
1.1 Definitions . . . 3
1.2 Problems in Current Life Cycle Models . . . 5
1.3 Proposed Life Cycle Model . . . 6
1.4 Transition Method . . . 8
1.5 Proposed Transition Method . . . 9
1.6 Thesis Objectives . . . 12

2 The Experiment **13**
2.1 A Case Study . . . 13
2.2 Case Study Mechanics . . . 14
2.3 Case Study Validity . . . 15

3 The ffortid Program **17**
3.1 Background . . . 17
3.2 ffortid Source Files . . . 19
3.3 Why We Chose ffortid . . . 21

4 Domain Software Reengineering of ffortid **22**
4.1 Software Units . . . 22
4.1.1 Software Unit Interface and Side-effects . . . 23
4.1.2 Software Sub-Units . . . 24
4.1.3 Service Flow Diagrams . . . 26
4.2 Reverse Engineering a Software Unit . . . 31
4.3 ffortid Version 3.0 Reverse Engineering . . . 32
4.4 ffortid Version 3.0 Architecture . . . 37
4.5 Author's Conclusions from Decomposition . . . 40
4.6 The Initial Domain . . . 41

5 ffortid Version 4.0 **43**
5.1 SWU Modifications . . . 43
5.2 The New Requirements . . . 45
5.3 Implementation . . . 46
5.4 Implementation Comparison . . . 49

6 ffortid Version 5.0 **53**
6.1 The New Requirements . . . 53
6.2 Implementation . . . 55

7 Experiment Results 57
7.1 Measuring Reuse . . . 57
7.2 Results . . . 58
7.3 Conclusions . . . 60
7.4 Acknowledgments . . . 61

A SFD Icons 62

B ffortid Ver 3.0 Manual Page 66

C ffortid Ver 4.0 Manual Page 70

D ffortid Ver 5.0 Manual Page 77

Bibliography 87

List of Figures

1.1 Relationship between terms 4
1.2 The legacy and reuse software life cycle 7
1.3 Overview of the proposed transition method 9
1.4 Augmented method processes 11

3.1 Stretching connecting letters with a filler 17
3.2 Example ditroff output not piped through ffortid 18
3.3 Same ditroff output piped through ffortid with stretching off 18
3.4 Last connecting letters in lines are stretched 18
3.5 Last connecting letters in lines stretched up to maximum amount . . 19
3.6 Stretch distributed between all last connecting letters in words . . . 19
3.7 ffortid example output with combined English, Hebrew and Arabic text 20

4.1 Example scope diagram 25
4.2 Major icons used in SFDs 27
4.3 A SFD of the SWU abstracting ffortid 28
4.4 SFD of dump.c SWU with its sub-units 29
4.5 A complex SFD 30
4.6 First part of ffortid Version 3.0 SWU 1 page 33
4.7 Third part of ffortid Version 3.0 SWU 1 page 34
4.8 Second part of ffortid Version 3.0 SWU 1 page 35
4.9 SWU 16 page in ffortid Version 3.0 decomposition 36
4.10 Overview of ffortid Version 3.0 decomposition 39

5.1 Connecting letters, fillers, and dynamic letters 45
5.2 Overview of ffortid Version 4.0 domain 50

6.1 Stretchable letter connections and fillers 54
6.2 Layout of slanted font words on line 54
6.3 Sample slanted output 55

7.1 Relationship between original and product application 57

List of Tables

3.1 ffortid version history 19
3.2 ffortid source files 21

4.1 ffortid Version 3.0 software units 38

5.1 SWU modification types 44
5.2 ffortid Version 4.0 software units 51

7.1 Experiment results 58
7.2 Experiment results analysis 59

Abstract

The problem of maintaining and enhancing existing systems has been recognized as a major problem in the field of software engineering. Researchers have proposed solving this problem by organizational changes and methods for systematic software reuse and automatic program generation.

One method for the transition to this futuristic vision of software engineering is the *Synthesis* approach proposed by the Software Productivity Consortium. This approach prescribes an ordered sequence of steps for the management, analysis, and specification of a *domain* which contains the architecture of a product family of reusable software components, and the decision rules needed for their selection. The top-down process of creating the domain is called *domain engineering*. New applications are constructed by selecting components from the domain, as indicated by the decision rules, in a process called *application engineering*.

This approach has so far proved to be very costly and risky. The heavy reliance on committed experts with extensive knowledge both in the application domain and in software engineering has a crippling effect. It requires these experts to build from scratch a library of reusable components to answer every possible application in the domain, a formidable task. Additionally, not a single application can be generated until the complete process is finished.

Ahrens and Prywes have proposed to augment the top-down domain engineering process with a bottom-up *domain reengineering* process. In this process, automated tools extract from good quality legacy code, domain and software knowledge that can be used to define application requirements and their supporting reusable software components. Legacy code becomes a key catalyst of the process and reduces the reliance on domain experts. Combined use of bottom-up reengineering and top-down engineering reduces the time and risk involved in generating new applications.

This work attempts to evaluate the proposed bottom-up domain reengineering process by performing a which-is-better type of case study on a real, small scale, legacy code application. The idea was to perform a controlled experiment of what happens naturally in real life situations. A software product is released and new feedback from users, evolving system architectures, and even competition force the creation of more advanced applications which are based on the original one. However, this can be a painstaking task with legacy applications because they are so difficult to adapt.

There were two roles in the experiment. The author and the control. Both started from the same legacy code application, but used different methods to create two subsequent generations of more advanced applications, each based on the previous generation.

The author used the domain reengineering process, to create an initial domain. He then evolved it in an evolutionary manner, to satisfy the new requirements of the more advanced applications in the domain, generating these new applications in the process. The control used a method, often called seat-of-the-pants, representing currently used maintenance methods that do not use any form of reverse or software reengineering. The control implemented the same two generations of applications

using the same requirements as the author.

The hypotheses of the case study were:

Hypothesis 1: The new method requires less time to produce an application than current maintenance methods.

Hypothesis 2: The new method produces more reusable code than current maintenance methods.

Hypothesis 3: The new method requires less code modifications to produce an application than current maintenance methods.

Hypothesis 4: The new method produces smaller applications than current maintenance methods.

During the experiment data was collected in order to prove or disprove the hypotheses. Special measures were taken in order to insure the validity of the experiment. For example, each generation of applications was tested against the same set of tests to make sure that despite the different methods used to create the applications both had exactly the same functionality, at least with respect to the test data.

The experiment results indicate that only Hypothesis 1, 2, and 4 hold. Only Hypothesis 3 has been disproved in this experiment. Therefore, the experiment shows that the new domain reengineering process does have promise in it. However, additional, formal experiments are necessary in order to strengthen these results. Additionally, it is the author's view that the domain reengineering process cannot be performed successfully on large scale legacy projects without dedicated CASE tools to assist in the process.

The chapters in this thesis follow the course of the experiment steps. The last chapter presents the results of the experiment. Chapter 4 presents conclusions reached on the enabling technology required to make this approach feasible on large scale projects. Additionally, a theory of *Software Units* was developed to assist the reverse and reengineering processes, and the adaptation of a domain according to new requirements. This theory is introduced as applicable throughout the experiment steps.

Chapter 1

Introduction

The problem of maintaining and enhancing existing systems has been recognized as a major problem in the field of software engineering. This thesis deals with state-of-the-art methods for software engineering, such as software reengineering, reverse engineering, domain and application engineering, that are seen as effective, partial solutions to this problem. The widespread use of these and other terms has caused much confusion in the field. The thesis therefore begins with a short section of definitions of the major terms that are used. Other terms will be defined as they are presented.

1.1 Definitions

This section defines and relates the following terms: software maintenance, forward engineering, reverse engineering, redocumentation, design recovery, restructuring and reengineering. The definitions are based mostly on a taxonomy by Chikofsky and Cross [1].

Software Maintenance is defined by ANSI to be the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [2]. There are four types of maintenance activities that can be performed on an existing software product [3]:

- *Corrective Maintenance* is the correction of software faults, i.e. deviations of functionality from specifications after the product has been delivered to the users.
- *Adaptive Maintenance* is the modification of a software system as a result of environmental changes such as new generations of hardware, new peripheral equipment, new operating systems, or new releases of old ones.
- *Perfective Maintenance* is the modification of a software system as a result of new requirements.
- *Preventive Maintenance* is the modification of a software system in order to prolong its lifetime or provide a better basis for future enhancements.

Although the following terms can be applied to any orderly life-cycle model of software development, for simplicity, Chikofsky and Cross define them using only three life-cycle stages:

- Requirements (specification of the problem).
- Design (specification of the solution).

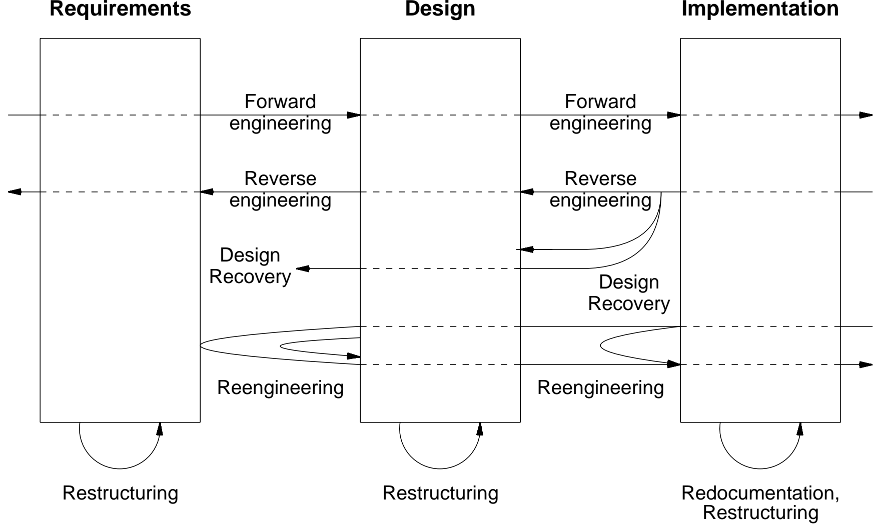


Figure 1.1: Relationship between terms

- Implementation (coding, testing, and delivery of the operational system).

Note that each of these stages is a different abstraction level of the same system. A subject system may be a code fragment, a single program, a complex set of interacting programs, etc. Figure 1.1 shows the relationship between the following terms.

Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent design to the physical implementation of a system. Forward engineering follows a sequence of steps going from requirements through design to implementation.

Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships or to create representations of the system in another form or at a higher level of abstraction. Reverse engineering is a process of examination, not a process of change or replication. Reverse engineering cannot capture all lost information, for example, rejected design alternatives. However, it can discover, for example, side effects that were not planned in the forward engineering process. In general, reverse engineering can be applied at any level of abstraction, or at any life-cycle stage. Two subareas of reverse engineering are redocumentation and design recovery:

- *Redocumentation* is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views, for example, data flow, data structure and control flow, intended for a human audience.

- *Design recovery* is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior. Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard.

Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering followed by some form of forward engineering or restructuring. In most cases, reengineered software reimplements the function of the original system, but at the same time, additional functionality is added or performance is improved in some respect according to new requirements.

Reverse engineering, restructuring and reengineering are usually all performed on existing systems and are therefore, by definition, a form of maintenance. However, each of these processes can be used in the development of new systems or evolutionary system development. Reverse engineering by itself is not maintenance, however it can be used as part of a maintenance effort to help understand an existing system in order to determine the changes needed to be performed on it. Restructuring of an existing system is by definition preventive maintenance. A Reengineering effort can either be adaptive, perfective, or preventive maintenance, or a combination of them.

1.2 Problems in Current Life Cycle Models

Most software today is developed using one, or a combination of, well known life-cycle models such as the waterfall [3], prototyping [3], spiral [4], and what have been called "fourth-generation techniques" [3]. These and other life-cycle models do not adequately represent software maintenance and reengineering activities which today account for the vast majority of software labor costs [5]. Additionally, they do not adequately represent state-of-the-art concepts for improving software engineering practices such as domain and application engineering [6] and software reuse [7].

Ahrens *et al* [5, 8] have identified several underlying assumptions about the nature of software processes and practices that possibly explain these and other inadequacies in current life-cycle models:

Assumption 1: Maintenance is a separate life-cycle phase. The view of maintenance as a separate software life-cycle phase that begins after software is released, originated with the waterfall model and is still widely accepted, as implied by the ANSI definition given above.[1] However, this division is unnatural because the same tasks of requirements analysis, specification, design, implementation, and testing are performed both in new software development and in corrective, adaptive, perfective, or preventive maintenance. Also, reengineering activities such as reverse engineering and restructuring which are traditionally seen as maintenance activities can also be applied during the development stages of new software, either to produce reusable components from existing software or to develop new software. Therefore, the unnatural division between development and maintenance is for administrative purposes only. A CSTB Report [9] states that this assumption seems to have legitimized higher costs, poor technological support, and poor management of maintenance activities. For example, different teams are created for the development and maintenance of the same software project.

Assumption 2: New software applications require new software development. This assumption is widely regarded as obsolete. The notion of reuse and the benefits to be gained from its use are well known [7]. Creating new applications completely or partially from reusable software components holds the prospect

[1] The original view of maintenance was similar to what I defined as corrective maintenance, i.e. correction of faults not found before the software was released. As software systems aged, they were adapted to new environments and perfected to answer new user needs. The adaptive and perfective maintenance terms were coined and added to the only life-cycle phase to which they seemed appropriate – the maintenance phase. As these systems aged even further and became unmanageable a fourth term, preventive maintenance, was added.

of creating higher quality applications more quickly while improving development productivity.

Assumption 3: New applications based on reusable components can and should be developed only in a top-down manner. This assumption can be found in state-of-the-art approaches to software engineering. ARPA's Domain Specific Software Architecture (DSSA) project and the Synthesis approach developed by the Software Productivity Consortium (SPC) [6] both describe an ordered sequence of steps for the management, analysis, and specification of a *domain* which contains the architecture of a product family of reusable software components, and the decision rules needed for their selection. The process of creating the domain is called *domain engineering*. New applications are constructed by selecting components from the domain, as indicated by the decision rules, in a process called *application engineering*. This is a very difficult, time consuming, costly, and therefore risky process. Instead of using a top-down approach to the creation of a domain, it is possible to use a bottom-up approach by reengineering legacy software and selecting from it candidate components for the domain library. Another possibility, is the use of off-the-shelf commercial software components (COTS), which can be modified to fit a certain domain. These alternatives are not disjoint. Part or all of them can be combined according to available resources and analyzed risks.

Assumption 4: CASE technology for forward software development can perform almost all maintenance. This assumption, as presented by Fuggeta [10], implies that software can be maintained more easily by redesigning the software and generating new code than by understanding the existing code prior to its redesign and restructuring. Reverse engineering techniques are seen as necessary only in specific circumstances. This assumption justifies the current CASE orientation towards forward software development. However, such CASE technology leaves out the great potential of using legacy software of which we have little or no design information for the creation of new applications. By using CASE tools that assist reverse engineering and restructuring, we can understand legacy software and harness the sometimes vital information in it for the creation of new applications.

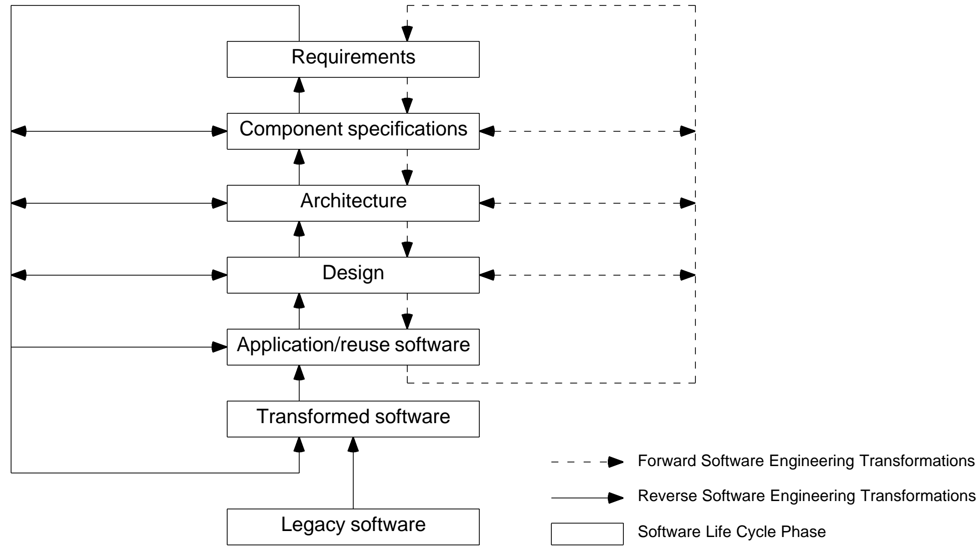
1.3 Proposed Life Cycle Model

The obsolete and erroneous assumptions about software engineering processes and activities in current life-cycle models presented in the previous section have led to the proposal of a more realistic life-cycle model that seeks to incorporate state-of-the-art ideas and technologies. Ahrens *et al* [11, 8] have proposed a new life-cycle model called the "legacy and reuse software life cycle" (LRSLC). In their own words:

> "It is a generalized model of the software life cycle that recognizes explicitly the critical contribution of legacy software to the attainment of software production from reusable software components."

The LRSLC is presented in Figure 1.2. Rectangles represent life-cycle product and information states. Transformation processes, denoted by arrows, convert artifacts in one state to information products in a neighboring state. Forward transformations are represented by dashed lines, reverse transformations are represented by solid lines.

The LRSLC model does not necessarily replace current models; it can be combined with them. For example, the model fits well into the larger scope of the risk-oriented, iterative nature of the spiral model. It can also be combined with other models such as the prototyping model to validate customer requirements. The model does however have some notable features:



Requirements—Domain or aplication software requirements defined in terms of functionality, capabilities, performance, user interface, inputs, and outputs.

Component specifications—Domain or application software requirements specified in terms of capabilities of hardware and software components and interfaces. This state is exemplified by the software specifications in Department of Defense Military Standard 498, ''Software Development and Documentation,'' December 1994.

Architecture—The hierarchy of software components, rules for component selection, and interfaces between components.

Design—Program interfaces, control flow, and logic, defined in greater detail.

Application/reuse software—In application software, a unique software product; in software reuse, a library of adaptable reusable software components. The reuse software components are tested, verified and validated.

Transformed software—Legacy software restructured and translated, if needed, into a modern programming language.

Legacy software—Application software created in a previous traversal of a software life cycle.

Figure 1.2: The legacy and reuse software life cycle

- The model defines the information products of the software life cycle, but leaves the transition processes between them open to various methods. This is similar to the spiral model which also defines the information products produced at the conclusion of a life cycle phase but leaves open the means of their attainment.
- The model integrates forward and reverse engineering processes for traversing the life cycle. Traversal is triggered by new information in one or more states and concludes when all states become consistent. Both forward and reverse engineering traversals can be generated from a single trigger.
- The model specifically incorporates the use of reverse engineered legacy software in the creation of software applications and reuse libraries.
- The model does not have a separate maintenance state. The integrated forward and reverse engineering processes enable the creation, maintenance, and evolution of software domains, reuse libraries, and applications over long time spans. The model is therefore of an evolutionary nature.

As an example use of the model, suppose one has already created a domain and built a first application from it. A customer, having used the application, now has a set of new requirements. To satisfy these requirements we decide to create some completely new components and to reengineer others from the legacy software. First, in a reverse traversal of the model, we reengineer the legacy software by analyzing, possibly translating, restructuring, and redocumenting components from the legacy software, adding them to our domain. We update the design, architecture, and component specification information states from the documentation extracted by the reverse engineering process. Then, forward traversal of the model is used to create the new domain components, updating in the process, the specifications, architecture, design, and reuse information states. Finally, a last forward traversal of the model is used to create the customer's new application by integrating previous, new, and reengineered software components.

1.4 Transition Method

The LRSLC presented in the previous section is a generalized life-cycle model that describes information product states rather than the processes for moving between them. This model is well suited for state-of-the-art software engineering methods aimed at the development of reusable building blocks of adaptable software components from which application software can be constructed. These methods aim to reach a state in which applications in a specific domain can be automatically generated from a library of reusable components according to customer requirements.

However, transition from present software practice to a state of automatic application generation has proved to be very difficult. The proposed transition methods, such as the synthesis approach of the SPC, advocate the creation of the domain from scratch, based on the expertise of domain experts. These experts, based on their knowledge and experience in the domain and in software engineering, define a knowledge base of potential application requirements. Software experts build in a top-down fashion, a library of adaptable, reusable software components to answer these potential requirements. Decision rules and automated processes for the selection and assembly of these components into applications are defined.

The problem with this approach is that their is a large dependency on domain experts. Also, a complete library of reusable components for a large family of applications needs to be created from scratch before a single application can be

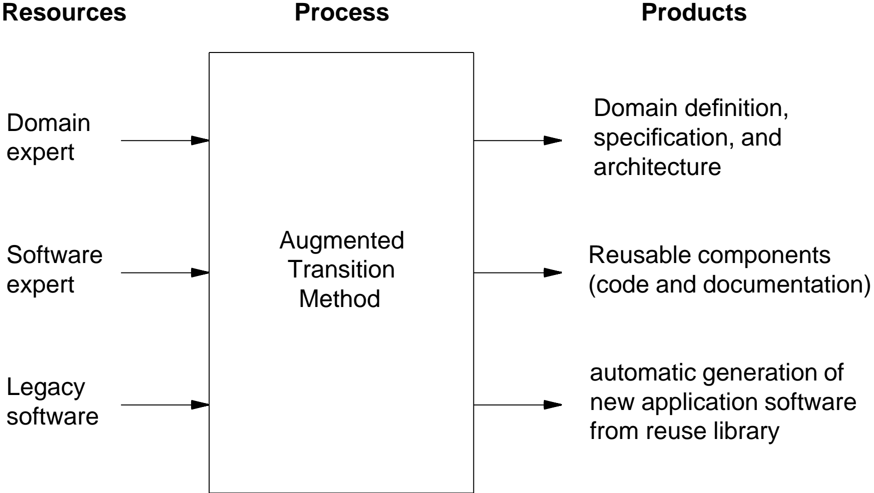


Figure 1.3: Overview of the proposed transition method

generated. This approach has so far proved to be very costly and risky [12, 13]. The initial investment is large and the returns are slow.

Ahrens and Prywes [11] propose a new method for the transition from current software practices to the LRSLC. It is an augmentation of the top-down synthesis method by the use of bottom-up legacy code component and knowledge extraction. Figure 1.3 presents an overview of this approach. Unlike synthesis, legacy software becomes a key resource in the transition process. It reduces the dependency on domain experts which are the bottleneck of the process and with the use of appropriate reverse engineering CASE technology provides a basis for the creation of a library of reusable components.

Only legacy software of reasonable quality and of proven reliable performance is a good candidate for such a process of component extraction. Most legacy software in day to day use answers these requirements. These are large, complex applications which have satisfied their users needs over a long period of time. They are too difficult to maintain and too costly to replace by completely new applications. They are a valuable resource of their organizations and therefore hold invaluable knowledge and code that can be extracted.

1.5 Proposed Transition Method

As described in Section 1.2, synthesis prescribes an ordered sequence of steps for the management, analysis, and specification of a *domain* which contains the architecture of a family of reusable software components, and the decision rules needed for their selection. The top-down process of creating the domain is called *domain engineering*. Following are the major steps in domain engineering:

1. Domain definition
2. Domain specification
3. Domain design
4. Domain verification
5. Domain implementation

6. Domain validation

New applications are constructed by selecting components from the domain, as indicated by the decision rules, in a process called *application engineering*. The following are the major steps in application engineering.

1. Define customer's application software requirements
2. Use rules in decision model to select reusable components
3. Generate application software
4. Test application software
5. Generate application documentation

Ahrens and Prywes have augmented the top-down domain engineering process with a bottom-up *domain reengineering* process that extracts architecture, design, business rules etc. from legacy software. The major steps in domain reengineering are

1. legacy application analysis and translation,
2. legacy application conversion to new hardware, operating systems etc,
3. augmentation and adaption of reusable components,
4. domain validation,
5. domain design update,
6. domain specifications update,
7. domain definition update, and
8. domain verification.

Figure 1.4 illustrates the augmented method which includes both processes. Application engineering in the augmented method is the same as in synthesis. Either process can be used to create the initial domain repository. The feedback loop shown in Figure 1.4 shows that both top-down and bottom-up processes can be interleaved and applied iteratively, incrementing the domain with each application of the process. Note that using the two processes in a different sequence will not necessarily lead to the same reuse library.

When the top-down process alone is selected, it is driven by iterations for designated domain areas, after which application software may be obtained from these partial domains. In later iterations, smaller additions to the domain are needed to produce software for a new application. When combined top-down and bottom up processes are selected, they are driven by iterations for extracting reusable legacy applications to produce domain increments. For example, first a top-down process is used to define a high-level architecture. Then a bottom-up process is interleaved for filling in the detailed architectural levels.

We can assume that the top-down approach by itself will require significantly more time than the combined approach to complete the first domain increment of reusable software components for an application for two reasons. First, the top-down approach requires more input from human domain experts. Second, the synthesis method requires the complete domain to be specified before applications are produced. However, the top-down approach by itself has an advantage when

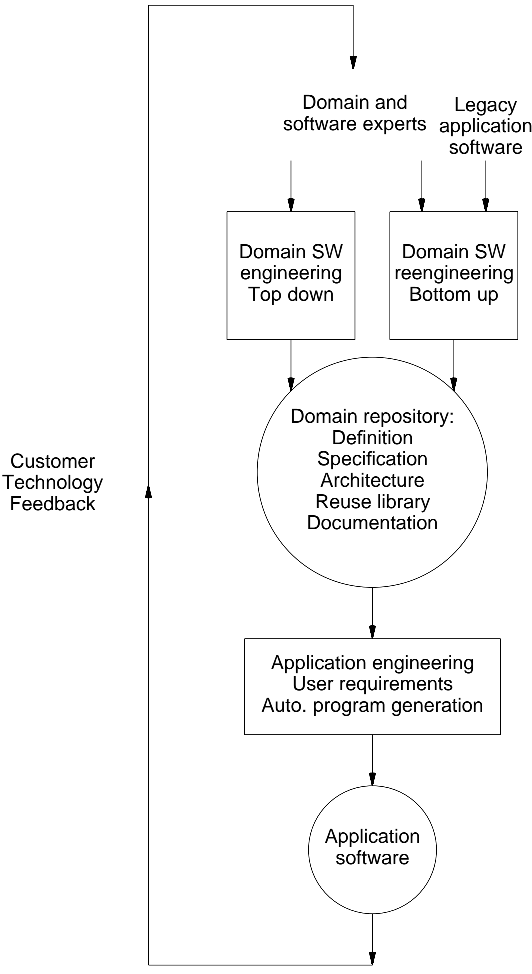


Figure 1.4: Augmented method processes

developing a domain for which there is sufficient domain expertise but no legacy applications or when the domain is not overly complex and can be defined manually.

The combined approach can more quickly add components from a legacy code application to the domain architecture, leading to faster and less expensive production of new software applications than the top-down approach. The combined approach also reduces reliance on the scarcer resource of domain experts by relying more on software experts extracting domain knowledge embedded in good legacy software. In summary, the combined approach presents an alternative for a faster and more economical transition to the LRSLC model.

Ahrens and Prywes emphasize the importance of an enabling technology to make their approach practical. Automated tools complement and help the cognitive effort required on the part of the software and domain experts in the domain and application engineering phases. They are especially important in the effort required to understand the legacy software in the processes.

1.6 Thesis Objectives

The objectives of this thesis are

- to evaluate the proposed domain reengineering process,
- to reach conclusions on the required enabling technology for the reengineering process,
- to develop a method for the evolutionary development of a domain according to external requirements,
- to reach conclusions on the required enabling technology for the new method, and
- to refine the theory of *Software Units* to support the above processes.

This thesis focuses on the bottom-up domain reengineering transition process because of the limited time and resources available in a masters thesis. Domain reengineering is used to create an initial domain and then to create two generations of applications from the domain, updating it in the process. The issues of creating adaptable components and the automatic generation of applications from them, are beyond the scope of this thesis. All processes were performed manually without the use of any automated tools.

The following chapters explain the course of the experiment and follow its steps, introducing the theory of Software Units as applicable to the problem. Finally, the thesis summarizes the experiment results and draws conclusions from their analysis.

Chapter 2

The Experiment

2.1 A Case Study

In order to achieve the objectives of this thesis, it was decided to conduct a *case study* [14]. In general, a case study can show the effects of a technology or method in a typical situation, but cannot be generalized to every possible situation. Although case studies are not as scientifically rigorous as *formal experiments* [14], they can provide us with sufficient information to judge if a method has any promise in it. It is not claimed that this case study gives a definite answer or proof as to the usefulness of the new method. It does, however, attempt to show that the new method is applicable to a real application domain and that one can produce quality applications using it. The intention is that this case study serve as the basis for further study either by additional case studies or by a fully controlled formal experiment. Such formal experiments are very difficult to perform, especially in the field of software engineering, and require careful planning and large resources.

This case study is a "which is better" type of case study in which the author wanted to examine which is better, the new method for legacy and code reuse or the common and often used seat-of-the-pants (SOTP) maintenance. SOTP maintenance does not mean maintenance with no method in it. The maintainer can indeed have a clear method for performing modifications to the software. However, such a method does not involve any form of reverse engineering or reengineering.

In order to perform a successful case study, we must have well defined hypotheses. The hypotheses are:

Hypothesis 1: The new method requires less time to produce an application than current maintenance methods.

Hypothesis 2: The new method produces more reusable code than current maintenance methods.

Hypothesis 3: The new method requires less code modifications to produce an application than current maintenance methods.

Hypothesis 4: The new method produces smaller applications than current maintenance methods.

2.2 Case Study Mechanics

Application of the new method was performed by the author. The control of the experiment was Daniel Berry who applied his own systematic SOTP maintenance method. We believe that this method is representative of the maintenance methods used by most programmers that do not apply any form of reverse or reengineering. Both the control and the author worked on similar UNIX systems and neither used any CASE tools. All work was done manually with the help of some common UNIX commands such as `grep`. The case study followed the following steps:

1. A valid legacy code program P was selected as the pilot.
2. The author domain reengineered P and created an initial domain architecture and reusable components.
3. A set of requirements R' was devised for a new version of P.
4. The control and the author each created individual implementations of P' according to the requirements R'. The author used his own method for the evolutionary development of a domain according to new external requirements to create his new version of P' using the initial domain as his basis. The control used his own systematic SOTP method of maintenance to create his new version of P' using P as his basis.
5. Both implementations of P' were tested against the same set of tests to make sure they had implemented correctly the requirements R', and therefore had the same functionality.
6. A second set of requirements R'' was devised for a new version P''.
7. Again, the control and the author each created individual implementations of P'' according to the requirements R'', each using his own method.
8. Both implementations of P'' were tested against the same set of tests to make sure they had implemented correctly the requirements R'', and therefore had the functionality.

The following measurements were collected during the experiment in order to validate or invalidate the experiment hypotheses:

- Each recorded the number of implementation hours for each application version and for each method.
- Each recorded the number of added, deleted, and modified code lines for each application version and for each method.

The case study was built to follow the steps of a typical software project in which one has a legacy code program of which one has very little knowledge, but must create new versions of the program to satisfy new user requirements. The first method to handle this problem is to use traditional SOTP maintenance. The second method is to reverse engineer the application, discovering its architecture and components and documenting them. A domain is created, storing all this knowledge and reusable components. New application requirements are satisfied by updating the domain in an evolutionary manner by improving, adding, and deleting reusable components as necessary and creating new applications from these reusable components according to the rules in the domain.

The requirements for both new versions were not known to the author before he had reached the stage were he had to know them. This is just as in real software

projects in which the developers of an application do not usually know beforehand what are the requirements for the next application version. The performance of two requirement cycles is really necessary in this experiment because only by implementing the second set of requirements can the two methods used in implementing the first set of requirements be compared.

The actual course of the experiment was very similar to the steps described above. The difference was in the timing of the steps of the control. The actual legacy program that was selected for the experiment was one of which the control had already created version P' for his own purposes before the experiment had begun. This was an advantage to the experiment because less effort would be required by the control, and was in no way an impediment to it. It did however mean that we could not compare the implementation hours for version P' because the control did not record these. This is not really a problem because even if we could collect these hours for version P' it would be wrong to compare them for both methods because the author was learning and developing his method during this step and therefore the hours measured would not reflect only the version implementation time.

2.3 Case Study Validity

Performing case studies correctly so that they have valid results requires careful planning. Several steps were taken to insure the validity of the experiment:

1. A typical legacy code program was selected to be the pilot program.
2. The pilot program for the experiment was selected to be one of which the author had no previous knowledge.
3. The author had no knowledge of the first and second sets of requirements before he reached the steps in which he needed to know them.
4. Only discussion of the requirements themselves was allowed between the author and the control. Neither discussed his method or encountered implementation problems with the other.
5. Similar implementation versions were compared against the same set of tests before proceeding to the next stage in order to make sure they have both implemented the same functionality. Each devised his own test cases and both programs were tested against both sets of test cases.

As any experiment in software engineering that involves several programmers, a possibly wide difference in the programmers capabilities can undermine the validity of the complete experiment. It is necessary to examine carefully how such a difference, if any, can affect the experiment.

For example, a 1965 experiment to show that interactive programming is more effective than batch programming failed to produce significant results because the effect of the independent variable, batch versus interactive programming, was drowned out by individual differences in programmers of equal experience. One programmer was found to be 28 times more effective than another programmer of equal experience [15].

In this case both the author and the control are experienced programmers in the language of the program, C, and both come from a strong programming background. Although it cannot be determined who is the better programmer, the control has some clear initial advantages over the author:

- The control has 29 more years of programming experience.

- The control has a much deeper understanding of the text processing system of which the selected program is a part, than the author, who had absolutely no such understanding before the experiment. The control had been involved since 1983 in writing and correcting programs in this text processing system.
- The control was the client and worked with all the authors of the previous versions of the legacy program. He also fixed some of the bugs found in the program from time to time. He therefore has a clear initial advantage in the understanding of the program. Needles to say, the author had absolutely no knowledge of the program, its function, or its source code before the experiment.
- During the course of the experiment, the control had prior knowledge of the next version's requirements because he was their initiator. The author learned of these requirements only when the requirements document, the manual page, was written by the control just before the start of programming.

Taking the above into consideration, it is claimed that if the experiment shows a clear advantage in the use of the new method over the SOTP method, then indeed there is promise in the method and it is worthy of further study. If, however the results are inconclusive or with a clear advantage to the current maintenance method then, nothing can be concluded.

It must be emphasized however, that even if the new method shows a clear advantage over the SOTP maintenance method, it is still possible that this is because the author is a better programmer than the control **or** that the author is a better programmer and the method he used is better. Therefore, in any case, further case studies or formal experiments are required to validate the results of this experiment.

Chapter 3

The ffortid Program

3.1 Background

ffortid [16, 17] is a UNIX ditroff [18, 19] (Device Independent Typesetter RunOff) post-processor. When combined with ditroff and its various pre-processors, it creates a formatting system that is able to format multilingual scientific documents, containing text in Hebrew, Arabic, or Persian, as well as other right-to-left languages, plus pictures, graphs, formulae, tables, bibliographical citations, and bibliographies.

ffortid takes as input ditroff output which is formatted strictly left-to-right, finds occurrences of text in a right-to-left font, such as Hebrew or Arabic, and rearranges each line so that the text in each font is written in its proper direction. Additionally, ffortid left justifies lines containing Arabic, Persian, or related languages by *stretching* instead of inserting extra white space between the words in the line. The stretching is achieved by inserting one or more filler characters between the last connecting letters of lines or words. Figure 3.1 (a), (b), and (c) show how a filler is inserted between pairs of connecting letters.

Figure 3.2 shows the ditroff output of an example combining Arabic, Hebrew, and English text. Figure 3.3 shows the same output after it is piped through ffortid with stretching turned off. Note how the text in Arabic and Hebrew has been reversed in-place, and justification of the lines is achieved by extra spaces inserted between the words. Different styles of stretching can be achieved in ffortid by using one of several stretch options. Figures 3.4, 3.5, and 3.6 are examples of the different stretch styles of ffortid. In Figure 3.4 connections to last connecting letters in lines are stretched. In Figure 3.5 connections to last connecting letters in lines are

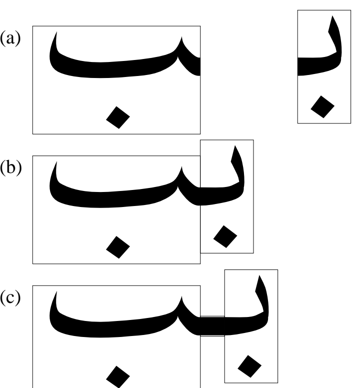


Figure 3.1: Stretching connecting letters with a filler

ةغللا فيفصتو ةعابطل لاثم اذه
ىرخأ تاغل عم ةيوس ةيبرعلا
.(תירבע) ةيربعلاو (English) ةيزيلكنلأاك
لك نيب قرفلا حضوت ةاطعلا ةلثملأا
دمو فيفصتلا بيلاسأ نم دحاو
.تاملكلا

Figure 3.2: Example ditroff output not piped through ffortid

هذا مثال لطباعة وتصفيف اللغة
العربية سوية مع لغات أخرى
كالأنكليزية (English) والعبرية (עברית).
ألأمثلة المعطاة توضح الفرق بين كل
واحد من أساليب التصفيف ومد
الكلمات.

Figure 3.3: Same ditroff output piped through ffortid with stretching off

stretched to a maximum amount, with any remainder going to preceding words. In Figure 3.6 the stretch is distributed between all the connections to last connecting letters in words in a line.

Figure 3.7 is the first page of a technical report [17] describing ffortid and is an example of ffortid output with combined English, Hebrew, and Arabic text. Note how the Arabic text at the bottom third of the page is left and right justified by the third style of stretching.

The first author of ffortid was Cary Buchman, an M.Sc. student at UCLA, and the first version was written during the years 1983-1984. That version could handle only Hebrew although it did have some hooks for Arabic that proved to be useless

هذا مثال لطباعة وتصفيف اللغـــــــة
العربية سوية مع لغات أخـــــــــــرى
كالأنكليزية (English) والعبريـــة (עברית).
ألأمثلة المعطاة توضح الفرق بين كــل
واحد من أساليب التصفيف ومـــــــد
الكلمات.

Figure 3.4: Last connecting letters in lines are stretched

هذا مثال لطباعة وتصفيف اللغة
العربية سوية مع لغات أخرى
كالأنكليزية (English) والعبرية (עברית).
ألأمثلة المعطاة توضح الفرق بين كل
واحد من أساليب التصفيف ومد
الكلمات.

Figure 3.5: Last connecting letters in lines stretched up to maximum amount

هذا مثال لطباعة وتصفيف اللغة
العربية سوية مع لغات أخرى
كالأنكليزية (English) والعبرية (עברית).
ألأمثلة المعطاة توضح الفرق بين كل
واحد من أساليب التصفيف ومد
الكلمات.

Figure 3.6: Stretch distributed between all last connecting letters in words

for later versions. The first external customer was the Hebrew University (HU). Mulli Bahr, a UNIX guru from HU, modified the code to optimize the output in 1986 during a visit to UCLA. Johny Srouji, an M.Sc. student at the Technion, extended ffortid for Arabic stretching during 1989-1991. Table 3.1 summarizes the different versions of ffortid.

The ffortid program described above is ffortid version 3.0. The complete manual page of ffortid version 3.0 can be found in Appendix B.

3.2 ffortid Source Files

ffortid was written in C. It is composed of 11 different source files, 5 of which are .c files, 1 of which is a lex file, and 5 of which are .h files. Table 3.2 shows all the source files with their respective number of lines and number of functions. Each of the 5 .c files is compiled separately to create a module. lex.dit is the lexical parser definitions file. The UNIX lexical parser generator lex takes lex.dit as input and generates from it a lexical parser source file which is included into main.c. This parser is used to parse the input to ffortid into tokens.

Version	Years	Author	From	Major Modification
1.0	1983-1984	Cary Buchman	UCLA	Hebrew
2.0	1986	Mulli Bahr	HU	Output Optimization
3.0	1989-1991	Johny Srouji	Technion	Arabic

Table 3.1: ffortid version history

TECHNION TECHNICAL REPORT, MARCH 1993)

Arabic formatting with ditroff/ffortid

JOHNY SROUJI (جوني سروجي, ג'וני סרוג'י) AND DANIEL BERRY (دانيال بيري, דניאל ברי)

Computer Science Department
Technion
Haifa 32000
Israel

SUMMARY

This paper describes an Arabic formatting system that is able to format multilingual scientific documents, containing text in Arabic or Persian, as well as other languages, plus pictures, graphs, formulae, tables, bibliographical citations, and bibliographies. The system is an extension of ditroff/ffortid that is already capable of handling Hebrew in the context of multilingual scientific documents. ditroff/ffortid itself is a collection of pre- and postprocessors for the UNIX ditroff (Device Independent Typesetter RunOFF) formatter. The new system is built without changing ditroff itself. The extension consists of a new preprocessor, fonts, and a modified existing postprocessor.

The preprocessor transliterates from a phonetic rendition of Arabic using only the two cases of the Latin alphabet. The preprocessor assigns a position, stand-alone, connected-previous, connected-after, or connected-both, to each letter. It recognizes ligatures and assigns vertical positions to the optional diacritical marks. The preprocessor also permits input from a standard Arabic keyboard using the standard ASMO encoding. In any case, the output has each positioned letter or ligature and each diacritical mark encoded according to the font's encoding scheme.

The fonts are assumed to be designed to connect letters that should be connected when they are printed adjacent to each other.

The postprocessor is an enhancement of the ffortid program that arranges for right-to-left printing of identified right-to-left fonts. The major enhancement is stretching final letters of lines or words instead of inserting extra inter-word spaces, in order to justify the text.

As a self-test, this paper was formatted using the described system, and it contains many examples of text written in Arabic, Hebrew, and English.

مقدمة

هذا المقال يصف برنامج لتوضيب اللغة العربية والذي يمكن مـــن
توضيب نصوص علمية متعددة اللغات ، محتوية على نص بالعربيـة
والفارسية بالاضافة للغات اخرى ، رسومات ، رسومات بيانيـــة ،
جداول ، مصادر بيبليوغرافية ، وبيبليوغرافيا . ألبرنامج هو تحسيـن
ل-ditroff/ffortid القـادر الآن عـلى معالجـة العبريـة في وثائـق متعـددة
اللغـات . ditroff/ffortid عبـارة عـن قبـل معالـج (preprocessor) وبعـد معالــج
(postprocessor) لبرنامـــج الصـــف في UNIX ، ditroff (Device Independent Typesetter

Received 10 July 1992
Revised 1 February 1993

© 1993 by Johny Srouji and Daniel M. Berry

Figure 3.7: ffortid example output with combined English, Hebrew and Arabic text

Num	File	Size (lines)	Functions
1	lex.h	30	-
2	lex.dit	37	-
3	token.h	34	-
4	macros.h	20	-
5	connect.h	256	-
6	table.h	18	-
7	dump.c	704	10
8	lines.c	296	6
9	main.c	506	1
10	misc.c	129	5
11	width.c	480	10
Total		2510	32

Table 3.2: ffortid source files

3.3 Why We Chose ffortid

ffortid was chosen as the pilot in the case study. As described in section 2.2, there are two major criteria for selecting a program as a pilot. It should be a typical legacy code program, although perhaps on a small scale, and it should be possible to conduct an unbiased experiment using it. ffortid is a typical legacy code program because,

- it has been written over a long time span (9 years), by several different authors (3), and had several versions (3). All of the original authors were busy with their own lives, and therefore none of them were approached for help in understanding the design and architecture of the program,
- it is in working condition and in current use,
- there are no original design documents; there are some documents describing the program's external use and general underlying algorithms and motivation, but none of these documents actually describe the program's design or architecture,
- the program is reasonably well commented, although certainly not fully commented, and
- it is a real program, answering a real need, and it has real users.

ffortid is a good candidate program for experimentation because

- it is reasonably sized, with 2510 source lines, not too small to be considered a toy program and not too large for experimentation within the normal time span of a thesis,
- the author had no previous knowledge of the program; he had never used it or seen its code before the experiment; in fact, the author also had no prior experience in using the ditroff text processing system, and
- the control had already written a new version of ffortid using conventional maintenance methods; this saved some work in the experiment without affecting its results.

For all the above reasons, ffortid was considered a suitable program for the experiment. The only issue which is not addressed in this analysis is the issue of scale. This issue will be addressed in Section 4.5 and in Chapter 7 describing the experiment results.

Chapter 4

Domain Software Reengineering of ffortid

The previous chapter described and justified the selection of ffortid as the legacy application on which to base the case study. The next step in the experiment, as described in Section 2.2, is the creation of an initial domain from ffortid. This domain can be defined as the family of ditroff post-processors that can rearrange text in a right-to-left font so it is written in its proper direction and can stretch Arabic text so it is left and right justified on the line.

As the author had no previous knowledge of the ditroff text processing system or the specific domain before the beginning of the experiment, it was only natural to use bottom-up domain reengineering to extract the knowledge and code that already exists in ffortid about the domain. He used a method of reverse engineering to discover the architecture and design of ffortid. The method calls for the decomposition of ffortid into abstractions called *software units*. These software units will be the basis of the domain's reusable components library. The following section defines and describes the attributes of software units.

4.1 Software Units

A software unit (SWU) is a well-defined component of a software system, that provides one or more computational resources or services.

This is a definition of what most refer to as software components or modules.[1] However, SWUs are more general than modules. Any software module is by definition a SWU, but the SWU definition includes software components which would generally not be regarded as modules. For example, a single statement, a block of statements, a function, an object oriented class, a single definition and a group of declarations are all SWUs but would conventionally be considered too small to be modules. On the other hand, a complete program would not generally be considered a module, but it is a SWU under this definition.[2]

The SWU concept gives a uniform view of software. It crosses traditional boundaries of scale, language, storage medium, programming or design technique. It can be applied successfully to any software in any language because it captures the essence of software, to provide computational services. It can be applied equally successfully to machine languages, procedural languages, functional languages, or

[1] Not necessarily compilation units as in C.

[2] Here the software system of which the program is a component is the operating system environment or alternatively any other program which can invoke it.

fourth-generation languages. It can be applied to any software using any programming or design paradigm: functional decomposition, OOP etc. Therefore, the SWU concept and all the techniques described shortly are applicable to any software.

A SWU provides computational services, including resources, to other SWUs or to an external user of the software system. It can even provide services to itself, as in recursion. A SWU can either depend on other SWUs to provide its services or be *stand alone*. Clearly, the most basic SWUs in a software system will be stand alone, however, at least some SWUs must cooperate with other SWUs to provide their services or else, we will be left with a collection of low-level service providers. Every SWU has a scope, capabilities, interface, requirements, and type:

- The *scope of a SWU* is the body of code which it abstracts. The scope does not have to be contiguous.
- The *capabilities of a SWU* are the services it provides.
- The *interface of a SWU* is a description of how its services can be accessed by its clients, i.e., other SWUs or an external user, and how these services affect or might affect other SWUs.
- The *requirements of a SWU* are the services it needs or depends upon in order to provide its own services.
- The *type of a SWU* categorizes the SWU into one of several types of similar service providers.[3]

Note that the type of a SWU should reflect the kinds of services it provides and not the medium in which it is organized or stored. In some languages the name of the storage medium is also the name of the type. For example, in C, a file is both a storage medium and a type of SWU.

The *environment of a SWU* is all the software in the context in which it is used which is not in its scope. The environment of a SWU therefore depends on the context in which it is used and is different for each use.

When we wish to use or reuse a SWU in a software project, we are mainly interested in its capabilities, interface, and requirements. Its capabilities tell us what services it can provide our project. Its interface tells us how we can access these services and in what way, if any, do these services affect the rest of the SWUs in the project. Its requirements tell us what other services must already exist or be added to our project if we want to use this SWU. Its requirements can even decide the method by which the SWU will be included in the project.

If we wish to create a reusable component library, the above information should be all we need in order to make a successful reuse library. This information should be documented for each SWU in such a way that it will be easy for the potential user to find the needed SWU, and once found, to know how to include it into his software project, how to access it, and how it might affect the rest of the software in the project.

4.1.1 Software Unit Interface and Side-effects

Every SWU provides a set of services. We can divide the interface of each service into its *access interface* i.e. how the service is accessed or initiated and its *result interface* i.e. how the service results, if any, are returned.

[3] The different types are decided upon by the decomposer and are language dependent. Example types in C are: function, procedure, declaration, definition, groups of the above, file, module and program.

A *resource* is a SWU service that does not have a result interface, for example, the definition of a new type. Other SWUs that include this definition, via the access interface, can use the newly defined type. However, this inclusion does not generate any result and therefore this SWU has no result interface. Resources are usually definitions or declarations that do not provide computational services.

The result interface itself can be divided into 2 parts, results returned through the access interface of the service and results not returned through the access interface. The latter are called *side-effects*. In other words, a SWU service side-effect is a change in the SWU environment which is not clearly stated or visible in the service's access interface.

The problem with side-effects is well known. They cause SWUs to be dependent on each other in a non-clear fashion. Once a side-effect is generated, it can propagate through a system and cause unexpected results. Therefore, side-effects are something we generally wish to avoid.

However, not all side-effects are bad. Some of them are intentional or unavoidable. For example, a printing function that prints to a fixed stream will always have a side-effect, the printing, but this is intended and documented. If the stream to be printed is passed as a parameter to the printing function, then it is a matter of interpretation if the printing is or is not a side-effect. The hard-liners would argue that since the printing is an effect outside the environment of the program, to a permanent file, for example, this is still a side-effect. The soft-liners would view the stream parameter as representing the stream and would therefore argue it is not a side-effect. Personally, I believe the hard-liners are more precise in this case. Some examples of things that are, and are not, side-effects:

1. The value returned by a function call is not a side effect because the fact that a value will be returned is clearly stated in the function definition, the access interface.

2. Accessing an external variable and reading its value is not a side effect because their is no change in the value of the external variable. Note however, that a declaration of the external variable is necessary for the SWU to function correctly, but this does not affect the access interface of the SWU, just as the call in the body of one function to another does not add the second function to the access interface of the first. The external variable and the second function are part of the requirements interface of the SWU because they are necessary for it to provide its own services.

3. Memory allocation or deallocation inside a function is a side effect unless the allocated memory is deallocated before the function returns.

4.1.2 Software Sub-Units

The SWUs $s_1, \ldots, s_n$ are the sub-units of a SWU S if and only if the following two equations hold:

$$scope(S) = \bigcup_{i=1}^{n} scope(s_i) \tag{4.1}$$

$$\forall\, 1 <= i, j <= n, i \neq j \;\; scope(s_i) \bigcap scope(s_j) = \emptyset \tag{4.2}$$

Every non-trivial SWU can be decomposed into its software sub-units. A sub-unit is a SWU in its own right. The scope of each of the sub-units must be mutually exclusive and the union of the scopes must be equal to the scope of the parent SWU. As with SWUs, the scope of each sub-unit does not have to be contiguous.

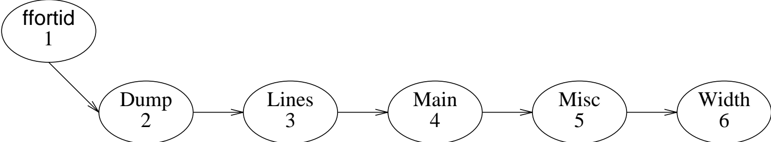


Figure 4.1: Example scope diagram

The sub-units of a single SWU do not all have to be of the same type or be recorded in a certain order, such as their scope order. However, the sub-units should be composed in a defined manner in order to create the parent SWU.

The decomposition of a SWU into its sub-units is not unique, and is dependent on a partitioning criteria provided by the decomposer. Figure 4.1 shows an example *scope diagram*, which is a graphical description of the decomposition of a SWU into its sub-units. It shows that a SWU named ffortid is decomposed into 5 sub-units. Each SWU in the diagram has a name and an identification number.

Section 4.2 examines partitioning criteria for SWUs. There is however, one rule which must be followed universally. This rule states that it is not desirable for a SWU to have more than 7 sub-units. The reason for this is purely psychological. The human brain has difficulty understanding more than 7 clusters at the same time [20], and having too many sub-units would therefore impede the understanding of the architecture of the SWU. If one does have a natural decomposition into more than 7 sub-units he or she should attempt to logically group some of them into a single sub-unit. If it does not seem natural to decompose the SWU into less than 7 sub-units, this usually indicates that one has some complexity problem in the SWU abstraction and it should, perhaps, itself be split into smaller abstractions.

The capabilities of a SWU are not necessarily the sum of the capabilities of its sub-units. Using the well known information hiding principle, a SWU can hide to its own clients some of its sub-units' capabilities which are seen to its creator as internal. This is achieved by also hiding the interfaces of the services we wish to hide. In this way we can achieve different levels of abstraction, and hide the internal details and workings of a SWU. Therefore we can define:

- The *hidden-capabilities* of a SWU are the subset of its capabilities that it does not to expose to its clients.
- The *hidden-interface* of a SWU is the subset of its interface that it does not to expose to its clients.

The requirements of a SWU are also not necessarily the sum of the requirements of its sub-units. The reason for this is that one sub-unit can answer some or all of the requirements of its brother sub-units. Therefore, it is quite possible to a have a stand-alone SWU with sub-units such that some or all of them do have requirements.

The side effects of a SWU are the side-effects of its sub-units that affect its environment. Sub-unit side-effects that affect only other sub-units are not side-effects of the parent SWU.

I can now relate more precisely between the attributes of a SWU and the attributes of its sub-units. Given a SWU S and its decomposition into sub-units $s_1, \ldots, s_n$ the following lemmas hold:

$$capabilities(S) = \bigcup_{i=1}^{n} capabilities(s_i) \setminus hidden\text{-}capabilities(S) \quad (4.3)$$

$$interface(S) = \bigcup_{i=1}^{n} interface(s_i) \setminus hidden\text{-}interface(S) \qquad (4.4)$$

$$requirements(S) = \bigcup_{i=1}^{n} requirements(s_i) \setminus \bigcup_{i=1}^{n} capabilities(s_i) \qquad (4.5)$$

$$side\text{-}effects(S) = (\bigcup_{i=1}^{n} side\text{-}effects(s_i)) \bigcap interface(S) \qquad (4.6)$$

The natural decomposition of a SWU into software sub-units which could themselves be decomposed into even smaller sub-units etc, creates a hierarchical map of SWUs describing the architecture of the root SWU. This gives rise to the following definition:

The *Architecture of a Software Unit S* is a rooted tree of SWUs, where the root of the tree represents S and each of the other nodes in the tree is a direct sub-unit of its parent node. The architecture of a SWU is a tree because, by the definition of sub-units, the scope of all the sub-units in the architecture is mutually exclusive. Since the decomposition of a SWU into its sub-units is not singular, we can have different architectures for the same SWU. This is an indication of the different perceptions different decomposers can have of the same SWU.

SWU architectures are graphically described by using multilayer scope diagrams (see Figure 4.10). Although they do not look like rooted trees, they are semantically equivalent and more readable.

4.1.3 Service Flow Diagrams

A *Service Flow Diagram* (SFD) is a graphical description of the service flow between one or more SWUs. Different graphical icons are used to describe the different types of SWUs and the different kinds of services that they can provide. A summary of the major icons used is shown in Figure 4.2. A complete list of the SFD icons with explanations can be found in Appendix A.

As shown in Figure 4.2 a SFD can show three types of service flow: data, function call, and declaration/definition use. The first shows the existence of data flow from one SWU to another. The second shows a function call from one SWU to another. The last shows the use of definitions or declarations from one SWU by another. It is, of course, possible to think of other interesting service flow types. However, these types were sufficient for the current experiment.

The precise semantics of a service flow are not shown in its SFD. If necessary, these could be documented in the interface section of the SWU.

The SFD of a single SWU can show different service flow views between the SWU and its environment. It can show the services that a SWU provides and how it provides them, i.e., the access and result interface, and/or it can show the services it requires in order to provide its services, i.e., its requirements. The boundary between what is internal and what is external to the SWU is shown by a dashed borderline.

Figure 4.3 shows the SFD of the SWU representing the complete ffortid program. It shows the interface of ffortid and the services required by it in one diagram. ffortid receives input from stdin and from the command-line through argc and argv and outputs to stdout and stderr. ffortid needs to read in a description file and several font files to provide its services. Note that without reading additional documentation or providing different views of the SFD one cannot always distinguish between interface and required services.

Software Unit	**IO File**	**Local Variable**
XXX *n* XXX is the name of the SWU. *n* is its number (optional)	XXX XXX is the name of the file	XXX XXX is the name of the variable
Parameter Variable	**Return Variable**	**External Variable**
XXX (P) XXX is the name of the variable	XXX (R) XXX is the name of the variable	XXX (E) XXX is the name of the variable
SWU Borderline	**Parameters Group**	**Data Flow Relationship**
XXX XXX is the name of the SWU	*func* XXX (P) Groups parameters of *func* for SWU entry point	A B Data flows from SWU A to SWU B
Bi-Directional Data Flow Relationship	**Call Relationship**	**Use relationship**
A B Data flows from SWU A to SWU B and vice-versa	A B SWU A calls a function in SWU B	A B SWU B uses declerations or definitions in SWU A

Figure 4.2: Major icons used in SFDs

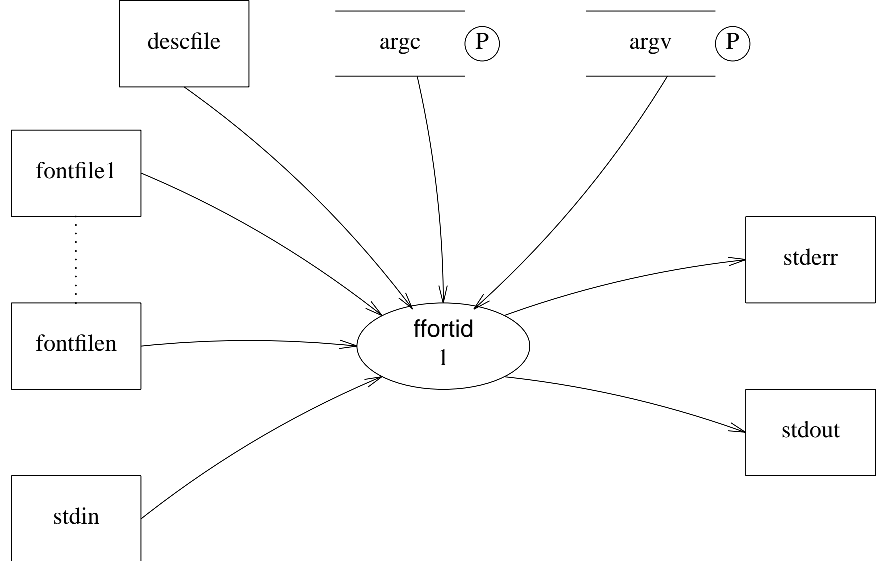


Figure 4.3: A SFD of the SWU abstracting ffortid

One can show service flow between any number of SWUs. It is interesting to show the SFD of a SWU and its sub-units in one SFD. Such a diagram shows which of its sub-units provides each service, and what services flow between the sub-units themselves. All the sub-units of the SWU are shown in the diagram. However, some of them could be internal in the sense that they only provide services to other sub-units and no services to the SWU environment.

Figure 4.4 shows an example of such a SFD. In it we see the SFD of the SWU abstracting the dump.c file in ffortid. dump.c has 5 sub-units, one of which, the "recalc_horiz" sub-unit, is hidden as an implementation detail of the SWU. It provides function call services to two other sub-units, "dump_line" and "reverse_line" and no services outside the SWU. Note how dump.c changes 6 global variables as a side-effect and one can see this side-effect originates in the "dump_line" sub-unit. Also note that this SFD does not show the services required by dump.c. For example, we do not see any services that the "dump_line" sub-unit requires in order to provide its own services that are not in any of the other sub-units. We do, however, see the side effects of any such required services, if there are any.

In the previous sub-section, I stated that it is not desirable for a SWU to have more than 7 sub-units. The SFD of such a SWU would probably be too complex to understand. I have observed that there is a correlation between the visual complexity of a SFD and the external complexity of the SWU. A SWU can be internally very complex. However, if it has a very simple interface, then it usually captures a very well defined concept and is therefore easy to understand by humans. A SWU that has a very large interface is more difficult to understand. However, if this large interface is really a collection of individually simpler interfaces, such as function calls, then it can more easily be grasped.

A SWU that has side-effects is more difficult to understand than one without any side-effects, especially in the context of the other SWUs. For example, a SWU that changes many global variables is difficult to understand. Figure 4.5 shows such an example SFD which is very difficult to understand. Perhaps a SFD can serve as an important indication of the external complexity of a SWU.

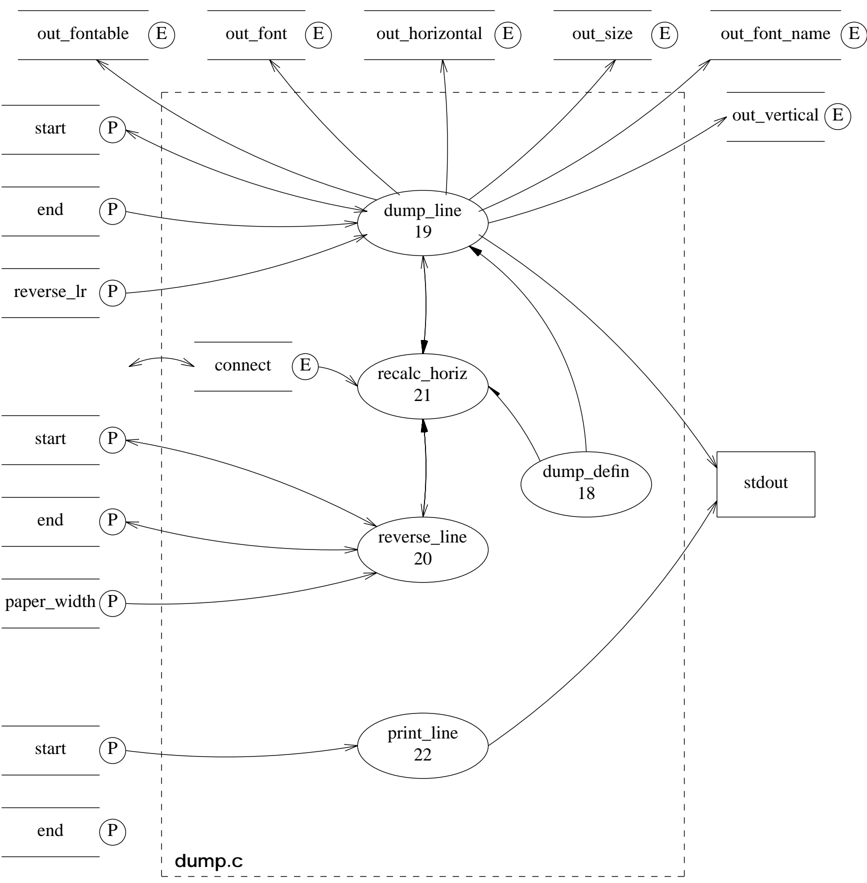


Figure 4.4: SFD of dump.c SWU with its sub-units

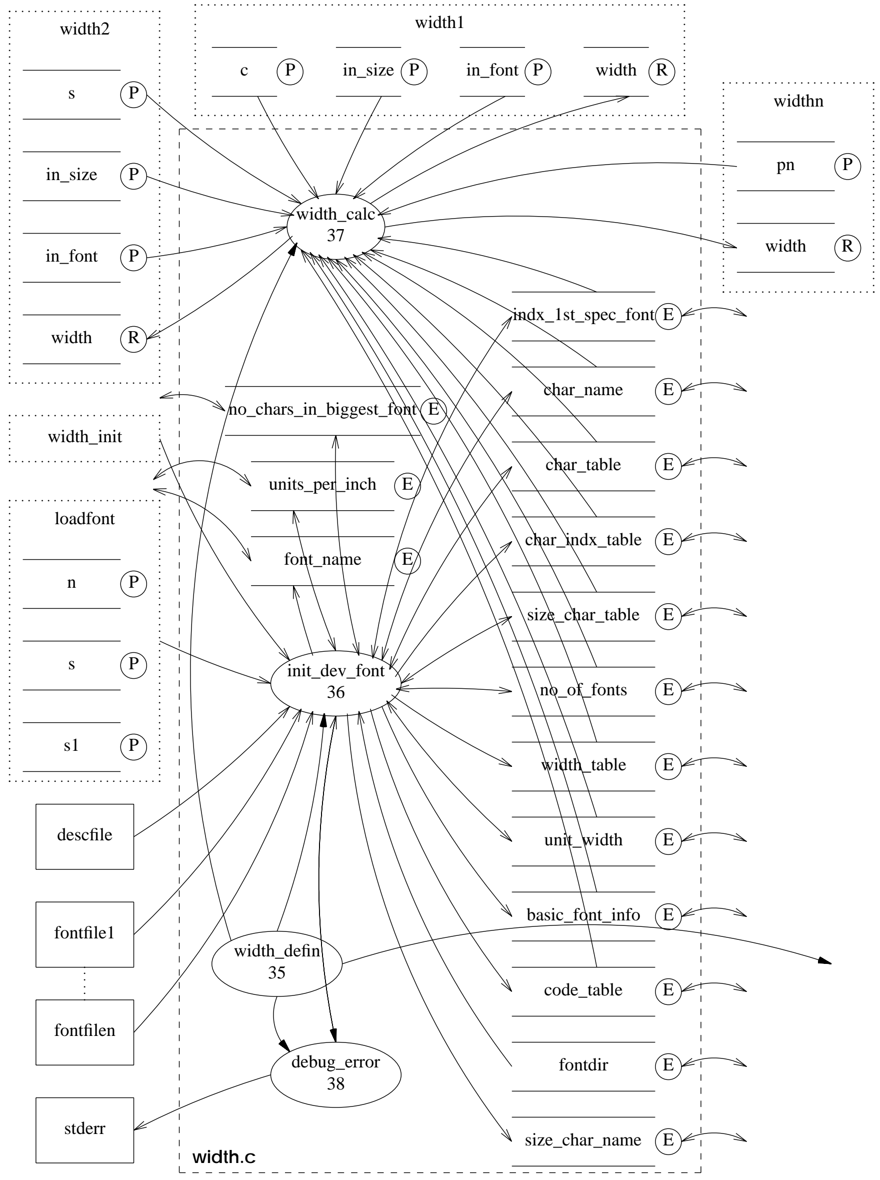


Figure 4.5: A complex SFD

4.2 Reverse Engineering a Software Unit

Reverse Engineering an existing SWU has two major goals [1]: to recreate the architecture and design of the SWU by decomposing it into sub-units identifying meaningful higher level abstractions and to assist understanding of the SWU by documenting the SWU and its sub-units. Additionally, reverse engineering a SWU has the following sub goals:

- to identify SWUs which are candidates for reusable software components,
- to recover information which is not documented in the source code, for example, modifications which were performed during maintenance but were never documented,
- to detect incorrect documentation, errors etc., which exist in the source code, and
- to detect side effects in the SWUs.

Understanding of the structure and functionality of the SWU is facilitated by providing the engineer with a top-down progression of more detailed information on the SWU and its sub-units. The capabilities of each SWU are documented with the aid of comments extracted from the source code. The interface and requirements of each SWU are also documented from the source code. Graphical representations of the service flow between SWUs (SFDs) are generated.

In order to recover the design of a SWU we must decompose it by recursively partitioning it into smaller and smaller sub-units. As defined in section 4.1.2, the architecture of the SWU is represented by a hierarchical map of SWUs, where descendant sub-units show an explosion of their parent SWU. All external service flow between partitioned SWUs are propagated up to a common ancestor SWU.

The decomposer must have some partitioning criteria to guide the decomposition process. This will usually be the syntactical structure of the code combined with the principles of cohesion and coupling. For example, a program in C will first be decomposed into its compilation units (modules). If there are too many modules, we can logically group related modules to create a smaller number of sub-units. This grouping will follow the principles of cohesion, i.e., strong service relations inside a SWU, and coupling, i.e., weak service relations between SWUs. Each of these module groups will be decomposed into its modules. In the next step, each compilation unit can be partitioned into its source files, again grouping some of them if there are too many of them. Source files will be decomposed into their global functions etc.

The decomposer must also decide on the desired level of detail and stop partitioning when that level is reached. We want to decompose SWUs down to a level which holds abstractions which are good reuse candidates. On the other hand, it is important to obtain SWUs with a granularity that does not clutter the visualization. The statement level in a function is usually too low to be a good reuse candidate. A good decomposition level in C is the function or group of functions level. Of course, sometimes we find very large functions from which we can find groups of statements that are good abstractions and therefore good reuse candidates. This implies that the function was too large to begin with, and should have been split into several functions in the original design.

As explained in Section 1.1, reverse engineering is a process of examination. We are trying to recapture the architecture and design of the SWU as understood by its creators and modifiers. This is not necessarily the best possible architecture and reverse engineering is not concerned with improving the design in any way.

Any improvements we believe can be inserted into the design can and should be recorded, but should not be implemented during the reverse engineering process. In later life-cycle phases, we may be able to justify some or all of these changes and perform them as necessary.

The process of decomposition is best performed by starting from the SWU to be decomposed, determining its sub-units and proceeding recursively. However, the attributes of each sub-unit should be determined in a bottom-up fashion. The reason for this is that the major attributes of a sub-unit, its capabilities, interface, and requirements are difficult to determine accurately without first determining the same attributes of its own sub-units. Section 4.1.2, showed that the capabilities of a SWU are determined by the capabilities of its sub-units. The side-effects of a SWU are the side-effects of its sub-units minus those that do not affect the outside environment of the SWU. The requirements of a SWU are the requirements of its sub-units that are not satisfied by any of the other sub-units.

It is therefore natural to view the capabilities, and interfaces, of the SWUs as being propagated from the most low-level SWUs up through the SWU architecture. When determining the capabilities of a certain SWU we can decide not to pass on a certain service, thereby hiding it and creating higher level abstractions. A SWU requirement will be propagated up until it is satisfied by a SWU at a certain level, at which point it disappears. Side effects are propagated up from their originating SWU until they reach a level, if at all, in which they are no longer considered side effects because their effect becomes internal to the SWU at the new level.

To summarize, there are 4 major points in reverse engineering a SWU:

1. Partition the SWU into sub-units and continue recursively.
2. Partition the SWU according to the syntactical structure of the SWU and the principles of coupling and cohesion.
3. Partition the SWU down to the desired abstraction level of good reuse candidates.
4. Determine the attributes of the sub-units in a bottom-up fashion.

4.3 ffortid Version 3.0 Reverse Engineering

The author performed a process of reverse engineering as described above on ffortid Version 3.0. The process was performed completely manually using only traditional methods of text editing and UNIX commands such as grep. All SFDs were drawn manually using Pic [21]. The whole process was very laborious and it was completed successfully only because ffortid is a relatively small program.

The following SWU classifications were chosen as types: Program, Module, Source file, Declarations source file, Definitions source file, Data file, Definitions block, Declarations block, Procedure group, Function group, Procedure, and Function.

It was decided that the required level of detail will be no smaller than functions or procedures. It turned out to be unnecessary to carry out all refinements to this level.

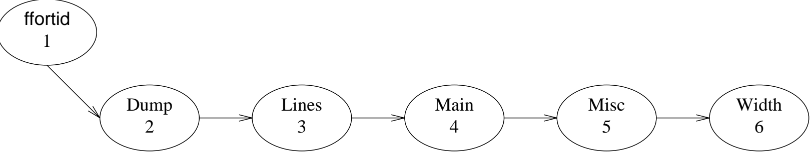
The partitioning criteria was similar to the one described in the previous section. In some cases it was decided to partition a SWU in one way, and later on after understanding the SWU better, a different partitioning that captured the new understanding more precisely was used. This is a natural and expected phenomena. As one learns more about the software and understands it better, one might see the abstractions of the software differently.

Software Unit #1 — ffortid

1.1 Software Unit Type

Program. (lex.h, lex.dit, token.h, macros.h, connect.h, table.h, dump.c, lines.c, main.c, misc.c, width.c)

1.2 Scope Diagram



1.3 Capabilities

ffortid takes from its standard input dtroff output, which is formatted strictly from left-to-right, finds occurrences of text in a right-to-left font and rearranges each line so that the text in each font is written in its proper direction. Additionaly, ffortid left and right justifys lines containing Arabic & Persian fonts by stretching connections in the words instead of inserting extra white space between the words in the lines.

1.4 Interface

command line options:
ffortid [−**r***font-position-list*] ... [−**w***paperwidth*] [−**a***font-position-list*] ...
[−**s**[**n**|**f**|**l**|**a**]] ...

The **-r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. The **-w***paperwidth* argument is used to specify the width of the paper, in inches, on which the the document will be printed. The **-a***font-position-list* argument is used to indicate which font positions, generally a subset of those designated as right-to-left (but not necessarily), contain fonts for Arabic, Persian or related languages. The **-s** argument specifies the kind of stretching to be done for all fonts designated in the **-a***font-position-list*

1. **-sn** — Do no stretching at all for all the fonts.
2. **-sf** — Stretch the last stretchable word on each line.
3. **-sl** — Stretch the last stretchable word on each line up to a maximum length.
4. **-sa** — Stretch all stretchable words on the line by the same amount.

The default is no stretching at all.

Manual connection stretching can be achieved by using explicitly the base-line filler character **\(hy** in the dtroff input. It can be repeated as many times as necessary to achieve the desired connection length.

Side effects:
1. ffortid reads dtroff output from stdin and prints dtroff output to stdout.
2. ffortid prints encountered errors to stderr and halts program.
3. ffortid allocates and frees memory from the heap. If out of heap memory ffortid prints a ``out of memory´´ message to stdout and halts program.

Figure 4.6: First part of ffortid Version 3.0 SWU 1 page

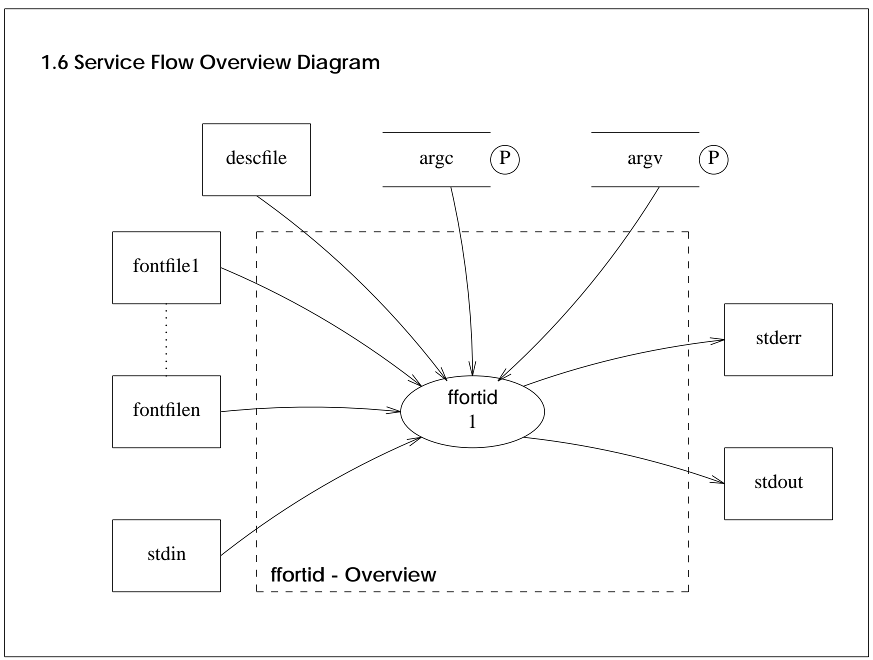


Figure 4.7: Third part of ffortid Version 3.0 SWU 1 page

Each SWU in the decomposition was documented in what is called a *Software Unit Page*. Figure 4.6 shows the first part of the page documenting the SWU abstracting the complete ffortid program (SWU 1). The first section of the page titled "Software Unit Type" describes the SWU type and scope. In this case, the type of the SWU is "Program" and its scope is all the source files of the program. Section 2 (1.2) of the page is the scope diagram of the SWU showing graphically the SWU and its sub-units. The sub-units of ffortid are the 5 modules from which it is created. This is a natural decomposition which captures the architecture of the program. Note that each of the sub-units has, of course, its own SWU page which describes it completely in the same fashion.

The third section of a SWU page, titled "Capabilities", gives a verbal description of the capabilities of the SWU. This should be a precise and concise description of the SWU capabilities in a language which is clear to the domain and software expert. In the case of ffortid, this section describes the capabilities of the complete program.

The fourth section of the SWU page, titled "Interface", gives a precise description of the interface of the SWU. This is simply a list of the different services provided by the SWU. The interface section describes, as described in Section 4.1.1, the side effects of the SWU. In the case of ffortid, the interface is the command-line options of the program. These are described completely in the section.

As defined previously, side effects are changes in the SWU's environment which are not clearly visible or stated in the SWU service's access interface. In this case, the environment of ffortid is the operating system environment. Therefore, all effects which are not clear from the command-line options must be considered side effects, although some or all of them might be part of the normal function of the program. Reading and writing to files/streams, allocation and deallocation of dynamic memory are therefore all side effects vis-a-vis ffortid and they are all recorded in this section.

In most SWU pages, the fifth section, which holds a SFD of the SWU, is the last section of the page. To SWU 1 it was decided to add a sixth section, titled "Service Flow Overview Diagram", which is also a SFD describing the services

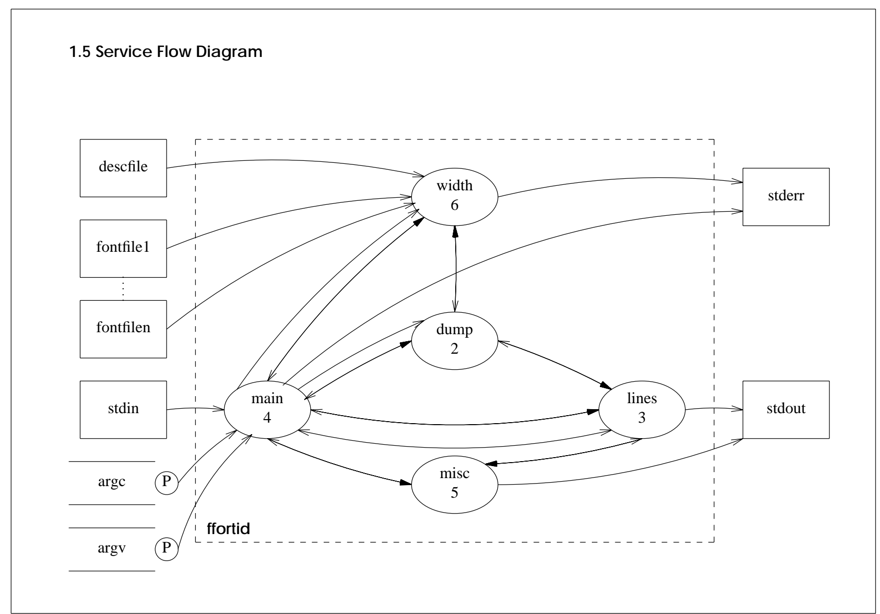


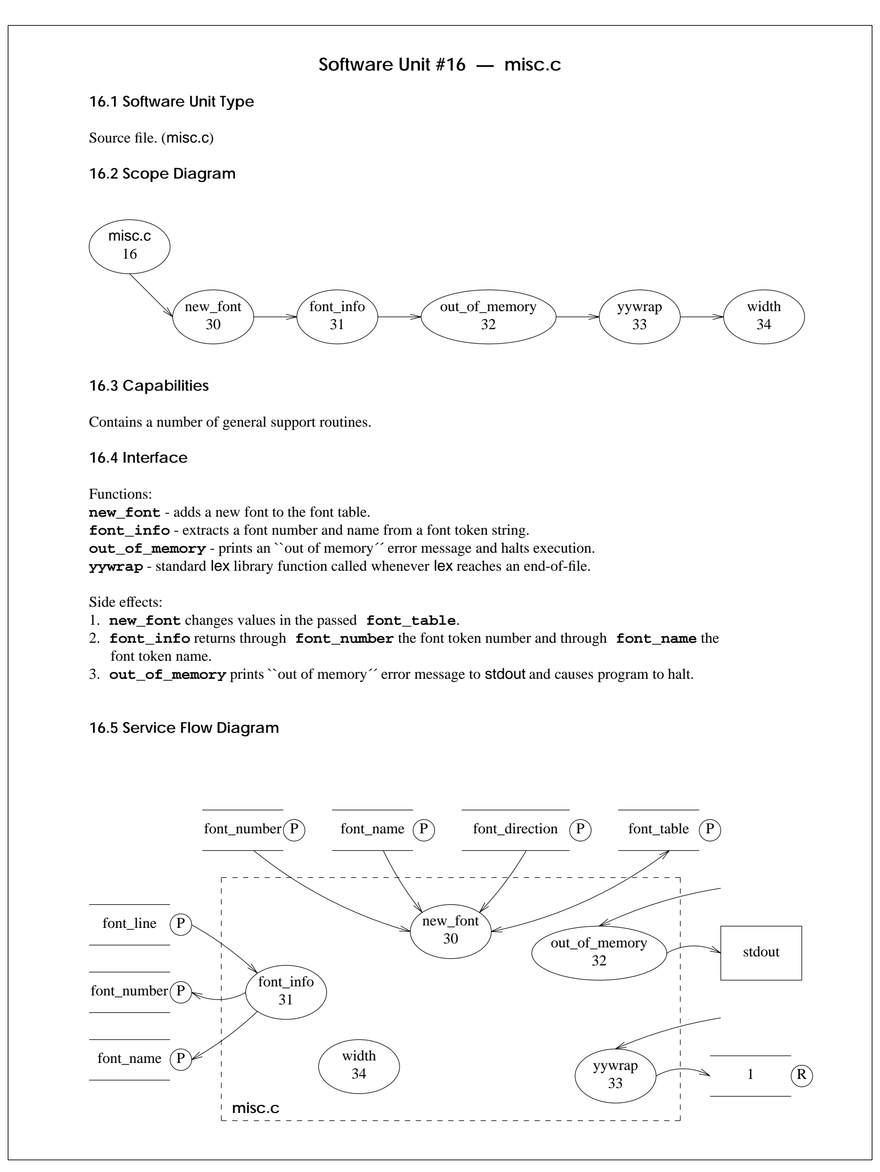
Figure 4.8: Second part of ffortid Version 3.0 SWU 1 page

and side-effects of the SWU, but without showing the internal structure or service flow of the SWU. This SFD is a simplification of the SFD of the previous section with the intention of showing the SWU as a "black box". It does not add any information to the previous SFD. In Figure 4.7, we see that ffortid receives input from the command-line, `argc` and `argv` variables, from the standard input stream, and from a number of files, and outputs to the standard output and standard error streams. The allocation and deallocation of dynamic memory is not shown in the diagram, although with appropriate icons it certainly could have been.

As mentioned previously, the fifth section of a SWU page holds a SFD that graphically describes the services provided by a SWU to its environment, the side effects of these services, and the service flow between the sub-units of the SWU. Figure 4.8 is this SFD of SWU 1. Unlike the SFD in Figure 4.7, it shows in detail the service flow between the sub-units of ffortid and their relation to the environment. For example, in it we can see that the command-line options and the standard input are read by SWU 4 which abstracts the main module. The other input files are read by the width module. Standard output is generated only by the lines and misc modules, and output to standard error is generated only by the main and width modules. In the SFD one can see which module calls functions in other modules and from which modules does data flow to other modules.

There is no requirements section in our SWU pages. The reason for this is that the reverse engineering process was performed before the significance of such a section was realized. Clearly, such a section is needed for the potential user of a SWU. In it, he or she will find what environment must exist for the SWU to function properly.

As stated previously, each of the SWUs in the decomposition was documented by a SWU page. Figure 4.9 shows the SWU page of an intermediate SWU in the decomposition, SWU 16. Note how the function `width`, SWU 34, is a sub-unit of SWU 16 but is not part of its interface. The reason for this is that `width` provides no services. This can be seen in the SFD of SWU 16, `width` has no parameters and returns no value. This function is an archeological relic of some earlier development phase of ffortid.



Software Unit #16 — misc.c

16.1 Software Unit Type

Source file. (misc.c)

16.2 Scope Diagram

16.3 Capabilities

Contains a number of general support routines.

16.4 Interface

Functions:
`new_font` - adds a new font to the font table.
`font_info` - extracts a font number and name from a font token string.
`out_of_memory` - prints an ``out of memory´´ error message and halts execution.
`yywrap` - standard lex library function called whenever lex reaches an end-of-file.

Side effects:
1. **`new_font`** changes values in the passed **`font_table`**.
2. **`font_info`** returns through **`font_number`** the font token number and through **`font_name`** the font token name.
3. **`out_of_memory`** prints ``out of memory´´ error message to stdout and causes program to halt.

16.5 Service Flow Diagram

Figure 4.9: SWU 16 page in ffortid Version 3.0 decomposition

4.4 ffortid Version 3.0 Architecture

Altogether, ffortid Version 3.0 was decomposed into 41 SWUs of which 28 are low-level. Each SWU was given a name and identification number. Additionally, the size of each SWU, and the number of lines in its scope, were recorded. Table 4.1 summarizes this information for all the SWUs in the decomposition.

Figure 4.10 shows the complete decomposition of ffortid into its SWUs in the form of a scope diagram. Due to the diagram's length, it was broken into 5 scope diagrams, one for each of ffortid's direct sub-units. They are shown one on top of the other but should be connected as shown by the dashed arrows. Note that SWUs abstracting header files (such as token.h) appear several times in the diagram because they are included by different SWUs.

The process of reverse engineering ffortid by decomposing it into SWUs, creating a page documenting each SWUs scope, capabilities, interface, and SFD proved itself as very effective in advancing the author's understanding of the architecture of a program about which he initially knew nothing.

In general, ffortid's architecture is a rather outdated form of structured programming. There is heavy use of global variables, which in some cases can be justified, but could always have been avoided to achieve higher independence between modules. This outdated architecture is exemplified in the SWUs which are sometimes not as reusable as desired, since they are abstractions of the code as is, without any modification.

The basic idea in ffortid is to read in the ditroff output tokens, convert them to an internal representation, perform any calculations and alterations to the lines of tokens as necessary in order to change text direction and/or justify lines, and then output the token lines in the same format as ffortid input.

ditroff output is a stream of well-defined tokens which are device-independent commands to a typesetter (usually a printer). These commands include such things as device resolution definition, font mounting, character printing, horizontal and vertical movements etc. This stream of tokens is parsed by SWU 14, which is generated by Lex based on SWU 27, into lexical tokens, SWU 8. SWU 4, the Main module, parses the command line options and stores them in global variables. It then reads in token by token using SWU 14, and depending on the token type either immediately outputs it as is, or if it is a character token, stores the token in a token structure, SWU 7, which holds lines of character tokens. Main simulates the actions of the typesetter by recording its changing state as fonts and point sizes are changed and movements are performed. Main uses services in Lines, SWU 3, to create and free token structures, some miscellaneous services in Misc, SWU 5, and services in Width, SWU 6, that calculate the width of characters according to their font and point size. This information is needed for line width calculations and character transformations within lines.

The heart of ffortid is in SWU 2, Dump. In it, lines of character tokens are transformed according to the command-line options stored in global variables and then output using services in Lines. For its calculations, Dump needs some width services from Width. Dump reverses characters of the fonts that are specified in the command-line as those to be reversed and stretches lines that contain characters in the fonts specified in the command line as those to be stretched. The stretching of the lines is performed according to the stretch style requested by the -s option, as described in the SWU 1 page.

The complete decomposition is available in Adobe Acrobat pdf format with hypertext links between the different SWU pages [22]. This electronic manual can be read using Adobe's Acrobat reader which is available free of charge. The hypertext links enable easy traversal between SWU pages, source code and all other relevant documents.

Num	Name	Type	Size (lines)	Low-Level
1	ffortid	Program	3422	
2	Dump	Module	1044	
3	Lines	Module	398	
4	Main	Module	1299	
5	Misc	Module	201	
6	Width	Module	480	
7	token.h	Declarations source file	34	*
8	lex.h	Definitions source file	30	*
9	macros.h	Definitions source file	20	*
10	connect.h	Data file	256	*
11	dump.c	Source file	704	
12	table.h	Declarations source file	18	*
13	lines.c	Source file	296	
14	lexer	Lex generated source file	691	
15	main.c	Source file	506	
16	misc.c	Source file	129	
17	width.c	Source file	480	
18	dump_defin	Definitions block	34	*
19	dump_line	Procedure	103	*
20	reverse_line	Procedure	83	*
21	recalc_horiz	Function group	463	
22	print_line	Procedure	21	*
23	lines_defin	Definitions block	35	*
24	new_free_token	Function group	85	*
25	insert_tokens	Procedure group	52	*
26	put_tokens	Procedure group	124	*
27	lex.dit	Lex source file	37	*
28	main_defin	Definitions block	58	*
29	main	Function	448	*
30	new_font	Procedure	41	*
31	font_info	Procedure	41	*
32	out_of_memory	Procedure	17	*
33	yywrap	Function	13	*
34	width	Function	17	*
35	width_defin	Definitions block	47	*
36	init_dev_font	Procedure group	229	*
37	width_calc	Function group	122	*
38	debug_error	Procedure group	82	*
39	recalc_horiz_2	Procedure	53	*
40	calc_total	Function	48	*
41	stretch	Function group	361	*

Table 4.1: ffortid Version 3.0 software units

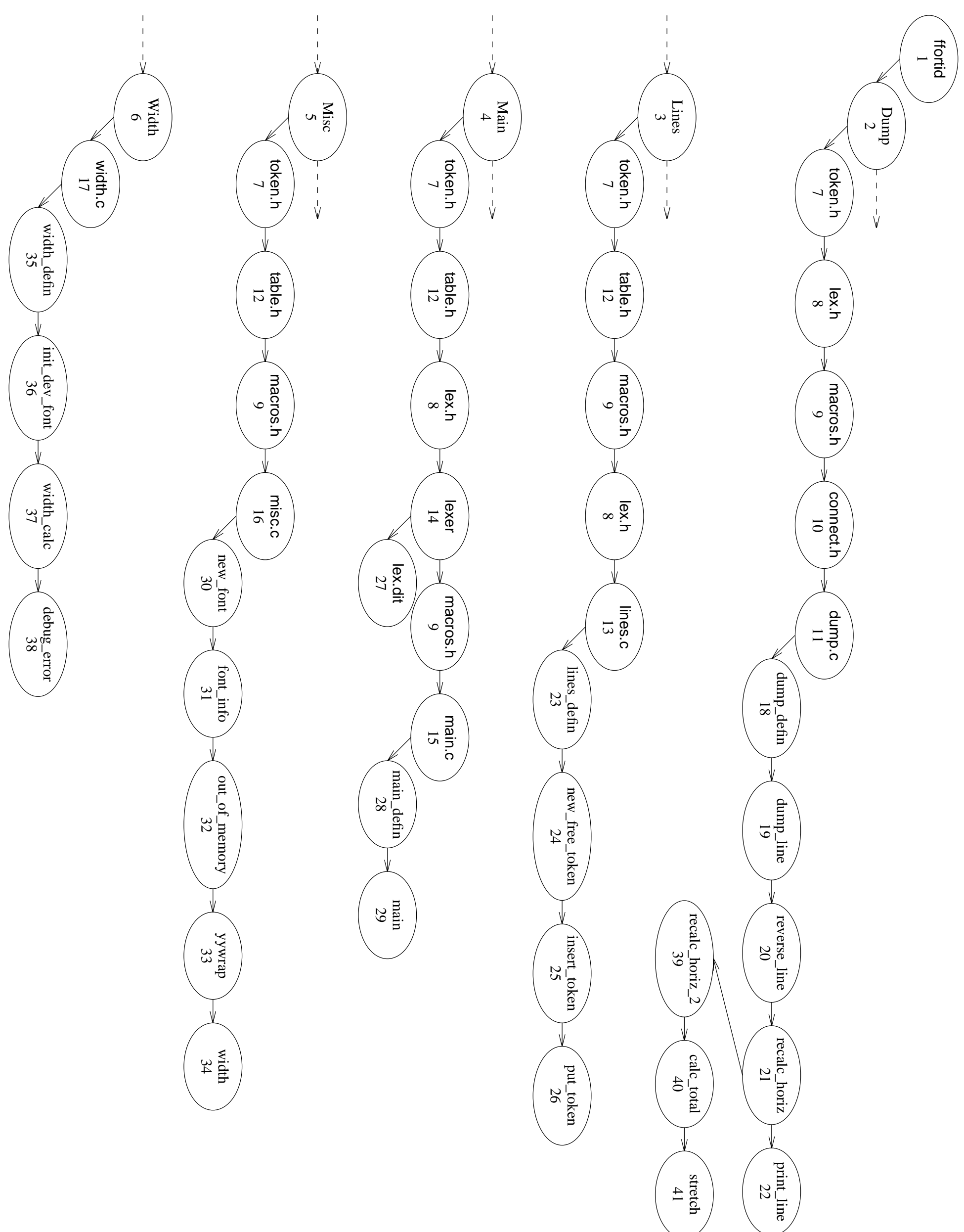


Figure 4.10: Overview of ffortid Version 3.0 decomposition

4.5 Author's Conclusions from Decomposition

Performing a decomposition of a legacy program has a lot in common with archeology. One discovers mixed layers of architectures and changes performed by different programmers at different times and with different programming paradigms. A good legacy program is one that is relatively homogeneous despite the various changes it has undergone throughout its lifetime.

As I examined the code during the partitioning phase, I added my own comments to help me understand what each piece of code was doing. During this phase I had found several small bugs, erroneous comments, unused code and variables and even a gross differentiation from the documentation in the manual page. Clearly, this is a result of the many modifications performed on the code. In general, the code was readable and had enough significant names in it to help understand the overall architecture of the program. I did not however attempt to understand the details of each and every algorithm, but instead to gain insight into the structure of the program.

I have found that building the SWU pages was best performed by starting from the lowest level SWUs and working my way up to higher level SWUs. The reason for this was that all the capabilities, interfaces and side-effects of the SWUs propagate up from the low level to the high level SWUs. It is simply not possible to document correctly a higher level SWU without first documenting its lower level SWUs.

I found it important to be able to understand the interaction between the different parts of the code. This includes recognizing the use of global variables, function calls etc. and where these were defined. I used `grep` to do these simple tasks, but the ability to perform automated queries on the code, just as in a database, and generate different views of the code, in my view, can greatly advance the software understanding process.

The SFD were actually the last thing I added to the decomposition. I found that they were a lot of work and did not help much in the decomposition itself, i.e., deciding on the partitioning criteria. Additionally, I have found that they did not help much in the understanding of SWUs especially in levels lower than function or procedure. I found it much easier to read statements of code than to understand a graphical description of these statements. However, the graphical documentation was very helpful giving a global view of high level SWUs services especially when you try to understand a SWU you have not worked with in a while.

The manual reverse engineering process I performed, helped me reach conclusions on what functions a dedicated CASE tool should provide to aid this process. There is much paperwork in this method and without such dedicated CASE tools no single or group of engineers can be expected to complete it in a reasonable period of time on large legacy systems. Fortunately, most of this paperwork can be automated successfully. In my view, this method of reverse engineering is viable on real, large volume, complex legacy code systems only with such CASE tools.

In general, a CASE tool should automate everything that can be automated and leave to the human operator that which cannot be automated well. The same is true for a reverse engineering CASE tool. It does not need to, and should not, replace the human decisions needed in the process. The documentation of SWU interfaces and SWU requirements, extraction of relevant comments, and generation of SFDs can all be automated. Precise partitioning of a SWU and capabilities documentation must still be mostly manually performed. Such tools are therefore semi-automatic reverse engineering tools.

With the use of expert knowledge, a reverse engineering tool can provide suggestions to help the human operator make faster and more knowledgable decisions. For example, it can suggest one or several options for partitioning a SWU according to its syntactic structure and/or the resulting service flow dependencies between

the sub-units. The human operator can then decide to accept one of the suggested decompositions or provide one of his own. More advanced tools could even try to learn new partitioning criteria from previous human partition decisions.

To summarize, a CASE tool can help a process of reverse engineering by:

- suggesting criteria, alternatives, implications, and places to partition a SWU, perhaps using AI knowledge expert technologies,
- generating automatically the SFDs,
- generating automatically all or most of the side-effects of a SWU,
- generating automatically the requirements of a SWU,
- handling most of the paperwork involved; a change in one SWU should propagate automatically to all affected SWUs,
- extracting or pointing to comments in the code which might be of use in documenting a SWU,
- building a database of the SWU architecture on which different queries can be performed, and
- allowing the addition of new comments to the source code as additional comments and not as part of the code.

Such a dedicated CASE tool should not only provide assistance in the reverse engineering process itself, but it should provide an environment in which the discovered architecture can be traveled through i.e., to move from one SWU to its parent SWU or to one of its sub-units, to view a SWU's attributes, to transfer between the abstraction and the source code, and to see different views of the stored information such as all uses of a global variable, function calls, etc. The reverse engineering tool should build a SWU database which can be viewed and easily changed as we change previous decomposition decisions. This database will later serve as s basis for changing the source code or SWU structure.

The fact that the complete SWU architecture is stored in a database will greatly help the following steps of changing the different components and realizing the effects of these changes.

4.6 The Initial Domain

The domain under consideration is the family of applications that are ditroff postprocessors capable of reversing text in right-to-left fonts and capable of left and right justifying lines by stretching Arabic text.

The author decided to use the architecture of ffortid Version 3.0 as discovered by the reverse engineering process as the basis of the initial domain architecture. Each of the SWUs in the decomposition is a reusable component in the domain. Some of the reusable components are low-level. Others are themselves composed of lower level reusable components. The SWU pages document the capabilities and interface of each SWU and therefore of the reusable components.

The initial domain has only one SWU representing an application, SWU 1, and one way of composing the different reusable components to create it. SWU 1 is directly composed from 5 high level reusable components:

- SWU 2 - Dump - a module that contains routines to reverse and stretch internal token lines.

- SWU 3 - Lines - a module that contains routines to allocate, free, and output internal token lines.
- SWU 4 - Main - a module that parses the command-line options and runs the main ffortid driver routine.
- SWU 5 - Misc - a module that contains some general support routines.
- SWU 6 - Width - a module that contains global variables to store the font and width tables and routines to initialize them and return character widths based on them.

The reusable components created from these SWUs are not always very adaptable or reusable. Some of them are not as independent from other components as would be desired. The intention is that they be transformed in an evolutionary manner to more adaptable components by a continuous flow of new external requirements for more advanced applications in the domain. It is possible to speed up this natural process by performing, at selected life-cycle points, a top-down domain engineering effort to refurbish the components for future requirements.

For example, it was possible not to use the architecture and components recovered from ffortid as is, but instead to use them as a basis for a domain with object oriented reusable components by extracting and analyzing the knowledge and code in the components. This method is arguably faster and less costly than building a domain from scratch because the designers have to their advantage the knowledge and experience of previous generations of programmers embedded in the legacy code. However, this technique is highly dependent on the domain and quality of the legacy application being leveraged.

In very complex domains with large legacy applications, such a preventive maintenance effort would be very costly and risky and therefore difficult to justify. In such a case, it is probably better to let the domain evolve in an evolutionary manner. This is the case to be checked in this experiment. Do our reusable components become better as more applications are created from the domain? How does the domain adapt under these circumstances?

Chapter 5

ffortid Version 4.0

After successfully building an initial domain architecture and reusable components it was time to proceed to the next experiment stage. According to the experiment design, a new set of previously unknown requirements must be devised for a new application in the domain.

Berry had already created, as part of his research, a new version of ffortid according to a set of requirements he devised. He used his own systematic maintenance method to implement these requirements. Only after I had finished creating the domain, was I presented with this new set of requirements, so they could not have affected in any way the architecture of the domain I created.

I was to implement these new requirements using the legacy and reuse based life-cycle model as my basis. As previously described, the intention is that the domain develop in an evolutionary manner according to external requirements. Therefore, no modifications will be made to the domain unless they can be completely justified by new requirements, or perhaps, by errors found in existing components. The new application is named ffortid Version 4.0.

5.1 SWU Modifications

All modifications to a SWU are performed on its scope, i.e. its source code. A modification of a SWU can potentially affect the 3 major attributes of a SWU: its capabilities, its interface, and its requirements. All these attributes are orthogonal and therefore each one of them can either be, or not be, affected by each modification.

The type of a SWU will usually not change by a modification, unless it is a major modification in which case it is not clear if the new SWU is logically the same as the old one. A SWU is an abstraction of a concept in the domain. One can update the abstraction as the domain changes, however, a major change in the abstraction does not leave us with the same SWU.

Most modifications are performed to change a SWU's capabilities and/or interface. Modification of a SWU's requirements is usually a side-effect of these changes unless it is itself the required change. We have categorized the 4 types of modifications possible looking only at how they affect the capabilities and interface of a SWU (see Table 5.1).

A *type I modification* of a SWU is simply a reimplementation of the SWU without changing its capabilities or interface. Such a modification will usually be performed either as part of a preventive maintenance effort or in order to increase the performance of the SWU in some respect, e.g., time, space, etc. Such modifications do not change any of the services provided by the SWU or how it interacts with its

Type	Capabilities Modified	Interface Modified
I	no	no
II	no	yes
III	yes	no
IV	yes	yes

Table 5.1: SWU modification types

environment.

A *type II modification* of a SWU does not change any of its services but does change the way the environment accesses them. Usually such a modification will result in a simpler, easier to understand, easier to use interface and this is the main motivation for such a change. A type II modification can include type I modifications as well.

A *type III modification* of a SWU changes the services of the SWU without changing its interface. In other words, this is a semantic change of the SWU without changing its syntax. A semantic change does not necessarily mean the SWU concept changes. On the contrary, for example, a SWU abstracting a square root calculating function can be modified to increase the accuracy of its result. The SWU concept has not changed, we are still providing square root calculating services. Even a local fault correction, is a type III modification. A type III modification can include type I modifications as well.

A *type IV modification* of a SWU is a change both in the semantics and in the syntax of a SWU. This usually occurs when a modification of the services provided by a SWU also require the change of its interface, either to increase its input bandwidth for needed new information, or to increase the output bandwidth for the new services results. A type IV modification can include type I, type II, and type III modifications as well.

Any of the above types of SWU modifications can potentially also affect the requirements of the SWU. Modifications to a SWU's requirements are either side-effects of other modifications or part of a preventive maintenance effort. The fewer requirements a SWU has the more independent it is in terms of how it can be incorporated in a project. We should always strive for SWUs that are more independent and therefore have fewer requirements, however, most low and medium level SWUs must interact with other SWUs to provide their services and therefore must have a minimal number of requirements.

Type I and type II modifications are normal in preventive maintenance. Type III and type IV modifications are typical in corrective, adaptive, and perfective maintenance.

Type II and type IV modifications are generally to be avoided because they affect not only the specific SWU that was modified but also all the SWUs that use the service whose interface has changed. Modifications of type IV that do not change the current interface of a SWU but instead add to it, do not fall under the category of modifications to be avoided because they do not cause this type of modification ripple effect. We will call these kind of modifications type IV*.

We can define in a similar fashion type II* modifications. However, it is not clear why one would add to an existing interface without changing the capabilities of the SWU. Type III* modifications are modifications of type III that do not change current capabilities but instead add completely new services without a need to change the SWU's interface. This is possible if the SWU interface was defined well enough to allow such future enhancements.

Software modifications are a natural phenomenon of software evolution. We

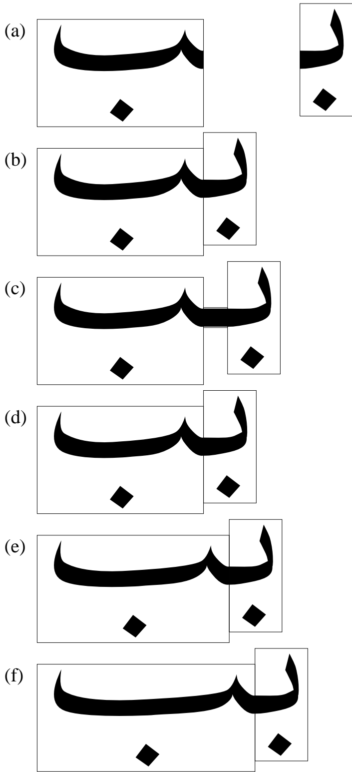


Figure 5.1: Connecting letters, fillers, and dynamic letters

should however, attempt to reach a situation in which we do not have modification ripple effects in which a modification in one SWU causes many modifications in other SWUs. In other words we want to reach a situation in which we perform only type I, II*, III, III*, and IV* modifications. This will occur only when we have good domain abstractions with carefully planned interfaces that pass high-level information abstractions.

5.2 The New Requirements

ffortid version 4.0 should have all the functionality of version 3.0 plus the capability to left and right justify lines containing Arabic, Persian or related languages by *stretching letters* and not just by inserting fillers between connecting letters. This requires the use of dynamic fonts in which letters can actually be stretched. Figure 5.1 (a), (b), and (c) show the current method of inserting a filler between two connecting letters, increasing the word's width. Parts (d), (e), and (f) of Figure 5.1 show the new method of stretching a letter in a word to achieve the same effect. Note that not all arabic letters can be stretched. Only those letters with a large, mostly horizontal stroke, are traditionally stretched in Arabic calligraphy.

The new requirements were specified as precisely as possible by Berry by writing a new manual page describing the new version of ffortid. It can be found in Appendix C. From a comparison of the old and new manual pages we created a list of 3 required enhancements:

1. Change the command-line options and add the capability to automatically stretch letters and/or connections according to these options and the new stretch information in the width tables.

2. Add the capability to manually stretch letters.
3. Add the capability to control automatic stretching of words with manually stretched letters and/or connections by two new command-line options.

Enhancement 1 modifies the command-line options to express the new possibilities of automatic stretching created by the use of letter stretching. In ffortid Version 3.0 all stretching was performed by inserting fillers and the stretch style option specified where to stretch. In ffortid Version 4.0 automatic stretching is specified by two relatively independent dimensions: where to stretch (the *stretch place*) and how to stretch (the *stretch mode*). The stretch place is similar to the previous stretch style. The stretch mode allows the stretching of only connections, only letters, either letter or connection, whichever comes later in a word, or both.

This enhancement includes inserting the functionality that performs the actual stretching of the lines according to the specified options. This includes reading new width tables fields that specify the stretchability and connectivity of each character. This information is needed so ffortid can know which characters are stretchable.

Enhancement 2 adds the functionality needed to accept new input tokens that specify manual letter stretch commands. These enable the user to manually stretch specific letters by any required amount. The manual stretch information must be stored in the character token as an integral part of the character.

Enhancement 2 adds the capability to manually stretch letters in words. When automatic line stretching is enabled, these words can be additionally stretched to left and right justify a line. In some cases, it would be desirable to prohibit automatic stretching of words already manually stretched. Enhancement 3 adds two new command-line options to achieve this effect: the `-msc` option prohibits the automatic stretching of words containing manual connection stretch commands, and the `-msl` option prohibits the automatic stretching of words containing manual letter stretch commands.

The actual stretching of letters by ffortid is achieved by a new output token that is preceded by ffortid to every character that it wants stretched. This output token includes the amount of stretch of the character. An application called psdit that reads ditroff output and translates it into postscript was modified to accept this new ditroff command and translate it into postscript commands causing the actual character stretching. This application is not part of ffortid as such and is therefore not part of the experiment.

We made sure the 3 enhancements cover all the new requirements by comparing the old and new manual pages. Our main concern in the division of the new requirements into enhancements was to make each enhancement as independent from the others as possible from the users point of view. All the enhancements are independent except for enhancement 3 which depends on enhancement 2. The idea is that in principle each enhancement could be performed separately and therefore could be tested separately in an incremental manner.

5.3 Implementation

We now have a domain and a set of requirements which we must implement by adapting the domain as necessary and generating a new application answering these requirements. Our new requirements are enhancements to ffortid Version 3.0 on which our initial domain is based. In the domain's current state, we can generate only applications with similar architectures because we have only one SWU representing an application. Such SWUs tell us how to build applications from our reusable components. This will not always be the case in more advanced domains in which applications with completely different architectures could be generated.

In such domains, we would have several SWUs each representing an application architecture.

We propose here a systematic method for domain adaptation according to a new set of requirements. This method can be used to implement the requirements in an incremental manner or in a single batch.

Usually requirements are expressed in a user-oriented, high level fashion. Our first step is the expression of the requirements in a more detailed fashion by listing them at the lower software level as interface and capability changes to the SWU representing the complete application. In our case this is SWU 1. The interface changes of SWU 1 are:

I–1 Modify the command-line options.

I–2 Modify the structure of the width table.

I–3 Add the acceptance of input manual stretch commands.

I–4 Add the printing to output of stretch commands.

Note how some of the changes are modifications to existing interfaces, I–1 and I–2, and some are completely new additions to the application interface, I–3 and I–4. I–1, I–2, and I–3 are all access interface changes and I–4 is the only result interface change. The capability changes of SWU 1 are:

C–1 Treat manual stretch values in character tokens as part of the character.

C–2 Add the storage of automatic stretch values in character tokens.

C–3 Modify automatic stretching to stretch according to the new command-line options and width table information.

Note again how some of the capability changes are completely new capabilities, C–1 and C–2, and some are modifications of existing capabilities, C–3.

The next step in our method is the implementation, first of the access interface changes, and then of the capability changes, and finally of the result interface changes. Capability changes can and usually do depend on new information in the input interface and must therefore be implemented only after we have designed and implemented the access interface changes. The result interface changes can and usually do depend on the new capability changes and must therefore be implemented after them.

The division of the requirements into application interface and capability changes serve several purposes. First, it helps separate the internal and external changes to the application. Secondly, it assists the implementation of the changes using the method in the previous paragraph.

Each interface or capability change is implemented using the same technique. Using the domain hierarchy of SWUs, we perform a top-down search for all the low-level SWUs that should be modified by the change. The search is a focused search, directed by the interface or capability description of each SWU. If we are modifying an existing interface, or capability, we search for the current low-level SWUs that possess the to-be-modified interface or capability. If we are adding a new interface or capability we search for the SWU to which it should logically be added.

For example, I-1 is a modification of the current command-line options. According to the SWU Lemmas in Section 4.1.2, there exists a sub-unit of SWU 1 that provides this interface service. By interface service, we mean that there exists a sub-unit that reads in, parses, and stores the command-line options for the use of

other sub-units. The top-down search flows from SWU 1 to SWU 4 to SWU 15 and finally to SWU 29 (see Figure 4.10). We must therefore modify SWU 29 which is the `main` function to implement I-1. This includes modifying the parse mechanism of the command-line to accept the new options and modifying the global variables to store the new options.

This modification causes a series of modification side-effects. SWU 29 requires SWU 28 for the definition of the global variables holding the command-line options. These variables need to be changed because of the change in SWU 29. SWU 18 holds external declarations of the same variables for the dump module. Therefore, SWU 18 requires the definitions in SWU 28 and a modification in them requires a similar modification in SWU 18. These external definitions are used only in SWU 41 where automatic stretching is performed according to these options. Therefore, SWU 41 must also be modified.

A single modification causing such a modification side-effect chain reaction is something we generally wish to avoid. In this case, the interface change in SWU 1 required a capability and interface change in SWU 29. The global variable interface used to convey the command-line options is not a high level enough abstraction of this information. If we had used a user-defined type that abstracted the command-line options we would need only have changed this type's capabilities i.e., its fields, in SWU 29 and changed SWU 41 to use these fields. No other SWUs would have been affected. We can see that in some cases modifications are bound to have side-effects, but we should keep these side-effects to the necessary minimum.

It is interesting that some of the modification side-effects can be detected automatically by a CASE tool. For example, if in a SWU, a programmer changes the definition of variables used in other SWUs, the CASE tool can warn the programmer that these other SWUs must be modified as a consequence of the definition change. The programmer can then correct the other SWUs, perhaps causing other modification side-effects. Such a tool will help the programmer not to forget to modify affected SWUs in the cases it can detect.

The SWU database should hold a tree of service dependencies between the SWUs. This tree should be checked by the tool for possible modification ripple effects. If a global variable is not used any more in SWU 28 and it depends on SWU 29 for its definition then the tool should notify the programmer of this change. When the definition of a global variable is deleted, as in SWU 29, then the programmer must be notified of all the SWUs that use this variable, such as SWU 41.

Modification I-2 is an example of a modification causing a change in the domain hierarchy. The top-down search leads us from SWU 1 to SWU 6 to SWU 17 and finally to SWU 36. There, we add the functionality needed to read in the additional stretch and connectivity fields in the width tables. Keeping in line with the design philosophy of the modules we must add global variables to SWU 35 to store this additional information. We decided however to provide functions that access this new information so it does not have to be accessed through the global variables. We grouped these functions in a new SWU 48 called char_info, and as they belong to width.c we made it a sub-unit of SWU 17. New macros needed for the functions in SWU 48 were added to the macros.h, SWU 9, and it was included in file width.c therefore it became a sub-unit of SWU 6, Width.

This modification did not generate any modification side-effects, except for SWU 35, largely because it was an addition to the current interface without any changes to the previous one.

Modification C-1 calls for the viewing of manual stretch values in character tokens as part of the character. A top-down search of the domain architecture revealed that there is no character width concept in the domain. The functions in width return only the font table width of characters. Therefore, we created such a concept by creating a function `tokenBasicWidth` and another function which which

we thought is an important concept `tokenStretch`. The former returns the width of a character token before it is automatically stretched and the latter returns the total stretch amount of a character token. These were grouped in a new SWU inquire_token, SWU 42, and added as a sub-unit to SWU 13, lines.c. We then had to examine the complete code looking for calculations based on a character's width and change them to call the function `tokenBasicWidth`. This modification caused no side-effects.

Modification C-3, implementing the new automatic stretching according to the new command-line options, resulted in two fundamental changes to the domain architecture. The first fundamental change was caused by the fact that we realized SWU 41 which is the heart of the line stretching algorithm would require complete refurbishing in order to implement the modification because it does not have the abstractions necessary to represent the new required functionality. We therefore created a new SWU 43 instead with several sub-units each performing part of the line stretching algorithm with the new letter stretching functionality inside. Of course this does not mean we could not use some of the code in SWU 41 in the new SWUs. We did. However, most of SWU 43's code was completely new.

The second fundamental change in the domain architecture as a result of modification C-3 was that of finding a serious conceptual bug in the original ffortid while testing the modification. We realized that the original designers of ffortid made a serious mistake in deciding when to reverse part of the tokens in a line. This mistake is only evident in certain test cases. This realization resulted in the deletion of SWU 21, some modifications to SWU 19 and SWU 39 and the alteration of the domain architecture to reflect these changes.

Several additional minor bugs were found in the original code but they were corrected without any ripple effects or major domain architecture changes.

Figure 5.2 shows the updated domain architecture of ffortid Version 4.0 after all the above modifications were performed. As ffortid Version 4.0 included only enhancements over ffortid Version 3.0 we saw no need to preserve SWUs that have been deleted or replaced by better SWUs. In more advanced domains where one could have a choice between several components this should be reflected in the domain architecture and scope diagram.

Table 5.2 shows all information on the SWUs in the updated domain.

5.4 Implementation Comparison

The initial domain had 28 low-level SWUs with 2510 lines altogether. We deleted from the domain 3 low-level SWUs (10, 34 and 41) and one high-level SWU (21). We added 6 low-level SWUs (42, 44, 45, 46, 47, and 48) and one high-level SWU (43).

The 3 low-level SWUs deleted had altogether 634 lines (lines include comment lines). Of the 25 low-level SWUs carried on to the modified domain, modifications were made to 18 of them. Altogether 320 lines were added, 50 were deleted and 22 modified. The 6 new low-level SWUs have altogether 649 lines. The new domain therefore has 31 low-level SWUs and 2795 lines.

Berry implemented the same requirements using the same original ffortid version as his basis. He used his own SOTP systematic maintenance method to implement these requirements. The major steps in the his method were:

- Make a list of all the changes.
- Mentally plan the changes to the implementation to achieve these changes, mainly in data structures and key new algorithms.

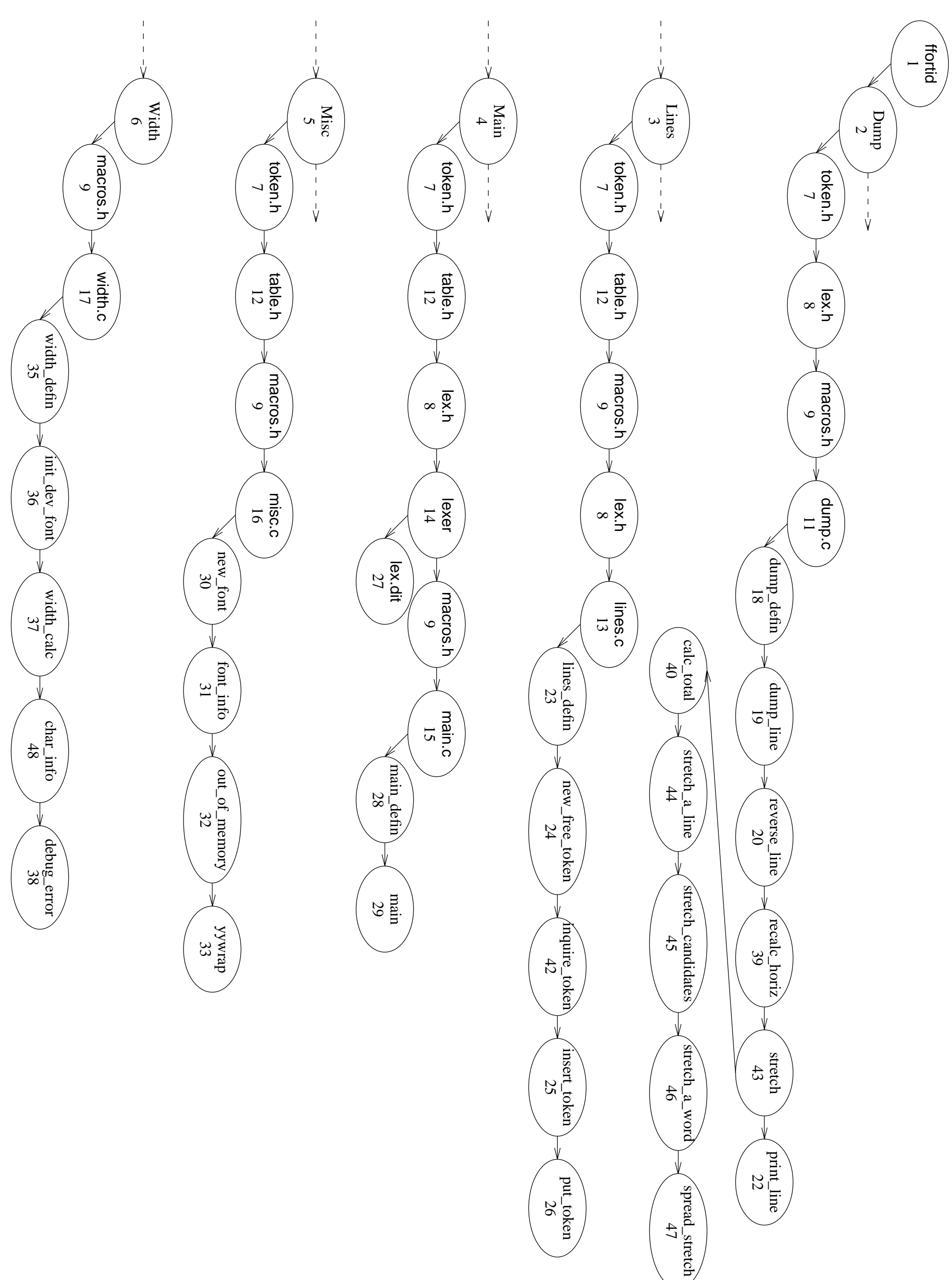


Figure 5.2: Overview of ffortid Version 4.0 domain

Num	Name	Type	Size (lines)	Low-Level
1	ffortid	Program	3803	
2	Dump	Module	1020	
3	Lines	Module	493	
4	Main	Module	1457	
5	Misc	Module	204	
6	Width	Module	629	
7	token.h	Declarations source file	39	*
8	lex.h	Definitions source file	31	*
9	macros.h	Definitions source file	33	*
11	dump.c	Source file	917	
12	table.h	Declarations source file	18	*
13	lines.c	Source file	372	
14	lexer	Lex generated source file	705	
15	main.c	Source file	631	
16	misc.c	Source file	114	
17	width.c	Source file	596	
18	dump_defin	Definitions block	30	*
19	dump_line	Procedure	125	*
20	reverse_line	Procedure	87	*
22	print_line	Procedure	21	*
23	lines_defin	Definitions block	33	*
24	new_free_token	Function group	91	*
25	insert_tokens	Procedure group	74	*
26	put_tokens	Procedure group	135	*
27	lex.dit	Lex source file	38	*
28	main_defin	Definitions block	73	*
29	main	Function	558	*
30	new_font	Procedure	42	*
31	font_info	Procedure	42	*
32	out_of_memory	Procedure	17	*
33	yywrap	Function	13	*
35	width_defin	Definitions block	51	*
36	init_dev_font	Procedure group	260	*
37	width_calc	Function group	151	*
38	debug_error	Procedure group	82	*
39	recalc_horiz	Procedure	47	*
40	calc_total	Function	55	*
42	inquire_token	Function group	45	*
43	stretch	Function group	607	
44	stretch_a_line	Function group	182	*
45	stretch_candidates	Function group	131	*
46	stretch_a_word	Function group	116	*
47	spread_stretch	Function group	123	*
48	char_info	Function group	52	*

Table 5.2: ffortid Version 4.0 software units

- Add to each change on the list, a list of modules affected by the change.
- Make hard copies of each of the modules and write in all the changes by following the list and going to each affected module. As you do this, discover additional changes necessary, so-called ripple effects, and either do them immediately if they are small and in the same module or add them to the list of changes to be done with the right modules listed next to them.
- Desk check the changes by going module by module trying to make sure that the module is consistent.
- Enter all the changes and recompile after each change.
- If you can see an order to doing the changes, i.e.
  - all changes which do not invalidate current functionality
  - all changes which change current functionality
  - all changes which add new functionality

  and each such set makes a testable program, follow that order of adding the changes and compiling and testing.

Note that this method did not include any form of reverse or reengineering. Using this method to implement the above requirements Berry found it was possible to create an order of implementation that enabled the implementation of each modification and its subsequent testing. Berry found that a typical test would expose two or three bugs and about half of these were unforeseen ripple effects.

To the original 2510 line ffortid program Berry added 1997 lines, deleted 784 lines and modified 36 lines. Therefore, his implementation of ffortid Version 4.0 had altogether 3723 lines. The most striking difference between the two implementations is the number of added lines, 321 by the author compared to 1997 by Berry. This requires some explanation.

Both implementors had the same goal in mind, to perform the minimum amount of modifications needed to implement the new requirements. However, as happens so often with programmers, they chose different designs for their implementation. Berry chose to store any information calculated in appropriate data stores so it will never need to be calculated twice. This required the addition of new data types and was not in keeping with the original design of ffortid.

The author on the other hand, as a consequence of the reverse engineering and modification method used, based his design largely on the original design. The focused search method used to find the SWU to be modified tends to focus the programmer on the current abstractions when they exist and not on creating new abstractions. Only when these do not exist in the current architectures must one justify and then add the new abstractions with as much coherence with the current design as possible. In other words, the author claims that the difference in the number of added lines between the two implementations is not an accident of different programming styles but a consequence of the methods used. Clearly one could have done exactly the same changes as the author had done without reverse engineering ffortid. However, using this method these changes came naturally and easily.

As a general remark, we want to point out another lesson we had learned during this phase of the experiment. The fact that we decided to have a written contract for the requirements, i.e., the manual page, brought to the surface mistakes and misconceptions Berry had of his program. The fact that another person, ignorant at that, had to implement the same manual page, resulted in the clarification of many points which had seemed clear to Berry, but on second thought were not well defined.

Chapter 6

ffortid Version 5.0

After we had successfully compared both implementations of the first set of requirements, we proceeded to the final experiment stage. According to the experiment design, another set of more advanced requirements must be devised and implemented by the experimenter and control on their latest application versions. Needles to say, these requirements were not known to the author before reaching this stage.

6.1 The New Requirements

The new requirements were specified as precisely as possible by Berry by writing an additional manual page describing the new version of ffortid. It can be found in Appendix D. From a comparison of the previous and new manual pages a list of 7 required enhancements was created:

1. Use new font width table information on type of connection stretching requested and accept new manual connection stretch commands.
2. Arrange words in slantable fonts on a slanted base line.
3. Use new ditroff commands to properly handle embedded text of the the opposite direction containing sub-text , e.g., numerals, of the original direction.
4. Add -msw option to prevent the automatic stretching of words containing manual stretch of any kind.
5. Use new font width table information on type of stretching requested.
6. Allow stretching of all types of characters, not just
   N named character.
7. Change -a command-line option to --.

Enhancement 1 allows fonts to specify the type of automatic connection stretching to be performed when connection stretching is needed according to the current stretch mode and place command-line options. Their are 3 possible types of connection stretching:

- Fixed filler — is the type of connection stretching used in ffortid Versions 3.0 and 4.0. Connections are fixed size fillers inserted between connecting letters.
- Stretchable filler — the use of stretchable letters allows the use of a stretchable filler character. This character is usually and normally of width zero but can be stretched to any needed length and then inserted between connecting letters.

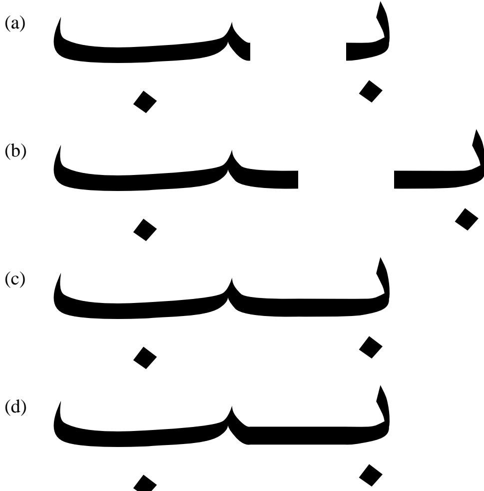


Figure 6.1: Stretchable letter connections and fillers

- Stretchable connections — the connecting portions of all connecting letters are themselves stretchable in the same way as stretchable letters are. In this case, to achieve a total connection stretch of size x, one would pass $x/2$ to each of the connecting-after portion of the before letter and the connecting-before portion of the after letter.

The stretchable filler solves the problems caused by the fact that the amount of a given connection stretch may not be integrally divisible by the width of the fixed size filler. The use of stretchable connections improves the appearance of the connection stretch by replacing the flat straight filler with a smooth curved connection.

For example, figure 6.1 (a) shows two regular connecting letters. Part (b) shows how their connecting parts are dynamically stretched by a given amount and part (c) how the stretched letters are joined. Part (d) of the same figure shows the same connecting letters stretched by inserting a stretched filler between the letters. Note the more pleasing result using the first method in (c) than using the second in (d).

This enhancement includes the acceptance of new manual connection commands for the two new types of connection stretching, i.e., manual stretchable filler commands and manual letter connection stretch commands.

Enhancement 2 enables ffortid to handle slantable fonts. These fonts have a fixed character slant which requires special handling in laying out words and lines. Each word in such a font is printed on a slanted baseline that crosses the original baseline of the line at the center of the word. Figure 6.2 shows each word's baseline as a solid arrow and the line's baseline as a dotted arrow.

Figure 6.3 shows a sample slanted output created by ffortid. Note how it handles correctly a combination of slanted, unslanted, left-to-right, and right-to-left fonts. All the command-line options and different types of stretching are available with slanted fonts as well.

Sometimes right-to-left text contains some embedded left-to-right text, such as a street address in Hebrew that contains a numeral using traditional western

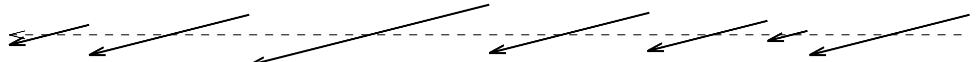


Figure 6.2: Layout of slanted font words on line

هذا مثال لطباعة وتصفيف اللغة العربية سوية

مع لغات أخرى كالأنكليزية (English) والعبرية

(עברית). الأمثلة المعطاة توضح الفرق بين كل

واحد من أساليب التصفيف ومد الكلمات.

Figure 6.3: Sample slanted output

digits with the most significant digit to the left. If this right-to-left text were embedded inside left-to-right text, e.g., an English sentence announcing a Hebrew street address, inside a left-to-right document, then the numeral, being left-to-right text, would be treated as left-to-right text that separates two right-to-left chunks inside a left-to-right document. Enhancement 3 solves this problem by recognizing two ditroff commands that surround the embedded left-to-right text and cause ffortid to recognize that the surrounding text should be treated as a single right-to-left unit.

Enhancement 4 adds a new command-line option `-msw` that prevents the automatic stretching of words containing manual connection or letter stretch commands. This is useful for preventing the messing up of finely tuned manual stretch commands by the automatic stretch mechanism.

Enhancement 5 allows better control over the type of stretching fonts provide. A new line in each font's width table describes the stretchability of the font as either connections only, letters only, or letters and connections. This enables the font to limit the type of automatic stretching allowed on the font despite the actual available stretchability of each character. Therefore, the same font can be mounted several times, each time with different stretch properties.

ffortid Version 4.0 allowed only the stretching of characters entered by their numerical code. Enhancement 6 enables the stretching of characters also entered by their ascii or two letter synonym.

Enhancement 7 is a trivial enhancement which simply changes the syntax of the `-a` command-line option, specifying the stretchable fonts, to the more intuitive `--` syntax.

6.2 Implementation

The most substantive enhancements in ffortid 5.0 are the two new types of connection stretching, slanted fonts, and the handling of embedded reversed text. These enhancements represent completely new functionality in ffortid. The rest of the enhancements are mostly technical because they only alter or improve current functionality without adding something completely new.

The author implemented these requirements using the same method described in the previous chapter. He implemented them serially, one by one, testing each new enhancement as it was implemented, and found no problem in doing this.

Berry implemented the same requirements using his own SOTP method de-

scribed in the previous chapter. He had found that unlike the first set of requirements, it was not possible for him to implement each enhancement separately. He implemented the whole program all at once. He then tested the old features first to make sure that they have been preserved and then the new features.

Chapter 7

Experiment Results

Before giving the results and conclusions from the experiment, it is necessary to describe how to compare different application versions for their amount of reuse and modification.

7.1 Measuring Reuse

Figure 7.1 shows as a Venn diagram what happens when we create a new application from some original application. Part of the original code is deleted and is not included in the product . Another part of the original code is modified and included in the product application. Finally, part of the code is reused as-is in the product application. The final product application consists of the modified and reused code from the original application and of completely new code which is added to existing code. Note that this analysis is true not only of applications but also of any other type of SWU.

In order to qualitatively measure the amount of reuse achieved in a project based on some original application producing some product application, we have defined three important ratios. These ratios use as the basic unit of their numerator and denominator, the number of code lines including comments. It is of course possible to choose some other basic unit such as lines without comments, statements etc. However, code lines have been shown to be a good estimate of code size over the years. Additionally, in my view, comments are also reused in projects and not only program statements; therefore, it is logical to count them as well in program size.

The *reuse ratio* (see Equation 7.1) measures the number of directly reused lines in the product application relative to the original application size. This ratio tells us how much of the original application was reused as-is.[1] Clearly, a small reuse ratio means that we did not reuse a lot of code from the original application. On the

[1] Some use a reuse ratio which includes not only directly reused lines but also modified lines. This measures what is known as *leveraged reuse*.

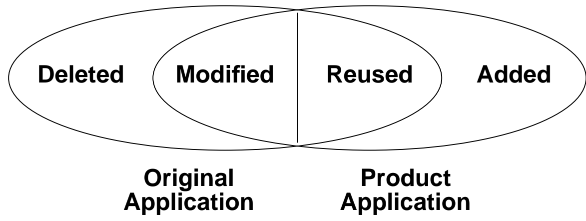


Figure 7.1: Relationship between original and product application

$$\text{Reuse Ratio} = \frac{\text{reused lines}}{\text{original lines}} \tag{7.1}$$

$$\text{Modification Ratio} = \frac{\text{modified lines}}{\text{original lines}} \tag{7.2}$$

$$\text{Addition Ratio} = \frac{\text{added lines}}{\text{product lines}} \tag{7.3}$$

	Del. Lines	Mod. Lines	Added Lines	Final Lines	Implementation Time
Experiment 4.0	684	22	969	2795	—
Control 4.0	784	36	1997	3723	—
Experiment 5.0	126	23	947	3616	39
Control 5.0	44	82	789	4468	77-94.5

Table 7.1: Experiment results

other hand, a large reuse ratio indicates a large amount of reuse from the original application.

The *modification ratio* (see Equation 7.2) is similar to the reuse ratio except that it measures the number of modified lines in the product application relative to the original application size.[2]

The *addition ratio* (see Equation 7.3) measures the number of new lines in the product application relative to the product application size. This measure is important because it tells us how much of the final application is from completely new code and how much is from directly reused and modified code ($1 -$ addition ratio). It is possible to have a case with a high reuse ratio and a high addition ratio. This indicates that although we reused a large proportion of our original application, this reuse amounts to only a small fraction of our final product application and therefore we cannot say we have reached a high level of reuse altogether. In fact, such a situation would probably imply that the whole reuse effort was futile and that perhaps it was better to create the complete product application from scratch as most of it was created so in any case.

Therefore, in order to state that a high level of reuse was achieved in a certain project we should have a large reuse, or leveraged reuse, ratio and a small addition ratio. How much is large and small? That depends on the specific project and its goals. If we implement two different versions of the same application, the comparison is simpler because we can compare the ratios of each version and decide accordingly which had a higher level of reuse.

7.2 Results

For each application version of the author and the control we recorded the number of deleted, modified, and added lines from the original application it was derived from. Additionally, we recorded the final number of lines in the product application and the implementation time in hours, including testing, of the second version. The results for all the different application in the experiment can be found in Table 7.1.

Both the author's and the control's versions of ffortid 4.0 started from the same application – ffortid 3.0, which had 2510 lines. Both deleted approximately the same number of lines and modified nearly the same insignificant number of lines. The

[2] Therefore adding the reuse and modification ratios gives us a leveraged reuse ratio.

	Reuse Ratio	Modification Ratio	Addition Ratio
Experiment 4.0	72%	1%	35%
Control 4.0	67%	1.5%	54%
Experiment 5.0	95%	1%	26%
Control 5.0	97%	2%	18%

Table 7.2: Experiment results analysis

major difference between the versions is the number of added lines. The control added more than twice the number of lines the author added in order to achieve the same functionality, and therefore his application version was much larger, almost 1,000 lines more, than the author's (See Section 5.4).

The reason for this wide difference is that the control took a design decision different from that of the author, adding a new complex data structure and all the code needed to initialize, handle and extract the information in this new data structure. Clearly, this decision was not mandatory, as the author achieved the same functionality without adding this new data structure.

Remember that both implementors had the same goal in mind: to perform the minimum amount of modifications needed to implement the new requirements. However, as happens so often with programmers, they chose different designs for their implementation. Berry chose to store any information calculated in appropriate data stores so it will never need to be calculated twice. This required the addition of new data types and was not in keeping with the original design of ffortid.

The author, on the other hand, as a consequence of the reverse engineering and modification method used, based his design largely on the original design. The focused search method used to find the SWU to be modified tends to focus you on the current abstractions when they exist and not on creating new abstractions. Only when these do not exist in the current architecture must one justify and then add the new abstractions with as much coherence with the current design as possible. In other words, it is claimed that the difference in the number of added lines between the two implementations is not an accident of different programming styles but a consequence of the methods used. One could have done exactly the same changes the author had done without reverse engineering ffortid. However, using this method these changes came naturally and easily.

No implementation time is recorded for either application version of ffortid 4.0 for two reasons: First, the control's version was written before this experiment was conceived and therefore no time recordings were performed. Secondly, the author was learning and inventing his method as the application was being created and therefore it is not possible to objectively compare the two versions implementation times.

The analysis of these results can be found in Table 7.2. As expected from the collected data, the direct reuse ratio of both versions was more or less the same, as both deleted and modified more or less the same number of lines. The modification ratio of both versions was small and insignificant. Finally, the addition ratios of both versions were quite different. The addition ratio of the author (35%) is much lower than the addition ratio of the control (54%), because he added significantly fewer new lines to the product application.

According to the analysis in the previous section, the author's version 4.0 had a higher level of reuse of the original application, ffortid 3.0, than the control's because it has a slightly larger reuse ratio and a significantly lower addition ratio. In order to judge which ffortid 4.0 version was more reusable, i.e. which method resulted in a more reusable application, we must compare the level of reuse achieved by each

method in the second experiment step.

The author and the control both started their versions of ffortid 5.0 from their individual versions of ffortid 4.0. Both the author and the control deleted very few lines from their original application although, as shown in Table 7.1, the author deleted more lines than the control. This indicates that both implementors added good code to ffortid 4.0 versions because they reused most of it in the their ffortid 5.0 versions. Both the author and the control modified very few lines, with the control modifying more lines, and both added nearly the same number of new lines. The final application of the control, therefore, still had significantly more lines than the author's because he started with a larger original application.

Implementation time was recorded in this experiment step. The control recorded a minimum and maximum implementation time because he could not give exact hours due to the nature of his working environment. He was interleaving other professional duties while coding and was continually thinking even while not coding. Even if we take the minimum hours recorded by the control, they are approximately twice the implementation hours of the author. I believe this can be attributed to the fact that the author's original application was documented in the form of a domain, while the control had only the immediate documentation in the code itself. This documentation allowed the author to quickly trace where each modification needs to be performed using the SWU architecture. The control on the other hand had only the decomposition of his application into modules to guide him. In the author's view, the overall result of this was better software understanding by the author and the ability to perform modifications quicker.

The analysis of these results using the above ratios can be found in Table 7.2. The direct reuse ratios of both versions are similar and very high. The modification ratios of both versions are similar and insignificantly small. The addition ratio of the author is higher than the control's because he added slightly more lines but had a smaller final application size therefore resulting in the higher addition ratio. If we take the different original application size into account, there is no significant difference between the version's addition ratios.

7.3 Conclusions

There was no significant difference between the reuse and modification ratios of both methods in both experiment steps. There was a significant difference in the addition ratio of the methods in ffortid 4.0. As explained in the previous section, I believe this shows the advantage of using the proposed method which directs the implementor to use existing SWUs over creating new abstractions.

Although there was no significant difference in the ratios of the different implementations of ffortid 5.0, if we examine the overall results, the author claims they indicate clearly that the author's version of ffortid 4.0 was more reusable than the control's because:

- It took a significantly shorter time to perform the necessary changes on it.
- It has a significantly smaller application size.
- It is better documented.
- It has higher quality code.[3]

[3] This conclusion was reached by examining both applications source code and comparing basic principles of software engineering such as function length etc. This is clearly a subjective conclusion.

It is, however, necessary to put some hedges over these results. Although we had taken several steps to make sure the results of the experiment are valid and that we had taken into account the possible difference between the implementors capabilities it is still possible that the author was a much better programmer than the control and this explains the difference in the implementation time, etc. The author does not believe this to be the case, but in any case further case studies and formal experiments are needed in order to strengthen these results.

Another point to keep in mind in such experiments, is that there is a possible difference between the actions of an ignorant and knowledgable person. It is well known that an ignorant person performs better on some tasks because of a lack of tacit assumptions a knowledgable person makes due to his increased knowledge [23]. This can also be a possible explanation for the different design decision taken by the control in his implementation of ffortid 4.0.

The author has shown it is possible to use the proposed domain reengineering method to produce an initial domain, and has proposed a systematic method for the implementation of requirements in such a domain. In the current case study, this method did result in a high level of reuse and low level of modification. This method can be highly automated using dedicated CASE tools.

Regarding the SWU theory refined in the thesis, it has been shown that it is applicable to real software and that it can be used to document the architecture and design of existing systems and to advance the systematic implementation of new requirements. It also allows a high degree of automation of the proposed method using CASE tools which are a must if we wish to use such methods of reengineering on large scale systems.

In the author's view, only the use of such methods and automated tools offer hope of handling the problem of maintaining and reusing the large number of legacy systems that exist. Further experiments, perhaps with real, full scale, legacy systems, using state-of-the-art CASE tools will help refine the proposed methods and advance the technology of software reengineering. Only with such experiments will we be able to realize the full potential of these technologies.

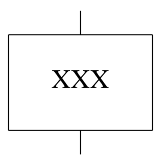
As a final not I would like to quote a remark pointed out by Berry while reading Section 5.3 "Now I am beginning to understand the advantage you had in tracking things down to avoid ripple effects and to just plain find what to change and the sources of bugs!". This remark recognizes the potential in the methods used in this experiment.

7.4 Acknowledgments

I wish to thank Prof. Daniel M. Berry who not only performed all the functions of an advisor in an exceptional way but also performed a vital part of the experiment itself by acting as its control. I would also like to thank Prof. Noah Prywes for describing to me his methods on which this thesis is based and for exchanging ideas during the experiment.

Appendix A

SFD Icons

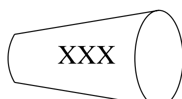


Assignment

a=b+c

XXX is the name of the variable on the left hand side of the assignment.

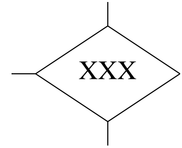
Procedure/Function Call



my_procedure(arg1,arg2)

XXX is the procedure name.

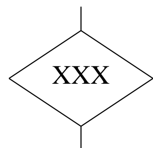
Condition



if (my_var)...else...   switch(c)

XXX is either IF or SWITCH.

Simple Condition



if (my_var)

XXX is always IF without an else.

IO File

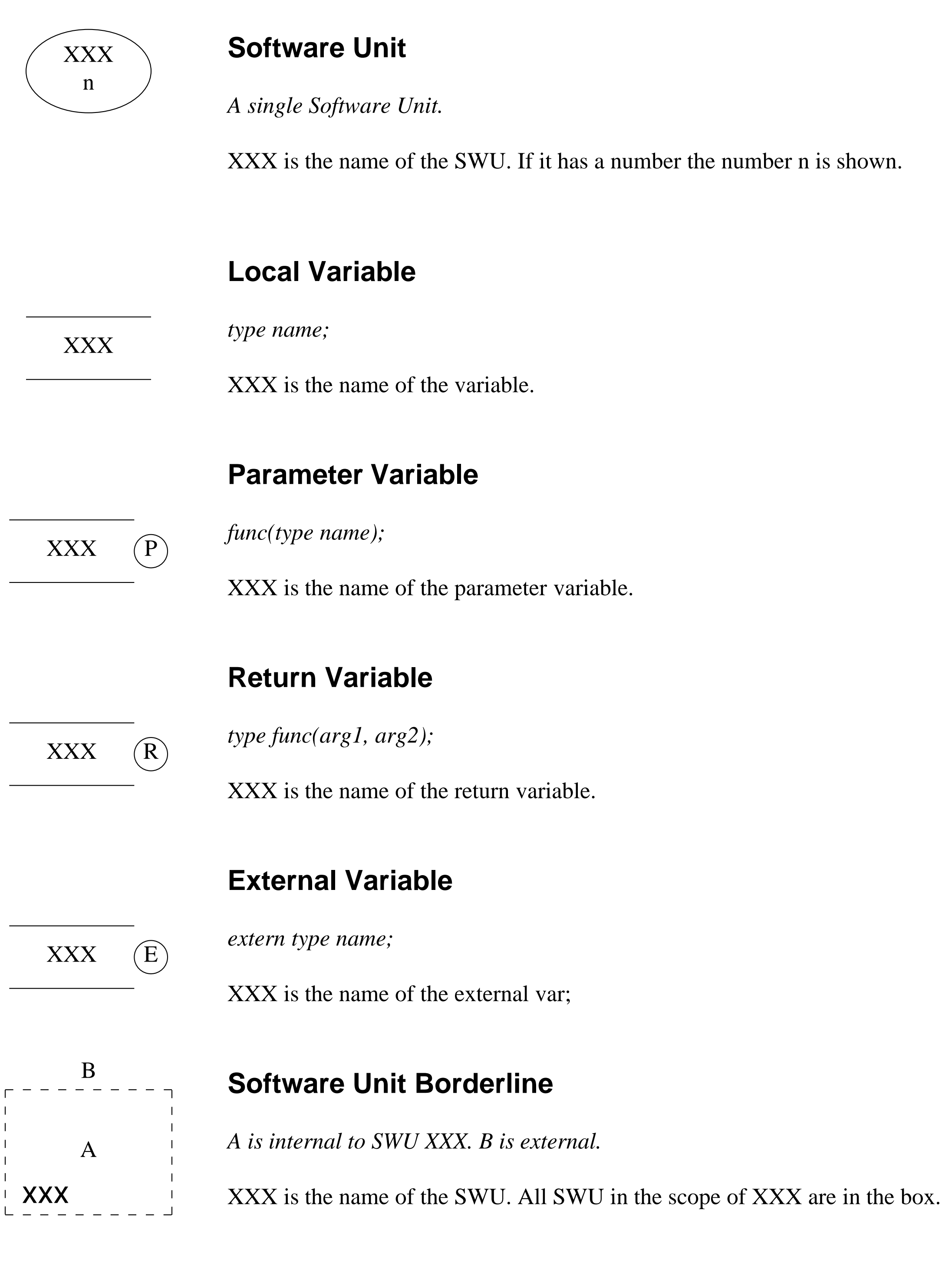
FILE fd;*

XXX is the name of the variable or the name of the file in quotes.

Loop

for(i=0;i<n;i++)...   while (cond) do ...   do statement while (cond)

XXX is the type of statment, e.g. FOR, WHILE or DO.



Software Unit

A single Software Unit.

XXX is the name of the SWU. If it has a number the number n is shown.

Local Variable

type name;

XXX is the name of the variable.

Parameter Variable

func(type name);

XXX is the name of the parameter variable.

Return Variable

type func(arg1, arg2);

XXX is the name of the return variable.

External Variable

extern type name;

XXX is the name of the external var;

Software Unit Borderline

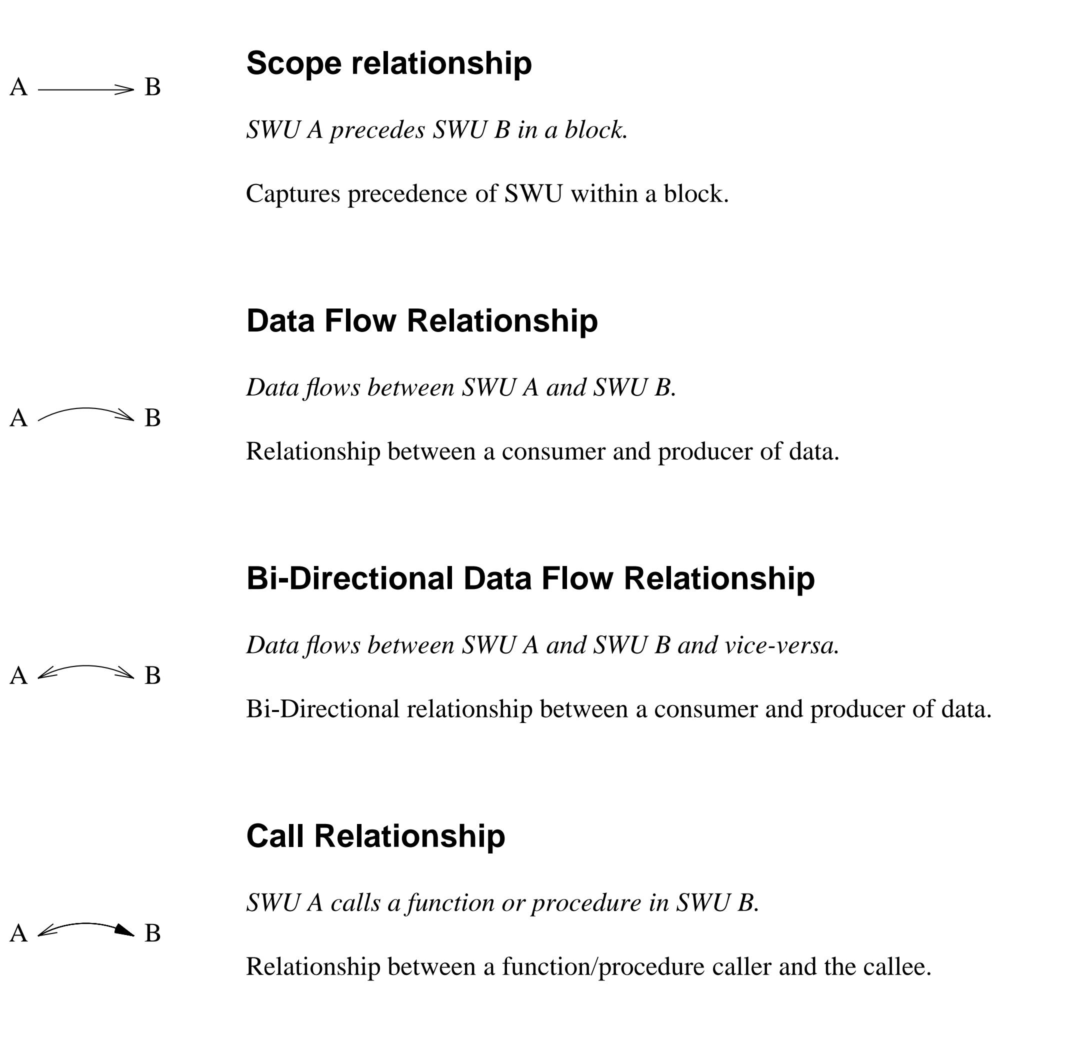
A is internal to SWU XXX. B is external.

XXX is the name of the SWU. All SWU in the scope of XXX are in the box.

Function/Procedure parameters group

proc is the name of the function/procedure. YYY is a parameter.

Groups function/procedure parameters and return value for SWU entry point.



Scope relationship

SWU A precedes SWU B in a block.

Captures precedence of SWU within a block.

Data Flow Relationship

Data flows between SWU A and SWU B.

Relationship between a consumer and producer of data.

Bi-Directional Data Flow Relationship

Data flows between SWU A and SWU B and vice-versa.

Bi-Directional relationship between a consumer and producer of data.

Call Relationship

SWU A calls a function or procedure in SWU B.

Relationship between a function/procedure caller and the callee.

Use relationship

SWU B uses declerations or definitions in SWU A.

Relationship between decleration/definition in a SWU and it’s use in another SWU.

Appendix B

ffortid Ver 3.0 Manual Page

NAME

ffortid − in dtroff output, find and reverse all text in designated right-to-left fonts and carry out stretching in Arabic and Farsi text

SYNOPSIS

ffortid [−**r***font-position-list*] ... [−**w***paperwidth*] [−**a***font-position-list*] ...
[−**s**[**n**|**f**|**l**|**a**]] [*file*] ...

DESCRIPTION

ffortid's job is to take the *dtroff*(1) output which is formatted strictly left-to-right, to find occurrences of text in a right-to-left font and to rearrange each line so that the text in each font is written in its proper direction. *ffortid* deals exclusively with *dtroff* output, it does not know and does not need to know anything about any of *dtroff*'s preprocessors. Therefore, the results of using *ffortid* with any of *dtroff*'s preprocessors depends only on the *dtroff* output generated as a result of the input to *dtroff* from the preprocessors. Furthermore, the output of *ffortid* goes on to the same device drivers to which the *dtroff* output would go; therefore, *ffortid*'s output is in the same form as that of *dtroff*.

In the command line, the −**r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. A *font-position-list* is a list of font positions separated by white space, but with no white space at the beginning. *ffortid*, like *ditroff*, recognizes up to 256 possible font positions (0-255). The actual number of available font positions depends only on the typesetting device and its associated *ditroff* device driver. The default font direction for all possible font positions is left-to-right. Once the font direction is set, either by default or with the −**r***font-position-list* argument, it remains in effect throughout the entire document. Observe then that *ffortid*'s processing is independent of what glyphs actually get printed for the mounted fonts. It processes the designated fonts as right-to-left fonts even if, in fact, the alphabet is that of a left-to-right language. In fact, it is possible that the same font be mounted in two different positions, only one of which is designated as a right-to-left font position. This is how a single font can be printed left-to-right and right-to-left in the same document.

In addition to the specified font directions, the results of *ffortid*'s reformatting also depends on the document's *current formatting direction*, which can be either left-to-right or right-to-left. The default formatting direction is left-to-right and can be changed by the user at any point in the document through the use of the

```
x X PL
```

and

```
x X PR
```

commands in the *dtroff* output. These commands set the current formatting direction to left-to-right and right-to-left, respectively. These commands are either meaningless or erroneous to *dtroff* device drivers; therefore they are removed by *ffortid* as they are obeyed. These commands can be generated by use of

```
\X'PL'
```

and

```
\X'PR'
```

escapes in the *dtroff* input. For the convenience of the user, two macros

```
.PL
```

and

```
.PR
```

are defined in the *−mX2* and *−mXP* macro packages, that cause generation of the proper input to *ffortid*. They are defined by

```
..de PL
\\X'PL'
..
.de PR
\\X'PR'
..
```

If the current formatting direction is left-to-right, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from left to right. In each *dtroff* output line, any sequence of contiguous right-to-left font characters is reversed in place.

If the current formatting direction is right-to-left, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from right to left. Each *dtroff* output line is reversed, including both the left and right margins. Then, any sequence of contiguous left-to-right font characters is reversed in place.

The **−w***paperwidth* argument is used to specify the width of the paper, in inches, on which the document will be printed. As explained later, *ffortid* uses the specified paper width to determine the width of the right margin. The default paper width is 8.5 inches and like the font directions, it remains in effect throughout the entire document.

It is important to note that *ffortid* uses the specified paper width to determine the margin widths in the reformated output line. For instance, suppose that a document is formatted for printing on paper 8.5 inches wide with a left margin (page offset) of 1.5 inches and a line length of 6 inches. This results in a right margin of 1 inch. Suppose then that the text specifies a current formatting direction of right-to-left. Then, *ffortid*'s reformatting of the output line results in left and right margins of 1 and 1.5 inches, respectively. This margin calculation works well for documents formatted entirely in one direction. However, as a document's formatting direction changes, the resulting margins widths are exchanged. Thus, **.PL**'s right and left margins end up not being the same as **.PR**'s right and left margins. The user can make *ffortid* preserve the left and right margins by specifying, with the **−w***paperwidth* argument, a paper width other than the actual paper width. This artificial paper width should be chosen so that both margins will appear to *ffortid* to be as wide as the desired left margin. For example, for the document mentioned above, a specified paper width of 9.0 inches results in reformatted left and right margins of 1.5 inches each. The resulting excess in the right margin is just white space that effectively falls of the edge of the paper and does not effect the formatting of the document.

There is one exception to these simple rotation rules in that *ffortid*, at present, makes no attempt to reverse any of *dtroff*'s drawing functions, such as those used by *pic*(1) and *ideal*(1) (which are also available directly to the user). It is therefore suggested that these drawing functions, and thus *pic* and *ideal*, be used only when the current formatting direction is left-to-right. Additionally, due to the cleverness of the *dtroff* output generated by most substantial *eqn*(1) equations, it is suggested that *eqn*'s use also be limited to a left-to-right formatting direction for all but the simplest forms of equations. These rules do not in any way restrict the use of right-to-left fonts in the text dealt with by any of the preprocessors, but simply suggest that these particular preprocessors be used only when the formatting direction is left-to-right.

An additional point to keep in mind when preparing input both for *dtroff*'s preprocessors and for *dtroff* itself is that *ffortid* rotates, as a unit, each sequence of characters of the same direction. In order to force *ffortid* to rotate parts of a sequence independently, as for a *tbl*(1) table, one must artificially separate them with a change to a font of the opposite direction.

The **−a***font-position-list* argument is used to indicate which fonts positions, generally a subset of those designated as right-to-left, contain fonts for Arabic, Farsi, or related languages. For these fonts, left and right justification of a line is achieved by stretching instead of inserting extra white space between the

words in the line. Stretching is done on a line only if the line contains at least one word in a −**a** designated font. If so, stretching is used in place of extra white space insertion for the entire line. There are several kinds of stretching, and which is in effect for all −**a** designated fonts is specified with the −**s** option, described below. If it is desired not to stretch a particular Arabic, Farsi, or other font, while still stretching others, then the particular font should not be listed in the −**a***font-position-list*. Words in such fonts will not be stretched and will be spread with extra white space if the containing line is spread with extra white space. The −**r** and the −**a** specifications are independent. If a font is in the −**a***font-position-list* but not in the −**r***font-position-list*, then its text will be stretched but not reversed. This independence can be used to advantage when it is necessary to designate a particular Arabic, Farsi, or other font as left-to-right for examples or to get around the above mentioned limitations in the use of *eqn*, *ideal*, *pic*, or *tbl*.

The kind of stetching to be done for all fonts designated in the −**a***font-position-list* is indicated by the −**s** argument. The choices are:

−sn

Do no stretching at all for all the fonts.

−sf

Stretch the last stretchable word on each line. A stretchable word is a word containing a stretchable character (if the font is dynamic) or a stretchable connection to a character (if the font has a straight base line). *Currently only stretchable connections to characters are handled; a future version will deal with dynamic fonts*. If no stretchable word exists on the line, then spread the words in the line as does *dtroff*.

−sl

Stretch the last stretchable word on each line. If the amount of stretch for that word is longer than a limit equal to the current point size times the length of the base line filler used to achieve the stretched connection, then stretch the penultimate stretchable word up to that limit, and if necessary, then stretch the stretchable word before that, etc. If no stretchable word exists on the line, or some extra stretch is left after stretching all stretchable words to the limit, then spread the words in the line as does *dtroff*.

−sa

Stretch all stretchable words on each line by the same amount (different amount for each line). If no stretchable word exists on the line, then spread the words in the line as does *dtroff*. This is the default for all −**a** designated fonts.

FILES

/usr/lib/tmac/tmac.*	standard macro files
/usr/lib/font/dev*/*	device description and font width tables

SEE ALSO

Cary Buchman, Daniel M. Berry, *User's Manual for Ditroff/Ffortid, An Adaptation of the UNIX Ditroff for Formatting Bi-Directional Text,*
Johny Srouji, Daniel M. Berry, *An Adaptation of the UNIX Ditroff for Arabic Formatting*
troffort(l), ptrn(l)

Appendix C

ffortid Ver 4.0 Manual Page

NAME

ffortid − in dtroff output, find and reverse all text in designated right-to-left fonts and carry out stretching in Arabic, Hebrew, and Persian text

SYNOPSIS

ffortid [−**r***font-position-list*] [−**w***paperwidth*] [−**a***font-position-list*] ...
[−**s**[**n**|[[**l**|**c**|**e**|**b**][**f**|**2**|**m**[*amount*]|**a**|**ad**|**al**]]] [−**ms**[**c**|**l**] ...

DESCRIPTION

ffortid's job is to take the *dtroff*(1) output which is formatted strictly left-to-right, to find occurrences of text in a right-to-left font and to rearrange each line so that the text in each font is written in its proper direction. *ffortid* deals exclusively with *dtroff* output, it does not know and does not need to know anything about any of *dtroff*'s preprocessors. Therefore, the results of using *ffortid* with any of *dtroff*'s preprocessors depends only on the *dtroff* output generated as a result of the input to *dtroff* from the preprocessors. Furthermore, the output of *ffortid* goes on to the same device drivers to which the *dtroff* output would go; therefore, *ffortid*'s output is in the same form as that of *dtroff*. *ffortid* reads its input from the standard input and write to the standard output.

In the command line, the −**r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. A *font-position-list* is a list of font positions separated by white space, but with no white space at the beginning. *ffortid*, like *ditroff*, recognizes up to 256 possible font positions (0-255). The actual number of available font positions depends only on the typesetting device and its associated *dtroff* device driver. The default font direction for all possible font positions is left-to-right. Once the font direction is set, either by default or with the −**r***font-position-list* argument, it remains in effect throughout the entire document. Observe then that *ffortid*'s processing is independent of what glyphs actually get printed for the mounted fonts. It processes the designated fonts as right-to-left fonts even if, in fact, the alphabet is that of a left-to-right language. In fact, it is possible that the same font be mounted in two different positions, only one of which is designated as a right-to-left font position. This is how a single font can be printed left-to-right and right-to-left in the same document.

In addition to the specified font directions, the results of *ffortid*'s reformatting also depends on the document's *current formatting direction*, which can be either left-to-right or right-to-left. The default formatting direction is left-to-right and can be changed by the user at any point in the document through the use of the

x X PL

and

x X PR

commands in the *dtroff* output. These commands set the current formatting direction to left-to-right and right-to-left, respectively. These commands are either meaningless or erroneous to *dtroff* device drivers; therefore they are removed by *ffortid* as they are obeyed. These commands can be generated by use of

\X'PL'

and

\X'PR'

escapes in the *dtroff* input. For the convenience of the user, two macros

.PL

and

```
.PR
```

are defined in the **−mX2** and **−mXP** macro packages, that cause generation of the proper input to *ffortid*. They are defined by

```
..de PL
\\X'PL'
..
.de PR
\\X'PR'
..
```

If the current formatting direction is left-to-right, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from left to right. In each *dtroff* output line, any sequence of contiguous right-to-left font characters is reversed in place.

If the current formatting direction is right-to-left, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from right to left. Each *dtroff* output line is reversed, including both the left and right margins. Then, any sequence of contiguous left-to-right font characters is reversed in place.

The **−w***paperwidth* argument is used to specify the width of the paper, in inches, on which the document will be printed. As explained later, *ffortid* uses the specified paper width to determine the width of the right margin. The default paper width is 8.5 inches and like the font directions, it remains in effect throughout the entire document.

It is important to note that *ffortid* uses the specified paper width to determine the margin widths in the reformated output line. For instance, suppose that a document is formatted for printing on paper 8.5 inches wide with a left margin (page offset) of 1.5 inches and a line length of 6 inches. This results in a right margin of 1 inch. Suppose then that the text specifies a current formatting direction of right-to-left. Then, *ffortid*'s reformatting of the output line results in left and right margins of 1 and 1.5 inches, respectively. This margin calculation works well for documents formatted entirely in one direction. However, as a document's formatting direction changes, the resulting margins widths are exchanged. Thus, **.PL**'s right and left margins end up not being the same as **.PR**'s right and left margins. The user can make *ffortid* preserve the left and right margins by specifying, with the **−w***paperwidth* argument, a paper width other than the actual paper width. This artificial paper width should be chosen so that both margins will appear to *ffortid* to be as wide as the desired left margin. For example, for the document mentioned above, a specified paper width of 9.0 inches results in reformatted left and right margins of 1.5 inches each. The resulting excess in the right margin is just white space that effectively falls of the edge of the paper and does not effect the formatting of the document.

There is one exception to these simple rotation rules in that *ffortid*, at present, makes no attempt to reverse any of *dtroff*'s drawing functions, such as those used by *pic*(1) and *ideal*(1) (which are also available directly to the user). It is therefore suggested that these drawing functions, and thus *pic* and *ideal*, be used only when the current formatting direction is left-to-right. Additionally, due to the cleverness of the *dtroff* output generated by most substantial *eqn*(1) equations, it is suggested that *eqn*'s use also be limited to a left-to-right formatting direction for all but the simplest forms of equations. These rules do not in any way restrict the use of right-to-left fonts in the text dealt with by any of the preprocessors, but simply suggest that these particular preprocessors be used only when the formatting direction is left-to-right.

An additional point to keep in mind when preparing input both for *dtroff*'s preprocessors and for *dtroff* itself is that *ffortid* rotates, as a unit, each sequence of characters of the same direction. In order to force *ffortid* to rotate parts of a sequence independently, as for a *tbl*(1) table, one must artificially separate them with a change to a font of the opposite direction.

The **−a***font-position-list* argument is used to indicate which fonts positions, generally a subset of those designated as right-to-left, contain fonts for Arabic, Hebrew, Persian, or related languages, whose fonts support stretching of letters and/or connections. For these fonts, left and right justification of a line

can be achieved by stretching instead of inserting extra white space between the words in the line. If requested by use of the **−s** argument described below, stretching is done on a line only if the line contains at least one word in a **−a** designated font. If so, stretching is used in place of the normal distributed extra white space insertion for the entire line. The intention is that stretching soak up all the excess white space inserted by *dtroff* to adjust the line. If there are no opportunities for stretching or there are too few to soak up all the excess white space, what is not soaked up is distributed in between the words according to *dtroff*'s algorithm. There are several kinds of stretching, and which is in effect for all **−a** designated fonts is specified with the **−s** argument, described below. If it is desired not to stretch a particular Arabic, Hebrew, Persian, or other font, while still stretching others, then the particular font should not be listed in the **−a***font-position-list*. Words in such fonts will not be stretched and will be spread with extra white space if the containing line is spread with extra white space. The **−r** and the **−a** specifications are independent. If a font is in the **−a***font-position-list* but not in the **−r***font-position-list*, then its text will be stretched but not reversed. This independence can be used to advantage when it is necessary to designate a particular Arabic, Hebrew, Persian, or other font as left-to-right for examples or to get around the above mentioned limitations in the use of *eqn*, *ideal*, *pic*, or *tbl*.

The kind of stetching to be done for all fonts designated in the **−a***font-position-list* is indicated by the **−s** argument. There are two relatively independent dimensions that must be set to describe the stretching, what is stretched and the places that are stretched. A stretch argument is of the form

−s*mp*
or
−sn

where *m* specifies the stretching mode, i.e, what is stretched, and *p* specifies the places that are stretched. The *m* and *p* must be given in that order and with no intervening spaces. The **−sn** means that there is *no* stretching and normal spreading of words is used even in **−a** designated fonts. The choices for the mode *m* are:

l (letter ell)
In the words designated by the *p*, stretch the last stretchable letter.

c
In the words designated by the *p*, stretch the last connection to a letter.

e
In the words designated by the *p*, stretch either the last stretchable letter or the last connection to a letter, whichever comes later.

b
In the words designated by the *p*, stretch either the last stretchable letter or the last connection to a letter, whichever comes later, and if it is a letter that is stretched and it is a connect-previous letter then also stretch the connection to the letter.

To our knowledge, all Arabic and Persian fonts, have a baseline filler that can be used to achieve the stretching of connections. It is fairly easy for such a filler to be added to any font definition that does not have it, and moreover to make it the character that is addressed by **\(hy**, which is normally the code for the hyphen character. (Therefore no account is taken of the possibility that stretching connections is not possible.) Since Arabic and Persian do not have a hyphen and hyphenation is turned off when in an Arabic or Persian font, it is safe to use **\(hy** to name the filler. Of course, this requires that the width table for Arabic and Persian fonts have an entry for **hy** designating the filler character in the font, for example:

```
hy    15 0 0267   filler
```

Giving the filler character an explicit *dtroff* two-character name allows *dtroff* to deal with it uniformly despite that it might be in a different position in each font.

On the other hand, stretching of letters requires a dynamic font which, by its very nature of not having a constant bitmap for a given font name, point size, and character name, cannot be type 1 (in PostScript terminology) and cannot be a bitmapped font. Therefore, not all Arabic, Hebrew, and Persian fonts support stretching of letters. Moreover, within a dynamic font, not all characters are stretchable. Historically, only characters with strong horizontal components are stretchable, such as those in the stand-alone and connect-previous forms of the *baa* family. Obviously, one cannot stretch totally vertical characters such as *alif*. Therefore, it is necessary to specify by additional information in the *ditroff* width table for a font which characters are stretchable. In the width table for an Arabic, Hebrew, or Persian font, for each character that is not also an ASCII character, i.e., not also a digit or punctuation, and thus is neither connected or stretchable, one specifies after the name, width, ascender-descender information, and code, two additional fields, the connectivity and the stretchability of the character, in that order. The connectivity is either

N for stand-alone,
A for connect-after,
P for connect-previous,
B for connect-both, or
U for unconnected (because it is punctuation or a diacritical, etc.),

and the stretchability is either

N for not stretchable,
S for stretchable,

Some examples of width table lines are:

```
%      125 2 045   percent

---    55 0 0101   U      N      comma
---    70 0 0105   U      N      hamza

---    129 0 0106  N      S      baa_SA
---    36 2 0102   N      N      alef_SA

---    113 0 0177  A      N      sad_CA
---    66 2 0215   A      S      caf_CA

---    43 2 0225   P      N      alef_CP
---    120 0 0274  P      S      baa_CP

---    53 0 0230   B      N      baa_CB
---    73 2 0261   B      S      caf_CB
```

Recall that `---` in the name field of a character means that it can be addressed only by **\N'***n***'**, where *n* is the decimal equivalent of the character's code. Only such lines will have the connectivity and stretchability fields.

For a Hebrew font, for which there is no notion of connectivity of letters, and therefore, the position of the letters is irrelevant for deciding stretching, there is only the possibility of stretching letters. Some examples of width table lines for such fonts are:

```
%      132 3 045   percent

---    95 3 0140   U      N      quoteleft=alef
---    92 3 0141   U      S      a=bet
```

Below, "stretchable unit" refers to what is a candidate for stretching according to the mode. The choices for *p*, which specifies places of stretching, are:

f

In any line, stretch the last stretchable unit.

2

Assuming that the mode is **b** (both), in any line, stretch the last two stretchable units, if they are the connection leading to a stretchable connect-previous letter and that letter, and stretch only the last stretchable unit otherwise. If the mode is not **b**, then this choice of places is illegal.

m*n* or **m**

In any line, stretch the last stretchable unit by an amount not exceeding *n* emms. If that does not exhaust the available white space, then stretch the next last stretchable unit by an amount not exceeding *n* emms, and so on until all the available white space is exhausted. If *n* is not given, it is assumed to be **2.0**. In general *n* can be any number in floating point format.

a, **ad**, or **al**

In any line, stretch all stretchable units. In this case, the total amount available for stretching is divided evenly over all stretchable units on the line identified according to the mode. Since the units of stretching are the units of device resolution, the amount available might not divide evenly over the number of places. Therefore, it is useful to be able to specify what to do with the remainder of this division. This specification is given as an extension of the stretching argument. The choices are **d** or **l**, with the former indicating that the excess be distributed as evenly as possible to the spaces between words and the latter indicating that the excess be distributed as evenly as possible in stretchable letters that were stretchable units according to the current mode and place. The latter is the default if no choice is specified. The stretched item for the **l** choice must be a letter rather than a connection because only a stretchable letter is stretchable to any small amount that will be the remainder.

In general, the stretch is divided as evenly as possible between all stretchable units in a line. Specificly, in stretch mode **b**, if we have a connection leading to a stretchable connect-previous letter and that letter, then any stretch remainder we have from stretching the connection will be added to the stretch of the letter.

Sometimes, it is desirable to be able to manually stretch connections or letters to achieve special effects, e.g., more balanced stretching or stretching in lines that are not otherwise adjusted, e.g., centered lines. Stretching a connection can be achieved by using the baseline filler character explicitly as many times as necessary to achieve the desired length. Note that the *troff* line drawing function can be used to get a series of adjacent fillers to any desired length, e.g.,

```
\l'2m\(hy'
```

will draw a string of adjacent base-line fillers of length 2 emms.

To achieve stretching of letters, one should immediately preceed, with no intervening white space, the letter to be stretched by

```
\X'stretch'\h'n'
```

where *n* is a valid length expression in *troff*'s input language. *ffortid* is prepared to deal with the output from *dtroff* generated by this input to generate output that will cause the letter immediately following it to be stretched by the length specified in *n*. For example,

```
\X'stretch'\h'1m'\N'70'
```

will cause the character whose decimal code is 70 to be stretched by 1 emm. The output will fail to have the desired effect if the letter following is not a stretchable letter.

For finer control over stretching, it may be desirable to inhibit automatic stretching on manually stretched connections and letters. In particular, when manual stretching is done on a letter or its connection for balancing purposes, one does not want additional automatic stretching to be done on the same to mess up the balance. Accordingly, two command line flags are provided for this purpose:

−msc

Do not automatically stretch manually stretched connections.

−msl

Do not automatically stretch manually stretched letters.

These flags are understood as eliminating potential stretching places, letters or connections, that were identified on the basis of the stretch mode, **l**, **c**, **e**, or **b**. (In the following description, parenthesized text is a comment stating what is true at this point and not what needs to be done.)

For any letter *l* that is a candidate for stretching by the mode,

> **if** both the letter itself and its connection to the previous letter are candidates **then**
>
> > **if** either kind of manual stretch is in the letter and that kind of manual stretch cannot be stretched additionally, **then** neither part of *l* is any longer a candidate;
>
> **otherwise** (only the letter itself is a candidate OR only its connection to the previous letter is a candidate)
>
> **if** the letter itself is a candidate for stretching by the mode,
>
> > **if** there is manual stretching in the letter and manually stretched letters cannot be stretched more, **then** *l* is no longer a candidate;
>
> **otherwise** (the connection of *l* is a candidate for stretching by the mode),
>
> > **if** there is manual stretching in the connection of *l* to the previous letter and manually stretched connections cannot be stretched more, **then** *l* is no longer a candidate.

FILES

/usr/lib/tmac/tmac.∗	standard macro files
/usr/lib/font/dev∗/∗	device description and font width tables

SEE ALSO

Cary Buchman, Daniel M. Berry, *User's Manual for Ditroff/Ffortid, An Adaptation of the UNIX Ditroff for Formatting Bi-Directional Text,*
Johny Srouji, Daniel M. Berry, *An Adaptation of the UNIX Ditroff for Arabic Formatting*
troffort(l), ptrn(l)

Appendix D

ffortid Ver 5.0 Manual Page

NAME

ffortid − in dtroff output, find and reverse all text in designated right-to-left fonts, carry out stretching in Arabic, Hebrew, and Persian text, and arrange that words in slantable fonts are printed on a slanted base line.

SYNOPSIS

ffortid [−**r***font-position-list*] [−**w***paperwidth*] [**−−***font-position-list*] ...
[−**s**[**n**|[[**l**|**c**|**e**|**b**][**f**|**2**|**m**[*amount*]|**a**|**ad**|**al**]]] [−**ms**[**c**|**l**|**w**] ...

DESCRIPTION

ffortid's job is to take the *dtroff*(1) output which is formatted strictly left-to-right, to find occurrences of text in a right-to-left font and to rearrange each line so that the text in each font is written in its proper direction. *ffortid* deals exclusively with *dtroff* output, it does not know and does not need to know anything about any of *dtroff*'s preprocessors. Therefore, the results of using *ffortid* with any of *dtroff*'s preprocessors depends only on the *dtroff* output generated as a result of the input to *dtroff* from the preprocessors. Furthermore, the output of *ffortid* goes on to the same device drivers to which the *dtroff* output would go; therefore, *ffortid*'s output is in the same form as that of *dtroff*. *ffortid* reads its input from the standard input and write to the standard output.

In the command line, the −**r***font-position-list* argument is used to specify which font positions are to be considered right-to-left. A *font-position-list* is a list of font positions separated by white space, but with no white space at the beginning. *ffortid*, like *dtroff*, recognizes up to 256 possible font positions (0-255). The actual number of available font positions depends only on the typesetting device and its associated *dtroff* device driver. The default font direction for all possible font positions is left-to-right. Once the font direction is set, either by default or with the −**r***font-position-list* argument, it remains in effect throughout the entire document. Observe then that *ffortid*'s processing is independent of what glyphs actually get printed for the mounted fonts. It processes the designated fonts as right-to-left fonts even if, in fact, the alphabet is that of a left-to-right language. In fact, it is possible that the same font be mounted in two different positions, only one of which is designated as a right-to-left font position. This is how a single font can be printed left-to-right and right-to-left in the same document.

In addition to the specified font directions, the results of *ffortid*'s reformatting also depends on the document's *current formatting direction*, which can be either left-to-right or right-to-left. The default formatting direction is left-to-right and can be changed by the user at any point in the document through the use of the

x X PL

and

x X PR

commands in the *dtroff* output. These commands set the current formatting direction to left-to-right and right-to-left, respectively. These commands are either meaningless or erroneous to *dtroff* device drivers; therefore they are removed by *ffortid* as they are obeyed. These commands can be generated by use of

\X'PL'

and

\X'PR'

escapes in the *dtroff* input. For the convenience of the user, two macros

.PL

and

.PR

are defined in the **−mX2** and **−mXP** macro packages, that cause generation of the proper input to *ffortid*. They are defined by

```
..de PL
\\X'PL'
..
.de PR
\\X'PR'
..
```

If the current formatting direction is left-to-right, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from left to right. In each *dtroff* output line, any sequence of contiguous right-to-left font characters is reversed in place.

If the current formatting direction is right-to-left, all formatting, filling, adjusting, indenting, etc. is to appear as occurring from right to left. Each *dtroff* output line is reversed, including both the left and right margins. Then, any sequence of contiguous left-to-right font characters is reversed in place.

The −**w***paperwidth* argument is used to specify the width of the paper, in inches, on which the document will be printed. As explained later, *ffortid* uses the specified paper width to determine the width of the right margin. The default paper width is 8.5 inches and like the font directions, it remains in effect throughout the entire document.

It is important to note that *ffortid* uses the specified paper width to determine the margin widths in the reformated output line. For instance, suppose that a document is formatted for printing on paper 8.5 inches wide with a left margin (page offset) of 1.5 inches and a line length of 6 inches. This results in a right margin of 1 inch. Suppose then that the text specifies a current formatting direction of right-to-left. Then, *ffortid*'s reformatting of the output line results in left and right margins of 1 and 1.5 inches, respectively. This margin calculation works well for documents formatted entirely in one direction. However, as a document's formatting direction changes, the resulting margins widths are exchanged. Thus, **.PL**'s right and left margins end up not being the same as **.PR**'s right and left margins. The user can make *ffortid* preserve the left and right margins by specifying, with the −**w***paperwidth* argument, a paper width other than the actual paper width. This artificial paper width should be chosen so that both margins will appear to *ffortid* to be as wide as the desired left margin. For example, for the document mentioned above, a specified paper width of 9.0 inches results in reformatted left and right margins of 1.5 inches each. The resulting excess in the right margin is just white space that effectively falls of the edge of the paper and does not effect the formatting of the document.

There is one exception to these simple rotation rules in that *ffortid*, at present, makes no attempt to reverse any of *dtroff*'s drawing functions, such as those used by *pic*(1) and *ideal*(1) (which are also available directly to the user). It is therefore suggested that these drawing functions, and thus *pic* and *ideal*, be used only when the current formatting direction is left-to-right. Additionally, due to the cleverness of the *dtroff* output generated by most substantial *eqn*(1) equations, it is suggested that *eqn*'s use also be limited to a left-to-right formatting direction for all but the simplest forms of equations. These rules do not in any way restrict the use of right-to-left fonts in the text dealt with by any of the preprocessors, but simply suggest that these particular preprocessors be used only when the formatting direction is left-to-right.

An additional point to keep in mind when preparing input both for *dtroff*'s preprocessors and for *dtroff* itself is that *ffortid* rotates, as a unit, each sequence of characters of the same direction. In order to force *ffortid* to rotate parts of a sequence independently, as for a *tbl*(1) table, one must artificially separate them with a change to a font of the opposite direction.

The **−−***font-position-list* argument is used to indicate which fonts positions, generally a subset of those designated as right-to-left, contain fonts for Arabic, Hebrew, Persian, or related languages, whose fonts support stretching of letters and/or connections. For these fonts, left and right justification of a line can be achieved by stretching instead of inserting extra white space between the words in the line. If requested by use of the **−s** argument described below, stretching is done on a line only if the line contains at least one word in a **−−** designated font. If so, stretching is used in place of the normal distributed extra white space insertion for the entire line. The intention is that stretching soak up all the excess white space inserted by *dtroff* to adjust the line. If there are no opportunities for stretching or there are too few to soak up all the excess white space, what is not soaked up is distributed in between the words according to *dtroff*'s algorithm. There are several kinds of stretching, and which is in effect for all **−−** designated fonts is specified with the **−s** argument, described below. If it is desired not to stretch a particular Arabic, Hebrew, Persian, or other font, while still stretching others, then the particular font should not be listed in the **−−***font-position-list*. Words in such fonts will not be stretched and will be spread with extra white space if the containing line is spread with extra white space.

The **−r** and the **−−** specifications are independent. If a font is in the **−−***font-position-list* but not in the **−r***font-position-list*, then its text will be stretched but not reversed. This independence can be used to advantage when it is necessary to designate a particular Arabic, Hebrew, Persian, or other font as left-to-right for examples or to get around the above mentioned limitations in the use of *eqn*, *ideal*, *pic*, or *tbl*.

The kind of stetching to be done for all fonts designated in the **−−***font-position-list* is indicated by the **−s** argument. There are two relatively independent dimensions that must be set to describe the stretching, what is stretched and the places that are stretched. A stretch argument is of the form

−s*mp*
or
−sn

where *m* specifies the stretching mode, i.e, what is stretched, and *p* specifies the places that are stretched. The *m* and *p* must be given in that order and with no intervening spaces. The **−sn** means that there is *no* stretching and normal spreading of words is used even in **−−** designated fonts. The choices for the mode *m* are:

l (letter ell)
In the words designated by the *p*, stretch the last stretchable letter.

c
In the words designated by the *p*, stretch the last connection to a letter.

e
In the words designated by the *p*, stretch either the last stretchable letter or the last connection to a letter, whichever comes later.

b
In the words designated by the *p*, stretch either the last stretchable letter or the last connection to a letter, whichever comes later, and if it is a letter that is stretched and it is a connect-previous letter then also stretch the connection to the letter.

Not all modes are available for all fonts. For example, fonts for Hebrew, whose letters are not connected do not support connection stretching. While Arabic, Hebrew, and Persian traditionally do have letter stretching, not all fonts for them support letter stretching. *ffortid* attempts to stretch all **−−** designated fonts in the specified modes, but in any text, ends up doing only those stretches that are possible given in the text's current font. To allow *ffortid* to know what stretches are possible, the width tables for stretchable fonts have some additional lines that must come somewhere after the **name** line and before the **charset** line.

```
stretchable: letters connections
stretchable: connections letters
```

```
stretchable: connections
stretchable: letters
```

Each such stretchable font must have one of the first four lines. We now discuss the various ways that different kinds of stretch are achieved in the available fonts and how *ffortid* deals with each.

To our knowledge, all Arabic and Persian fonts, have a baseline filler that can be used to achieve the stretching of connections. It is fairly easy for such a filler to be added to any font definition that does not have it, and moreover to make it the character that is addressed by **\(hy**, which is normally the code for the hyphen character. Since Arabic and Persian do not have a hyphen and hyphenation is turned off when in an Arabic or Persian font, it is safe to use **\(hy** to name the filler. Of course, this requires that the width table for Arabic and Persian fonts have an entry for **hy** designating the filler character in the font, for example:

```
hy     15 0 0267   filler
```

Giving the filler character an explicit *dtroff* two-character name allows *dtroff* to deal with it uniformly despite that it might be in a different position in each font.

On the other hand, stretching of letters requires a dynamic font which, by its very nature of not having a constant bitmap for a given font name, point size, and character name, cannot be type 1 (in PostScript terminology) and cannot be a bitmapped font. Therefore, as mentioned, not all Arabic, Hebrew, and Persian fonts support stretching of letters. Moreover, within a dynamic font, not all characters are stretchable. Historically, only characters with strong horizontal components are stretchable, such as those in the stand-alone and connect-previous forms of the *baa* family. Obviously, one cannot stretch totally vertical characters such as *alif*. Therefore, it is necessary to specify by additional information in the *dtroff* width table for a font which characters are stretchable. In the width table for an Arabic, Hebrew, or Persian font, for each character, one specifies after the name, width, ascender-descender information, and code, two additional fields, the connectivity and the stretchability of the character, in that order. The connectivity is either

N for stand-alone,
A for connect-after,
P for connect-previous,
B for connect-both, or
U for unconnected (because it is punctuation or a diacritical, etc.),

and the stretchability is either

N for not stretchable,
S for stretchable,

Some examples of width table lines are:

```
%      125 2 045   percent

---    55 0 0101   U      N      comma
---    70 0 0105   U      N      hamza

---    129 0 0106  N      S      baa_SA
---    36 2 0102   N      N      alef_SA

---    113 0 0177  A      N      sad_CA
---    66 2 0215   A      S      caf_CA

---    43 2 0225   P      N      alef_CP
```

```
---    120 0 0274   P      S      baa_CP

---    53 0 0230    B      N      baa_CB
---    73 2 0261    B      S      caf_CB
```

Recall that `---` in the name field of a character means that it can be addressed only by **\N'***n***'**, where *n* is the decimal equivalent of the character's code.

For a Hebrew font, for which there is no notion of connectivity of letters, and therefore, the position of the letters is irrelevant for deciding stretching, there is only the possibility of stretching letters. Some examples of width table lines for such fonts are:

```
%      132 3 045    percent

---    95 3 0140    U      N      quoteleft=alef
---    92 3 0141    U      S      a=bet
```

In a dynamic font, there are two additional, alternative ways that stretching of connections can be achieved.

- The filler is a stretchable letter, normally of width zero, to which the total width of the filler is passed as the stretch amount.
- The connecting portions of all connecting letters are themselves stretchable in the same way as the stretchable letters are. In this situation to achieve a total connection stretch of *x*, one would pass *x/*2 to each of the connecting-after portion of the before letter and the connecting-before portion of the after letter.

The use of the first of these solves the problems caused by the fact that amount of a given connection stretch may not be integrally divisible by the width of the filler. A stretchable filler can be stretched to any amount. The use of the second improves the appearance of the connection stretch. While letter stretching is done with nice, smooth curves, connection stretching using the very straight filler is noticeably flatter and there are observable corners where the filler meets the generally curved connecting parts of its adjacent letters. While the fixed-size filler seems to be available on all Arabic and Persian fonts, stretchable fillers and stretchable connecting parts are available only with type 3 PostScript fonts, although it would be possible to provide a stretchable filler as the only locally defined character in a type 3 font that falls to another type 1 font for all the other characters, which are only virtual in the type 3 font.

The *dtroff* width table for any font providing a stretchable filler or stretchable connecting parts must have an additional line to specify the nature of the connection stretch in the font, which must be one of the following.

```
connection stretch: fixed filler
connection stretch: stretchable filler
connection stretch: stretchable connections
```

This line must come somewhere after the **name** line and before the **charset** line. If none is specified, it is assumed to be the first. Therefore, it is not necessary to say anything new for the typical type 1 or bit-mapped font with a fixed-sized filler. Note that if a font allows different kinds of connection stretching, only one can be specified per mounting of the font specified in a single width table. If one wants to use the same font with different ways of stretching connections, one must mount the same font under different names in different width tables, each specifying a different kind of connection streching.

ffortid implements the connection stretching that is requested by the **−s** command-line arguement as well as it can using the kind of connection stretching available for the font being used. Thus, if one is not using fixed-sized fillers, *ffortid* ignores the various options put in to deal with the fact that an integral number of

fillers may not fulfill the needed stretch.

Below, “stretchable unit” refers to that which is a candidate for stretching according to the mode. The choices for *p*, which specifies places of stretching, are:

f
: In any line, stretch the last stretchable unit.

2
: Assuming that the mode is **b** (both), in any line, stretch the last two stretchable units, if they are the connection leading to a stretchable connect-previous letter and that letter, and stretch only the last stretchable unit otherwise. If the mode is not **b**, then this choice of places is illegal.

m*n* or **m**
: In any line, stretch the last stretchable unit by an amount not exceeding *n* emms. If that does not exhaust the available white space, then stretch the next last stretchable unit by an amount not exceeding *n* emms, and so on until all the available white space is exhausted. If *n* is not given, it is assumed to be **2.0**. In general *n* can be any number in floating point format.

a, **ad**, or **al**
: In any line, stretch all stretchable units. In this case, the total amount available for stretching is divided evenly over all stretchable units on the line identified according to the mode. Since the units of stretching are the units of device resolution, the amount available might not divide evenly over the number of places. Therefore, it is useful to be able to specify what to do with the remainder of this division. This specification is given as an extension of the stretching argument. The choices are **d** or **l**, with the former indicating that the excess be distributed as evenly as possible to the spaces between words and the latter indicating that the excess be distributed as evenly as possible in stretchable letters that were stretchable units according to the current mode and place. The latter is the default if no choice is specified. The stretched item for the **l** choice must be a letter rather than a connection because only a stretchable letter is stretchable to any small amount that will be the remainder. Of course, if the method of stretching a connection is dynamic, then a connection could be stretched to any amount, but then there would not be a remainder in the first place.

In general, the stretch is divided as evenly as possible between all stretchable units in a line. Specificly, in stretch mode **b**, if we have a connection leading to a stretchable connect-previous letter and that letter, then any stretch remainder we have from stretching the connection will be added to the stretch of the letter.

Sometimes, it is desirable to be able to manually stretch connections or letters to achieve special effects, e.g., more balanced stretching or stretching in lines that are not otherwise adjusted, e.g., centered lines.

If fixed-sized fillers are used to achieve connection stretching, then one can use the filler character explicitly as many times as necessary to achieve the desired length. Note that the *troff* line drawing function can be used to get a series of adjacent fillers to any desired length, e.g.,

```
\l'2m\(hy'
```

will draw a string of adjacent baseline fillers of length 2 emms.

How to manually stretch connections that are done by a stretchable filler or by stretchable connection parts is described after describing how to manually stretch letters themselves.

To achieve stretching of letters, one should immediately preceed, with no intervening printable text, the letter to be stretched by the escape sequence

\X'stretch'\h'*n***'**

where *n* is a valid length expression in *troff*’s input language. *ffortid* is prepared to deal with the output from *dtroff* generated by this input to generate output that will cause the letter immediately following it to be stretched by the length specified in *n*. For example,

```
\X'stretch'\h'1m'\N'70'
```

will cause the character whose decimal code is 70 to be stretched by 1 mm. The output will fail to have the desired effect if the letter following is not a stretchable letter.

If connection stretching is achieved by having a stretching filler, then one manually stretches the filler character by the desired amount as if it were a letter.

\X'stretch'\h'*n***'\(hy**

Here, though the stretch parameter *n* is the total length of the filler, as the filler is of length zero if it is not stretched.

To stretch the connecting parts of letters, two additional escape sequences are provided that may be placed before, with no intervening printable text, the letter to which they apply,

\X'BCstretch'\h'*nb***' \X'ACstretch'\h'***na***'**

where *nb* and *na* are valid length expressions in *troff*'s input language. These specify the amounts of stretch in the before and after connecting parts of the immediately following letter. The order in which the **\X'stretch'\h'***n***'**, **\X'BCstretch'\h'***nb***'**, and **\X'ACstretch'\h'***na***'** for a letter appear is irrelevant, but in between them and after the last of them, there is *no* printable text, including white space (including new lines), and the letter to which they apply immediately follows the last. Suppose that two consecutive, in logical order, letters have decimal codes 70 and 80. Suppose also that 70 connects after to the connecting before 80. Suppose finally that this connection from 70 to 80 is to be stretched by 1 emm and the letter 80 is to be stretched by 2 emms. Then the input would look as follows:

```
\X'ACstretch'\h'.5m'\N'70'\X'BCstretch'\h'.5m'\X'stretch'\h'2m'\N'80'
```

Note that the connection stretch of 1 emm was split into two stretches of .5 emm for each of the connecting after and the connecting before parts.

For finer control over stretching, it may be desirable to inhibit automatic stretching on manually stretched connections and letters. In particular, when manual stretching is done on a letter or its connection for balancing purposes, one does not want additional automatic stretching to be done on the same to mess up the balance. Accordingly, three command line flags are provided for this purpose:

−msc
: Do not automatically stretch manually stretched connections.

−msl
: Do not automatically stretch manually stretched letters.

−msw
: Do not automatically stretch any word containing any manual stretching.

These flags are understood as eliminating potential stretching places, letters or connections, that were identified on the basis of the stretch mode, **l**, **c**, **e**, or **b**. (In the following description, parenthesized text is a comment stating what is true at this point and not what needs to be done.)

For any letter *l* that is a candidate for stretching by the mode,

> **if** *l* is in a word containing a manually stretched letter or connection and **−msw** is set, **then** *l* is no longer a candidate
>
> **otherwise**
>
> **if** both the letter itself and its connection to the previous letter are candidates **then**

if either kind of manual stretch is in the letter and that kind of manual stretch cannot be stretched additionally, **then** neither part of *l* is any longer a candidate;

otherwise (only the letter itself is a candidate OR only its connection to the previous letter is a candidate)

if the letter itself is a candidate for stretching by the mode,

if there is manual stretching in the letter and manually stretched letters cannot be stretched more, **then** *l* is no longer a candidate;

otherwise (the connection of *l* is a candidate for stretching by the mode),

if there is manual stretching in the connection of *l* to the previous letter and manually stretched connections cannot be stretched more, **then** *l* is no longer a candidate.

ffortid is able to arrange that text in slantable fonts is printed with each word in a line of text in a slanted baseline that crosses the baseline of the line at the center of the word. The figure below shows each words baseline as a solid arrow and the line's baseline as a dotted arrow.

figure baselines.ps

Observe that in this style of printing the beginning of a non-first word is directly over the end of its previous word. Moreover, within a word there will generally be stretching to allow this property to hold; that is, if there were no stretching to achieve left justification, it might be necessary to have a horizontal gap between two consecutive words.

For *ffortid* to implement this slanted-baseline printing for a font, it is necessary that some non-standard information be supplied in the *dtroff* width table for the font. First, there is a line that specifies the slant in degrees.

```
slant 22.0
```

The argument can be a floating point number. This line must come somewhere after the **`name`** line and before the **`charset`** line. The argument should be the slant in degrees and should match the slant implied by the first two values in the **`FontMatrix`** of the font. Specifically the ratio of the second to the first should be the tangent of the slant. *ffortid* uses this slant value to know by how much to displace the beginning of a word vertically so that as it flows downward in the right-to-left direction, the center of the word crosses the line's baseline.

In addition, in order that there appear to be no horizontal white space between words, the **`spacewidth`** of the font must be set to one.

```
spacewidth 1
```

Actually, the spacewidth should be zero, but *dtroff* refuses to set the it to zero, setting it to an emm width if you tell it zero. To the human eye, at the typical resolutions specified in the *DESC* files, in the mid hundreds, a spacewidth of one is close enough.

A few suggestions to the user are in order. While *dtroff* supports font changes in the middle of words, *ffortid* does not support and reports as an error font changes that change the slant in the middle of words, either to another nozero slant or to no slant at all. Besides it being a pain to implement, it is not clear what the behavior should be in such a situation. Recall also that there is typically no horizontal separation between slanted words; all the separation comes from the end of one word being separated vertically from the beginning of the next. If words are too short, there may not be enough vertical clearance between consecutive words. To insure adequate vertical clearance, it may be necessary to combine several words into what *dtroff* and *ffortid* consider one word. For this purpose, each such slantable font should have a special

character called **\(ps** (for "permanent space", whose width is set to what would normally be the spacewidth and which can be used as an unpaddable blank between two words that are to be treated as a single, unbreakable word by *dtroff* and *ffortid*.

```
ps     72 0 040    permanent space
```

Note that the normal *dtroff* unpaddable space, "**\ **", cannot be used, because its width is defined to be that of the regular space, i.e., the spacewidth, and would end up being one in this case. If one wants the guaranteed white space, but wants to allow a word break, one can make the **\(ps** the last character or the first character in a regular, white-space-delimited word.

FILES

/usr/lib/tmac/tmac.∗	standard macro files
/usr/lib/font/dev∗/∗	device description and font width tables

SEE ALSO

Cary Buchman, Daniel M. Berry, *User's Manual for Ditroff/Ffortid, An Adaptation of the UNIX Ditroff for Formatting Bi-Directional Text,*
Johny Srouji, Daniel M. Berry, *An Adaptation of the UNIX Ditroff for Arabic Formatting*
troffort(l), ptrn(l)

Bibliography

[1] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[2] ANSI/IEEE. *IEEE standard glossary of software engineering terminology*. IEEE, 1983. ANSI/IEEE standard 729.

[3] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 3rd edition, 1992.

[4] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[5] Judith D. Ahrens, Noah S. Prywes, and Evan Lock. Software process reengineering: Toward a new generation of case technology. *Journal of Systems and Software*, 30(1 and 2):71–84, July–Aug 1995.

[6] Software Productivity Consortium. Reuse-driven software process guidebook. Technical Report SPC-92019-CMC, Version 02.00.03, Software Productivity Consortium, Herndon, Virginia, 1993.

[7] Software Productivity Consortium. Software reuse: The competitive edge. Technical Report SPC-91047-N, Software Productivity Consortium, Herndon, Virginia, 1991.

[8] Judith D. Ahrens and Noah Prywes. Reengineering the software life cycle and enabling technology. Technical report, Computer Command and Control Company, July 20 1994.

[9] CSTB Report. Scaling up: A research agenda for software engineering. *Communications of the ACM*, 33(3):281–293, March 1990.

[10] Alfonso Fuggeta. A classification of case technology. *Computer*, 26(12):25–38, December 1993.

[11] Judith D. Ahrens and Noah S. Prywes. Transition to a legacy and reuse-based software life cycle. *Computer*, 28(10):27–36, October 1995.

[12] Rebecca Joos. Software reuse at motorola. *IEEE Software*, 11(5):42–47, September 1994.

[13] Wayne C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Software*, 11(5):23–30, September 1994.

[14] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[15] H. Sackman, W.J. Erickson, and E.E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, January 1968.

[16] Cary Buchman and Daniel M. Berry. *User's Manual for ditroff/ffortid, An adaption of the UNIX Ditroff for formatting bi-directional text.* Berry Computer Scientists, Los Angeles, CA, 1987.

[17] J. Srouji and D. M. Berry. Arabic formatting with ditroff/ffortid. *Electronic Publishing*, 5(4):163–208, December 1992.

[18] B. W. Kernighan. A typesetter-independant TROFF. Computing Science 97, Bell Laboratories, Murray Hill, NJ, March 1982.

[19] J. F. Ossana. NROFF/TROFF user's manual. Technical report, Bell Laboratories, Murray Hill, NJ, October 11 1976.

[20] G.A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, March 1956.

[21] B.W. Kernighan. Pic — a graphics language for typesetting, revised user manual. Computing Science 116, Bell Laboratories, Murray Hill, NJ, December 1984.

[22] Harry I. Hornreich. ffortid version 3.0 decomposition manual. Available from ftp://csgo.cs.technion.ac.il/pub/misc/dberry/hornreich.work, April 1996. In Adobe Acrobat pdf format.

[23] D.M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, February 1995.

מקרה מחקר בהנדסה מחדש של תוכנה

הרי הורנרייך

מקרה מחקר בהנדסה מחדש של תוכנה

חיבור על המחקר

לשם מילוי חלקי של הדרישות לקבלת תואר מגיסטר למדעים
במדעי המחשב

הרי הורנרייך

הוגש לסנט הטכניון - מכון טכנולוגי לישראל
אדר התשנ"ח חיפה פברואר 1998

חיבור על המחקר נעשה בהנחיית פרופ. דניאל מ. ברי בפקולטה למדעי המחשב

ברצוני להודות לקרן הטכניון על המילגה שניתנה לי במהלך מחקר זה

מחקר זה מוקדש לזכרו של אבי פרופ' ריצ'רד הורנרייך ז"ל

ברצוני להודות לאשתי על תמיכתה הרבה במהלך מחקר זה

תוכן ענינים

תקציר 1

1 מבוא 3
1.1 הגדרות 3
1.2 בעיות במודלים קיימים למחזור החיים 5
1.3 מודל מוצע למחזור החיים 6
1.4 שיטות מעבר 8
1.5 שיטת המעבר המוצעת 9
1.6 מטרות המחקר 12

2 הניסוי 13
2.1 מקרה מחקר 13
2.2 שלבי מקרה המחקר 14
2.3 תוקף מקרה המחקר 15

3 התוכנית ffortid 17
3.1 רקע 17
3.2 קבצי המקור של ffortid 19
3.3 למה בחרנו ב ffortid 21

4 הנדסת תחום מחדש של ffortid 22
4.1 יחידות תוכנה 22
4.1.1 מינשק יחידת תוכנה והשפעות צד 23
4.1.2 יחידות מישנה של תוכנה 24
4.1.3 דיאגרמות זרימת שירותים 26
4.2 הנדסה לאחור של יחידת תוכנה 31
4.3 הנדסה לאחור של ffortid גירסה 3.0 32
4.4 הארכיטקטורה של ffortid גירסה 3.0 37
4.5 מסקנות המחבר מהפירוק 40
4.6 התחום הראשוני 41

5 ffortid גירסה 4.0 43
5.1 ביצוע שינויים ביחידת תוכנה 43
5.2 הדרישות החדשות 45
5.3 המימוש 46
5.4 השוואת המימושים 49

6 ffortid גירסה 5.0 53
6.1 הדרישות החדשות 53
6.2 המימוש 55

7 **תוצאות הניסוי** 57
7.1 מדידת שימוש חוזר 57
7.2 תוצאות 58
7.3 מסקנות 60
7.4 תודות 61

א **סימנים בדיאגרמות זרימת שירותים** 62

ב **דף תאור** ffortid **גירסה** 3.0 66

ג **דף תאור** ffortid **גירסה** 4.0 70

ד **דף תאור** ffortid **גירסה** 5.0 77

ביבליוגרפיה 87

רשימת איורים

1.1 הקשרים בין ההגדרות 4
1.2 מודל מחזור החיים לשימוש חוזר בתוכנות ישנות 7
1.3 מבט על של שיטת המעבר המוצעת 9
1.4 התהליכים בשיטה המאוחדת 11

3.1 מתיחת אותיות מתחברות עם ממלא מקום 17
3.2 דוגמא של ditroff שלא עברה דרך ffortid 18
3.3 אותה דוגמת ditroff שהועברה דרך ffortid 18
3.4 אותיות מתחברות אחרונות בשורה נימתחות 18
3.5 אותיות מתחברות אחרונות בשורה נימתחות עד גודל מקסימלי 19
3.6 המתיחה מפוזרת בין כל האותיות המתחברות האחרונות במלים 19
3.7 דוגמא לפלט ffortid המכילה ערבית, עברית ואנגלית 20

4.1 דיאגרמת טווח לדוגמא 25
4.2 סימנים עיקריים בדיאגרמת זרימת שרותים 27
4.3 דיאגרמת זרימת שרותים של ffortid 28
4.4 דיאגרמת זרימת שרותים של dump.c עם יחידות המשנה 29
4.5 דיאגרמת זרימת שרותים מורכבת 30
4.6 חלק ראשון של דף יחידת תוכנה 1 של ffortid גירסה 3.0 33
4.7 חלק שלישי של דף יחידת תוכנה 1 של ffortid גירסה 3.0 34
4.8 חלק שני של דף יחידת תוכנה 1 של ffortid גירסה 3.0 35
4.9 דף יחידת תוכנה 16 בפירוק של ffortid גירסה 3.0 36
4.10 מבט על של פירוק ffortid גירסה 3.0 ליחידות תוכנה 39

5.1 אותיות מתחברות, ממלאי מקום ואותיות דינמיות 45
5.2 מבט על של ספרית ffortid גירסה 4.0 50

6.1 חיבורי אותיות נימתחים וממלאי מקום 54
6.2 סידור מלים בכתיב משופע בשורה 54
6.3 דוגמא לפלט בכתיב משופע 55

7.1 יחסים בין יישום מקורי ליישום לאחר שינויים 57

רשימת טבלאות

3.1 ההיסטוריה של ffortid 19
3.2 קבצי המקור של ffortid 21

4.1 יחידות התוכנה של ffortid גירסה 3.0 38

5.1 סוגי השינויים ביחידות תוכנה 44
5.2 יחידות התוכנה של ffortid גירסה 4.0 51

7.1 תוצאות הניסוי 58
7.2 ניתוח תוצאות הניסוי 59

תקציר

הבעיה של תחזוקה ושיפור מערכות קיימות מוכרת כבעיה מרכזית בתחום של הנדסת תוכנה. בעיה זו מודגשת במיוחד ביישומי תוכנה ישנים שהפכו עם השנים לחלק חשוב באופן הפעולה של הארגונים בהם הם קיימים. המהנדסים של היישום המקורי כבר אינם בד"כ, ומרוב שינויים שנעשו ביישום במהלך השנים הוא נהיה כה מורכב שאיש אינו מבין אותו לעומק. מצד שני, אצור ביישום ידע רב שנצבר במהלך השנים, ידע שעשוי להיות חיוני לארגון. לכן לא ניתן לזרוק את היישום ולהתחיל מאפס במיוחד שעלות פיתרון מאין זה יקרה במשאבים ובזמן. חוקרים הציעו לפתור בעיה זו ע"י שינויים ארגוניים וע"י שיטות לשימוש חוזר בתוכנה ויצירה אוטומטית של תוכנה.

שיטה אחת למעבר לחזון עתידני זה של הנדסת תוכנה היא גישת הסינתזה שהוצעה ע"י ה Software Productivity Consortium בארה"ב. גישה זו מציעה סדרת צעדים סדורה לניהול, ניתוח והגדרת דרישות של תחום המכיל את הארכיקטקטורה של רכיבי תוכנה לשימוש חוזר במשפחת יישומים, ואת חוקי הבחירה הדרושים לשם בחירת הרכיבים השונים. תהליך זה של יצירת תחום מלמעלה למטה נקרא הנדסת תחום. יישומים חדשים ניבנים ע"י בחירת רכיבים מהתחום, עפ"י חוקי הבחירה, בתהליך שנקרא הנדסת יישום.

גישה זו הוכיחה את עצמה עד כה כגישה יקרה ומסוכנת. הצורך הרב להסתמך על מומחים עם ידע נרחב הן בתחום היישום והן בהנדסת תוכנה גובה מחיר יקר. גישה זו דורשת ממומחים אלה לבנות מאפס, ספריה של רכיבי תוכנה הניתנים לשימוש חוזר ועונים על הצרכים של כל יישום אפשרי בתחום, וזו משימה קשה ביותר. בנוסף, לא ניתן לבנות ולו יישום אחד לפני שכל התהליך כולו הסתיים.

אהרנס ופריווס הציעו להוסיף לתהליך הנדסת התחום מלמעלה למטה, תהליך מלמטה למעלה הנקרא הנדסת תחום מחדש. בתהליך זה, כלים אוטומטיים מוציאים מתוכנה ישנה באיכות טובה, ידע בתחום ובתוכנה אשר ניתן להשתמש בה על מנת להגדיר את דרישות היישומים בתחום ואת רכיבי התוכנה מהם ניתן לייצור אותם. בצורה זו, תוכנה ישנה הופכת להיות מאיץ עיקרי בתהליך ומקטינה את התלות במומחים בתחום. שילוב של הנדסה מלמעלה למטה והנדסה מחדש מלמטה למעלה, מפחיתה את הזמן והסיכון הכרוך ביצירת יישומים חדשים.

מחקר זה מנסה להעריך את השיטה המוצעת להנדסה מחדש מלמטה למעלה ע"י עריכת מקרה מחקר השוואתי על יישום תוכנה אמיתי וישן בגודל קטן. הרעיון היה לבצע ניסוי מבוקר של מה שקורה במציאות כל הזמן. יישום תוכנה חדש משוחרר ומשוב ממשתמשים, מערכות הפעלה משתנות, ואפילו תחרות, יוצרים את הצורך לייצור יישומים מתקדמים יותר המבוססים על היישום המקורי. תהליך זה יכול להיות קשה ומורכב ביותר בתוכנות ישנות בגלל הקושי הרב לבצע בהם שינויים.

היו שני תפקידים בניסוי: המחבר והבקר פרופ' ברי. שניהם התחילו מאותו יישום ישן,

אבל השתמשו בשיטות שונות לייצור את שני הדורות הבאים של היישומים שלהם. כל דור של היישום שנעשה ע"י המחבר או הבקר, היה מבוסס על הדור הקודם שלו.

המחבר השתמש בהנדסה מחדש על מנת ליצור תחום ראשוני. לאחר מכן, הוא פיתח את התחום בצורה אבולוציונית על מנת לענות על הדרישות החדשות של היישומים המתקדמים יותר בתחום, ותוך כדי כך הוא יצר את היישומים החדשים הנדרשים. הבקר השתמש בשיטה המייצגת את שיטות התחזוקה הנהוגות כיום, כלומר שיטות אשר אינן עושות כל שימוש בהנדסה לאחור או בהנדסה מחדש. הבקר יישם את אותם שני דורות של יישומים שיצר המחבר עפ"י אותם דרישות.

ההשערות של חקר המקרה היו:

השערה 1: השיטה החדשה יוצרת יישומים בזמן קצר יותר מאשר השיטות הקיימות.

השערה 2: השיטה החדשה יוצרת יותר קוד שניתן לשימוש חוזר מאשר השיטות הקיימות.

השערה 3: השיטה החדשה דורשת פחות שינויים בתוכנה על מנת לייצור יישום מאשר השיטות הקיימות.

השערה 4: השיטה החדשה יוצרת יישומים קטנים יותר מאשר השיטות הקיימות.

ביצוע מחקר מקרה דורש תכנון מוקדם על מנת לוודא שהתוצאות יהיו ברות תוקף. מספר צעדים נלקחו על מנת לוודא זאת בניסוי זה:

- נבחר יישום ישן טיפוסי למחקר.
- למחבר לא היה כל ידע מוקדם על היישום הנבחר.
- למחבר לא היה כל ידע על קבוצת הדרישות של הדור השני והשלישי של היישום לפני שהיה עליו לדעת אותם.
- המחבר והבקר שוחחו ביניהם אך ורק על הדרישות עצמם ולא על צורת היישום שלהם.
- יישומים מאותו דור נבחנו כנגד אותה קבוצת בדיקות לפני שהניסוי עבר לשלב הבא, על מנת לוודא שהם פונקציונלית זהים.

כמו בכל ניסוי בהנדסת תוכנה אשר כולל מספר מתכנתים, הבדלים ניכרים ביכולת המתכנתים יכול לבטל את תוקף הניסוי כולו. במקרה זה, הן המחבר והן הבקר הם מתכנתים בעלי ניסיון רב בשפת היישום C. למרות שלא ניתן לקבוע מי המתכנת הטוב יותר, ניתן לומר שלבקר מספר יתרונות ברורים על המחבר. יש לו 29 שנות ניסיון רבות יותר מאשר למחבר. יש לו הבנה עמוקה מאוד של מערכת עיבוד התמלילים שהיישום שנבחר לניסוי הוא חלק ממנה, ואילו למחבר לא היתה כל הבנה שכזו. הבקר היה מעורב מאז שנת 1983 בכתיבה, תיקון והזמנת יישומים מדורות קודמים של היישום שנבחר לצורך הניסוי. כמו כן, במהלך הניסוי, לבקר היה ידע מוקדם על הדרישות לדור היישום הבא כיוון שהוא היה היוזם שלהן. לפיכך, אם תוצאות הניסוי יראו יתרון ברור לשיטת המחבר על פני שיטת הבקר, אזי יש עתיד לשיטה המוצעת והיא ראויה למחקר נוסף. אם לעומת זאת התוצאות אינן חד משמעיות או עם יתרון ברור לשיטת הבקר, אזי לא ניתן להסיק דבר.

היישום שנבחר לצורך הניסוי נקרא ffortid. יישום זה מהווה חלק ממערכת עיבוד התמלילים המקורית של UNIX שנקראת ditroff. כאשר יישום זה משולב במערכת עיבוד התמלילים, הוא מאפשר כתיבת מסמכים מדעיים מרובי שפות הכוללים עברית, ערבית או פרסית כמו שפות אחרות מימין לשמאל כולל תמונות, גרפים, נוסחאות, טבלאות וציטוטים ביבליוגרפיים. יישום זה נכתב לאורך 9 שנים על ידי 3 מתכנתים שונים אשר לא ניתן להשיגם עוד. היישום פועל ונמצא בשימוש, אך אין כל תיעוד על המבנה הפנימי שלו. התוכנה היא בגודל 2510 שורות ולכן אינה קטנה מדי להיחשב תוכנית צעצוע ולא גדולה מידי לזמן העומד לרשות המחקר.

כיוון שלמחבר לא היה כל ידע על היישום הנבחר ועל התחום בו הוא פועל, היה זה אך טבעי להשתמש בהנדסת תוכנה מחדש מלמטה למעלה על מנת להוציא את הידע והקוד שכבר היה קיים ביישום על התחום. המחבר השתמש בשיטה של הנדסה לאחור שפותחה במהלך הניסוי, על מנת לגלות את הארכיטקטורה והתיכון של היישום. בשיטה זו, מפרקים את היישום ליחידות מופשטות הנקראות יחידות תוכנה. יחידות תוכנה אלו יהוו את הבסיס לסיפריית הרכיבים של התחום. המחבר ביצע את כל פעולות הפירוק באופן ידני על מנת להסיק מסקנות לגבי היכולות הדרושות לכלים ממוחשבים שיבצעו את אותן פעולות.

יחידת תוכנה היא רכיב מוגדר היטב של מערכת תוכנה אשר מספקת לפחות שירות או משאב חישובי אחד. במחקר מוגדרים הטווח, היכולות, הממשק, הדרישות והסוג של יחידת תוכנה. כמו כן מוגדרים גם הסביבה והשפעות הצד שעלולים להיווצר כתוצאה מהפעלת יחידת תוכנה. במחקר מוגדר היטב מה הן יחידות משנה של יחידת תוכנה ומה הקשר בין כל התכונות הנ"ל של יחידת תוכנה לבין אלו של יחידות המשנה שלה. המחקר עושה שימוש בגרפים הנקראים דיאגרמות זרימת שירותים שבעזרתם ניתן לתאר בצורה גרפית את הסוגים השונים של יחידות תוכנה והשירותים השונים שהם מספקים. במחקר מתוארת שיטה מסודרת לפירוק תוכנה קיימת ליחידות תוכנה וכיצד ניתן לתעד פירוק זה.

כאמור, המחבר והבקר יצרו שני דורות של היישום ffortid. כל אחד מהם רשם את מספר השורות שנמחקו, שונו והוספו בכל דור של היישום. כמו כן נרשמו המספר הסופי של השורות בכל דור ומספר השעות שנדרשו למימושו כולל בדיקות.

על מנת למדוד כמותית את מידת השימוש החוזר שהושג בכל דור הוגדרו שלושה יחסים:

- יחס השימוש החוזר מודד את מספר השורות שנעשה בהם שימוש חוזר בתוצר יחסית למספר השורות המקורי של היישום. יחס זה מודד בכמה מהיישום המקורי נעשה שימוש חוזר כפי שהוא.
- יחס השינוי דומה ליחס השימוש החוזר אלא שהוא מודד את מספר השורות ששונו בתוצר יחסית למספר השורות המקורי של היישום.
- יחס ההוספה מודד את מספר השורות החדשות שהוספו לתוצר יחסית למספר השורות הסופי בתוצר. יחס זה חשוב כיוון שהוא אומר לנו כמה מהתוצר הסופי נוצר מקוד חדש לחלוטין וכמה ממנו נוצר מקוד שנבע משימוש חוזר ומשינויים.

לפיכך, על מנת לומר שרמה גבוהה של שימוש חוזר הושגה בפרויקט כלשהוא צריך להיות יחס שימוש חוזר גבוה ויחס הוספה נמוך. יחס שימוש חוזר גבוה מעיד על כך שלא היה צורך לזרוק קוד רב מהיישום המקורי. יחס הוספה נמוך מעיד על כך שרוב היישום החדש נוצר מקוד קיים ולא מקוד חדש.

תוצאות הניסוי מראות שלא היה הבדל מהותי ביחס השימוש החוזר וביחס השינוי בין

שתי השיטות בשלבי הניסוי השונים. לעומת זאת, היה הבדל מהותי ביחס ההוספה בדור הראשון של היישום. לדעת המחבר הבדל זה מצביע על היתרון בשימוש בשיטתו כיוון שזו מדריכה את המתכנת להשתמש ביחידות תוכנה קיימות ולא ליצור הפשטות חדשות. למרות שלא היה הבדל מהותי ביחסים השונים בדור היישום השני, במבט על כלל התוצאות ניתן לטעון שהן מצביעות על כך שהדור הראשון של היישום של המחבר היה טוב יותר לשימוש חוזר מאשר של הבקר כיוון ש

- לקח משמעותית פחות זמן לבצע שינויים על גירסה זו,
- גודל היישום היה משמעותית קטן יותר,
- הוא היה מתועד טוב יותר,
- והקוד שבו היה באיכות גבוהה יותר.

אם זאת, יש להדגיש שדרושים מקרי מחקר וניסויים פורמליים נוספים על מנת לחזק תוצאות אלה ולבטל השפעות חיצוניות שייתכן והשפיעו על תוצאות הניסוי.

במחקר זה, הראה המחבר, שניתן להשתמש בשיטה המוצעת להנדסת תוכנה לאחור על מנת ליצור תחום ראשוני ושניתן לבצע חלקים ניכרים משיטה זו בעזרת כלים אוטומטיים, כלים שהם תנאי הכרחי לשימוש בשיטה זו על יישומים מסדרי גודל גדולים יותר מאשר היישום במחקר. לדעת המחבר, רק שימוש בשיטות וכלים אלה נותנים תקווה לפתור את בעיית התחזוקה והשימוש החוזר בתוכנות ישנות ומורכבות. ניסויים נוספים, שבשאיפה יבוצעו על מערכות גדולות אמיתיות ובעזרת כלים ממוחשבים מהשורה הראשונה, יעזרו לשפר את השיטות המוצעות ולמצות את מלוא הפוטנציאל של טכנולוגיה זו בהנדסת תוכנה.