A Case Study of Software Reengineering

Harry I. Hornreich
and
Daniel M. Berry
Computer Science Department
Technion
Haifa 32000
Israel
harry@ubique.com, dberry@csg.uwaterloo.ca [*]

[*]Current Address of Corresponding Author: Prof. Daniel M. Berry, Computer Systems Group, University of Waterloo, 200 University Ave. West, Waterloo, Ontario N2L 3G1, Canada

Abstract

This paper describes a case study attempting to validate the effectiveness of the Ahrens-Prywes (AP) top-down domain engineering method as it applies to maintaining and enhancing legacy systems. Hornreich was totally unfamiliar with a particular legacy system and Berry was intimately familiar with the system. The case study has Hornreich apply the AP method to do two consecutive enhancements of the system and has Berry apply the traditional seat-of-the-pants (SOTP) method to do the same two enhancements of the system. By several measures of software and software development quality, Hornreich produced better code faster than did Berry. Therefore, the case study indicates that the AP method has promise as a software reengineering method.

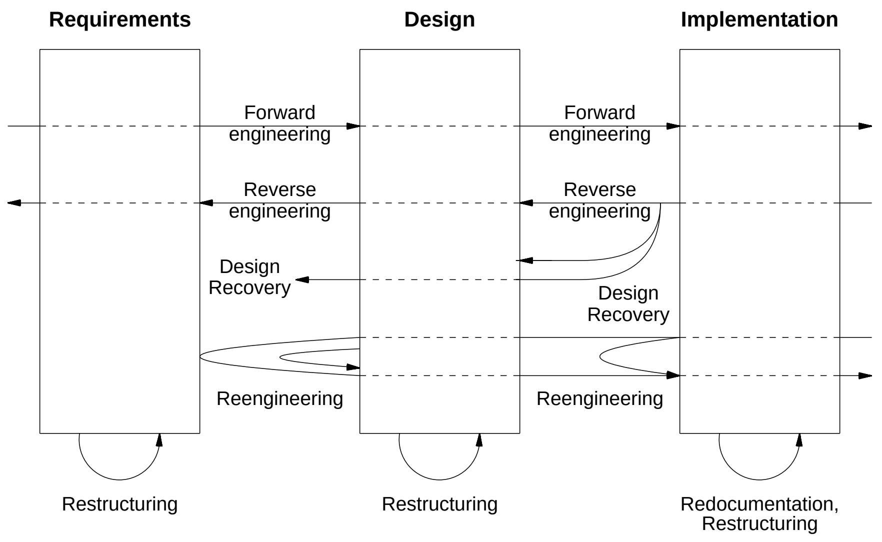


Figure 1: Relationship between terms

1 Introduction

The problem of maintaining and enhancing existing systems has been recognized as a major problem in the field of software engineering [10].

1.1 Definitions

This paper assumes that the reader is familiar with the vocabulary of software maintenance, forward engineering, reverse engineering, redocumentation, design recovery, restructuring and reengineering. Their definitions can be found in the literature [10, 5, 22]. Figure 1 shows the relationship between these terms.

Reverse engineering, restructuring, and reengineering are usually all performed on existing systems and are, therefore, forms of maintenance. However, each of these processes can be used in new system development or in 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 needed changes. Restructuring of an existing system is effectively preventive maintenance. A reengineering effort can either be adaptive, perfective, or preventive maintenance, or some combination of them.

1.2 Current Life-Cycle Models

Most software today is developed using one, or a combination of, well known life-cycle models such as the waterfall [22], prototyping [22], the spiral [8], and what have been called *fourth-generation techniques* [22]. 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 [3]. Additionally, they do not adequately represent state-of-the-art concepts for improving software engineering practices such as domain and application engineering [12] and software reuse [11].

1.3 Proposed Life-Cycle Model

Ahrens and Prywes [2, 1] have proposed a new life-cycle model called the *legacy and reuse software life cycle* (LRSLC), which "... 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 mapped out in Figure 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, and reverse transformations are represented by solid lines.

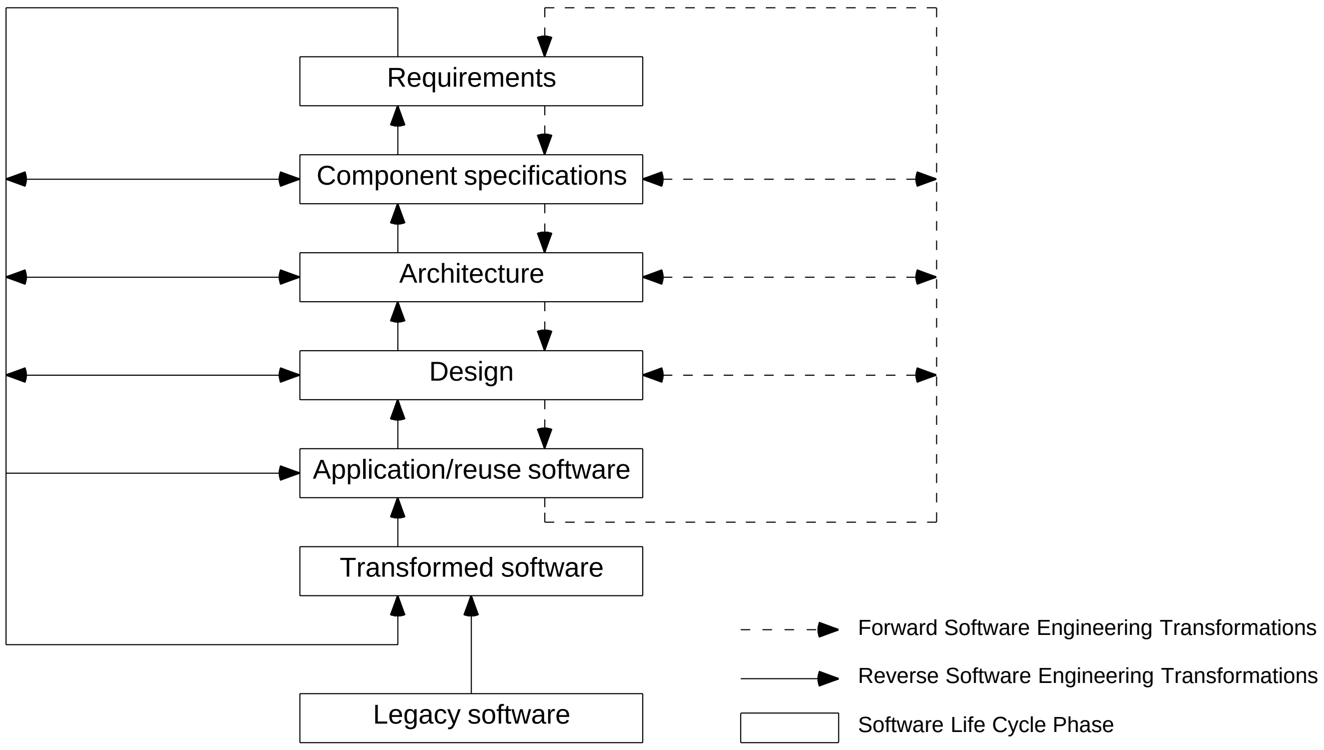
The LRSLC model does not necessarily replace current models; it can be combined with them. For example, the model fits well in the risk-oriented, iterative spiral model. It can also be combined with other models such as the prototyping model to validate customer requirements. The model does however have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

- 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 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, evolutionary.

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, 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 specifications from the documentation extracted by the reverse engineering process. Then, forward traversal is used to create the new domain components, updating in the process, the specifications, architecture, design, and reuse information. Finally, a last forward traversal is used to create the customer's new application by integrating previous, new, and reengineered software components.

1.4 Transition Method

The LRSLC model 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


Requirements
Component specifications
Architecture
Design
Application/reuse software
Transformed software
Legacy software
Forward Software Engineering Transformations
Reverse Software Engineering Transformations
Software Life Cycle Phase

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 2: The legacy and reuse software life cycle

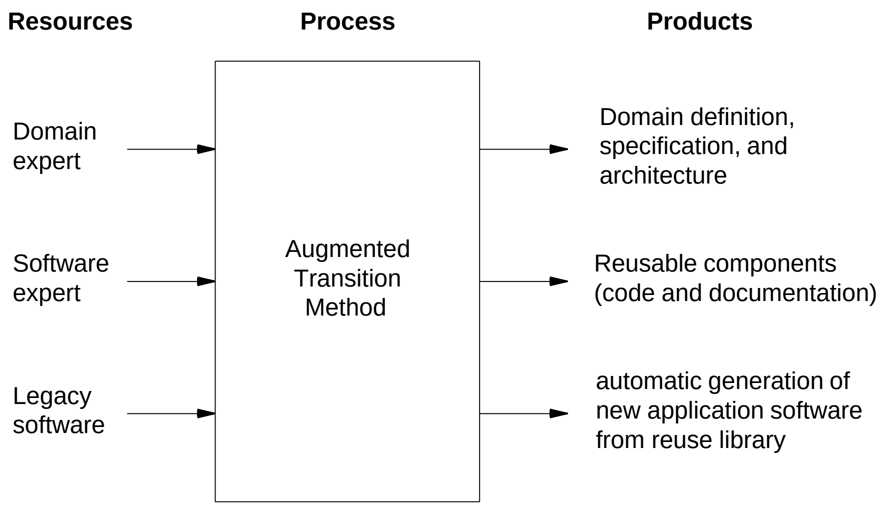

Resources
Process
Products
Domain expert
Software expert
Legacy software
Augmented Transition Method
Domain definition, specification, and architecture
Reusable components (code and documentation)
automatic generation of new application software from reuse library

Figure 3: Overview of the proposed transition method

blocks of adaptable software components from which application software can be constructed. The ultimate goal of these methods is to be able to automatically generate applications in a specific domain from a library of reusable components according to customer requirements.

However, transition from present software practice to automatic application generation has proved to be very difficult. The proposed transition methods, such as the synthesis approach of the Software Productivity Consortium, 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 it depends on having 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 generated. This approach has so far proved to be very costly and risky [15, 19]. The initial investment is large and the returns are low and slow.

Ahrens and Prywes [2] propose a new method for the transition from current software practices to the LRSLC, an augmentation of the top-down synthesis method by the use of bottom-up legacy code component and knowledge extraction. Figure 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 who are the bottleneck of the process. With the use of appropriate reverse engineering CASE technology, it 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 that have satisfied their users needs over a long period of time. They are too difficult to maintain without degredation [6] and too costly to replace by completely new applications. They are valuable resources of their organizations and therefore hold invaluable knowledge and code that can be extracted.

1.5 Augmented Transition Method

As described in Section 1.2, synthesis prescribes an ordered sequence of steps for the management, analysis, and specification of a *domain* that contains the architecture of a family of reusable software components, and it provides the decision rules needed for the selection of components. The top-down process of creating the domain is called *domain engineering*, which consists of six main steps: domain definition, specification, design, verification, implementation, and validation. New applications are constructed by selecting components from the domain, as indicated by the decision rules, in a process called *application engineering*, involving: defining the application's software requirements, selecting reusable components according to rules in a decision model, generating and testing the application software, and finally writing the application's 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, in eight major steps: (1) analyzing and translating the legacy application, (2) converting the legacy application to new hardware and operating system, (3) augmenting and adapting reusable components, (4) validating the domain, (5) updating the design of the domain, (6) updating the domain's specifications, (7) updating the domain's definition, and (8) verifying the domain.

Figure 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 4 shows that both top-down and bottom-up processes can be interleaved and applied iteratively, adding to 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 to fill in the detailed architectural levels.

Two factors indicate that the top-down approach by itself will require significantly more time than the combined approach to complete the first domain's increment of reusable software components for an application. First, the top-down approach requires more input from human domain experts. Second, the synthesis method requires the complete domain be specified before applications are produced. However, the top-down approach by itself has an advantage when 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 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 scarce resource of domain experts by relying more on general software experts extracting domain knowledge embedded in good legacy software. In summary, the combined approach presents an alternative for a faster and more economical application of 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

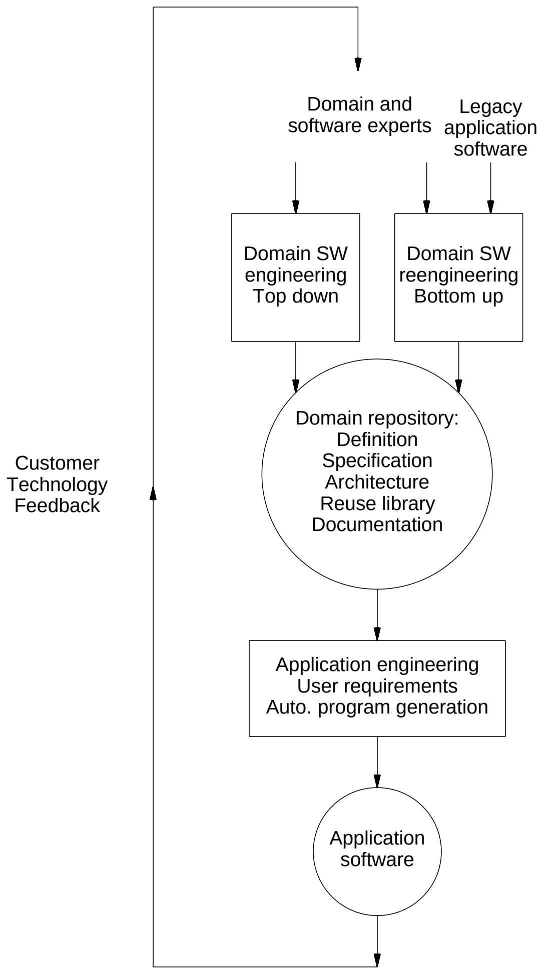


Figure 4: Augmented method processes

in the effort required to understand the legacy software in the processes.

The rest of this paper describes a case study amounting to a preliminary validation of the augmented LRSLC (ALRSLC)[1] model. It also lays out requirements for tools that help carry out the method.

This paper is based on the M.Sc. thesis of the first author. The thesis contains details that are omitted from this paper due to space limitations. Occasionally the reader is referred to the thesis [13], which can be obtained at [2]

ftp://ftp.cs.technion.ac.il/pub/misc/dberry/hornreich.work/Thesis.pdf .

The same directory contains other materials from the experiment, including source programs and the various documents described in this paper.

[1]This ALRSLC model is what is referred to as the Ahrens-Prywes method in the abstract; we thought that more readers would recognize the names of the authors than the name of the method.

[2]This thesis and the other materials mentioned here can be read using Adobe's Acrobat reader, which is available free of charge from

http://www.adobe.com/.

2 The Experiment

2.1 A Case Study

To demonstrate the effectiveness of the augmented LRSLC (ALRSLC) model, we decided to conduct a *case study* [18]. In general, a case study can show the effects of a technology or method in a typical situation, but the result cannot be generalized to every possible situation. Although case studies are not as scientifically rigorous as formal experiments [18], 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 quality applications can be produced 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 experiments. 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 intended to examine which is better, the new, ALRSLC method for legacy code reuse and enhancement or the common and often used seat-of-the-pants (SOTP) method for code maintenance. SOTP maintenance does not mean maintenance with no method in it. The maintainer may indeed have a systematic method for performing modifications to the software. However, such a method does not involve any systematic 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 ALRSLC method requires less time to produce an application than current maintenance methods.

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

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

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

2.2 Case Study Mechanics

Hornreich, the first author, served as the subject of the experiment, who applies the ALRSLC method. Berry, the second author, served as the control, who applied his own systematic SOTP maintenance method. We believe that this SOTP method, described by example in Section 5.5, is representative of the maintenance methods used by most programmers that do not apply any systematic form of reverse or reengineering. Both the subject and the control 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 subject domain reengineered P and created an initial domain architecture and a set of reusable components.

3. A set of requirements R' was devised for a new version of P.

4. The control and the subject each created individual implementations of P' according to the requirements R'. The subject used the ALRSLC method for the evolutionary development of a domain according to new external requirements to create his version of P' using the initial domain as his basis. The control used his own systematic SOTP method of maintenance to create his 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 subject 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 to be 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 designed to follow the steps of a typical software project in which one has a legacy program of which he or she has very little knowledge, but must create new versions of the program to satisfy new user requirements.

The requirements for both new versions were not known to the subject before he had reached the stage in which 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 version of the application. Doing two enhancements with each method allowed us to see if the ALRSLC method produced, in the first enhancement, code that was easier to enhance than before and than with the SOTP method.

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. However, it did 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. Even if we could collect these hours for version P', it would be wrong to compare them for both methods because the subject was learning and developing the method during this step. 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 subject had no previous knowledge.
3. The subject 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 subject 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 in 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 [23].

In the present case, both the subject 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 had some clear initial advantages over the subject:

- The control had more than 29 years of programming experience, while the subject had only 6 years.
- The control has a much deeper understanding of the text processing system of which the selected program is a part, than the subject, 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. Needless to say, the subject 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 and author. The subject learned these requirements only just before the start of programming, when the control gave the subject the requirements document, i.e., the manual page the control had written.

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 the advantage appeared because the subject is a better programmer than the control *or* that the subject 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.

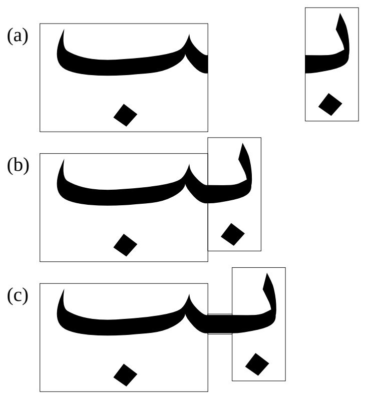


Figure 5: Stretching connecting letters with a filler

3 The ffortid Program

3.1 Background

ffortid [9, 24] is a UNIX ditroff [16, 21] (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 Arabic, Hebrew, 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 that is formatted strictly left-to-right, finds occurrences of text in any right-to-left font, such as Arabic or Hebrew, 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 5 (a), (b), and (c) show how a filler is inserted between pairs of connecting letters. Each character, including the filler, is enclosed by its bounding box.

Figure 6 shows the ditroff output of an example combining Arabic, Hebrew, and English text. Figure 7 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 8, 9, and 10 are examples of the different stretch styles of ffortid. In Figure 8, connections to last connecting letters in lines are stretched. In Figure 9, connections to last connecting letters in lines are stretched to a maximum amount, with any remainder going to preceding words. In Figure 10, the stretch is distributed among all connections to last connecting letters in words in a line.

Figure 11 is the first page of a technical report [24] 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 the University of California at Los Angeles (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 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,

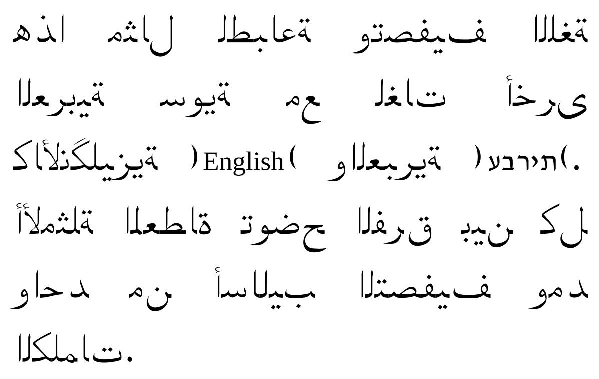

ةغللا فيفصتو ةعابطل لاثم اذه
ىرخأ تاغل عم ةيوس ةيبرعلا
ةيزيلكنلأاك)English(ةيربعلاو)תירבע(.
لك نيب قرفلا حضوت ةاطعملا ةلثملأا
دمو فيفصتلا بيلاسأ نم دحاو
تاملكلا.

Figure 6: Example ditroff output not piped through ffortid

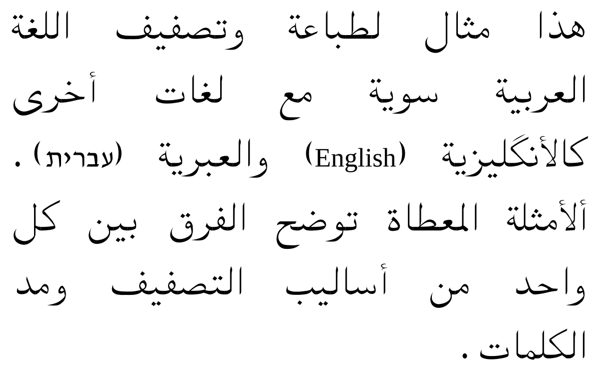

هذا مثال لطباعة وتصفيف اللغة
العربية سوية مع لغات أخرى
كالأنكليزية (English) والعبرية (עברית) .
ألأمثلة المعطاة توضح الفرق بين كل
واحد من أساليب التصفيف ومد
الكلمات .

Figure 7: Same ditroff output piped through ffortid with stretching off

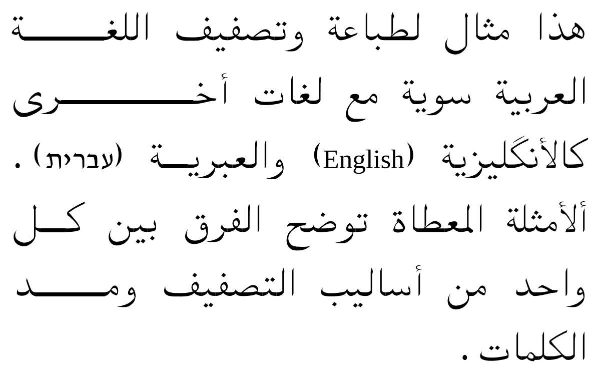

هذا مثال لطباعة وتصفيف اللغـــــــة
العربية سوية مع لغات أخـــــــــــرى
كالأنكليزية (English) والعبريـــة (עברית) .
ألأمثلة المعطاة توضح الفرق بين كـــل
واحد من أساليب التصفيف ومـــــــد
الكلمات .

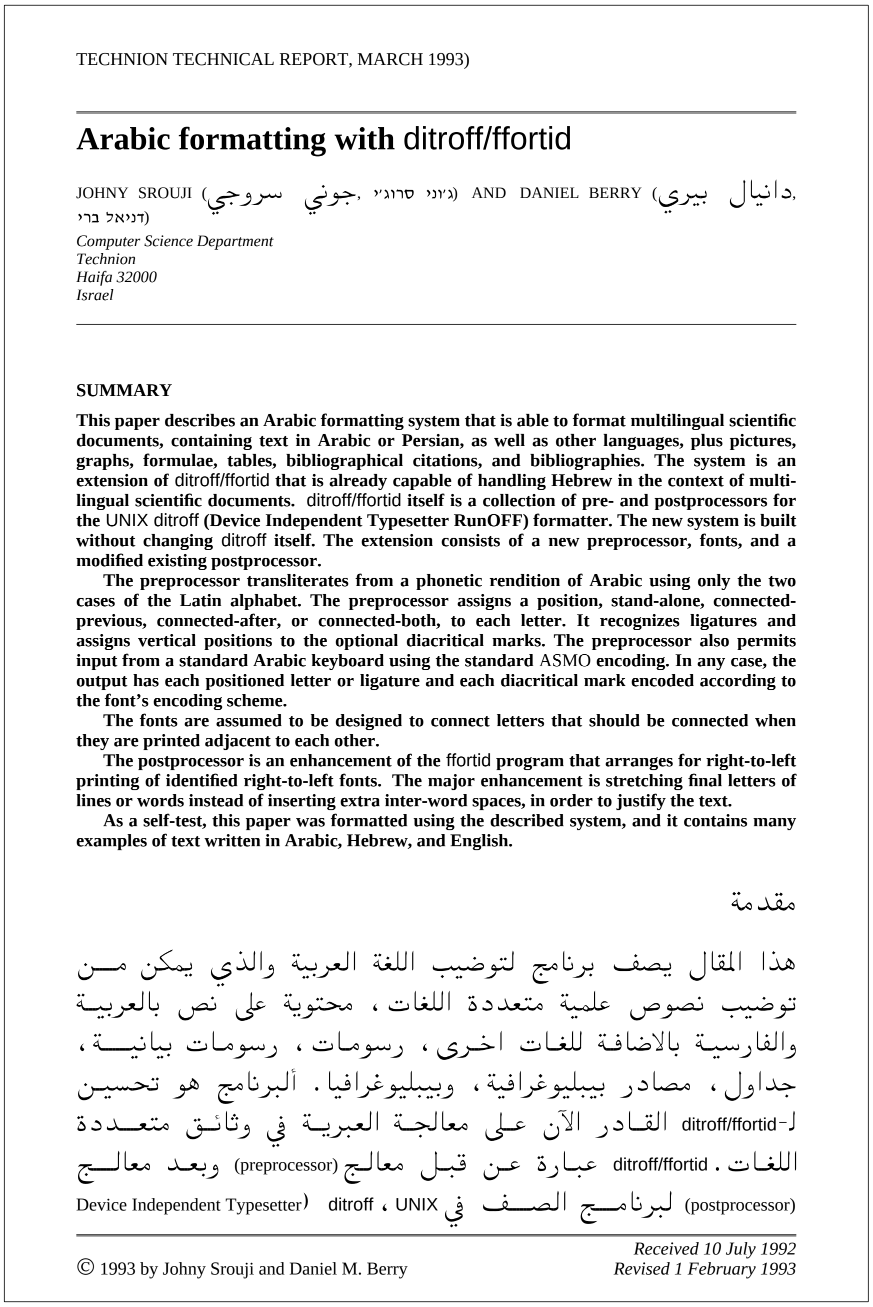
Figure 8: Last connecting letters in lines are stretched

هذا مثال لطباعة وتصفيـــف اللغـــــة
العربية سوية مـــع لغـــــات أخـــــرى
كالأنكليزية (English) والعبريـــة (עברית) .
ألأمثلة المعطاة توضح الفرق بين كــــل
واحد من أساليب التصفيـــف ومـــــــد
الكلمات .

Figure 9: Last connecting letters in lines stretched up to maximum amount

هــذا مثـال لطباعـة وتصفيـف اللغـة
العربيـــة سويـــة مـــع لغـــات أخـــرى
كالأنكليزيـة (English) والعبريـة (עברית) .
ألأمثلة المعطاة توضح الفرق بيـن كل
واحـــد مـــن أساليـــب التصفيـــف ومـــد
الكلمات .

Figure 10: Stretch distributed between all last connecting letters in words



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 من
توضيب نصوص علمية متعددة اللغات ، محتوية على نص بالعربية
والفارسية بالاضافة للغات اخرى ، رسومات ، رسومات بيانية ،
جداول ، مصادر بيبليوغرافية ، وبيبليوغرافيا . ألبرنامج هو تحسين
لـ-ditroff/ffortid القادر الآن على معالجة العبرية في وثائق متعددة
اللغات . ditroff/ffortid عبارة عن قبل معالج (preprocessor) وبعد معالج
(postprocessor) لبرنامج الصف في UNIX ، ditroff (Device Independent Typesetter

Received 10 July 1992
© 1993 by Johny Srouji and Daniel M. Berry *Revised 1 February 1993*

Figure 11: ffortid example output with combined English, Hebrew and Arabic text

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 1: ffortid version history

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 2: ffortid source files

a M.Sc. student at the Technion, extended ffortid for Arabic stretching during 1989-1991. Table 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 of the thesis [13].

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 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.

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 describes 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 M.Sc. research,
- 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,[3] and
- the control had already written a new version of ffortid using conventional maintenance methods; this saved some time in the experiment without affecting its results.[4]

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

[3]This is why this paper was prepared using TEX rather than ditroff.

[4]Because of the control's other duties as a professor, he ended up being the bottleneck in the case studying, slowing up any step in which his participation was required.

4 Domain Software Reengineering of ffortid

The control's next step in the experiment, as described in Section 2.2, is his 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 that it is written in its proper direction and can stretch Arabic text so it is left and right justified on the line.

As the subject 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 the ALRSLC 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 end up being 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.[5] However, SWUs are more general than modules. Any software module is by definition a SWU[6], but the SWU definition includes software components that 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.[7]

The SWU concept gives a uniform view of software. It transcends traditional boundaries of scale, language, storage medium, programming or design technique. It can be applied successfully to any program in any language because it captures the essence of software, which is to provide computational services. It can be applied equally successfully to programs written in machine languages, procedural languages, functional languages, fourth-generation languages, or even job-control 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 are stand alone. However, at least some SWUs must cooperate with other SWUs to provide their services or else we are left with only a collection of low-level service providers. Every SWU has a scope, capabilities, an interface, requirements, and a type:

- The *scope of a SWU* is the body of code that it abstracts. The scope does not have to be contiguous.
- The *capabilities of a SWU* are the services it provides.

[5] Not necessarily compilation units as in C.

[6] All acronyms are pronounced as their expansions, hence "a SWU" rather than "an SWU".

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

- 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. 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.

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 the SWU is used that is not in the SWU's scope. The environment of a SWU therefore depends on the context in which the SWU 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 is 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 the software project, how to access it, and how it might affect the rest of the software in the project.

4.1.1 Software Sub-Units

Every non-trivial SWU can be decomposed into its software sub-units. A sub-unit is a SWU in its own right. The scopes of the sub-units must be mutually exclusive, and the union of these 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.

The sub-units of a single SWU do not all have to be of the same type or to be recorded in a certain order, such as the order of their scopes. 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 criterion provided by the decomposer. The diagram in the manual page in Figure 15 below 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 that 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 a SWU does have a natural decomposition into more than 7 sub-units, some

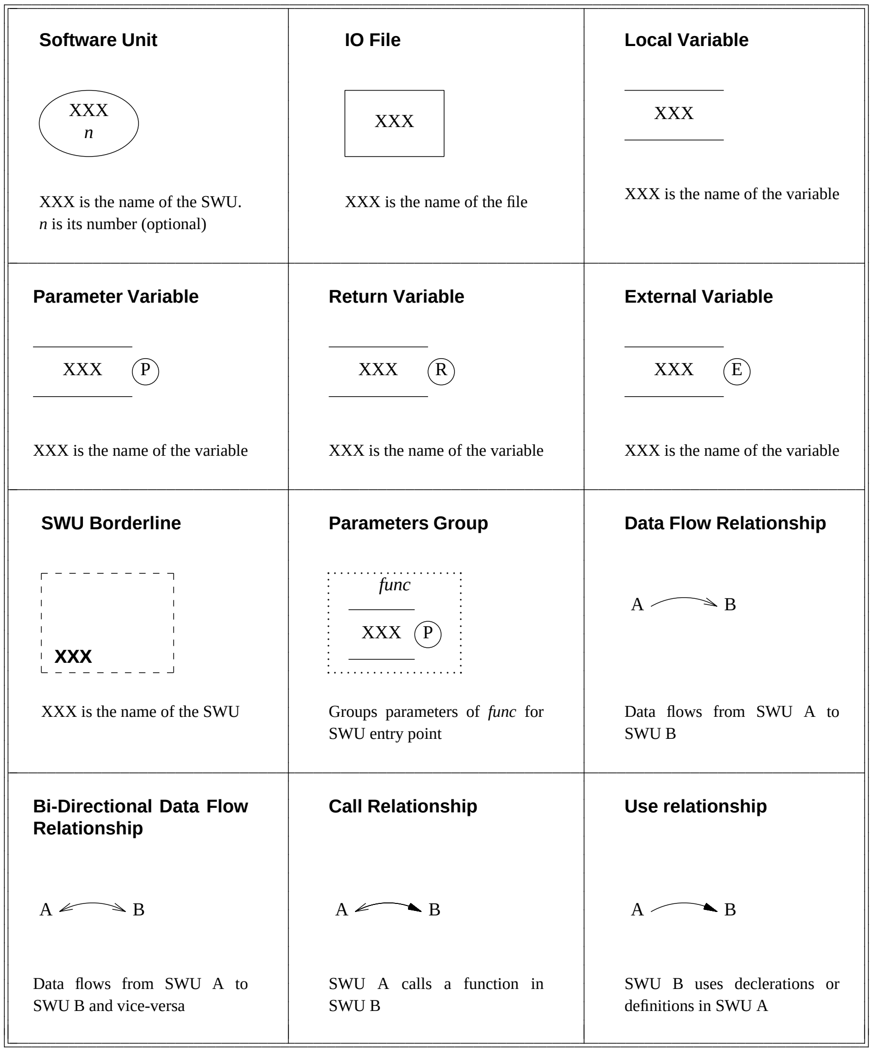

Software Unit
XXX
n
XXX is the name of the SWU.
n is its number (optional)
IO File
XXX
XXX is the name of the file
Local Variable
XXX
XXX is the name of the variable
Parameter Variable
XXX P
XXX is the name of the variable
Return Variable
XXX R
XXX is the name of the variable
External Variable
XXX E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

Figure 12: Major icons used in SFDs

of them should be grouped into a single sub-unit. If it does not seem natural to decompose the SWU into less than 7 sub-units, then usually, there is some complexity problem in the SWU abstraction, and perhaps the SWU itself should be split into smaller abstractions.

4.1.2 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 in a SFD is the contents of Figure 12.

Figure 16 below 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 sends 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

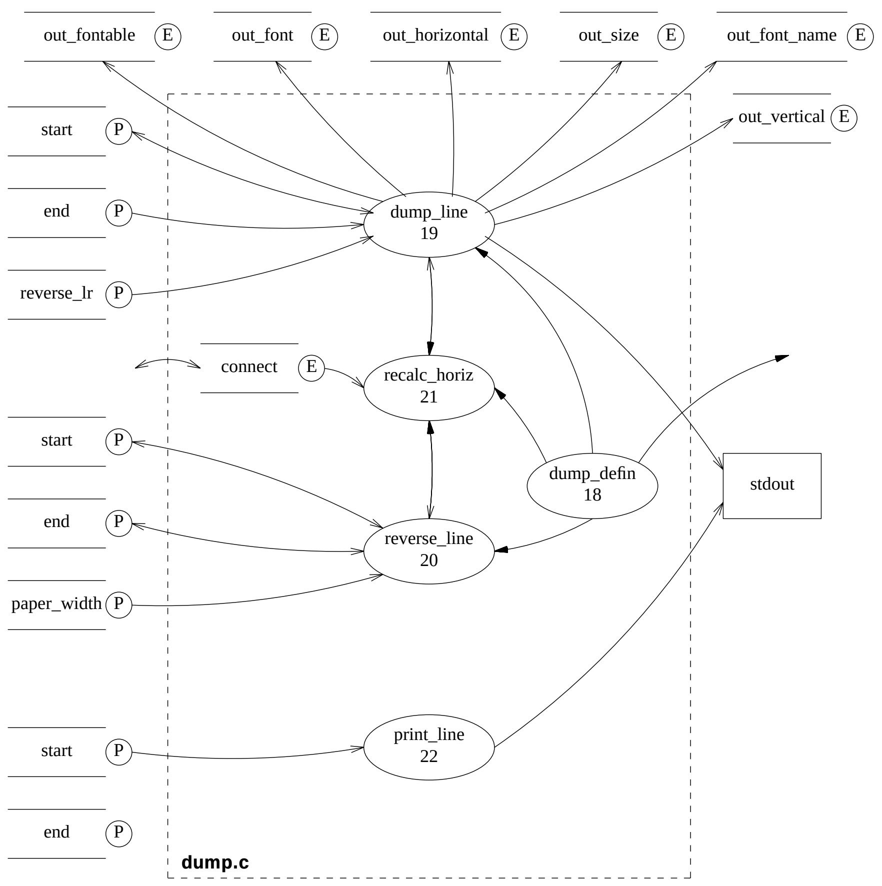


Figure 13: SFD of dump.c SWU with its sub-units

interface and required services.

Figure 13 shows 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 that dump.c changes 6 global variables as a side-effect, and 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, we suggested 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. We 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, easily understood 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 14 shows a SFD that is very difficult to understand. Perhaps a SFD can serve as an important visual indication of the external complexity of a SWU.

4.2 Reverse Engineering a Software Unit

Reverse Engineering an existing SWU has two major goals [10]: 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 that are candidates to be reused as software components,
- to recover information that is not documented in the source code, for example, about modifications that were performed during maintenance but were never documented,
- to detect incorrect documentation, errors etc., 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 programmer 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. 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 4.1.1, the architecture of the SWU is represented by a hierarchical map of SWUs, in which child sub-units show the details 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. The criteria are usually based on the syntactic structure of the code combined with the principles of high cohesion and low coupling. For example, a program in C is first decomposed into its compilation units (modules). If there are too many modules, logically related modules can be grouped to create a smaller number of sub-units. This grouping follows the principles of high cohesion, i.e., strong service relations inside a SWU, and low coupling, i.e., weak service relations between SWUs. Each of these module groups is 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 are 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 that holds abstractions that 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, are good reuse candidates. Doing so implies that the function was too large from the beginning, 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 design improvements we recognize can and should be recorded, but they

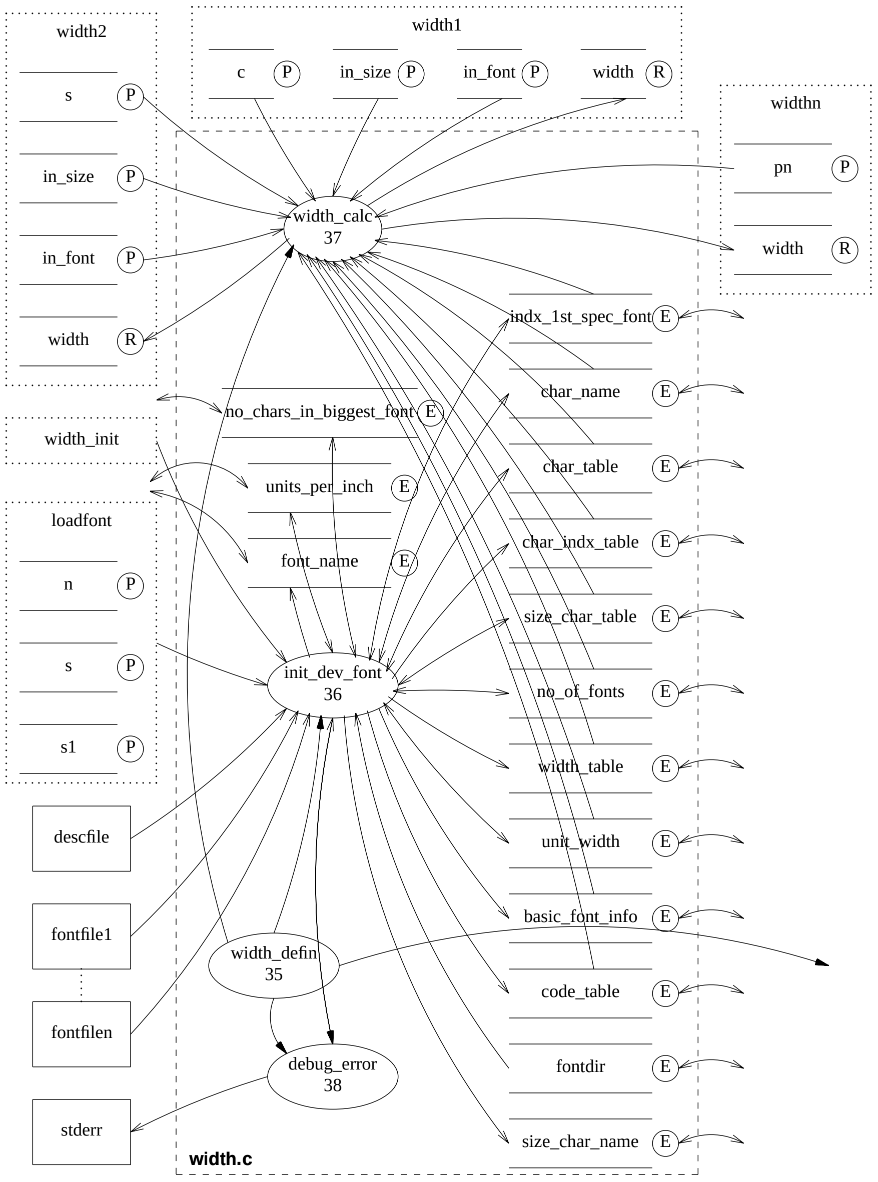


Figure 14: A complex SFD

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, because 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 sub-units. Section 4.1.1 shows 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 lowest 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 is propagated up until it is satisfied by a SWU at a higher level, at which point it disappears. A side effects is propagated up from its originating SWU until it reaches a level, if any, in which it is no longer considered a side effect because its effect is internal to the SWU at the new level.

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

1. Partition the SWU into sub-units and continue recursively.

2. Partition the SWU according to the syntactic structure of the SWU and the principles of high coupling and low 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

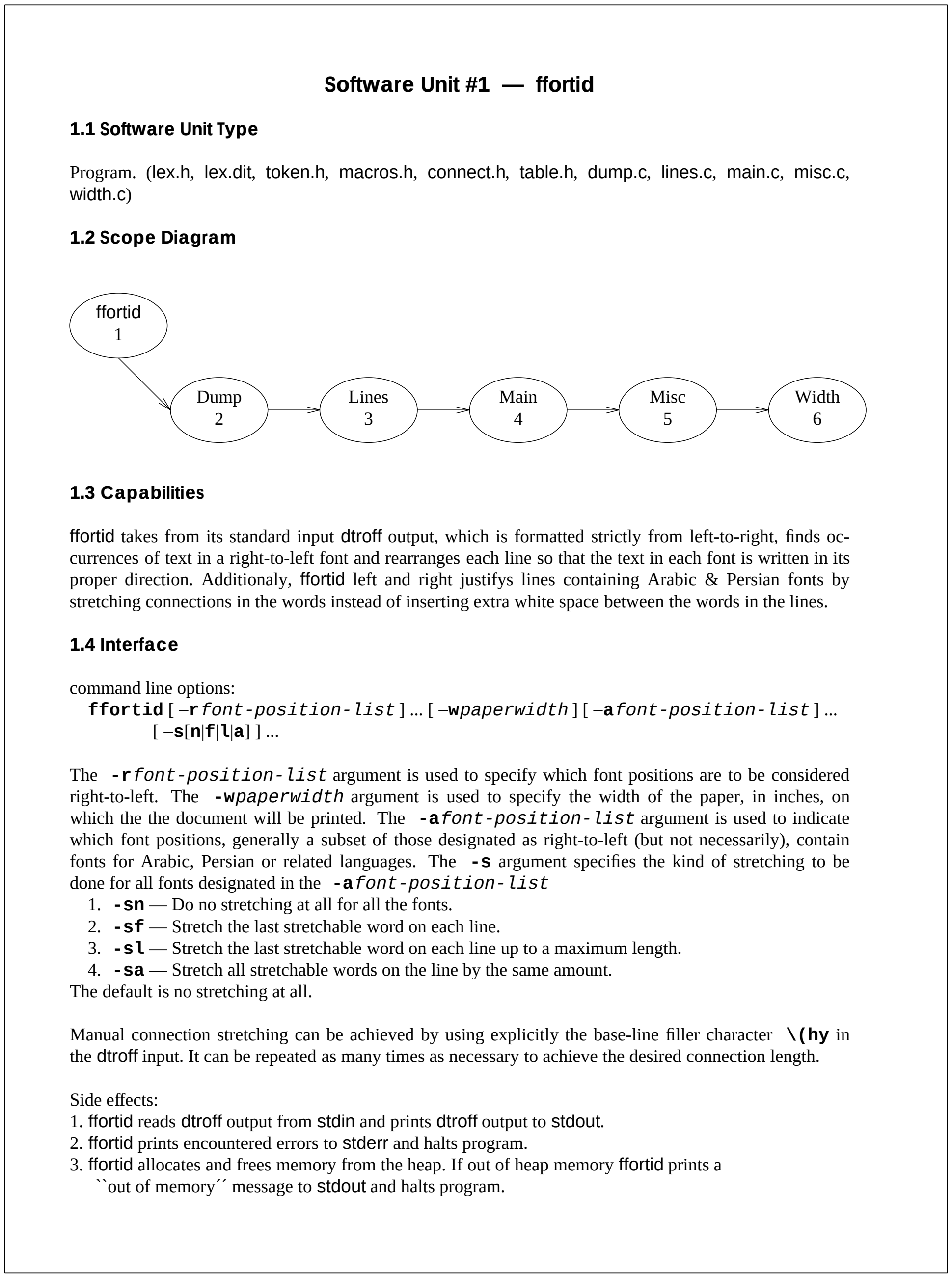
The subject performed a process of reverse engineering as described above on ffortid Version 3.0. The process was performed completely manually using only traditional methods of text editing and UNIX commands such as grep. All SFDs were drawn manually using pic [17]. The whole process was very laborious 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 lowest level SWU to be found was the function or procedure. It turned out to be unnecessary to carry out all refinements even to this level.

The initial partitioning criterion was that 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, he or she might see the abstractions of the software differently.

Each SWU in the decomposition was documented in what is called a *Software Unit Page*. Figure 15 shows the first part of the page documenting SWU 1, which abstracts the complete ffortid program. 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. The second section



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

$$\begin{tikzcd}
\text{ffortid } 1 \arrow[dr] & & & & & \\
& \text{Dump } 2 \arrow[r] & \text{Lines } 3 \arrow[r] & \text{Main } 4 \arrow[r] & \text{Misc } 5 \arrow[r] & \text{Width } 6
\end{tikzcd}$$

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 15: First part of ffortid Version 3.0 SWU 1 page

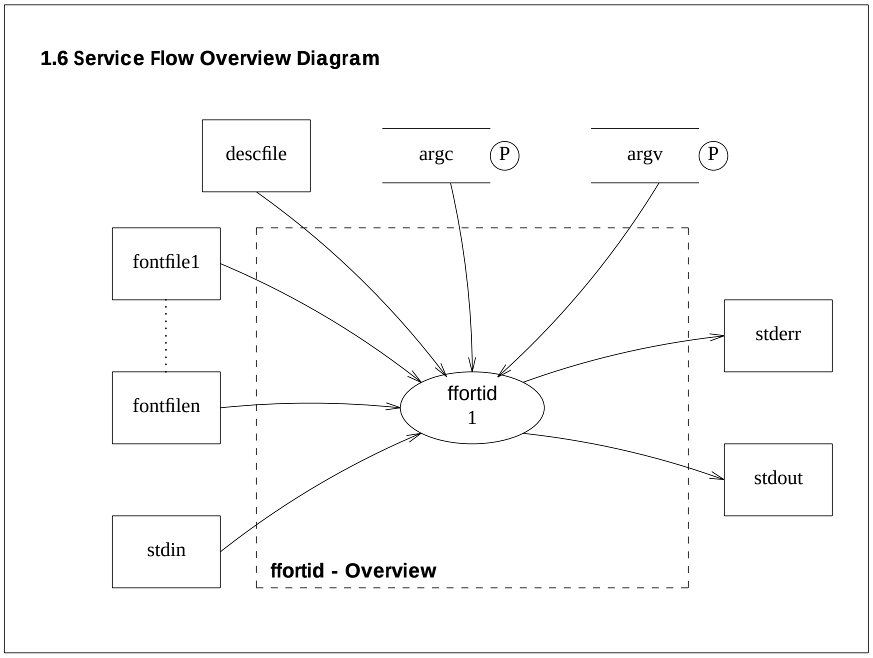


Figure 16: Third part of ffortid Version 3.0 SWU 1 page

of the page, titled “Scope Diagram”, contains 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 that captures the architecture of the program. Each of the sub-units has its own SWU page that describes the sub-unit completely in the same fashion.

The third section of the SWU page, titled “Capabilities”, gives a verbal description of the capabilities of the SWU, a precise and concise description of the SWU capabilities in a language that is clear to the domain and software expert. In the case of the ffortid SWU page, 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, simply a list of the different services provided by the SWU. The interface section describes the side effects of the SWU. In the case of the ffortid SWU page, the interface is the command-line options of the program. These options are described completely in the section.

As defined previously, side effects are changes in the SWU’s environment that 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 that 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 or streams and allocation and deallocation of dynamic memory are, therefore, all side effects of 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. For SWU 1 it was decided to add a sixth section, titled “Service Flow Overview Diagram”, which is also a SFD describing the services 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 16, 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 sends output to the standard output and the

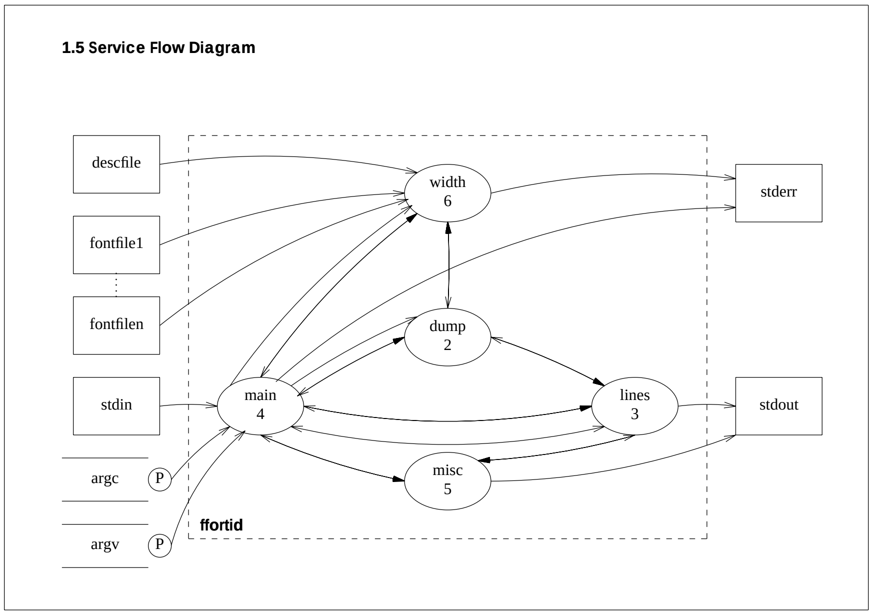


Figure 17: Second part of ffortid Version 3.0 SWU 1 page

standard error streams. Allocation and deallocation of dynamic memory are not shown in the diagram, although with appropriate icons, they certainly could have been.

As mentioned previously, the fifth section of a SWU page contains 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 17 is the SFD of SWU 1. Unlike the SFD in Figure 16, it shows in detail the service flow between the sub-units of ffortid and the relation of these sub-units to the environment. 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. The SFD shows which module calls functions in other modules and from which modules data flow to other modules.

There is no requirements section in our SWU pages, because 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, to determine 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 18 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.

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 3 summarizes this information for all the SWUs in the decomposition.

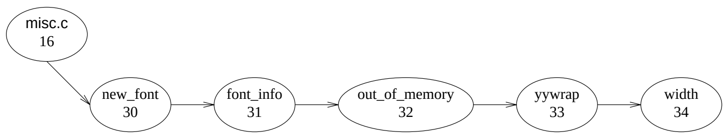
Figure 19 shows the complete decomposition of ffortid into its SWUs in the form of a scope diagram.

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

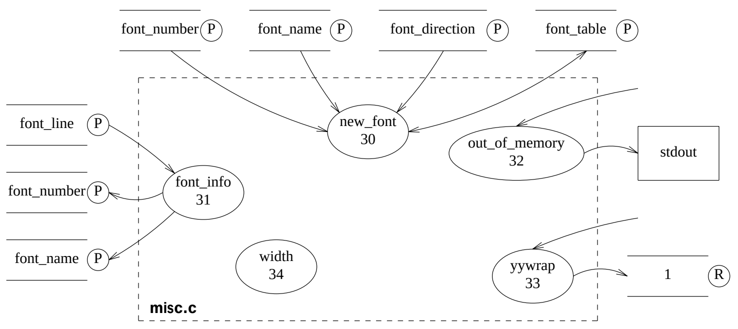


Figure 18: SWU 16 page in ffortid Version 3.0 decomposition

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 3: ffortid Version 3.0 software units

Due to the diagram's length, it was broken into 5 scope diagrams, one for each of ffortid's direct subunits. 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 SWU's scope, capabilities, interface, and SFD proved very effective in advancing the subject'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 that 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 justify lines, and then output the token lines in the same format as ffortid input.

ditroff output is a stream of well-defined tokens that are device-independent commands to a typesetter, usually a laser 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.dit based on SWU 27, into lexical tokens, SWU 8.

As the program starts, SWU 4, the "Main" module, parses the command line options and stores them in global variables. It then reads in each 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, to 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 subject's complete decomposition of this (and the next) version of ffortid is available in Adobe Acrobat pdf format at [14]

ftp://ftp.cs.technion.ac.il/pub/misc/dberry/hornreich.work/FFVER*.PDF.

The decomposition has hypertext links between the different SWU pages, enabling easy traversal between SWU pages, source code, and all other relevant documents.

4.5 Subject's Conclusions from Decomposition

This section, written in the first person, describes in the subject's own words what he learned from doing the decomposition.

Performing a decomposition of a legacy program has a lot in common with archeology. I discovered mixed layers of architectures and changes performed by different programmers at different times and

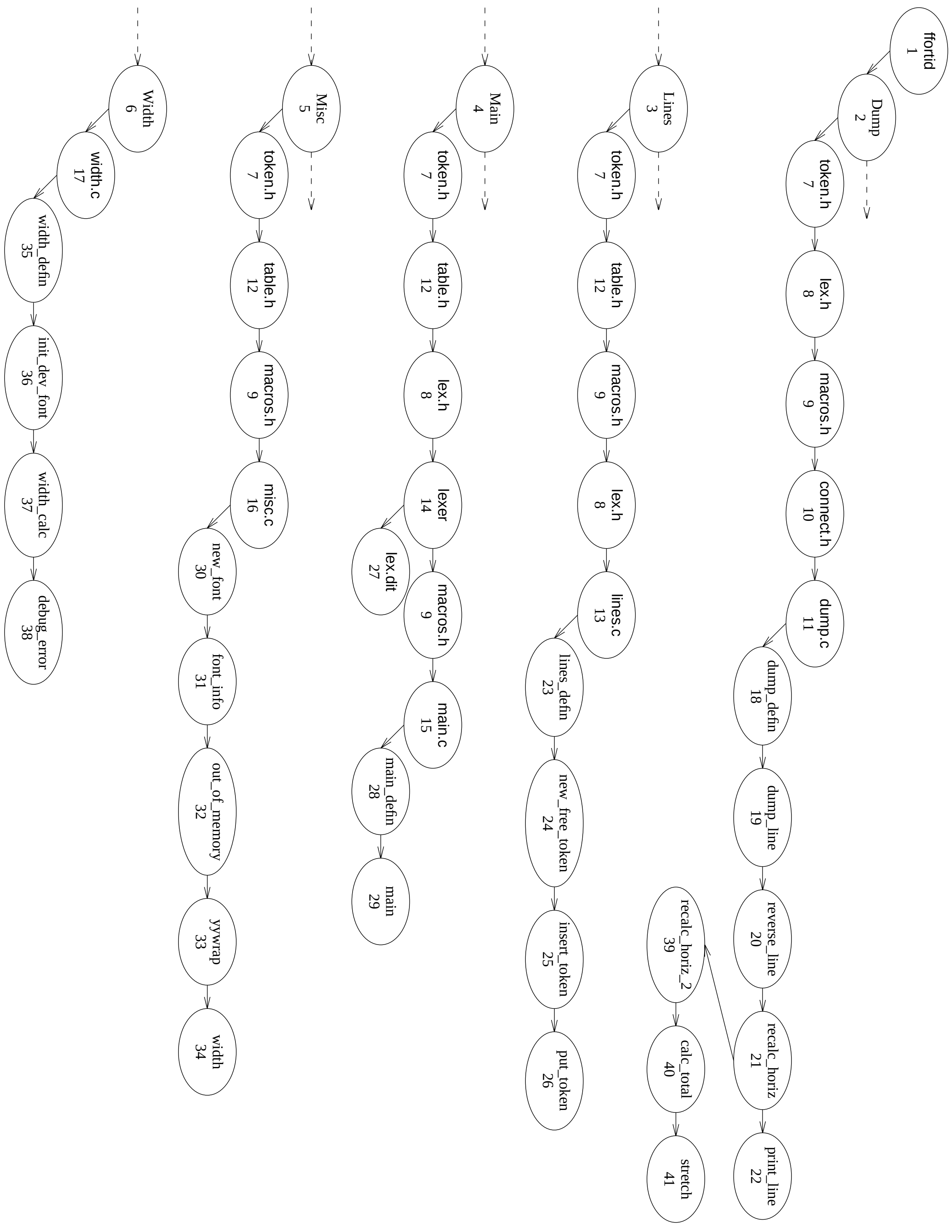


Figure 19: Overview of ffortid Version 3.0 decomposition

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 found several small bugs, erroneous comments, unused code and variables and even a gross difference 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. However, I did not 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, because all the capabilities, interfaces and side-effects of the SWUs propagate 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, including recognizing the use of global variables, function calls etc. and where these were defined. I used grep to do these simple tasks, but the abilities to perform automated queries on the code, just as in a database, and to generate different views of the code, in my view, would greatly advance the software understanding process.

The SFDs 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 decomposing, i.e., deciding on the partitioning criterion. Additionally, I have found that they did not help much in understanding 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 in getting a global view of high level SWUs services especially when I tried to understand a SWU on which I had not worked 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 the method in a reasonable period of time on a large legacy system. 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. See Section 7.4 for a discussion of the requirements for such CASE tools.

4.6 The Initial Domain

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

The subject 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 composed directly of 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 of 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 would be 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 this case, it would probably be 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?

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.

The control 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 SOTP maintenance method to implement these requirements. Only after the subject had finished creating the domain, was he presented with this new set of requirements, so they could not have affected in any way the architecture of the domain he created.

The subject was to implement these new requirements using the ALRSLC model as his basis. As previously described, the intention is that the domain develop in an evolutionary manner according to external requirements. Therefore, no modifications were to be made to the domain unless they could 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

Software modifications are a natural phenomenon of software evolution. We should however, attempt to avoid ripple effects in which a modification in one SWU causes modifications in other SWUs. Having good domain abstractions with carefully planned interfaces that pass high-level information abstractions helps to avoid ripple effects.

5.2 The New Requirements

ffortid version 4.0 was to 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 [4] in which letters can actually be stretched. Figure 20 (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 20 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 stretched in traditional 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 of the thesis [13]. From a comparison of the old and new manual pages, we created a list of three 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

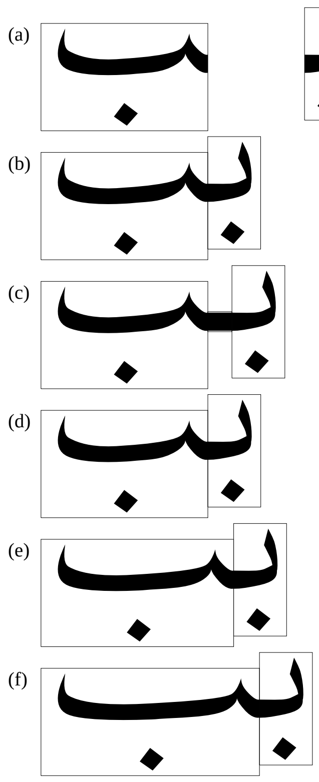


Figure 20: Connecting letters, fillers, and dynamic letters

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 the word; or both letters and connections.

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 and connectable.

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, thus, adds the capability to manually stretch letters in words. When automatic line stretching is enabled, these words may be stretched even more in order 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 having ffortid precede each letter that it wants to stretch by a new output token. 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 output token 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 three 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.

5.3 Subject's 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 that represents an application. This SWU tells us how to build applications from our reusable components. Having only one is not always the case, especially in more advanced domains in which families of applications have been planned or in which applications with completely different architectures could be generated. In such domains, we would have several SWUs, each representing a different application or 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.

Two of the changes, I–1 and I–2, are modifications to existing interfaces, and two, I–3 and I–4, are completely new additions to the application interface. 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 Add the ability to store manual stretch values as part of character tokens.

C–2 Add the ability to store automatic stretch values as part of character tokens.

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

Two of the capability changes, C–1 and C–2, are completely new capabilities, and one, C–3, is a modification of existing capabilities.

The next step in the ALRSLC 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 changes into interface and capability changes serves several purposes. First, it helps separate the internal and external changes to the application. Secondly, it permits the implementation of the changes using the method described 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. The example below of the method is described in the first person of the subject.

For example, I-1 is a modification of the current command-line options. According to the theory of SWUs detailed in the thesis [13], there exists a sub-unit of SWU 1 that provides this interface service. By interface service, I mean that there exists a sub-unit that reads in, parses, and stores the command-line options for the use of other sub-units. In Figure 19, it can be seen that the top-down search flows from SWU 1 to SWU 4 to SWU 15 and finally to SWU 29. I therefore modified SWU 29, which is the “main” function, to implement I-1. The changes required 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 caused 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 had also to 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 I had used a user-defined type that abstracted the command-line options, I would have needed to change only this type's capabilities, i.e., its fields, in SWU 29 and to change SWU 41 to use these fields. No other SWUs would have been affected. It is clear that in some cases, modifications are bound to have side-effects, but these side-effects should be kept to the 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 change 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 fact. 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 that caused a change in the domain hierarchy. The top-down search led me from SWU 1 to SWU 6 to SWU 17, and finally to SWU 36. There, I added 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, I added global variables to SWU 35 to store this additional information. I decided however to provide functions that access this new information so it does not have to be accessed through the global variables. I 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 macros.h, SWU 9, and that new code was included in width.c. Therefore, SWU 48 became a sub-unit of SWU 6, "Width". This modification did not generate any modification side-effects, except in SWU 35, largely because it was an addition to the current interface without any changes to the previous one.

Modification C-1 calls for viewing the manual stretch value in a character token 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, I created new concepts by creating two functions, "tokenBasicWidth" and "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. I 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 I realized that SWU 41, which is the heart of the line stretching algorithm, would require complete refurbishing in order to implement the modification. It did not have the abstractions necessary to represent the new required functionality. I, 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 change did not mean I could not use some of the SWU 41 code in the new SWUs. I 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. I 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 evident only 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.

I found several additional minor bugs in the original code, but I corrected them without any ripple effects or major domain architecture changes.

Figure 21 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, I saw no need to preserve SWUs that have been deleted or replaced by better SWUs. In more advanced domains in which there could be a choice between several components, this choice should be reflected in the domain architecture and scope diagram.

Table 4 shows all information on the SWUs in the updated domain.

5.4 Subject's Implementation Data

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

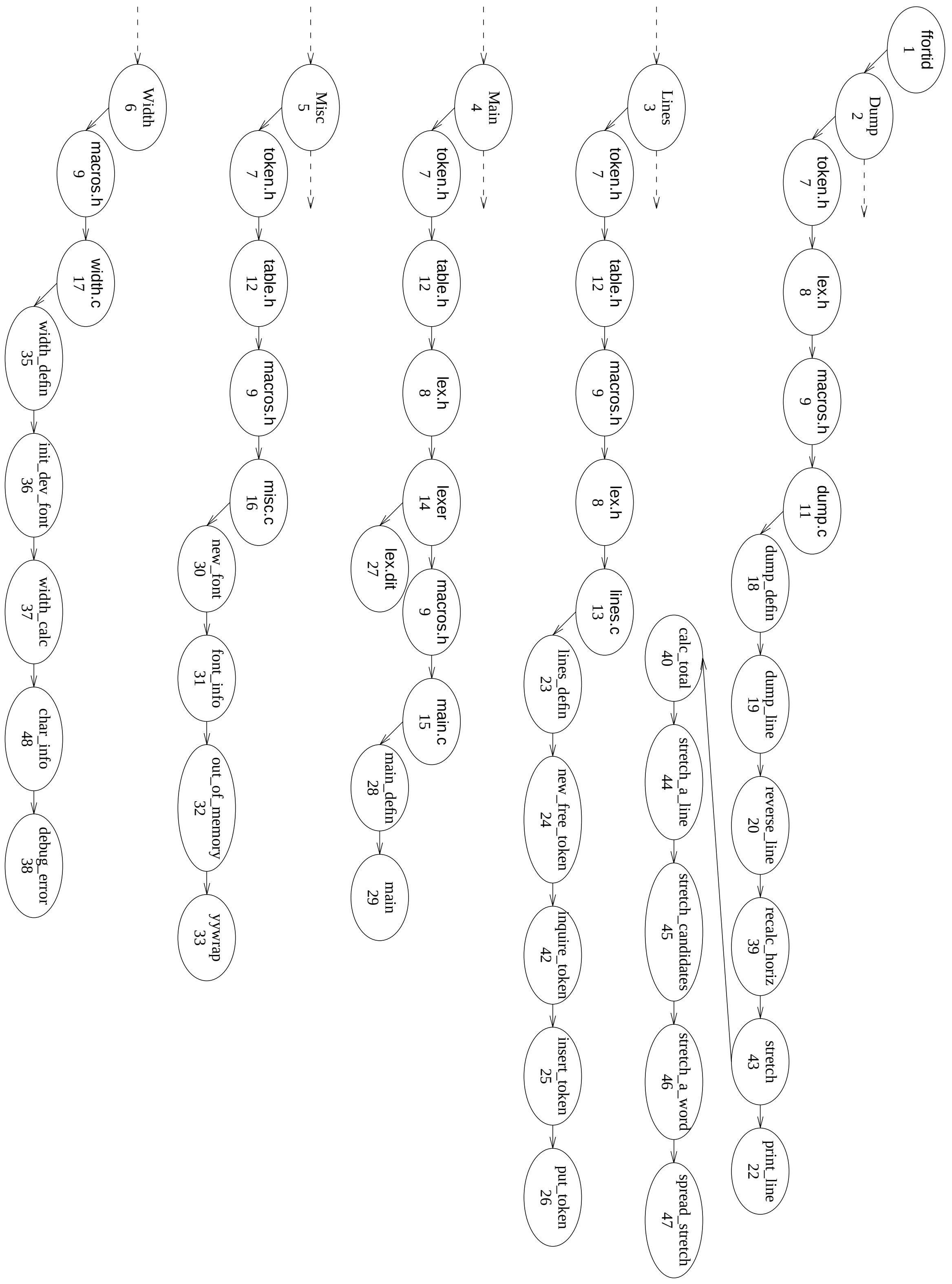


Figure 21: Overview of ffortid Version 4.0 domain

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 4: ffortid Version 4.0 software units

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.

5.5 Control's Implementation

The control implemented the same requirements using the same original ffortid version as his basis. He used his own systematic SOTP maintenance method to implement these requirements. The major steps in 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.
- 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 this is done, 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 that do not invalidate current functionality
  - all changes that change current functionality
  - all changes that 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, the control found it was possible to create an order of implementation that enabled the implementation of each modification and its subsequent testing. The control found that a typical test would expose two or three bugs and about half of these were unforeseen ripple effects.

5.6 Subject's Implementation Data

To the original 2510 line ffortid program, the control added 1997 lines, deleted 784 lines and modified 36 lines. Therefore, his implementation of ffortid Version 4.0 had altogether 3723 lines.

5.7 Implementation Comparison

The most striking difference between the two implementations is in the number of added lines, 321 by the subject compared to 1997 by the control. This difference 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. The control 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 subject 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, we claim 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 do exactly the same changes as the subject had done without reverse engineering ffortid. However, by using the ALRSLC 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 the control 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 that had seemed clear to the control, but were, on second thought, not well defined.

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. Another set of more advanced requirements were to be devised and implemented by the subject and the control on their individual latest application versions. These requirements were from the control's mind and were not yet known to the subject.

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 of the thesis [13]. 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 characters.
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. There 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.
- 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.

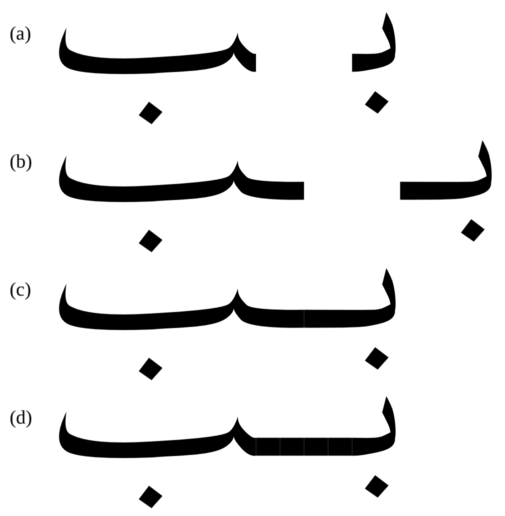


Figure 22: Stretchable letter connections and fillers

For example, figure 22 (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 23 shows each word's baseline as a solid arrow and the line's baseline as a dotted arrow.

Figure 24 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 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 providing for two ditroff commands that surround the embedded left-to-right text and that cause ffortid to recognize that the surrounding text should be treated as a single right-to-left unit.

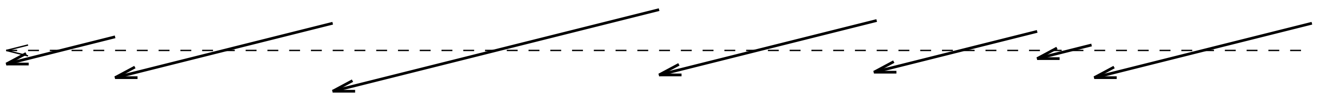
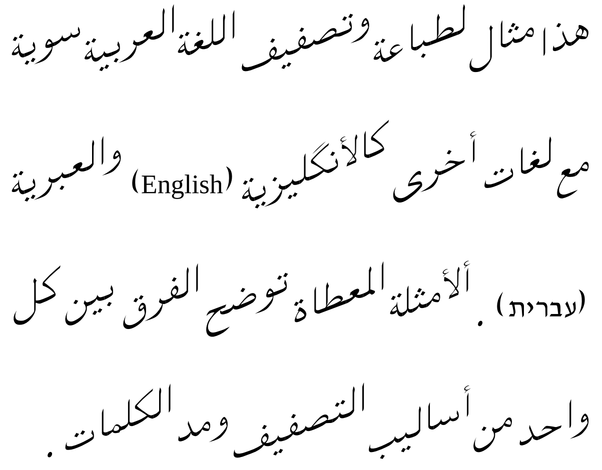


Figure 23: Layout of slanted font words on line



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 24: Sample slanted output

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

Enhancement 5 allows better control over the type of stretching fonts provided. 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 allows the stretching of only characters entered by their numerical code as the argument of a `\N` escape. Enhancement 6 enables the stretching of characters also entered by their ASCII or two letter synonym.

Enhancement 7 is a trivial enhancement that 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 subject 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.

The control implemented the same requirements using his own SOTP method described 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 he tested the new features.

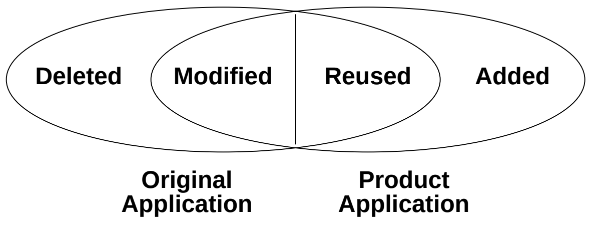


Figure 25: Relationship between original and product applications

$$\text{Reuse Ratio} = \frac{\text{reused lines}}{\text{original lines}} \tag{1}$$

$$\text{Modification Ratio} = \frac{\text{modified lines}}{\text{original lines}} \tag{2}$$

$$\text{Addition Ratio} = \frac{\text{added lines}}{\text{product lines}} \tag{3}$$

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 25 shows as a Venn diagram what happens when we create a new, product application from some original application. Part of the original code is deleted and is not included in the product application. Another part of the original code is modified and included in the product application. Finally, still another part of the original 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 the existing code. 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 an acceptable measure of code size over the years. Additionally, comments are also reused in projects and not only program statements; therefore, it is logical to count them as well in measuring program size.

The *reuse ratio*, defined in Equation 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.[8] Clearly, a small reuse ratio means that little code was reused from the original application, and a large reuse ratio means that much code was reused from the original application.

The *modification ratio*, specified in Equation 2, is similar to the reuse ratio except that it measures

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

	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 5: Experiment results

the number of modified lines in the product application relative to the original application's size.[9]

The *addition ratio*, specified in Equation 3, measures the number of new lines in the product application relative to the original application's size. This measure is important because it tells us how much of the final application, $(1 - \text{addition ratio})$, is from completely new code and how much is from directly reused and modified code. It is possible to have a case with a high reuse ratio and a high addition ratio. This situation indicates that although we reused a large proportion of our original application, the reuse amounts to only a small fraction of our final product application. 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 would have been better to create the complete product application from scratch, as most of it was written from scratch 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 subject and the control, we recorded the number of deleted, modified, and added lines from the original application it was derived from. Additionally, for the second version of each, we recorded the final number of lines in the product application and the implementation time in hours, including testing. The results for all the different applications in the experiment can be found in Table 5.

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

The reason for this wide difference is that the control took a design decision different from that of the subject, 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 subject 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. The control chose to store any information

[9]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 6: Experiment results analysis

calculated in appropriate data stores, so that no item will 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 subject, 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 user on the current abstractions when they exist and not on creating new abstractions. Only when the needed abstractions do not exist in the current architecture must one add them 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 subject had done without reverse engineering ffortid. However, with the ALRSLC method, these changes came naturally and easily.

No implementation time is recorded for either version of ffortid 4.0 for two reasons: First, the control's version was written before this experiment was conceived, and no time data had been recorded. Secondly, the subject was learning and inventing his method as the application was being created, and therefore, it would not be fair to compare the two versions' implementation times. This lack of time data was one reason for including the creation of ffortid 5.0 in the experiment; that creation could be timed from the beginning by both the subject and the control.

The analysis of these results can be found in Table 6. 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 subject's addition ratio, 35%, is much lower than the control's addition ratio, 54%, because the subject added significantly fewer new lines to the code than did the control.

According to the analysis in the previous section, the subject'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.

Each of the subject and the control started his version of ffortid 5.0 from his own version of ffortid 4.0. Each deleted very few lines from his original application, although, as shown in Table 5, the subject deleted more lines than the control. These data indicate that each implementor added good code to his version of ffortid 4.0, because each reused most of his own ffortid 4.0 in his ffortid 5.0. Neither the subject nor the control modified many lines, with the control modifying a few more lines, and both adding nearly the same number of new lines. The final application of the control, therefore, still had significantly more lines than the subject's because the control 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 found himself thinking about the coding in the background while doing other activities totally unrelated to the coding. The subject worked more contiguous hours and rarely thought about the coding during the other hours, having had his full during the regular working hours. Even if we take the minimum hours recorded by the control, they are approximately twice the implementation hours of the subject. This can be attributed to the fact that the subject's original application was documented in the form of a domain, while the control had only the commentary in the code itself. The domain documentation allowed the subject 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 subject's view, the overall result of this was better software understanding by the subject and the ability to perform modifications quicker.

The analysis of these results using these ratios can be found in Table 6. 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 subject is higher than that of the control, because the subject 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 versions' 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. This difference, we believe, shows the advantage of using the proposed ALRSLC 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, we claim that they indicate clearly that the subject'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; and
- it has higher quality code.[10]

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 subject was a much better programmer than the control and this difference explains the difference in the implementation time, etc. We do 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 knowledgeable person. It is well known that an ignorant person performs better on some tasks because of a lack of tacit assumptions a knowledgeable person makes due to his

[10]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. However, both authors agree with this conclusion!

increased knowledge [7]. This can also be a possible explanation for the different design decision taken by the control in his implementation of ffortid 4.0.

The subject has used the ALRSLC method to produce an initial domain, and has carried out 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.

In the subject's view, only the use of such methods with 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.

A remark made by Berry while reading Hornreich's first draft of Section 5.3 recognizes the potential of the methods studied in this experiment. "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!".

7.4 CASE Tool Requirements

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 documentation of capabilities must still be performed mostly manually. The desired tools are therefore semi-automatic, reverse-engineering tools.

With the use of expert knowledge, a reverse engineering tool could provide suggestions to help the human operator make faster and more knowledgeable decisions. For example, it could suggest one or several options for partitioning a SWU according to its syntactic structure or the resulting service flow dependencies between the sub-units. The human operator could then decide to accept one or more of the suggested decompositions or to provide one of his or her own. More advanced tools could even try to learn new partitioning criteria from previous human partition decisions.

To summarize, a CASE tool would help 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 that 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.

The desired CASE tool should not only provide assistance in the reverse engineering process itself, but it should also provide an environment in which the discovered architecture can be navigated, e.g., 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 that can be viewed and easily changed as we change previous decomposition decisions. This database would serve later a basis for changing the source code or SWU structure.

Storing the complete SWU architecture in a database would greatly help changing the different components and realizing the effects of these changes.

7.5 Acknowledgments

The authors thank Prof. Noah Prywes for describing his methods and for exchanging ideas during the experiment.

References

[1] J.D. Ahrens and N.S. Prywes. Reengineering the software life cycle and enabling technology. Technical report, Computer Command and Control Company, July 20 1994.

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

[3] J.D. Ahrens, N.S. Prywes, and E. 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.

[4] J. André and B. Borghi. Dynamic fonts. PostScript *Language Journal*, 2(3):4–6, 1990.

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

[6] L.A. Belady and M.M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, March 1976.

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

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

[9] C. Buchman and D.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.

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

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

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

[13] H.I. Hornreich. A case study of software reengineering. Master's thesis, Technion, Haifa 32000, Israel, 1996. Available in Adobe Acrobat PDF format from ftp://ftp.cs.technion.ac.il/pub/misc/dberry/hornreich.work/.

[14] H.I. Hornreich. ffortid version 3.0 decomposition manual. Technical report, Technion, Haifa 32000, Israel, 1996. Available in Adobe Acrobat PDF format from ftp://ftp.cs.technion.ac.il/pub/misc/dberry/hornreich.work/.

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

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

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

[18] B. Kitchenham, L. Pickard, and S. Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

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

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

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

[23] 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.

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