On the Maintenance Costs of Formal Software Requirements Specifications Written in the Software Cost Reduction and in the Real-time Unified Modeling Language Notations

by

Irwin Kwan

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

Author's Declaration for Electronic Submission of a Thesis

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

A formal specification language used during the requirements phase can reduce errors and rework, but formal specifications are regarded as expensive to maintain, discouraging their adoption. This work presents a single-subject experiment that explores the costs of modifying specifications written in two different languages: a tabular notation, Software Cost Reduction (SCR), and a state-of-the-practice notation, Real-time Unified Modeling Language (UML). The study records the person-hours required to write each specification, the number of defects made during each specification effort, and the amount of time repairing these defects. Two different problems are specified—a Bidirectional Formatter (BDF), and a Bicycle Computer (BC)—to balance a learning effect from specifying the same problem twice with different specification languages. During the experiment, an updated feature for each problem is sent to the subject and each specification is modified to reflect the changes.

The results show that the cost to modify a specification are highly dependent on both the problem and the language used. There is no evidence that a tabular notation is easier to modify than a state-of-the-practice notation.

A side-effect of the experiment indicates there is a strong learning effect, independent of the language: in the BDF problem, the second time specifying the problem required more time, but resulted in a better-quality specification than the first time; in the BC problem, the second time specifying the problem required less time and resulted in the same quality specification as the first time.

This work demonstrates also that single-subject experiments can add important information to the growing body of empirical data about the use of formal requirements specifications in software development.

Acknowledgements

I would like to thank my supervisors, Dr. Joanne Atlee and Dr. Daniel Berry, for their help with this thesis. Jo gave me the inspiration that started this work, and has lent me her guidance and knowledge for the past two years. Dan has provided me with numerous ideas, and much-needed encouragement that kept me on the path. Sometimes he would lead me, and I would follow; other times, he provided that extra push to keep me going. To both of you, thank you for being so patient, and so supportive.

I would like to thank my readers Dr. Michael Godfrey and Dr. Richard Trefler, who took their time to read my thesis and attend my presentation, and for providing valuable input to improve the quality of this research.

Thanks to my friends, for giving me points of advice when I needed them, and distractions when I did not need them! Cheers to #beer.

Finally, and most importantly, I would like to thank Dad and Mom, who love me and nurture me.

Irwin Kwan, September 2005

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

Collecting and communicating the software requirements of a system is an important part of the software-development process. Requirements that are elicited are usually specified using natural language, and the resulting software requirements specification (SRS) is passed to developers for implementation. However, an SRS written in natural language is usually ambiguous, imprecise, and often incomplete. Developers who need to transform the requirements into code often discover conflicting requirements and missing details.

One method proposed to smooth the transformation from the SRS to the implementation is to use a formal specification language to write the SRS. A formal requirements specification is a document that describes the functional requirements of a software system precisely and unambiguously. The formal specification provides an unambiguous representation of the requirements and should assist a requirements engineer in identifying conflicts, resolving ambiguities, and discovering defects in a requirements description. A formal specification should emphasize functionality, not design, and it should be cheaper to write than code. Anecdotal evidence suggests that formal methods improve quality [21, 15, 16, 12, 8], but there is still ongoing debate in the software-development community about the cost-effectiveness of formal methods. Many argue that formal methods are difficult to adopt and that a formal specification is expensive to maintain.

If a requirements engineer writes a formal specification for a system, he will have to deal with the inevitable pain of modifying the specification to reflect changing requirements. The ripple effects of change among software artifacts are difficult to contain, and developers will now need to maintain an additional artifact. What are the costs of modifying a formal SRS when the requirements change? What can be done to reduce the costs of modifying a formal SRS?

1.1.1 The Wager: Is Maintaining a Tabular Specification Easier than Maintaining a Visual State-Machine Specification?

Daniel Berry claims that, no matter what method or technology is used, there is always a "painful" part of that language or technology [4]. For a formal method, the pain in software development is coping with modifying the formal specification.

The choice of formal notation that is used to specify a software system may affect how painful modifying a specification is. Parnas Tables are an easy-to-read state-based tabular notation that uses one table per variable to describe every conditions for an assignment. With appropriate tool support, a requirements engineer can use a dependency graph to explore the dependencies of each table. This notation, which localizes assignments into tables, may be better at isolating changes and containing the ripple effects of change than traditional imperative state-based languages. Atlee believes that this style of formal specification can help reduce the pain of doing modifications when requirements change [1]. Berry is not convinced [5].

The bet is placed on the question: "Do Parnas Tables reduce the pain involved in maintaining a formal specification, compared to what is commonly used for specification today?" To help answer this question, a pilot study is executed to compare the cost of modifying specifications written in two different languages: The pilot study sets the groundwork for a fully controlled experiment that may be executed in the future. The purpose of the pilot study is not to answer the question, but rather to prepare a fully controlled experiment that will answer the question.

The pain of software development is not a quantifiable measure, so we use the following two metrics:

1. **The amount of time required to modify a specification to reflect a requirements change.** The lower this number is, the better, as it suggests that a language is easier to use and requires fewer person-hours to specify a problem.

2. **The amount of time required to repair defects discovered during the specification modifications.** The lower this number is, the better, as it suggests that the language helps the requirements engineer make fewer mistakes.

Given these data, the goal of this thesis is to establish

1. that the time required to modify a specification of a system written using a tabular notation to reflect a requirements change is less than the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

2. that the time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change is less than the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

To test these hypotheses, we would ideally run a fully controlled experiment, but such an endeavour is expensive. We instead execute a single-subject experiment as a pilot study to explore possible experimental designs for a fully controlled experiment that may be run in the future. Most of the framework that applies to the pilot study also applies to a fully controlled experiment.

We choose two languages to compare in both a fully controlled-experiment and a pilot study: one language that is representative of Parnas tables, and one language that is representative of the state-of-the-practice—that is, commonly used in industry.

Each language chosen should

1. **Be easy to learn.** Software development is already hard. According to Craigen, et al., it is paramount that the process not be further bogged down by complex languages [12]. A formal specification language that is easier to learn will be more likely to be adopted in practice.

2. **Be easy to read.** A language should be easily understood by software developers, especially if the specification is to be reused in subsequent revisions of the software. Obviously, an unreadable specification will not even be used.

3. **Support scenario-based verification.** Scenario-based verification, or simulation, is checking a formal specification against test cases based on scenarios. These scenarios are derived from a natural-language requirements document. Scenario-based test cases can be used also as criteria to determine when a specification is completed. These scenarios can be used also to check that two different specifications of the same system cover identical functionality. Scenario-based verification can be considered a lightweight formal method [27] because it does not employ proofs or model checking.

4. **Have tool support.** In order to support execution and scenario-based verification, the specification languages should have an executable semantics, and shoud have a simulator to execute models.

Software Cost Reduction: A Tabular Specification Language

Software Cost Reduction (SCR) is a tabular notation that uses Parnas Tables. In SCR, each table contains every assignment to a variable. SCR is advocated by the Naval Research Laboratory (NRL) as a language that easy to learn and is effective for software-requirements specification [26]. The unique design of SCR tabular specifications and the use of dependency graphs should assist requirements engineers in also making modifications. SCR is supported by the SCRtool (formerly known as SCR*) [25], which provides computer-based specification support and compilation of the specification into a prototype. The SCRsimulator provides an executable framework for the prototype.

Real-time Unified Modeling Language: A Graphical State-based Language

A commonly used language for requirements and design is the Unified Modeling Language (UML). Real-time UML (RT-UML) is an extension to the UML to adapt it for the specification of real-time systems [37]. The extension is supported by the Rational Rose RealTime CASE tool, which allows real-time simulation of software systems. The extension includes support for capsules, which provide a decompositional structure, and protocols, which define communication signals. The functionality of a specification is described using Statecharts. UML is the de-facto specification language used in industry, and is therefore representative of the state-of-the-practice. The most common use of UML, by far, is for the informal modelling of requirements, rather than for formal, detailed specifications. RT-UML is more formal than UML because specifications written in it can be executed.

1.2 Related Work

There are, to the author's and his advisors' knowledge, no studies that explore the costs of modifying an SRS under requirements change. The available case studies about the cost of formal methods tend to study the cost of writing a specification, not the cost of maintaining it. Our work also differs from existing research concerning the readability of languages, such as that by Zimmerman [45], because our work examines the cost of writing and modifying a specification using a language, as opposed to determining how easily a specification written in a language is understood.

Much of the research about the costs and the benefits of formal methods is based on experience reports [20, 33, 12, 14, 15, 27], although an increasing amount of it is based on data gathered during system specifications [16, 35, 30]. There have been also a few controlled experiments exploring the benefits of formal specification [29, 40].

Empirical research on formal specification methods so far has primarily investigated a method's ability to help a requirements engineer discover defects in a natural-language requirements document. We call this natural-language document, intended to be read by users, a **user requirements specification** (URS). For instance, Wing concluded that use of a formal language to write the specification resulted in a clearer, more complete specification of a library system [44]. A controlled experiment executed by Kamsties, et al. also explores how effectively formal specification languages help a requirements engineer discover defects in a URS [29]. The experiment used the languages SCR, UML, ROOM, and Statecharts, among others, but it did not compare the effectiveness of each language with each other. The study concludes that the process of writing a formal specification effectively detects incompleteness and conflicts in a URS. A study by King, et al. shows that formal proofs carried out at the specification level are extremely efficient. The study reports that doing a Z proof detects more faults per day in an SRS than testing [30]. There is strong evidence to show that rewriting a URS in a formal specification language

is an extremely effective way to discover defects in the URS.

Although the studies show that formal methods are effective at helping requirements engineers detect defects in a URS, it is unclear if formal methods are efficient enough to be used in most software projects. A technique is efficient if it helps requirements engineers to detect defects at a reasonable cost. It is commonly believed that formal methods are too expensive to use on non-safety-critical systems. To counter this belief, formal-methods proponents argue that using formal methods saves costs: either formal methods detect defects early and therefore save costs during downstream development, especially after deployment [21, 44, 14, 16], or formal methods somehow allow a project to stay on schedule and on budget because they force early decisions about requirements details and conflicts [15, 12, 9]. An experience with the CoRE method [15] reported that the C-130J SRS, at the time of writing, was on-time and on-budget. Craigen, in his survey, found a number of commercial projects that were said to be on schedule [12]. Hall claimed that inflated development costs are a myth [21], and Bowen claimed that delays in development are also a myth [8]. Despite these statements, many proponents of formal methods declared the Darlington Nuclear Power Plant Safety Shutdown system as a successful application of formal methods, even though it went far over budget [19]. There are undoubtedly success stories attributed to the application of formal methods, but it should be shown that cost savings and time savings are directly caused by the use of formal methods, and not to other circumstances, such as personnel; after all, people who can use formal methods are generally skilled at any task involving abstract thinking [3]. More exploration of the costs of formal methods are needed to confirm their usefulness on various types of systems.

To reduce the cost of using formal methods, while retaining some of the benefits, many practitioners have used lightweight formal methods. Breen used a lightweight specification method during an industry project [9], as did Easterbrook [14]. The Easterbrook study is particularly interesting because lightweight formal methods were adopted specifically to cope with requirements that were still volatile, and were used to feed back improvements to the URSs early in the project. Easterbrook's experience is positive: He concludes that the benefits of discovering defects in the URS early in development outweighs the required time to use the techniques. However, no cost-benefit analysis was performed in this study.

Instead of using formal methods, is it possible to improve a system's quality by devoting more time to requirements specification using informal techniques? According to Berry and Wing [7], formal methods may work because they make a person specify the system more than once. Daugulis, who did not use formal specification, reports on his experience writing a specification multiple times [13]. He and his team were assigned to specify the SRS for a power utility, but the project continuously ran into financial difficulty, leading to extended deadlines. Daugulis and his team used the unexpected time to generate three revisions of the SRS. Daugulis, in retrospect, states that had the project been allowed to continue with the first version of the SRS that he and his team wrote,

the project would have been a complete disaster. It has also been shown that projects that spend more time in the planning and requirements stages tend to suffer fewer cost overruns [17]. Perhaps, instead of employing formal methods, more care, time, and rigour is required upfront when writing an informal SRS.

Because there are so many claims that using formal methods can reduce the time, money, or effort required for software development, the software-engineering community needs to find a way to assess these claims. There is a continuing need for empirical validation [2, 42, 34, 39, 38] to advance the state of software engineering, and, in particular, a need for comparative studies in requirements engineering [18]. A recent survey of software-engineering journals for empirical studies reports that the majority of published articles in computer science and software engineering provide "little or no empirical validation and the proportion of controlled experiments is particularly low" [39]. Not all experiments need to be fully controlled, multiple-subject experiments; Harrison [22] advocates researchers to consider single-subject studies as one way to provide empirical support for software-engineering techniques.

Prior research has established that writing a formal specification is an effective method to uncover defects in a requirements document. There is also anecdotal evidence that using formal methods early in software development can result in strong benefits due to the early detection of defects in the requirements document. There is little question about the benefits of formal specification—it detects defects, and it helps specifiers create higher quality software.

1.2.1 Contributions of This Work

What separates this work from other related work is that this pilot study does a quantitative comparison between SCR, a tabular notation, and RT-UML, a commonly-used visual notation. The pilot study compares the times a specifier needs (1) to modify specifications, and (2) to repair defects in the modified specifications written in the two notations when the requirements change. The pilot study is intended to provide feedback on an experimental design that may be used for a fully controlled experiment executed in the future.

1.3 Organization of this Document

The rest of the document is laid out as follows. Chapter 2 describes the procedure used in this pilot study, including the evaluation criteria, the experimental design, the specification languages, and the software problems provided to the subject. Chapter 3 presents the sizes of the specifications and analyzes the time data and the defect data acquired during the specification process. Chapter 4 presents recommendations for a fully controlled experiment, and Chapter 5 concludes the study.

Chapter 2

Experimental Design and Procedure

This pilot study is intended to compare the cost of modifying formal specifications, with respect to two different languages: the Software Cost Reduction notation, and the Real-time Unified Modeling Language notation. The pilot study is a single-subject experiment, in which only one subject performs the tasks in the study. The intention of the pilot study is to debug the design of a fully controlled experiment that tests the experimental hypothesis. The pilot study helps also to create materials, to identify risks in a fully controlled experiment, and to provide some preliminary results.

This chapter presents one possible design of a fully controlled experiment, and then presents the details of the pilot study that is based off of the fully controlled experiment. The chapter describes the two languages used, the two candidate software problems a subject must specify, and the tracking techniques used to monitor the required time and detected defects.

2.1 Choosing Evaluation Criteria

To objectively compare the costs and quality of modifying formal specifications written in different languages, we must define metrics to use as the basis for the comparison. The Personal Software Process (PSP) [28] was used as a foundation for determining what data to measure. We initially intended to compare the specifications on the following criteria: the time required to modify a specification, the quality of a specification after modifications, and the size of a specification after modifications. Whereas time can be measured directly, quality and size are not as easily measured.

Problems with Measuring Quality The original idea was to measure the quality by using the **defect count** (DC), which is the number of defects made by the subject during

the specification task. It was believed that higher the DC was, then the lower the quality was. The PSP recommends also tracking the **defect-repair time** (DRT), which is the amount of time required for the requirements engineer to repair every defect discovered during the specification task. However, after the pilot study started, the accuracy of these metrics as an indicator of specification quality was called into question.

It is difficult to make statements about a specification's quality based on a DC value. Is a high DC value bad because it suggests that the language did not help prevent the developer from making mistakes, or is a high DC value good because it suggests that the language helped the user discover more defects that would have otherwise gone unnoticed? Without a list of known defects in the hands of the experimenters, the number of defects is not an indicator of quality. In this pilot study, a list of defects was not available. One reason for doing this pilot study is to create a list of defects in the user requirements specification for the experimenters.

DCs and DRTs from different specifications are difficult to compare. If two specifications of different systems are compared, one cannot say that the specification with a high DC value is worse than the specification with a low DC. Perhaps the specification with the high DC value is a large industrial system with hundreds of modules, whereas the specification with the low DC value is small system for a school project; it would be logical to presume that a large system will have a higher DC than a small system. A possible solution is to calculate *normalized metrics* that can be compared with other specifications. One normalized metric is the percentage of total specification time spent repairing defects, also called the **percentage of time repairing defects**, and another is the **average defect-repair time**.

DCs and DRTs are not precisely a measure of quality; they are more a measure of the requirements engineer's efficiency. If the percentage of specification time spent on repairing defects for a specification written in language A is less than the DRT for the same specification written in language B, then an argument can be made that language A helped the requirements engineer work more efficiently because the language reduces the amount of time that a requirements engineer spends fixing bugs. An increased proportion of the requirements engineer's time is *value-added work* used to add new functionality to the system. Although measuring the DRT is important and relevant, DRT does not report what the quality of the system is.

Another measurements of quality is customer satisfaction, but measuring the level of customer satisfaction can be problematic, due to the lack of criteria for evaluating a formal specification. The customer must evaluate also two specifications of the same system, and the results of the evaluation may be biased.

Problems with Measuring the Size of a Specification A useful metric would be the size of a specification, designed such that it can be compared to other specifications written in different languages. This metric can be used to analyze different specifications of the same system to see if the same set of requirements makes a noticeable difference

in the size of a specification.

The difference in size between two specifications of the same system, one written in SCR, and one written in RT-UML, is difficult to determine due to a lack of tools to analyze models written in the SCR and RT-UML notations. It is easy to count every possible assignment that can be made to an SCR variable, but difficult to count every possible assignment made to a RT-UML variable. Even if you are comparing two specifications written in the same language, counting the differences may not be a good indicator of a difference in size. For example, two specifications with an identical number of variables may be different from each other because one model may have more complex conditions than the other. In addition, counting elements can result in a lot of numbers that are overwhelming and do not offer useful information to a reader.

Because both specifications are executable, a comparison of specification sizes should be possible, but determining an appropriate metric to compare the size of changes between two specifications written in different languages with each other is beyond the scope of this thesis. However, being able to do a size comparison between an SCR specification and an RT-UML specification is worth exploring. With a good size metric, it may be possible to calculate the cost of doing a modification per unit size when using a certain language. The data acquired for cost of modification per unit size can be compared with data acquired for other languages.

Selected Evaluation Criteria

In light of the above issues, the following metrics were selected in this pilot study to compare the specifications written using two different languages.

- **Time:** The amount of time required to write a specification to reflect requirements and later to modify the specification to reflect a requirements change.
- **Efficiency:** The proportion of time required to write a specification that is used to repair defects discovered during the specification process.

If these metrics are useful in the pilot study, the fully controlled experiment may also use these metrics.

2.2 Hypothesis

The hypothesis is based on the wager that is presented in Chapter 1. The hypothesis of both the fully controlled experiment and the pilot study are as follows.

$H_\alpha 1$**:** The time required to modify a specification of a system written using a tabular notation to reflect a requirements change is less than the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_\alpha 2$: The time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change is less than the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

Given the above alternative hypothesis, the pilot study tests the following null hypotheses.

$H_0 1$: There is no difference between the time required to modify a specification of a system written using a tabular notation to reflect a requirements change and the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_0 2$: There is no difference between the the time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change and the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

A fully controlled experiment would test these hypotheses using multiple subjects that write and modify a specification for only one problem, as opposed to two in the pilot study.

2.3 Design of a Fully Controlled Experiment

To test these hypotheses with a fully controlled experiment, we would ideally select multiple subjects to participate in the experiment. Half of the subjects would be assigned to the control group, and would use RT-UML as a control language. The other half of the subjects would be assigned to the experimental group, and would use SCR as the experimental language. Every subject would get one copy of a *user requirements specification* (URS), which is the problem written using natural language and intended to be read by non-technical stakeholders. Each subject would specify the requirements from the URS using the language assigned to him. Once the specification is completed or a deadline is reached, every subject would submit the initial specification to the experimenters, and keep a copy for himself or herself. Each subject would then receive a URS describing a requirements change for the same system he received earlier, and would be ordered to modify the specification they wrote to reflect the changes.

The factors in the fully controlled experiment are

- the language factor, which is our treatment variable. There are two treatments, SCR and RT-UML.
- the subject factor, which is the individual ability of the subjects.

During the fully controlled experiment, each subject would track his time and defects in a log. These logs would be analyzed to determine the amount of time each subject spent doing specification activities. Also, the specifications generated by the subjects would have to be analyzed for quality, since it is unlikely that the specifications would be equivalent at the end of the study.

2.4 Pilot Study: A Single-Subject Experiment

Although a fully controlled experiment such as the one described above is reasonably straightforward, we do a pilot study first. Because a fully controlled experiment is expensive, we want to execute a "test run" that assesses the factors that the experimenters may not have considered, and determine which effects are worth studying further. Finally, the pilot study can provide some preliminary results that suggest what the results of a fully controlled experiment might look like.

This study reports the results of performing a single-subject experiment as a pilot study. A single-subject experiment requires a treatment and a measure of performance, and tests a hypothesis. Because there is only one subject, no statistical analysis can be performed, but rather, data are presented in such a way that differences can be observed.

A single-subject experiment is not a case study [22] because a case study is uncontrolled and establishes its hypothesis after the case study is completed. In this study, the hypothesis is provided up front, and the treatments are administered by an experimenter.

2.4.1 Experimental Design

The author of the thesis is the single subject in the pilot study, and is assigned the task of writing each specification. Prior to the study, two software problems unrelated to the pilot study were completed using both SCR and RT-UML to familiarize the subject with the languages.

Two customers, who are also the author's co-supervisors, each wrote the requirements for a software system. A requirements description is called a *software problem*, or simply, a *problem*. Each problem was provided in the form of a URS. The first problem is the Bidirectional Formatter (BDF), provided by the customer Daniel Berry. The second problem is the Bicycle Computer (BC), provided by the customer Joanne Atlee.

As a part of the pilot study, for each of the two problems, the subject writes a specification using each treatment language. The resulting artifact after this stage is called the initial specification (IS). After the IS for a problem is completed, the problem's customer gives the subject a URS outlining changes to the same problem, and the subject modifies the IS to reflect these changes. The resulting artifact after every change is made is called the specification modification (SM). In this thesis, the term "task" refers to either writing an IS, or making changes for the SM. During each task for each problem,

the amount of time used to specify the requirements, as well as the time spent repairing defects, is recorded using a time-tracking tool.

Defects were not seeded into either the initial URS or the URS changes. Each customer unknowingly injected his or her defects into his or her respective URS when writing it. No list of known defects in the URSs were generated before the pilot study.

2.4.2 Independent Variables

The pilot study manipulates the following independent variables:

1. The specification language, which can be either SCR or RT-UML, used to specify the problem as expressed in the URSs.
2. The software problem that the subject must specify, which can be either the BDF problem or the BC problem.

The specification language is the treatment variable in this pilot study.

2.4.3 Dependent Variables

There are three dependent variables measured in the pilot study.

1. The time, in minutes, required to write an IS, and later to do an SM.
2. The number of defects discovered while writing an IS, and later while doing an SM.
3. The total time, in minutes, required to repair defects discovered while writing an IS, and later while doing an SM.

Measurements are recorded for the IS and separately for the SM.

From these values, the following normalized data can be compared.

- The **percentage of time spent repairing defects**, which is the defect-repair time divided by the total time required to write a specification.
- The **average defect-repair time**, which is the defect-repair time divided by the number of defects.

2.4.4 Effect of Learning in the Experiment

In a single-subject experiment, if the same software problem is specified twice, once using the first treatment and once using the second treatment, then the second treatment has a distinct advantage because the subject is already familiar with the problem. The subject's learning of one of the problems has affected the outcome of the experiment. This effect

is called an **order effect** in experimental design, but will be referred to as the **learning effect** in this thesis.

In a single-subject experiment, the learning effect cannot be eliminated or factored out. The only recourse is to use an A-B, B-A design in an attempt to counterbalance the learning effect. Thus, the pilot study uses two problems, and reverses the application of the treatments for each problem, so for one problem, the subject writes the IS using SCR first, and then writes the IS using RT-UML. For the other problem, the subject writes the IS using RT-UML first, and then writes the IS using SCR. The same principle applies when doing the SMs. To further reduce the learning effect, a wash-out period of approximately a week is introduced between the first specification of the problem and the second specification of the same problem. To further minimize the learning effect, materials that were generated during the first specification of the problem could not be referred to during the second specification of the same problem.

One problem with using the swap design is that the treatment–problem interaction effect is confounded, or mixed, with the learning effect, in such a way that we cannot identify this in the analysis. A **treatment–problem interaction effect** is an effect from both the treatment and the problem have a strong effect on the response. In other words, the choice of problem has an effect on how well the treatment works, and vice-versa. The design makes the reasonable assumption that the learning effect of the specification language is negligible, since the subject practised using both RT-UML and SCR before starting the pilot study. However, we cannot assume that the learning effect when doing A-B is the same as the learning effect when doing B-A is reasonable, and therefore, in a "swap' design, there is no way to distinguish the proportion of the effect due to learning, and the proportion of the effect due to the treatment–problem interaction. One method to alleviate the confounding is to perform a replication of the pilot study in full by doing a total of sixteen runs, but the cost of doing so was too expensive. In a fully controlled experiment, the issue does not exist because there is no learning effect.

2.4.5 Run Order

Because there are two variables present, the pilot study uses an unreplicated 2^2 factorial design to determine the effects of the treatment and the problem. By running a factorial design, it is possible to determine if the treatment has an effect on the response, if the problem has an effect on the response, or if an interaction of the treatment and the problem have an effect on the response. See Christensen for a basic introduction to experimental factorial designs [11].

In an attempt to observe a cause-effect relationship of the learning effect to the pilot study, we assign a set run order of the specification tasks. Typically, runs should be randomized, but assigning a set run order may help provide observations of the learning effect. All observations based on the run order are anecdotal and would need to be verified using a controlled experiment.

The runs for the pilot study are shown in Table 2.1. The first run was determined

Run	Problem	Treatment	Task
1	BDF	SCR	Initial
2	BDF	RT-UML	Initial
3	BC	RT-UML	Initial
4	BC	SCR	Initial
5	BDF	RT-UML	Changed
6	BDF	SCR	Changed
7	BC	SCR	Changed
8	BC	RT-UML	Changed

Table 2.1: Run Order and Factors in Single-Subject Experiment

incidentally by the customers; the first customer to send his or her problem to the subject was assigned as the first run; in this situation, Daniel Berry's BDF problem was the first run. The subject randomly selected SCR as the specification language. Based on these circumstances, the run order used in the pilot study is shown in Table 2.1.

2.4.6 Determining When a Specification is Complete

The primary reason for choosing executable notations for the pilot study is to allow every specification to run against test cases to get a sense of when the specification was complete. To test the functionality of two specifications, we verify the specification against a set of test cases derived from scenarios uncovered in the URS. During each run of the experiment, the subject derives a set of test cases based on the reading of the URS. These test cases are specific to the treatment. For every treatment–problem combination, the subject ensures that, to the best of his ability, the test cases cover every scenario in the system. Each specification is verified against the test cases, and when the specification passes every one, it is considered complete.

After completing the specification tasks in the pilot study, the specifications written in different languages for the same problem will not be identical because of the differences between test cases written for two languages, making a direct comparison difficult. Doing the test cases before writing the IS or doing the SM was an option to ensure equivalent specifications was possible, but was considered unrealistic, as most requirements engineers do not derive test cases before modelling. In addition, defect lists were not available, so each test case had to be derived during each specification task. To reduce the learning effect, the subject doing the second treatment of the same problem is not allowed to allowed to refer to the test cases created when doing the first treatment. However, a subject working on the SM for a problem using a treatment is permitted to view the test cases from the IS for the same treatment. To check that two specifications of the same problem were the same, we compared each scenario in the test cases from one

specification to each scenario in the test cases from another, ensuring that they covered the same requirements.

2.5 Tracking the Dependent Variables

To gather the time and defect data required in this experiment, the subject uses a process based on the Personal Software Process (PSP) by Watts Humphrey [28]. The processes of coding a software system and of writing a software specification are similar, which allows the tracking process from PSP to be applied to this experiment. The PSP techniques for improving software-development techniques, such as planning and estimation, were not adopted to avoid affecting the outcome of the experiment.

During the writing or modifying the specification, the subject tracks data for each phase of each specification task as described in Section 2.8.1. The amount of time spent in each phase, the number of defects spent in each phase, and the amount of time repairing each defect is recorded in a **log**. Tracking data for the IS task is kept in a log separate from that for the SM task.

Defect Tracking

To keep track of the DCs and the DRTs, defect tracking techniques are borrowed from the PSP. In this thesis, a **defect** is defined as a problem that may be either a *fault* or a *failure*, as by the IEEE Software Engineering Glossary definition [41]. It is an appropriate term to use in the case when a problem is identified in a software system, but the cause of the problem is unknown.

A defect is logged when a problem is identified within one of the artifacts created during the specification process. When a defect is discovered, the subject takes steps to repair it. Once the defect is verified as repaired, the completion time is recorded in the log along with the following information:

- The phase during which the defect was injected into the system. Phases are defined in Section 2.8.1.
- The classification of the defect that was just repaired. A defect classification describes the type of defect, and is listed in Table D.1.
- The reason that the defect was injected, that is, what the subject did that caused the defect to be injected into the system. The possible reasons are listed in Table D.2.
- The component which was modified in order to repair the defect. If multiple components were modified, then the subject lists the component that required the most changes.

In as many cases as possible, the defect is fixed as soon as it is discovered. If the defect cannot be repaired, even after time has been spent exploring possible repairs, then it is closed with a reason indicating that the defect will not be fixed. In the case of a requirements defects that cannot be resolved by the subject alone, a defect is opened and closed immediately, and it is excluded from the defect analysis. When a corrections is received from the customer, a new defect is opened and the appropriate engineering changes are made to the specification to measure the effect that the requirements correction had on the specification.

2.5.1 Tracking Tools

Two tools were used in the tracking and the analysis of the time logs.

The first tool used was the the *pplog* emacs mode, developed by Lutz Prechelt [31], which records phase durations, interrupt durations, and defects. To record a log event, a keystroke is pressed that brings up the current date and time so the log entry can be made.

The *evalpplog* script, also developed by Lutz Prechelt, analyzes the logs created by the pplog tool. It lists the defects, calculates the total time required for specification, and reports a number of other information of interest to the PSP. The data that was most useful from the analysis were the total time numbers and the defect lists.

2.6 The Software Problems

The software problems were provided by two customers in this experiment. The BDF problem was provided by Daniel Berry. The BC problem was provided by Joanne Atlee. An overview of both problems and their requirements changes, are provided in this section. The full text descriptions of both problems, along with the requirements change, is available in Appendix A.

2.6.1 Bidirectional Formatter

The objective of the BDF system is to transform a text input file that contains markup commands to an output file that formats the text according to the markup. SGML-like markup can specify the text layout so that a continuous region, or a **chunk**, prints characters in a left to right order, or in a right to left order. The markup also sets the document type, which indicates whether chunks in a paragraph should be displayed from left-to-right, or from right-to-left. One markup tag can center text on the line. There are special markup tags for formatting numbers, and tags that insert different amounts of whitespace.

The system is useful for writing text using Hebrew, Arabic, Persian, and Urdu languages, which are written from right to left, mixed with text using Indo-European languages, which are written from left to right.

The BDF URS is 14 pages long. Six pages provide background information about the problem, three pages describe the software that solves the problem, and five pages provide examples of input and output.

BDF Requirements Change

The change in the BDF system adds new markup that specifies the justification of lines on a fixed-width output device. The output lines can be set to ragged justification, stretch justification, or spread justification. Ragged justification is the same as no justification. Stretch justification stretches the last stretchable character to fill the remaining space on the line. Spread justification adds spaces between the existing words to fill the remaining space on the line.

2.6.2 Bicycle Computer

The BC system provides relevant information to a user about a bicycle, such as its current speed. The computer's interface has an LCD that can display a number with two digits, and a number with five digits. The user can press a mode button, and a start/stop button. The computer monitors a bicycle's movement by reading a sensor attached to the front wheel, and continually displays the current speed on the LCD. The computer can display also the distance travelled, the elapsed time, the maximum speed achieved, and the average speed, but only one of these values at a time. Pressing the mode button cycles through these values. Monitoring can be suspended and resumed by pressing the start/stop button on the computer, and the computer can be reset to its default values by pressing both the mode button and the start/stop button at the same time. If the computer is idle for a specified amount of time, the display turns itself off.

The BC URS is one and a half pages long, and provides a small diagram with an example of the user interface.

BC Requirements Change

The change in the BC modifies the way that the bicycle's movement is monitored. Instead of getting information from a sensor on the front wheel, the computer periodically receives the bicycle's latitude, longitude, and altitude from a GPS in the computer and calculates the values that are shown on the LCD. The GPS is considered to be a hardware device that does not have to be specified.

2.7 The Specification Languages

2.7.1 Software Cost Reduction

Software Cost Reduction, or SCR, was first used by the Naval Research Laboratory (NRL) in the Requirements for the A-7E Flight Control Software [32] to show that concepts

such as information hiding and modularization are applicable to industrial systems. SCR requires knowledge of propositional logic, which most programmers are already familiar with. The notation models the software system as a set of function tables [23, 24].

We use the SCR two-variable model in this pilot study. **Monitored variables** are read from the environment, and **controlled variables** are sent to the environment. The relation *REQ* represents the system's requirements as a relationship between the monitored and controlled variables. Each controlled variable has one function table that contains every possible assignment to it. Each cell of the table expresses a condition for an assignment.

The following terminology is associated with an SCR specification.

Mode Class A **mode class** is a set of **modes of operation**, or simply **modes**, that transit to one another when an event is triggered. A specification can have multiple mode classes. Like controlled variables, mode classes are defined using function tables.

Terms A **term** is an intermediate variable defined in terms of monitored variables and mode classes. A term serves as a substitution macro. Like controlled variables and mode classes, terms are defined using function tables.

State variables A **state variable** is a monitored variable, a controlled variable, a mode class, or a term. A **state** is a snapshot of every state variable's current value.

Events An **event** occurs when a state variable changes value. An event is an expression over two states, the current state, in which the current variable values hold, and the next state, in which the updated variable values hold. An event is preceded by the `@` symbol. `@T(c)` means that the value of `c`c is false in the current state, and true in the next state; i.e. `c` becomes true in the next state. `@F(c)` means that the value `c` is true in the current state, and false in the next state. `@C(c)` means that the value of `c` changes in the next state. Thus, `@T(c)` can be defined as

$$\texttt{@T(c)} \stackrel{def}{=} \neg\ \texttt{c} \wedge \texttt{c'}$$

where unprimed variables represent their values in the current state, and primed variables represent their value in the next state.

Conditions A **condition** is a predicate defined on one or more state variables. A condition is evaluted in the current state. A **conditioned event** is an event whose occurrence is conditional on the value of other variables. A conditioned event has a `WHEN` condition, `WHEN d`, that must evaluate to true in the current state for the conditioned event to be deemed to have occured. Thus, a conditioned event such as `@T(c) WHEN d` is defined as

$$\texttt{@T(c) WHEN d} \stackrel{def}{=} \neg\ \texttt{c} \wedge \texttt{c'} \wedge \texttt{d}$$

where c and d are a boolean expressions. Note that a conditioned event evalutes the event expression, c, in both the current and next states, and evaluates the WHEN condition, d, in only the current state.

Function Tables Each controlled variable, term, and mode class is defined by a function table. There are two types of function tables. A **condition table** describes a set of disjoint conditional assignments to the table's output variable. The table's variable is assigned a value when one of the table entry's conditions, when evaluated in the current state, is true. An **event table** describes a set of disjoint conditional assignments to the table's output variable, in which assignment conditions are conditional events that are evaluated in the current state and the next state. The table's output variable is assigned a value when one of the table's conditioned events evaluates to true. A mode transition table is an event table whose output variable is a mode class.

Before an SCR specification can be executed, the following properties must be true. The **coverage property** must hold on each condition table. The coverage property states that some conditional assignment must apply in any current state of the model. A **circular dependency** occurs when the next values of two variables each depend on the other's next value, making it impossible to impose an order in which they are assigned. Finally, the **disjointness property** must hold on each function table. The disjointness property specifies that at most one conditional assignment applies in any current state.

Example SCR Table

Table 2.2 is an example of an SCR event table for a controlled variable called c_entrancegate. c_entrancegate controls an entrance gate that allows access into a parking garage. The gate opens only if there are available parking spots inside the garage. The table states that, if the Parking Garage is in mode free, meaning that parking spaces are available, then pressing the button, modelled by the event @T(m_entrancebutton = up), or inserting a card into a card reader, modelled by the event @T(m_entrancecardreader), will open the gate. The gate closes when the entrance gate timer reaches zero. Row occupied says that if the garage is occupied, then only a card holder can enter the garage by inserting her card into the card reader.

Tool Support for SCR

The CASE tool for SCR is called the SCRtool, and it is developed by the NRL. The tool allows the user to enter variables, modes, and function tables. The tool checks syntax on the fly and reports circular dependencies. The simulation program for SCR is called the SCRsimulator, which can execute an SCR specification. The user can manipulate

Modes for Garage	Events	
free	@T(m_entrancebutton = up) OR @T(m_entrancecardreader)	@T(c_entrancegate_timer = 0)
occupied	@T(m_entrancecardreader)	@T(m_entrancegate_timer = 0)
c_entrancegate' =	opened	closed

Table 2.2: Event Table for Entrance Gate

each monitored variable and see the results. The SCRsimulator provides a graphical user-interface builder to create a graphical user interface for a specification.

2.7.2 The Real-Time Unified Modeling Language

The Real-time Unified Modeling Language (RT-UML) is an extension to the UML to support real-time execution and simulation [37]. RT-UML was selected as the representative state-of-the-practice notation in the experiment because of its similarity to UML. Although we did not assess the degree to which RT-UML is in fact used in industry, UML is currently one of the most popular systematic system-description languages supported by tools.

RT-UML is an object-oriented notation that can be used for modeling behaviour at the requirements level and the design level. Traditionally, UML uses use cases to express requirements, and classes to model the domain. In RT-UML, a capsule is used to model each domain entity, and a connector is used to model each communication path between these entities.

Capsules A **capsule** is a structural component that communicates with other capsules and the environment. A capsule has four parts: 1) a structure diagram that displays its ports and other capsules, 2) a state machine that describes its behaviour, 3) a list of attributes, and 4) a list of operations, or methods.

A capsule's structure diagram displays each port inside the capsule, and, if applicable, other capsules. A capsule might contain other capsules as internal, black-box components. The state machine models dynamic behaviour using the Statecharts notation.

Ports A **port** is an interface to a capsule's environment. A port can send and receive signals to and from the capsule's environment.

A **protocol** is a logical set that contains one or more signals that transfer information to other capsules or the environment. A signal can be sent to and from the environment, or internally by the capsule's state machine. Each port is associated with a protocol, and the port can only send and receive signals in that protocol.

Each port is either an end port, or a relay port. A **relay port** is connected to another port using a connector, and relays every signal the relay port receives to the other port.

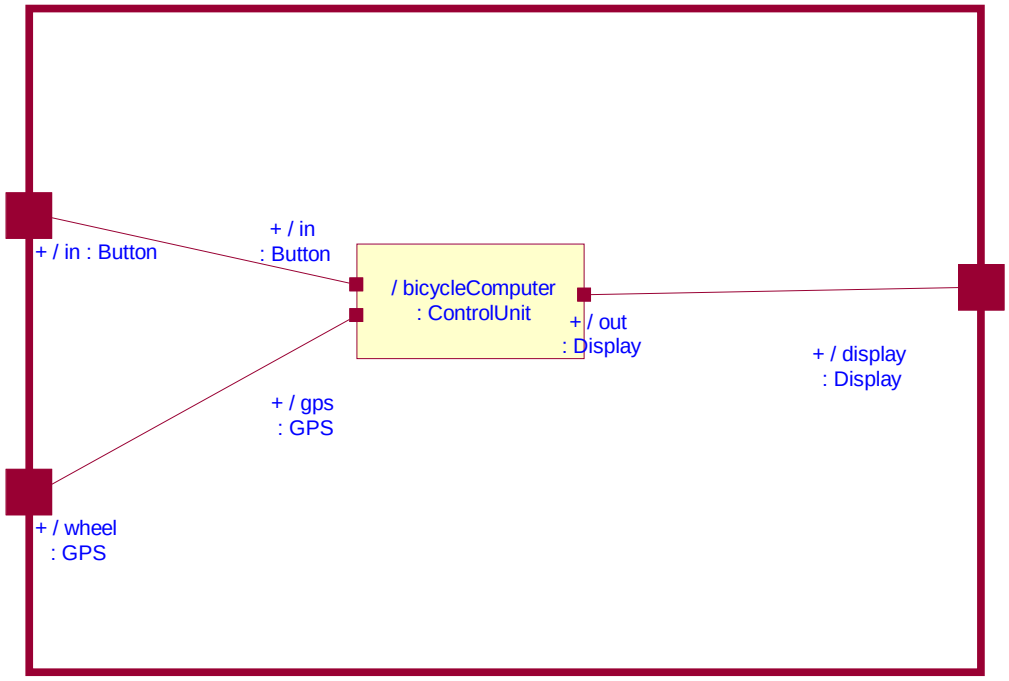


Figure 2.1: BicycleComputerSystem Capsule: Structure Diagram

An **end port** is attached to a capsule and every signal the end port receives is processed by the capsule's state machine. Transitions in a statechart are triggered by incoming signals to the capsule's ports. The statechart in a capsule can also send a signal through an end port to the capsule's environment.

Connectors A **connector** provides a communication path between two ports. A connector carries signals from one port to the other.

State Machines Each capsule has a state machine in the Statecharts notation. The Statechart is defined inside a capsule. An action may be executed on a transition, upon entering a state, or upon exiting a state. An action contains C++ or Java code that may include the sending of a signal, the assignment of an attribute, or the calling of a class's operation. Code in a transition usually contains two to five lines of code.

Example of Real-time UML Diagrams

An example of a Bicycle Computer System written in RT-UML is shown in Figure 2.1. The BicycleComputerSystem capsule is the main capsule that interacts with the environ-

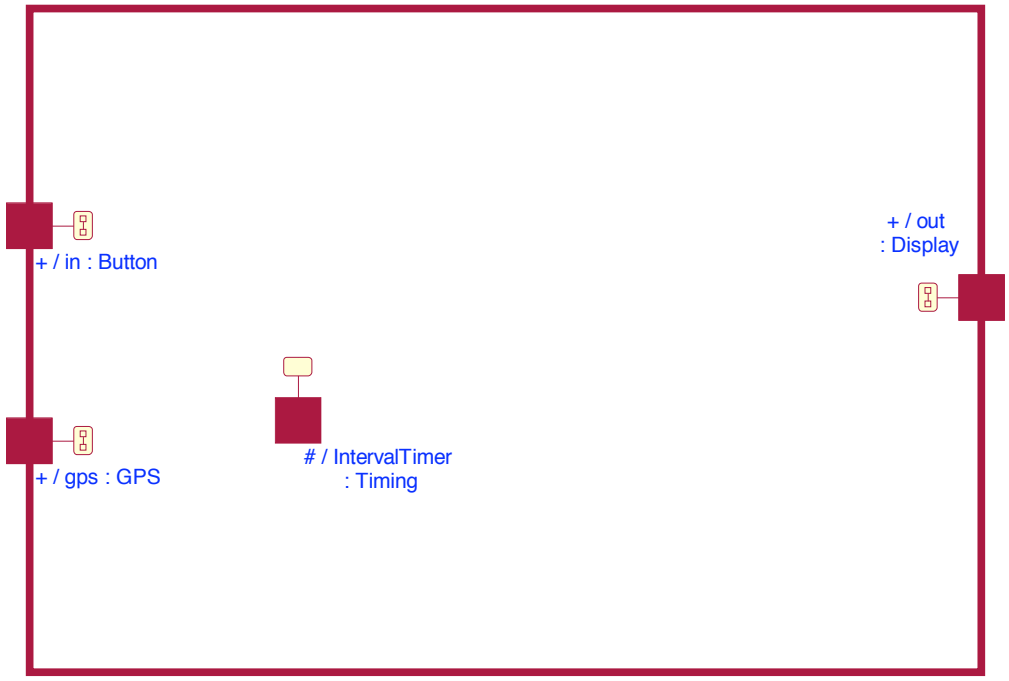


Figure 2.2: ControlUnit Capsule: Structure Diagram

ment. There is one capsule inside, a ControlUnit, but the BicycleComputerSystem allows the addition of other capsules if necessary.

The ControlUnit capsule is where all of the system's behaviour is modelled. Every port in Figure 2.2 is an end port. One port, a timer, is inside the capsule.

Figure 2.3 is the Statechart of the ControlUnit capsule. From the Active state, to trigger the transition Stop, the Statechart must receive the StartStopUp signal from the in port. The Active state, the Suspended state, and the Adjust_Circumference state each contain an icon with two bubbles, which means that each of these states is a superstate that represents a lower-level statechart.

Tool Support for Rational Rose RealTime

Rational Rose RealTime, now known as Rational Rose Technical Developer, is the CASE tool used in this experiment to model and simulate a RT-UML specification. The tool lets the user implement a model with either a C++ or a Java framework, so that the user can choose to write actions in either C++ code or Java code. With the tool, the user can create capsules, protocols, and classes, and display their relationships in a class diagram.

The RT-UML model can be executed and simulated in real-time. The user can inject

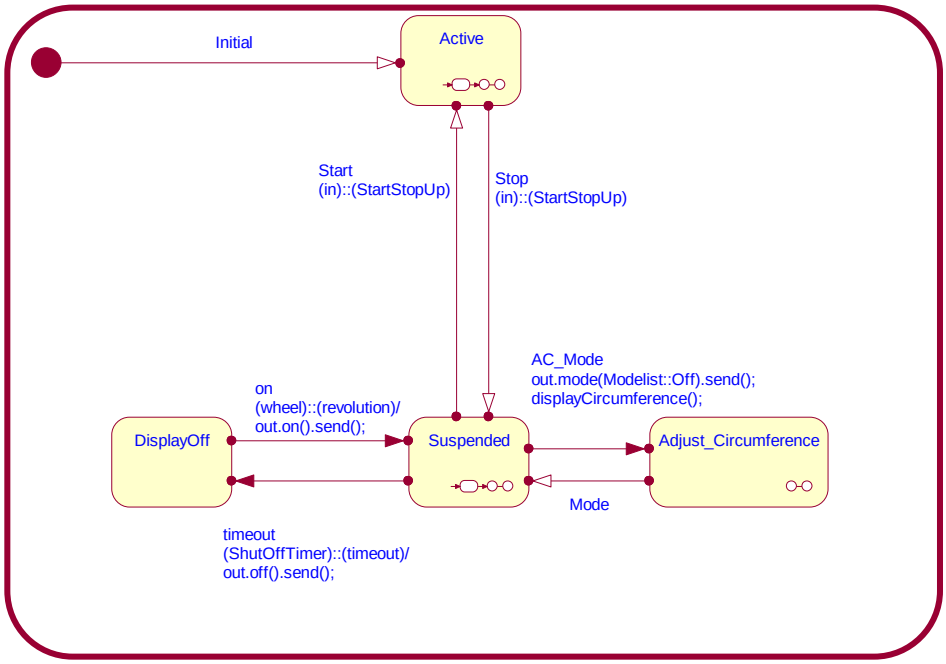


Figure 2.3: ControlUnit Capsule: Statechart Diagram

signals into the model with an argument; the model processes the signal and the argument, and responds by returning an output signal, or by assigning internal variables in the model.

The combination of visual notations and code allow the CASE tool to be used for not only requirements specification, but also design and implementation.

2.8 Processes Used For Specifying the Software Problems

The software-development process used can influence the outcome of the experiment. To reduce the influence, we standardize the process used to write each specification. We break down the specification process into **phases** that apply for each specification language and each problem. Within each phase, the specification languages work differently from each other, but the overall purpose of the phase is the same. This section describes the details of the process and the reasoning techniques used by the subject when writing specifications in each language.

2.8.1 Phases

The Reading Phase. The subject reads and understands the URS. Doing this phase is independent of the specification language used, so the time in the reading phase does not count toward the time to write or to modify a specification in the time analysis.

The Modelling Phase. Using a specification language, the subject creates an initial model that formally expresses the URS. This phase involves defining each domain element, defining each input and output variable, and defining behaviour. The subject uses a pencil and paper to sketch ideas for modelling the requirements.

The Specification Phase. The subject uses the CASE tool to write the specification based on the models developed in the modelling phase. The subject writes as complete of a specification as possible before moving on to compiling and testing the specification.

The Compiling phase. The subject compiles the specification to create an executable version of the model. This phase is usually very short.

The Test Design phase. The subject writes test cases during this phase. The test format is scenario based, so in this phase, the URS is re-read repeatedly to ensure every scenario is covered by some test case. Boundary testing and equivalence classes are among the testing techniques used. Different versions of the same scenario are grouped together.

When writing test cases, the following information is recorded: the purpose of the test, the instructions on how to run the test, and the expected outputs of the test case.

The Testing phase. The subject runs simulations to verify the specification against the test cases. When executing each test, three representations must be consistent—the specification, the test cases, and the subject's mental model of what the customer wants from the system. If the specification does not pass a test case, then either the test case, or the model, is incorrect. If the specification passes a test case, but the subject has doubts about the validity of the test case—perhaps from his domain knowledge, or from what he knows about the customer's needs—then either the specification and the test case must be modified to match the subject's thinking, or the subject must alter his mental model.

If the subject determines that the specification does not cover all of the required functionality at this stage, then he goes back to an earlier phase to add it to the specification.

2.8.2 Writing a Specification Using SCR

Modelling From the User Requirements Specification

The SCR model is drafted on paper to allow the subject freedom to express ideas in the modelling phase. The system boundary is defined based on the system's inputs and

outputs, which are described in the URS. First, the subject declares a *monitored variable* for each input event and a *controlled variable* for each output event.

Second, the subject introduces a mode class for each independent functionality of the system, and introduces modes for each mode of operation in each of the mode classes. Modes represent equivalence classes of input histories; they serve the same purpose as control states in a finite state machine notation. The transitions for these modes are modelled as events that respond to changes in the monitored variables.

Finally, the system is decomposed by each controlled variable. The subject, from reading the URS, discovers which possible values can be assigned to each variable, and each value is followed backwards to determine the possible conditions for the value's assignment. Terms are used to simplify conditions in function tables; terms are essentially macros. Instead of tracing forward from the inputs to the outputs, the design of SCR encourages going backwards from the outputs to the inputs, and discovering the global conditions for each assignment.

One limitation with SCR is that it cannot handle advanced data types, such as lists. If the URS calls for the use of an advanced data type, such as a history that the user can recall, then the subject must leave this functionality out of the SCR specification. For the above history example, instead of keeping the list and reading a value from the history, a monitored variable may instead represent the value to be read.

Specifying with the SCRtool

In the specification phase, the subject enters the model into the SCRtool. SCR's tabular notation helps the subject observe if every necessary assignment has been made, and if every situation is handled.

The SCRtool informs the user of circular dependencies, coverage errors, and syntax errors as soon as they are made. This automatic checking reduces the time of the compilation phase to almost zero, and helps keep the subject from building a model that can not be simulated.

Because SCR does not support decomposition by structure, the subject must write the entire requirements specification before testing. One advantage to writing the entire specification up front is that the specification must be carefully designed before the subject can proceed with the task. When changes need to be made to the specification, the subject modify the specification such that the increase in complexity is limited. When software evolves, the complexity of the software increases, and up-front planning may help control increasing complexity.

Testing an SCR Specification

When testing an SCR specification, the subject first builds a GUI for the specification using the SCRsimulator. A widget can be used to manipulate a monitored variable, or the widget can be used to display the value of a controlled variable. The subject writes

scripts that can be used to run through a scenario. In addition to the visual feedback provided by the GUI, the SCRsimulator also writes a log file that reports when each variable changes its value. The subject compares the expected output with these log files to determine if the test case has passed.

Modifying an SCR Specification

The process of modifying the SCR specification is similar to that of writing the initial specification. Modifications to an SCR specification involve extensive planning because it is difficult to build an SCR specification incrementally. SCR's tables are easy to browse, which makes isolating change easy, but modifying the specification may take time, especially if a new term or a new monitored variable is introduced.

2.8.3 Writing a Specification Using Real-time UML

Modelling from the User Requirements Specification

To write an RT-UML specification, the subject defines the system's boundary and determines an initial design for the model in the modelling phase. The system boundary is defined based on the system's inputs and outputs, which are described in the URS. A capsule that represents the structural border between a system and its environment is created, and is titled with the name of the system that is being modelled. This capsule is called the **top-level capsule**.

The subject defines a protocol for every communication path in the system. The subject defines input signals and output signals into the system, and sorts these signals into the appropriate protocol. Because a protocol usually represents a physical communication path, the sorting process is usually intuitive. If a capsule must send and receive signals using a particular protocol, the subject adds a port to the capsule's structure diagram and sets the port to use that protocol.

If the system can be logically decomposed into subcomponents, then an additional capsule representing each subcomponent is created inside the top-level capsule. This decomposition is modelled using object-oriented analysis techniques. For example, a URS may describe multiple hardware modules within one system, so creating a capsule for each module best represents the customer's requirements. If a capsule contains another capsule, the connectors must be laid out to ensure that signals can reach each capsule. If a subcomponent is passive, and does not need to respond to signals, then a class may be used to represent the subcomponent instead of a capsule.

If data storage is necessary, each appropriate attribute is defined with a capsule or a class. We apply good object-oriented design principles, such as keeping each operation simple, and exposing as little of the internal design as possible.

Specifying with Rational Rose RealTime

The specification phase involves entering the model into the CASE tool, Rational Rose RealTime. In addition to drawing visual diagrams, such as structure diagrams and state machines, the subject must also write code for each action in the model.

Each capsule contains a state machine that uses the Statecharts notation. Each transition in the state machine is triggered by a signal received by one of the capsule's end ports, and may optionally execute an action. The code that is used to execute these actions is kept as short and simple as possible; it is preferable to have the action call an operation if the action must be complex.

The compilation phase begins when the RT-UML specification is compiled after the functionality is specified. The compilation phase involves the correction of syntax errors, and appropriate defects are logged. Compiling is avoided until a majority of the functionality is covered. Unfortunately, compilation in Rational Rose RealTime involves coping with code syntax, in addition to coping with RT-UML syntax.

Testing an RT-UML Specification

RT-UML specifications are executed using the Rational Rose RealTime CASE tool. The subject can inject a signal into each port, monitor each port for output signals, and set a watch on attributes within the specification. A visual diagram displays the system's control flow using a state diagram. Running a test case involves injecting the appropriate input signal in sequence, watching each attribute, and waiting for output signals to be received by the environment. The value of each attribute and each output signal is verified against the test case.

The Rational Rose RealTime tool supposedly supports automatic execution of test scripts, but the user has been unable to figure out how to use this functionality even after extensively examining the help documentation.

Modifying an RT-UML Specification

Changes can be easily made to an RT-UML specification because RT-UML contains support for modular design techniques. In structure diagrams, a subject may reuse capsules, and in a Statechart, a subject can use substates and superstates to organize the state machine structure. There is also considerable flexibility with classes, attributes, and operations. Browsing an RT-UML specification and determining the best way to make modifications can be time-consuming because the specification language involves so many different elements. Using RT-UML encourages a subject to develop iteratively because very small modules can be developed and expanded. Iterative development may be an inexpensive way to implement changes immediately, but the complexity of the specification might increase at a greater rate than a specification technique that encourages more up-front planning.

2.9 Threats to Validity

The threats to validity associated with the pilot study are listed below, along with a description of threats to validity in the fully controlled experiment described back in Section 2.3. We describe the threats to validity in the fully controlled experiment before we describe the threats to validity in the pilot study.

2.9.1 Internal Threats to Validity in a Fully Controlled Experiment

Internal threats to validity are issues in the experiment that may influence the procedure, such that the conclusions made from the data are not properly attributed to the correct factors.

The following is an incomplete list of internal threats to validity for a proposed fully controlled experiment. Obviously, depending on the experimental conditions, other internal threats to validity may also exist.

Measurement error. There are certainly measurement errors in time and defect tracking. Time is tracked in detail by each subject, who must record the amount of work done in a day. The subject must diligently record the duration of interruptions, which include breaks and office visitors. Flagging interruptions to be accurate to even a few minutes is very difficult.

Inability to separate subject ability and individual tracking style. There may be differences in each subject's perception of how to track time and defects. Although the subject's ability can be separated, the individual subject's tracking style would be confounded with his ability.

Experimenter bias. The presence of experimenters can affect the experiment. Care must be taken to ensure that the experimenters interact with the subjects as infrequently as possible. The experimenters must analyze the specifications and logs provided by the subjects, which can also introduce bias.

2.9.2 Internal Threats to Validity in the Pilot Study

Because the pilot study has only one subject, a number of internal threats to validity are introduced. Because the pilot study is not replicated, a statistical analysis cannot be done. The design also introduces a threat due to its swap design. Finally, although individual subject ability is no longer an issue, subject bias can influence the outcome.

Internal threats to validity in the pilot study are identified below:

Lack of statistical analysis. The largest threat to the experiment's validity is the inability to perform any statistical analysis because of the lack of replications. The analysis performed on this experiment is informal, and is based on observations of trends. Even though the analysis is not based on statistics, the analysis method is not ad-hoc.

Cannot separate the learning effect and the treatment–problem interaction effect. The effect of the problem and the effect of the treatment can be separately observed. However, because of the way that the experiment is designed, we cannot determine the exactly how strong the learning effect is on the response and how strong the treatment–problem interaction effect is on the response; both are confounded into one reading in the data. Therefore, conclusions made about the interaction effect and the learning effect are speculative.

Measurement error. Measurement error is a problem in the single-subject experiment, as it is in any experiment. One way to reduce the error associated with measurement is through better time-tracking tools. However, we believe that the same subject tracking time through multiple specification tasks is probably more consistent than trying to compare different subjects, who may have differing opinions of what time should be tracked.

Subject bias. Because this experiment involves only one person, one threat is subject bias. Although the subject had no predisposition about the outcome of the pilot study, experiences during the study, such as frustration with using the specification tools, may have affected performance.

Experimenter bias. In the single-subject experiment, the subject was also the experimenter who performed the data analysis. As stated above, the subject had no predisposition about the outcome of the pilot study, but this in itself is a form of bias. The subject's co-supervisors had his or her individual bias, but because they were opposing beliefs, the subject was exposed to arguments for and against either side.

2.9.3 External Threats to Validity in a Fully Controlled Experiment

External threats to validity are issues in the experiment that suggest that the results of the experiment cannot be generalized. In software engineering, external validity usually means the applicability of the results to its targeted sample population, who are usually industry practitioners.

The problems used in this study are not representative of industrial-scale systems. This is an ongoing problem in empirical software engineering. A small system is not the same as a large system, and techniques that work well on small systems often do

not apply to large systems. There is no easy solution for this threat to validity, although measures that may help include alloting more time for the experiment, and using multiple groups of subjects to specify a large problem instead of a single subject to specify a small problem.

The environment may not be reflective of what is done in practice. The environment where the specification is done may not be reflective of a development environment in practice. For example, a development environment in industry may incorporate multiple analysts into one team, whereas this specification was written by only one person. A university may be a better environment for writing formal specifications than industry because the faculty members in software engineering are generally supportive of requirements specification and rigourous processes.

Real-time UML may not be representative of "state-of-the-practice". One reason for choosing Real-time UML as a treatment in this experiment is because the UML is extremely popular in industry for use in specification. However, the real-time extension is a non-standard extension that is used in Rational Rose RealTime (now Rational Rose Technical Developer), which may not be commonly used. Industry may in fact not do any specification at all with a visual state-based language.

2.9.4 External Threats to Validity in a Pilot Study

Most of the threats in the fully controlled experiment carry over to the pilot study. However, two additional threats are introduced.

Subject does not have specification experience. The subject working on the experiment does not have any previous experience with some of the formal specification languages, and therefore may not be representative of the average software analyst. However, it may also be the case that industry practitioners do not have extensive experience with the languages, either. This is a small threat, but is a threat nonetheless.

In the fully controlled experiment, this could still be a threat. Whether it is or not depends on the subjects that are recruited for the study, and if they are representative of the population the experiment is intended to generalize to. The levels of each subject may also be more variable, and will be harder to assess.

Tracking is extremely personal. The measurement techniques are subjective and would be difficult to generalize to a population. The subject in the pilot study measured active time only, in which work was actively being done on the specification task. A person in the target population may measure active time and passive time, which would include time spent thinking about the specification. The definition of a defect may differ among each subject; a subject also may handle a defect differently than another subject.

Therefore, an analyst must consider that the measurement techniques that one person uses may not be representative of the target population.

In the fully controlled experiment, this threat is not an external threat to validity, but is an internal threat to validity because the effect is confounded with the subject ability, and would affect the internal analysis.

2.10 Summary of Experimental Procedure

This chapter describes a single-subject experiment executed as a pilot study for a fully controlled experiment. The author of the thesis, who was the subject, wrote an IS using UML and an IS in SCR for two software problems, the BDF problem, and the BC problem. When each IS was completed, each customer provided the subject with updated requirements, and the subject made the appropriate changes to create the SM. The time required to write each IS and SM, the number of defects injected into each IS and SM, and the time required to correct each defect were tracked. Because of an anticipated learning effect associated with specifying the same problem twice, the order of applying the the languages is reversed for each problem. Although the design does not factor out the learning effect entirely, the design may provide some anecdotal evidence about the learning effect on the results.

Chapter 3

Observations and Analysis

This chapter describes the observations of the experiment, along with supporting analysis. We presents the subject's experiences with the specification process, examines the extent to which the specifications were equivalent, and discusses the sizes of the specifications. We present also observations of the time required to complete the specifications, and the defects discovered during the specification. Finally, the chapter discusses the results of the pilot study.

3.1 Subject's Experience With Specifying the Problems

The BDF problem was much more difficult and confusing to specify than the BC problem, in both RT-UML and SCR. Every BDF IS and BDF SM, written in both RT-UML and SCR, was also very difficult to read. The BDF problem was difficult to specify for the following reasons.

- **The abstractions used for formal specification were confusing.** Because both languages require that events be injected before the system can run, the subject had to devise an abstraction of the BDF problem whereby an input string was represented by an event containing a size and a direction, instead of characters. The space character was modelled as a separate input. The act of printing an output string was modelled by a number representing a cursor position. Based on the input, the cursor would move as if it were typing the characters on the line in the proper direction. In fact, using the abstraction might have caused the specification to become more difficult than using real strings for input and output. The abstraction was frustrating to the point where the subject started to believe that implementing a prototype in a regular programming language would have been easier, faster, and more useful.
- **There was a large number of transitions in the RT-UML specification.** There were over eighty transitions in RT-UML specification of the BDF problem

because the BDF problem had a number of commands that triggered state changes. The state of the BDF problem can be represented as a 4-tuple, *(document, centre, direction, number)*, where document has two values, centre has two values, direction has two values, and number has three possible values, for a total of 24 possible states. In the model, 8 states handle 5 transitions each, and 16 states handle 3 transitions each, for a total of 96 transitions. To avoid the need to add even more states in the BDF SM, a state variable was used. The large number of transitions made the BDF SM very tedious to specify, and required significant effort when specifying the states using the Rational Rose RealTime tool.

- **There was a large number of conditions in the SCR specification.** The controlled variable in the SCR specification that represented the cursor needed to respond to multiple conditions. As in the RT-UML specification, 96 conditions should have been present in the table, but many were missing.

Specifying the BC problem did not require an elaborate abstraction. The BC problem was more straightforward because it had fewer inputs and scenarios designed to respond to events. The BC problem is an event-driven system, which SCR and RT-UML are both designed to specify.

3.2 Differences Between the Specifications

To help ensure that the comparison of specification effort in the two languages is fair, we wanted to ensure that specifications written in the two languages were completed to comparable levels of detail and correctness. Thus, we compare the differences in functionality between the RT-UML specifications and the SCR specifications for each problem. To perform the comparison, we compare the RT-UML test cases with the SCR test cases to check that the system coverage is the same, and then verify that each specification passes every one of its respective test cases. The comparison was executed manually by the author. No blinding of the test cases or the specifications was used.

Bicycle Computer For the BC problem, there was no difference in the functionality between the specification written in RT-UML, and the specification written in SCR. The test cases were built around the same scenarios, and from a user perspective, the systems behaved the same.

Bidirectional Formatter For the BDF problem, there were differences in the functionality between the specifications written in RT-UML and the specifications written in SCR. The BDF IS written in SCR covered a subset of the functionality of the BDF IS written in RT-UML. In the SCR specification, the centring functionality was incorrect for Right-to-Left Documents, and if the system was in the Right-to-Left Document mode, it

did not send the correct output in response to events. Therefore, we say that the BDF IS written in RT-UML is of higher quality than the BDF IS written in SCR.

Neither the BDF SM written in SCR nor the BDF SM in RT-UML SM was completed. Because the BDF IS took a long time to write, the customer stated explicitly that if the modifications took too long, then they should not be completed. Consequently, the reader should view the analysis involving the SMs for the BDF problem skeptically; had this been a controlled experiment with replications, these runs would have been discarded. Nonetheless, we compare the difference in functionality between the portion of each SM that was completed. The modified SCR specification, which was performed after the modified RT-UML specification, covered more functionality, and took five hours longer to complete than the RT-UML specification took. Thus, the BDF SM written in SCR can be said to be of higher quality than the BDF SM written in RT-UML.

The fact that the specifications could not be completed is telling. According to personal correspondence with Berry, a C-language implementation of the BDF requirements change, written several years previously, involved minor changes in only a few modules [6]. In the BDF SM written in SCR, a large number of conditions would have had to be added to the c_cursor variable table, and every condition already in the table would have had to be modified. In the BDF SM written in RT-UML, the capsule that handled the way that the output was computed had to be completely overhauled. The large amounts of time required to make the modification to both specifications written in SCR and RT-UML suggests that the formal specification languages chosen for the pilot study may not have been appropriate for the BDF problem.

3.3 Sizes of the Completed Specifications

We summarize the sizes of the specifications, and describe the differences between the IS and the SM for each treatment–problem combination. We do not compare specifications from different problems or different treatments, due to the difficulties discussed in Section 2.1.

3.3.1 Bidirectional Formatter Specification in SCR

The BDF IS written in SCR consisted of 5 monitored variables, 4 controlled variables, 4 terms, and 4 mode classes. In the mode-transition tables, there were 9 modes and 10 transitions. In the function tables, there were a total of 56 conditions and 32 unique assignments. One variable in particular, c_cursor, was quite large and consisted of 32 conditions and 15 unique assignments alone.

The BDF SM written in SCR more than doubled the number of conditions and unique assignments. There was no change in the number of monitored variables, but 9 control variables were added, 5 terms were added, and 3 mode classes were added. After the mode class additions, the specification had 18 modes and 24 transitions in the mode transition

tables, which is more than double the size of the IS. In the function tables, there were a total of 117 conditions (63 were added or modified) and 72 assignments (46 were added or modified). The majority of the changes were to the c_cursor variable, where 27 conditions were added or modified, and 11 assignments were added or modified.

3.3.2 Bidirectional Formatter Specification in RT-UML

The BDF IS in RT-UML consisted of 3 capsules. There were a total of 5 operations, 11 attributes, 3 end ports, 25 states, and 115 transitions in these capsules. The transitions had anywhere from 1 to 4 non-comment, non-whitespace lines of code (LOC), with most of them containing 2 LOC. There were 13 input signals for the system, and 4 output variables.

The BDF SM in RT-UML added a large amount of code to the system. Most of the other changes involved the refactoring of existing specification elements. There are still 3 capsules in the system, but 1 class was added. There was a total of 52 operations (47 were added and 4 were changed), 17 attributes (12 were added and 3 were removed), 25 states, and 117 transitions (6 were added, 54 were changed, and 4 were removed). The number of LOC in the transitions remained unchanged even though the code was modified. In the Document capsule, 3 of the operations had over 22 LOC. There were 16 input signals for the system (3 were added), and 4 output signals that were unchanged. The changes were extensive and had to be made to every capsule in the specification.

3.3.3 Bicycle Computer Specification in SCR

The BC IS written in SCR consisted of 6 monitored variables, 3 controlled variables, 14 terms, and 3 mode classes. In the mode-transition tables, there were 13 modes and 17 transitions. In the function tables, there were a total of 50 conditions and 40 unique assignments.

The BDF SM written in SCR required that some portions of the IS be removed. Some states and variables were removed, but the intermediate calculations and conditions became more complex. 3 monitored variables were removed from the IS, and 3 new ones were added. There was no change in the controlled variables. 8 terms were added to the IS, and 2 terms were removed from the IS. In the mode-transition tables, 3 modes were removed from the 3 existing mode classes, 3 transitions were removed, and 4 were changed. In the function tables, there were a total of 57 conditions (12 added, 10 changed, 6 removed) and 48 assignments (12 added, 4 changed, and 4 removed). The changes propogated through most of the specification. Overall, the modifications increased the size of the specification slightly.

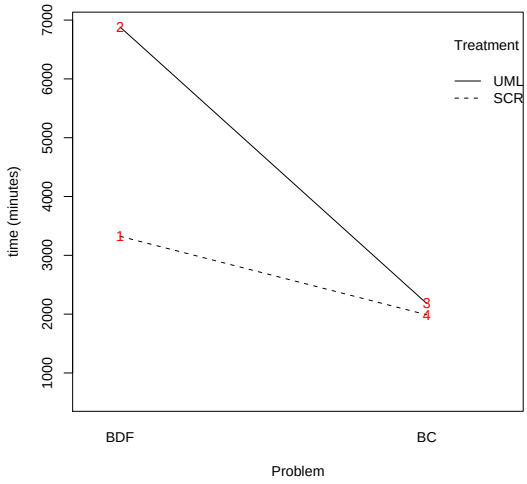


Figure 3.1: Sample Interaction Graph of Time Required For Initial Specification

3.3.4 Bicycle Computer Specification in RT-UML

The BC IS in RT-UML consisted of 2 capsules, one of which was a container for the other. There was also 1 class defined. There were a total of 14 operations, 20 attributes, 4 end ports, 11 states, and 43 transitions in these classes and capsules. There were 7 input signals and 5 output signals in the specification.

The BC SM in RT-UML consisted of the same 2 capsules and 1 class. For the SM, 1 class was added. There were a total of 24 operations (12 added, 2 removed, 3 changed), 29 attributes (16 added, 7 removed), 8 states (3 removed) and 37 transitions (6 removed). There were 9 input signals (3 added, 1 removed), and 5 output signals, which did not change. The change in the RT-UML system primarily involved the writing of code in the operations inside the added class. The change in the specification was fairly localized to a few operations.

3.4 Analysis Method

To analyze the results, we use interaction graphs to identify which independent variables of the experiment have a strong effect on the dependent variables. An interaction graph is shown in Figure 3.1. Interaction graphs are designed to highlight not only strong effects, but also the presence of an interaction between the two effects. See Christensen for an overview of factorial designs and interaction graphs [11].

Each level of the treatment effect is displayed as a different line style on the graph. Each level of the problem effect is represented on the x-axis. The response is graphed on

Run	Problem	Task	Treatment	Phases	Time (hours)	Time (minutes)
1	BDF	IS	SCR	10	**55:25**	3325
2	BDF	IS	RT-UML	28	**114:44**	6884
3	BC	IS	RT-UML	13	**36:21**	2997
4	BC	IS	SCR	5	**33:09**	2758
5	BDF	SM	RT-UML	21	**45:58**	2181
6	BDF	SM	SCR	15	**49:57**	2997
7	BC	SM	SCR	6	**21:52**	1312
8	BC	SM	RT-UML	5	**10:00**	600

Table 3.1: Time Required for Requirements Specification

the y-axis. If the two lines are almost parallel, and the lines are far apart vertically from each other, there is a strong treatment effect. If the slope of the lines is very high, then there is a strong problem effect. If the lines are far apart, and have a very high slope, then there is both a strong treatment effect and a strong problem effect.

If the lines are not parallel, a **treatment–problem interaction effect** is present, which means that the treatment and the problem simultaneously effect the response. In other words, the choice of problem has an effect on how well the treatment works, and vice-versa. Because the pilot study has no replications, the learning effect is confounded with the treatment–problem interaction effect. If we observe an interaction effect on an interaction graph, then an unknown portion of the interaction effect is actually due to the learning effect.

The numbers on each interaction graph indicate the order that the runs were performed in. The run order was presented in Table 2.1.

For the analysis, the reading phase was not included in time analysis, and requirements errors were not included in the defect analysis.

3.5 Time Observations and Analysis

The overview of the time data is shown in Table 3.1. The ISs took between 33 hours to 114 hours to specify. The SMs took between 10 hours to 50 hours to specify; note that the BDF SMs were both almost fifty hours, and neither was completed. The large amounts of time needed for the modifications to the BDF problem are indicative of the difficulties encountered with it.

The phases column in Table 3.1 refers to the number of phases (Section 2.8.1) that were started and finished throughout each specification task. A large number of phases suggests that there was incremental development or that a part of the specification had

Problem	Treatment	Total Specification Time (hours)	Modification Time (hours)	Percentage of Modification Time
BDF	SCR	105:22	49:58	**47%**
BDF	RT-UML	144:02	45:57	**28%**
BC	SCR	55:01	21:52	**40%**
BC	RT-UML	46:21	10:00	**23%**

Table 3.2: Time Required for Complete Development Cycle

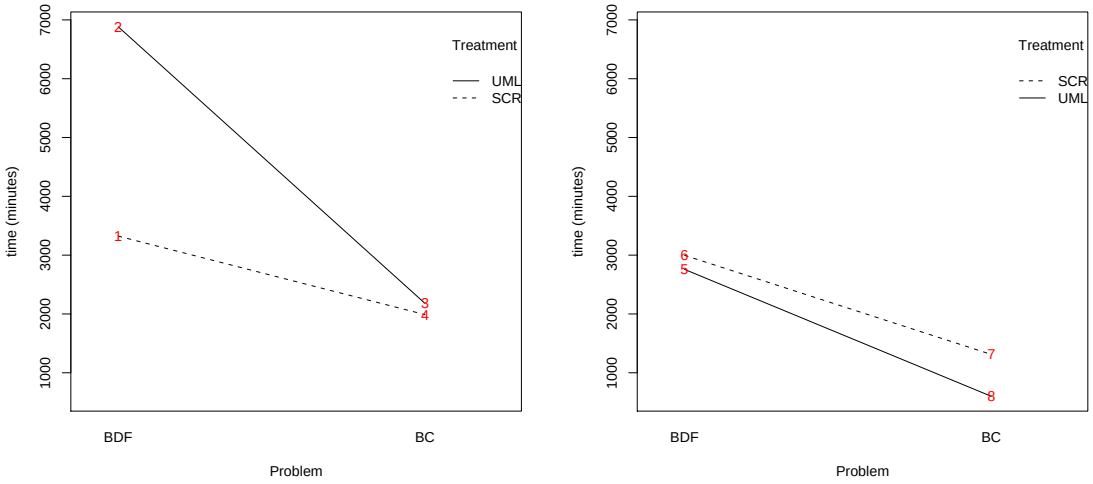
to be redesigned.

Table 3.2 displays the total specification time, which is the sum of times to complete the IS and the SM, required to do each treatment–problem combination. The percentage of modification time is represented as a fraction of the total specification time. The numbers suggest that, when using RT-UML, the percentage of modification time is lower than when using SCR. However, the BDF SMs were not completed, and it is possible that there was a learning effect that affected the modifications of the BC SM written in RT-UML, because the BC SM written in SCR was completed before starting the BC SM written in RT-UML.

Time to do Initial Specification Figure 3.2(a) shows a strong treatment–problem interaction effect on the time required to write the IS. The treatment effect on the time was large for the BDF problem, but was small for the BC problem, indicating that for the BC problem, the choice in specification language had little effect on the time required to complete the IS.

Time to do Specification Modifications Figure 3.2(b) shows that the problem has a strong effect on the time required to do the SM. Doing the BC SM required less time than doing the BDF SM. The treatment effect does not appear very large, although this point is uncertain because the BDF SMs were not completed. Although the time to specify the BC SM in SCR was double the time to specify the BC SM in RT-UML, the author does not have enough evidence to consider this a strong effect.

Time, in Run Order Figure 3.3 shows the time required to do the specification tasks, arranged in run order. Doing the second treatment, for both the BDF IS and the BDF SM, required more time than doing the first treatment. The fact that writing the BDF specifications used more time for the second treatment than for the first is contrary to intuition; many would think that writing a specification the second time would require less time than the writing the first.



(a) Time Required for Initial Specifications (b) Time Required for Specification Modifications

Figure 3.2: Interaction Graphs of Times Required for Specification

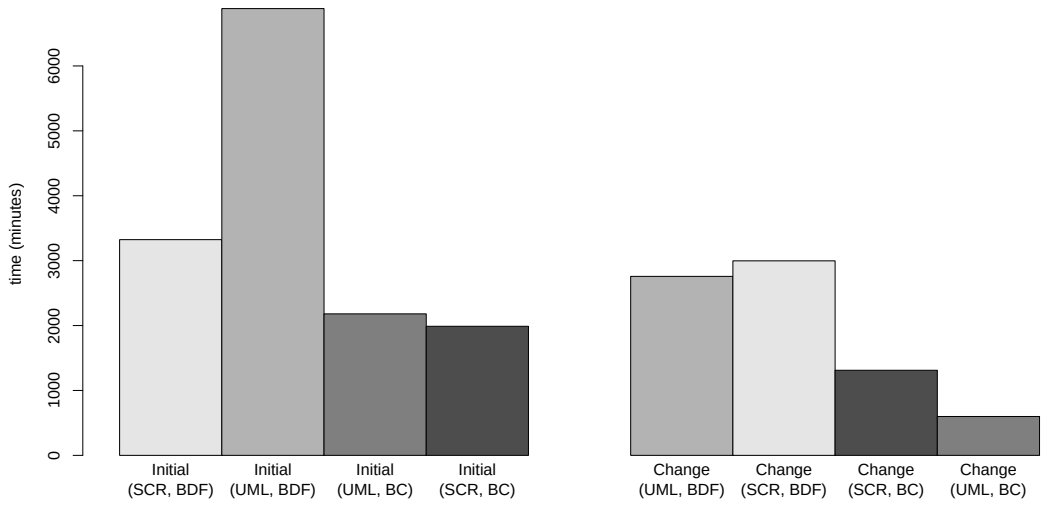


Figure 3.3: Times Required for Specification, in Run Order

Run	Problem	Task	Treatment	Specification Time (minutes)	Defect Count	Defect Repair Time (minutes)
1	BDF	IS	SCR	3325	**111**	**956**
2	BDF	IS	RT-UML	6884	**176**	**1676**
3	BC	IS	RT-UML	2181	**107**	**793**
4	BC	IS	SCR	1989	**58**	**910**
5	BDF	SM	RT-UML	2758	**72**	**1450**
6	BDF	SM	SCR	2997	**99**	**1066**
7	BC	SM	SCR	1312	**36**	**523**
8	BC	SM	RT-UML	600	**19**	**247**

Table 3.3: Defect Counts and Repair Times During Specification

Each first treatment, for either the IS or the SM of the BDF problem, required less effort than its second treatment, but yielded a specification with lower quality. This observation suggests that the learning effect may have had an effect on the time required to write the BDF IS and the BDF SM. When doing the first treatment, the subject thought that the IS was completed. However, when doing the second treatment, the subject re-examined the IS, and discovered that the problem was more complex than he originally thought. The subject then used extra time to ensure that the system was as complete as possible. The same explanation can be used for the SM, although neither SM was completed.

Each second treatment, for either the IS or the SM of the BC problem required less effort than its first treatment. This observation suggests that the learning effect decreases the amount of time required to write the same problem while maintaining the specification's quality, even when using a different language. One explanation for this is that the subject was comfortable with the quality of the specification after the first treatment, and was therefore able to write the second treatment faster.

3.6 Defect Repair Time Observations and Analysis

The defect count (DC) and the defect repair time (DRT) for each run are reported in Table 3.3. For the reasons examined in Section 2.1, the DRT and the DC by themselves are difficult to compare among specifications, so in addition to looking at the DC and the DRT for each specification, we examine also the percentage of time to repair defects, and the average defect-repair time (average DRT), shown in Table 3.4. The percentage of time to repair every defect is computed by dividing the DRT by the time required to

Run	Problem	Task	Treatment	Percentage of Time Repairing Defects	Average Defect-Repair Time (minutes)
1	BDF	IS	SCR	**29%**	**9.8**
2	BDF	IS	RT-UML	**24%**	**10.8**
3	BC	IS	RT-UML	**36%**	**20.1**
4	BC	IS	SCR	**46%**	**11.9**
5	BDF	SM	RT-UML	**53%**	**13.6**
6	BDF	SM	SCR	**36%**	**13.0**
7	BC	SM	SCR	**40%**	**12.0**
8	BC	SM	RT-UML	**41%**	**13.6**

Table 3.4: Percentage of Time Repairing Defects and Average Defect-Repair Times During Specification

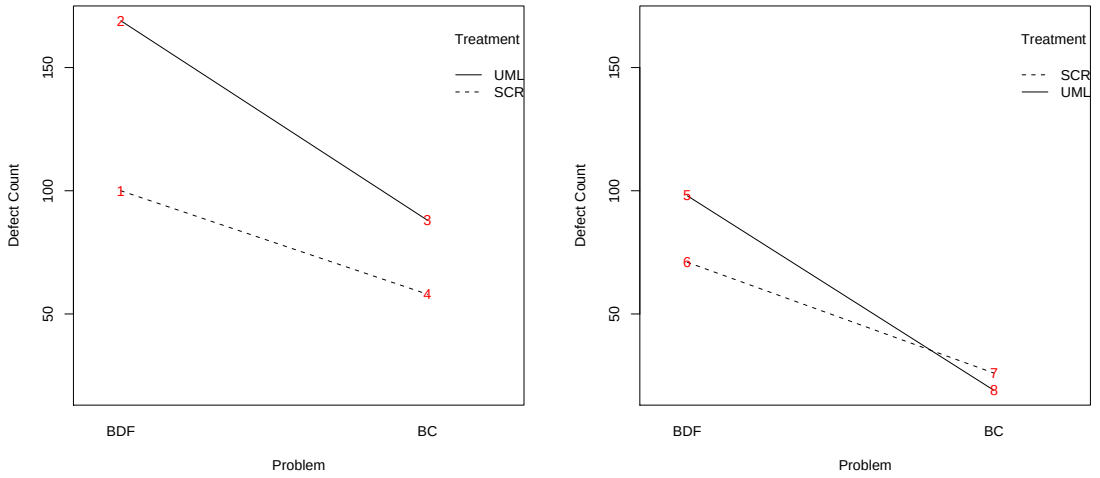
complete the specification. The average DRT, which is the average time to repair each defect, is computed by dividing the DRT by the defect count. The specification with the largest DRT in the study is the BDF IS written in RT-UML, yet, proportionally, the BDF IS written in RT-UML is actually the specification that has the smallest fraction of total time spent repairing defects. Writing the BDF IS in RT-UML required a lot of time, and it had a high DRT compared to the other specification tasks, but such a high DRT does not imply that using RT-UML to write the BDF IS is necessarily bad. The specification with the second-largest DRT in the study was the BDF SM written in RT-UML, but proportionally, this specification had the highest proportion of time spent repairing defects.

3.6.1 Defect Count

Defect Count During Initial Specification Figure 3.4(a) shows a strong treatment effect on the defect count, and a strong problem effect on the defect count. The subject discovered fewer defects when writing the BC IS than when writing the BDF IS. In addition, the study reveals that when writing an IS, using the SCR notation results in a lower DC than when using the RT-UML notation.

Defect Count During Specification Modifications Figure 3.4(b) shows a treatment–problem interaction effect present, but this effect is not very strong. The subject discovered fewer defects when writing the BC IS than when writing the BDF IS. In addition, the study reveals that when writing an MS, using the SCR notation results in a higher DC than when using the RT-UML notation.

Why would SCR be good for doing the IS, and RT-UML be good for doing the SM?



(a) Defect Count for Initial Specifications (b) Defect Count for Specification Modifications

Figure 3.4: Interaction Graphs of Defect Counts for Specification

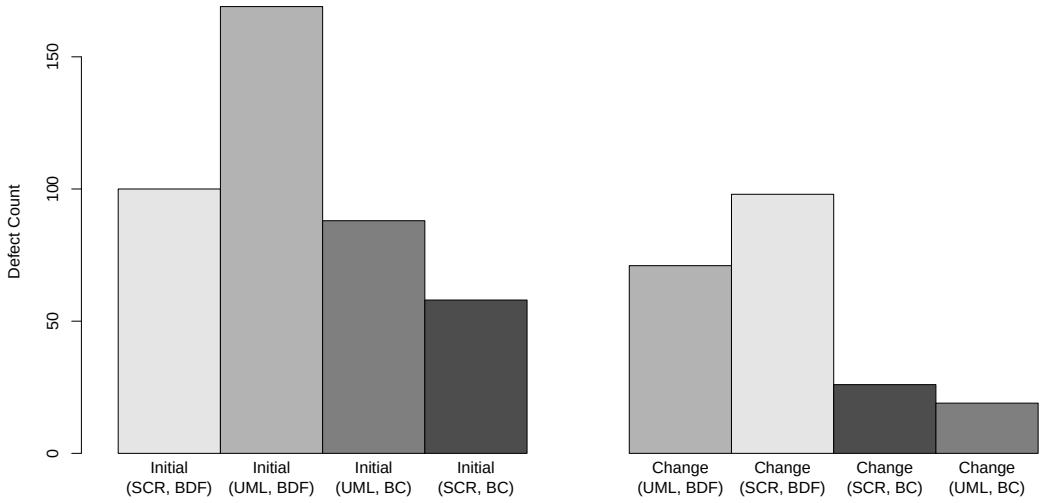


Figure 3.5: Number of Defects Discovered During Specification, in Run Order

There are two explanations. We could take the data at face value and claim that using the SCR notation reveals fewer defects during the initial specification. This explanation can be justified because SCR does error-checking on the fly. Using the RT-UML notation may require more work up front, but less work during maintenance. The other explanation is that the learning effect may have affected the number of defects made, and consequently discovered, during the specification process.

Defect Count, in Run Order Figure 3.5 shows the DC for each specification task, arranged in run order. For the BDF problem, for both the IS and the SM, the DC for the first treatment was less than the DC for the second treatment. For the BC problem, for both the IS and the SM, the DC for the first treatment was greater than the DC for the second treatment.

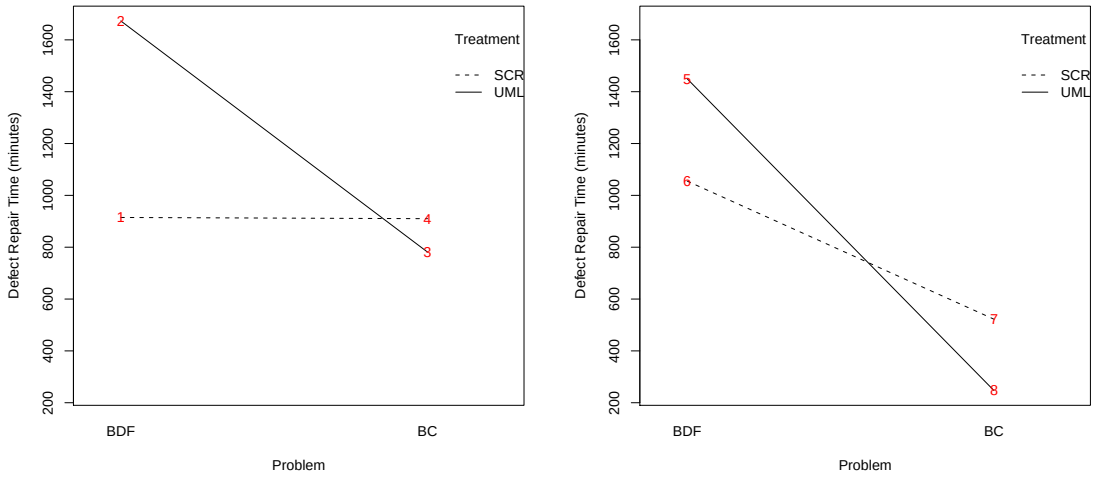
This effect is similar to what was observed in Figure 3.3. Each first treatment, for both the BDF IS and the BDF SM, had a lower DC than each second treatment, and therefore, the learning effect may have had an effect on the time required to write the BDF IS and the BDF SM. Because of the BDF IS's size and complexity, the subject did not feel confident in its quality after the first specification, and thus spent more time doing the second specification, specifying the functionality that was missed in the first treatment. The subject would therefore discover more defects, especially defects in the functionality, that were not included in the first specification. The same explanation follows for the SM, although neither SM was completed.

3.6.2 Defect Repair Time

Defect Repair Time During Initial Specification Figure 3.6(a) shows a strong treatment–problem interaction effect. It appears that when repairing errors in the BDF IS, the subject spends less time repairing errors when using SCR than when using RT-UML. The situation for the BC IS is reversed, and the subject repairs defects when using RT-UML in less time than when using SCR.

Defect Repair Time During Specification Modifications Figure 3.6(b) shows a strong treatment–problem interaction effect. It appears that when repairing errors in the BDF SM, the subject repairs defects in less time when using SCR than when using RT-UML. The situation for the BC SM is reversed, and the subject repairs defects in less time when using RT-UML than when using SCR.

Defect Repair Time, in Run Order Figure 3.7 shows the DRT for defects discovered during the specification process, ordered by run. With respect to the BDF problem, writing the IS and the SM using RT-UML results in a higher DRT than writing the respective IS and SM using SCR. With respect to the BC problem, writing the IS and



(a) Defect Repair Time for Initial Specifications

(b) Defect Repair Time for Specification Modifications

Figure 3.6: Interaction Graph of Defect Repair Time for Specification

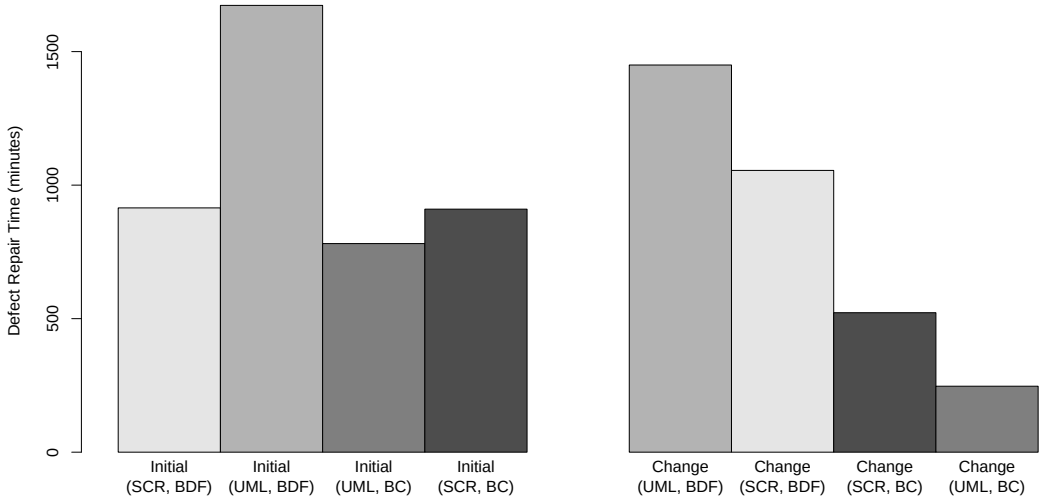
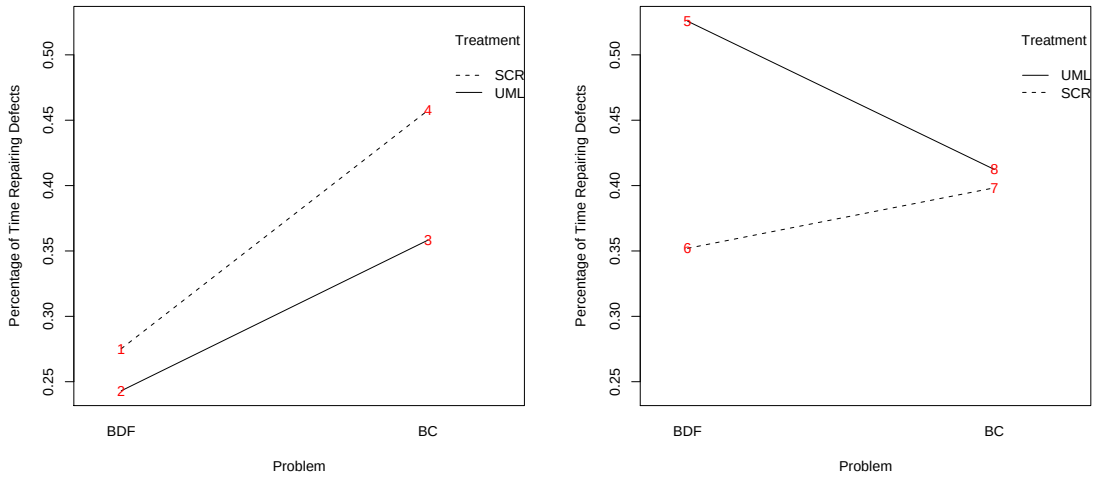


Figure 3.7: Defect Repair Times Discovered During Specification, in Run Order



(a) Percentage of Time Repairing Defects for Initial Specifications

(b) Percentage of Time Repairing Defects for Specification Modifications

Figure 3.8: Interaction Graph of Percentage of Time Repairing Defects for Specification

the SM using SCR results in a higher DRT than writing the respective IS and SM using RT-UML.

Because the SCR treatments have lower DRTs with the BDF problem, and the RT-UML treatments have lower DRTs with the BC problem, it appears that if an appropriate language is used with a problem, the DRTs can be kept low, thus reducing the number of mistakes and improving the amount of value-added work devoted to a specification.

3.6.3 Percentage of Time Repairing Defects

The fraction of the specification time spent repairing defects is computed by dividing the DRT by the time taken to write the specification. This value is called also the **percentage of time repairing defects**.

Percentage of Time Repairing Defects During Initial Specification Figure 3.8(a) shows a strong treatment effect on the percentage of time repairing defects. There is also a strong problem effect. It appears that when the subject used RT-UML to write either the BDF IS or the BC IS, the percentage of time repairing defects was less than the percentage of time repairing defects when he used SCR. It also appears that when the subject wrote the BDF IS, the percentage of time repairing defects was less than the percentage of time repairing defects when he wrote the BC IS.

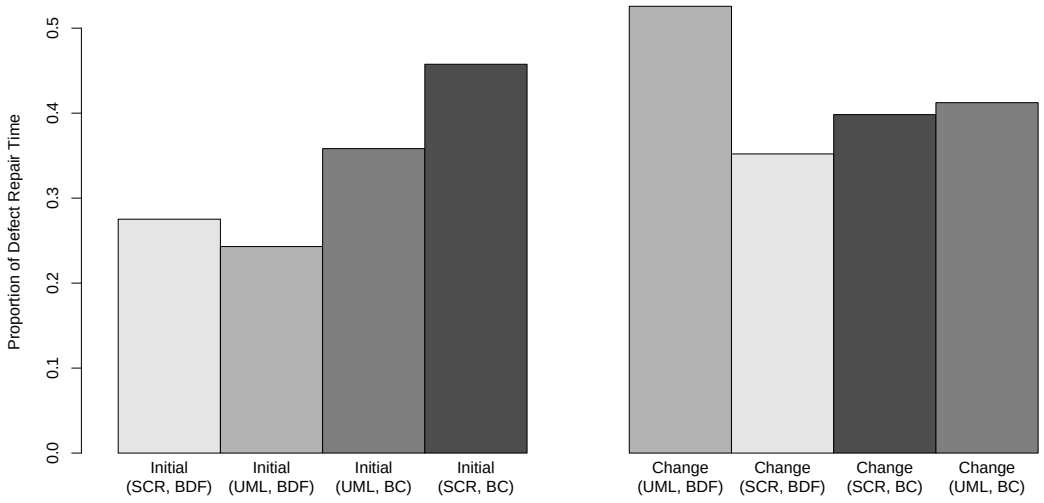
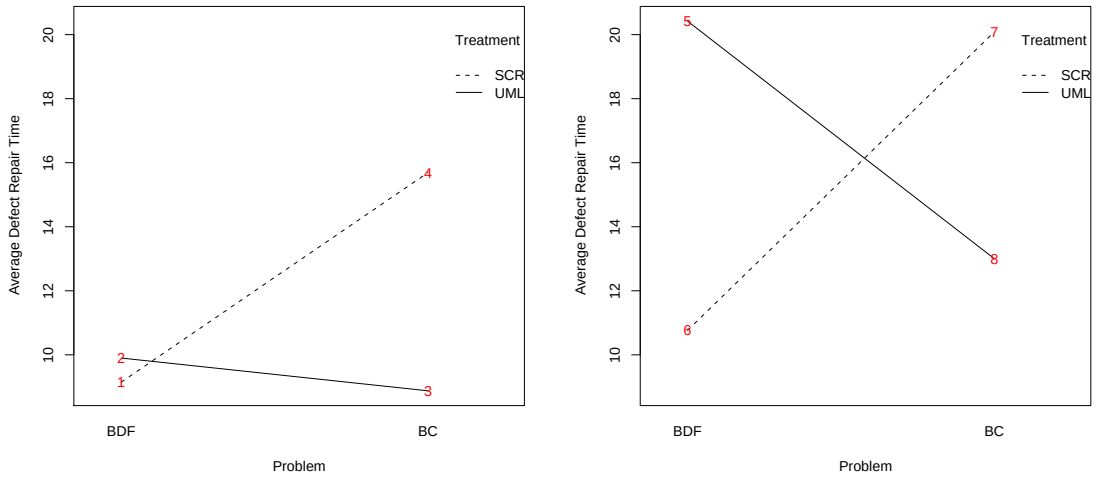


Figure 3.9: Percentage of Time Repairing Defects, in Run Order

One explanation for the difference in the percentage of time repairing defects between the BDF problem and the BC problem is, because the BDF problem was more difficult to specify using the event-driven languages, the subject required more time thinking and planning, as opposed to working on the specification. The BC problem, which was less abstract, was easier to specify. It is possible that the haste of specifying the BC problem contributed to the number of defects, but without more data from like-minded studies, we do not know if spending 35–50% of the specification time on repairing defects is normal.

Percentage of Time Repairing Defects During Specification Modification Figure 3.8(b) shows a treatment–problem interaction effect. The language used has a very strong effect on the percentage of time repairing defects in the BDF SM, but not the BC SM. It appears that the choice of language is important when doing an SM for the BDF problem, but less important when doing an SM for the BC problem.

Percentage of Time Repairing Defects, in Run Order The percentage of time repairing defects is displayed in Figure 3.9, in run order. The first treatment for both the BDF IS and the BDF SM had a higher percentage of time repairing defects than the second treatment. The first treatment for both the BC IS and the BC SM had a lower percentage of time repairing defects than the second treatment. This observed trend is actually opposite to the possible learning effect observed for time (Figure 3.3) and DC



(a) Average Defect-Repair Time for Initial Specifications

(b) Average Defect-Repair Time for Specification Modifications

Figure 3.10: Interaction Graph of Average Defect-Repair Time for Specification

(Figure 3.7).

The suggestion, from Figure 3.8(a), is that using RT-UML for an IS reduces the percentage of time repairing defects, but no such gain can be noticed in the SMs.

I suspect that there is not only a treatment–problem interaction effect present, but also a learning effect; it is possible that there is a combination of both factors. We know, from Figure 3.3, that the learning effect may have had an influence on the time required for specification. From Figure 3.7, we see that the choice of the appropriate language on the problem may have had an influence on the defect repair time. This combination of effects may have led to the observations for the percentage of time repairing defects.

Average Defect-Repair Time

The average defect-repair time (average DRT) is the DRT divided by the DC. Average DRT can be compared to other specifications to get an idea of how easy it may have been to fix a defect discovered during specification.

Average Defect-Repair Time During Initial Specification The interaction graph of the average DRT is shown in Figure 3.10(a). There is a strong treatment–problem interaction effect present. Using the RT-UML to write the IS had a strong effect on the

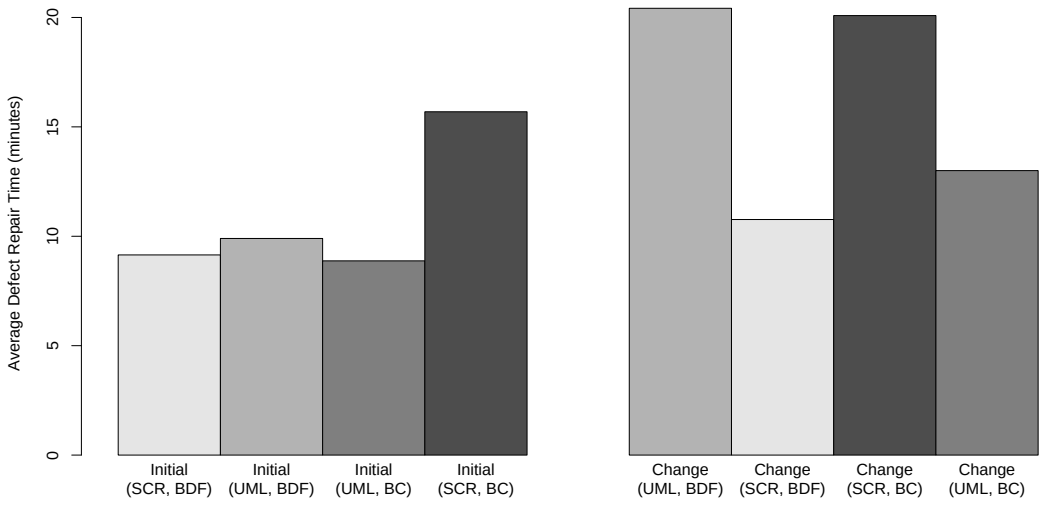


Figure 3.11: Average Defect-Repair Times Discovered During Specification, in Run Order

average DRT when doing the BC problem. The choice of language had little effect on the average DRT when writing the BDF IS.

The average DRT for the ISs is 10.3 minutes/defect.

Average Defect-Repair Time During Specification Modifications Figure 3.10(b) shows the effects on the average DRT during the SM. Again, the treatment–problem interaction effect here is strong. The average DRT when using SCR to do the BDF SM was less than the average DRT when using RT-UML for the BDF SM. The average DRT when using RT-UML to do the BC SM was less then the average DRT when using SCR for the BC SM.

The average DRT for the SMs is 12.93 minutes/defect.

Average Defect-Repair Time in Run Order Figure 3.11 shows the average DRT in run order. It appears that the average DRT when using SCR to write either the BDF IS or the BDF SM was less than the average DRT using RT-UML. For the BC problem, the situation is reversed. The average DRT when using RT-UML to write either the BC IS or the BC SM was less than the average DRT when using SCR.

Unlike the case for the percentage of time specifying defects, it appears that the DC (Figure 3.5) and the DRT (Figure 3.7) were each affected more by the treatment–problem interaction effect than by the learning effect. Therefore, the average DRT would also be

influenced the most by the treatment–problem interaction effect. Thus, selecting the appropriate language for a problem would reduce the average amount of time a subject spends repairing defects during the specification process.

3.7 Experimental Results from the Pilot Study

This section summarizes the results concluded from the data gathered during the pilot study, and presents some of their implications.

3.7.1 Time Required

It is clear from the data acquired in this experiment that there is a large difference between the time required to specify the BDF problem and the BC problem. This is not surprising, as the two problems are very different from each other. The BDF problem required more time to specify than the BC problem, for both the ISs and the SMs. However, the largest effect on time appears to be the learning effect, which we discuss below in Section 3.7.3.

The treatment effect on the time required to write the BDF IS is large (Figure 3.2(a)). Figure 3.2(b) shows that the treatment effect when writing the BDF SM is small. The situation is reversed for the BC problem: The treatment effect is small when writing the IS, but large when writing the SM.

That one language worked better than the other when specifying the IS for a problem does not imply that the same language will work better than the other when doing the SM for changes to the same problem, and vice versa. In other words, the times needed to do the IS and the SM appear to be independent. Although the pilot study examined the results of applying only one modification, the author hypothesizes that subsequent modifications would likely be independent of both the IS and the first SM.

3.7.2 Defect Repair Time and Suitability of a Language to a Problem

The treatment–problem interaction strongly affects the DRT for both the IS and the SM, as seen in Figure 3.7. With respect to the BDF problem, writing the IS and the SM using RT-UML results in a higher DRT than the writing the respective IS and SM using SCR. With respect to the BC problem, writing the IS and the SM using SCR results in a higher DRT than writing the respective IS and SM using RT-UML. Because the observation is consistent for both the IS and the SM, it is likely that a higher proportion of the effect is from the treatment–problem interaction, not from learning.

A similar trend is observed with average defect-repair time (Figure 3.11), where using SCR to write both the BDF IS and the BDF SM resulted in a lower average DRT than using RT-UML, and where using RT-UML to write both the BC IS and the BC SM resulted in a lower average DRT than using SCR. In addition, the average DRT when modifying specifications was higher than the average DRT when writing initial specifications.

This observation indicates that using SCR to write either the BDF IS or the BDF SM is more efficient than using RT-UML. Similarly, using RT-UML is better when writing the BC problem. It seems that if a language is well suited for a particular problem, then defects are easier and cheaper to fix, but making defects easier and cheaper to fix does not necessarily translate into cheaper costs of writing the IS and the SM for the problem.

The average DRT is higher when the subject used SCR to write the BC problem than when he used SCR to write the BDF problem. In Figure 3.11, the average DRT for the BC IS written in SCR is greater than the average DRT for the BC IS written in RT-UML. The average DRT for the BC SM written in SCR is also greater than average DRT for the BC SM written in RT-UML. Is it the case that SCR does not prevent errors made by the subject as effectively as RT-UML? One may argue that because SCR is a flat language, if a modification is made, more of the specification must be double-checked. One has to trace the conditions backward through the dependencies. One cannot make a change and be confident that the change is limited to the one module. The BC IS had a large number of small controlled variables, and the BC SM required the addition of many terms, which could have introduced more defects. Using SCR to specify the BDF problem may have resulted in lower average DRTs because the changes in the BDF problem were isolated to many conditions in only a few variables. It appears that the nature of the change would have a strong influence on the cost of modifications; if a requirements modification requires minor adjustments in many existing variables, then the change in SCR may be easier than the change in RT-UML. Alternately, if a requirements modification requires new assignments and new variables, then the change in RT-UML may be easier than the change in SCR. This is only speculative, and more user studies would be required to make a stronger conclusion.

The percentage of time spent repairing defects (Figures 3.8(a) and 3.8(b)) shows that there is a treatment–problem interaction effect when doing the SMs, but that there is a strong treatment effect when writing the IS. For the initial specification, using RT-UML appears to reduce the percentage of time spent repairing defects for both the BDF problem and the BC problem, but this is not the case for the SMs. One explanation for this data is that there may be a combination of the treatment–problem interaction effect and the learning effect that cannot be clearly determined, and this may be affecting the percentage of time repairing defects. More study of this variable is recommended.

There appears to be no clear connection between the percentage of time repairing defects and the amount of time spent on specification. A specification task that had a low percentage of time repairing defects did not suggest that the time required to do the specification would be low as well. The sample studied in this pilot study is too small to make a clear observation.

3.7.3 Influence of The Learning Effect

The effect of learning in this experiment is confounded with the treatment–problem interaction effect, which means that separating the learning effect from the treatment–problem

interaction effect is impossible. Speculation can be made as to the strength of the learning effect versus the strength of the treatment–problem interaction effect, but such speculation is anecdotal.

The learning effect may have a stronger effect on time than the problem-treatment interaction effect, because, even with wash-out periods, the previous specification was difficult to forget. Although the languages were different, the abstractions derived for the different treatments of each problem shared a number of similarities. The learning effect seemed to influence the model abstraction the most, and it was difficult to consider an alternative abstraction for the second treatment once one was decided upon for the first treatment. According to the trends observed in the data, the number of defects and the amount of time required to fix each defect seemed to be less influenced by learning, and more influenced by the treatment–problem interaction.

When writing the BDF problem, each second treatment for both the IS and the SM required more time than its first treatment, but yielded a specification with higher quality. This might be initially counterintuitive—why would writing a problem a second time increase the time required for specification? One explanation is that the BDF problem, due to its unintuitive abstraction and large number of conditions, was difficult to specify, and could be considered a reasonably complex problem. When writing the second treatment, more time is used to achieve higher quality that was not achieved the first time, implying that writing the specification once was not sufficient to ensure that the specification is complete.

When writing the BC problem, each second treatment, for both the IS and the SM, required less time than its first treatment, and the RT-UML specification and the SCR specification were equal in functionality. The BC problem was easier to specify than the BDF problem with respect to the languages used in this pilot study. This may be an explanation for why it was possible to specify the second treatment in less time than it took to specify the first treatment, yet maintain the same level of quality. The subject managed to understand the problem after the first treatment, so instead of trying to ensure completeness, he could instead concentrate on the specification process.

The statement about how the complexity of the problem affects learning is only a hypothesis. More studies of this observed phenomenon are recommended to explore the learning effect's impact on developing software.

3.7.4 Revisiting the Hypothesis

We revisit the hypothesis of this pilot study to determine if we should reject the null hypothesis. The alternative hypothesis for this experiment is as follows:

$H_\alpha 1$**:** The time required to modify a specification of a system written using a tabular notation to reflect a requirements change is less than the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_\alpha 2$**:** The time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change is less than the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

The null hypothesis that we are testing in this pilot is as follows:

$H_0 1$**:** There is no difference between the time required to modify a specification of a system written using a tabular notation to reflect a requirements change and the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_0 2$**:** There is no difference between the the time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change and the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

There is a small treatment effect on the time required to do specification modifications from Figure 3.2(b). It appears that the time required for specification when using RT-UML is slightly less than the time required for specification when using SCR: In fact, the small difference might suggest that the wrong alternative hypothesis was chosen for the experiment, and that making modifications to a specification written in RT-UML may cost less than making modifications to a specification written in SCR. Nonetheless, the difference is small enough such that I am reluctant to confirm this as a strong effect. In addition, the BDF SMs are data that is not as robust as I would have liked, and the BC SMs are affected by the learning effect. As a result, it is probably best to be conservative in this analysis. Based on this observation, *I do not reject the null hypothesis*, and I conclude that the time required to modify a specification of a system written using a tabular notation to reflect a requirements change is not less than the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

For the defect repair time when modifying a specification, Figure 3.6(b) shows a treatment–problem interaction effect, which means that the treatment effect is not the only influence on the defect repair time. Because there is no effect due only to the language, *I do not reject the null hypothesis*, and the time required to repair defects when modifying a specification of a system written in a tabular notation to reflect a requirements change is not less than the time required to repair defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

It appears that differences in the specification time and in DRT are due more to the treatment–problem interaction effect and the learning effect than the treatment effect

only. This result is only preliminary, and gathering more more evidence is recommended to confirm the results of the pilot study.

3.8 On Formal Specification

The main purpose of formal specification is to gain a full understanding of a problem and its solution before describing an implementation. This pilot study has presented anecdotal evidence of a learning effect that increases the quality of the second formal specification of the same URS. However, this learning effect seems to have two possible effects. The first possible effect is that writing a specification the second time results in a higher-quality specification than the first time, but requires more time to write. The second possible effect is that writing a specification the second time results in a specification that has the same quality as the first specification, but requires less time. An open problem is discovering a method how to determine whether doing the second specification will result in a higher-quality system.

The suggestion of this study is that the less that one knows about the problem, then the more important it is to do a complete formal description of some kind, such as a specification, or a prototype, before implementing code. Specifying a problem for the first time may take a short amount of time. Because code is a formal specification of a problem, writing a problem in code may be considered specifying the system a second time, and therefore, the resulting system should be either of a higher quality than the specification written the first time, or the same quality of the specification written the first time.

If the formal specification language is entirely incompatible with the problem, then the problem will likely be more difficult to specify than using a compatible formal specification language. It is not useful to specify a problem using a specification language that does not match, because the specification work will require lot of time, increase frustration, and result in an artifact that is not useful. In addition, choosing a language that is well suited for a particular problem appears to reduce the amount of time that requirements engineers spend repairing defects, although reducing the amount of time spent on repairing defects may not directly translate into cost savings.

3.9 Summary of Experimental Results

This chapter describes the results of the analysis of the data gathered during the writing and modifying of the specifications in the pilot study. The three primary results are summarized below.

1. The effect of a treatment on an IS and on an SM were not constant. The fact that one language worked better than the other when writing the IS for a problem does

not imply that the same language will work better than the other when writing the SM for changes to the same problem.

2. There is evidence that the learning effect does not simply reduce the time required to specify a problem. Specifying the BDF problem, which is a complex problem, using the second treatment required more time than using the first treatment, and resulted in higher quality. Specifying the BC problem, a less complex problem, using the second treatment required less time than using the first treatment, and resulted in a specification with the same level of quality. This observation suggests that requirements engineers should write a formal specification of complex systems that are not well-understood to benefit from this second-system effect.

3. There is no evidence to suggest that the choice of language has an effect on the time required to modify a specification. The total time required to write and modify a specification, and the total time required to repair defects appears to be affected more by the treatment–problem interaction and the learning effect than by the treatment effect alone.

Chapter 4

Recommendations for Future Experiments

Based on the subject's experience with the pilot study, this chapter provides suggested improvements to the single-subject experimental design, and provides recommendations for executing a fully controlled experiment that compares specifications written in different languages based on the cost of making modifications. The study presents also possible alternative designs to make a fully controlled experiment more manageable.

4.1 Experiences with the Single-subject Pilot Study

The single-subject experiment was intended to be an inexpensive method of screening the fully controlled experiment described in Section 2.3 for potential effects that could be explored in more detail for a future fully controlled experiment. This pilot study ended up to be rather expensive due to the amount of time required to specify the BDF problems, although the author believes that the cost was still less than the cost of a fully controlled experiment.

Based on the subject's experience, the following recommendations may improve the execution of a single-subject experiment.

Use stricter internal deadlines. Deadlines would help keep the subject motivated and would prevent a specification from taking too long. A deadline should be self-assigned by the subject, based on an estimate. Completeness of the specification should still be a priority. A specification's deadline should be allowed to slip if the problem is doable; every deadline and every slip should be recorded in the tracking log.

Use long wash-out periods. The wash-out periods in this study were approximately one week long, but were not strictly measured. If this experiment were to be repeated,

the wash-out periods should be a standard length in between every run. One week did not seem sufficient, and two weeks in between runs may be better in attempting to reduce the learning effects.

Use tracking guidelines. Implementing guidelines for time tracking may make the process easier and more consistent. Tracking is something that a subject can improve at; over time, the tracking precision improves. Specific recommendations for improved time tracking are discussed further in Section 4.2.

Repeat the single-subject experiment as a replication. In the pilot study, the subject did one run of each possible combination of treatment and problem. In the pilot study, it was important to do every run at least once to report on the design of the fully controlled experiment. The runs were not randomized because there was a possibility of determining cause-and-effect due to order. This could only be done on the assumption that the learning effect due to language is negligible, and even then, the learning effect was confounded with the interaction effect. If the results are important, but no controlled experiment can be run, then an experimenter may wish to consider running a replicate.

An **A-B-A design** would provide stronger results than just an A-B design. The A-B-A design means that one applies the control treatment A, then experimental treatment B, and finally applies treatment A again. The subject performs the same task for each run. The final application of treatment A can measure the extent to which a learning effect carries from A to B, and from B to A. If the response variable increases from A to B, and then decreases from B to A, the suggestion would be that applying treatment B has an effect on the tasks performed in the experiment. If the response variable increases from A to B, and then increases again when doing A, then there is a learning effect and the effect of B is not necessarily strong. The A-B-A design is highly recommended as a design for future single-subject experiments. Unfortunately, due to the limited amount of time in the pilot study in which to do eight runs, the control treatment could not be replicated.

4.1.1 Running Future Single-Subject Experiments

Single-subject experiments are a valid method of exploring techniques and methods. Instead of using a proof-of-concept to establish the usefulness of a new software development technique, a researcher may want to consider using the A-B-A design. The researcher can establish the research hypothesis, track data, and analyze the results to provide insight into the costs and benefits of using this technique.

4.2 Recommendations for a Fully Controlled Experiment

Executing the pilot study has brought to light some of the problems that may occur when running a fully controlled experiment. The issues that should be considered when designing and running a fully controlled experiment are

- defining a target population,
- choosing a problem appropriate for use with the specification languages,
- evaluating specification quality,
- evaluating subject efficiency,
- improving time-tracking, and
- reducing the experimenters' influence.

Defining a Target Population

Defining the target population for a fully controlled experiment is important in order to ensure external validity. The subjects that would be selected for a fully controlled experiment must be a representative sample of the population that the experiment's results apply to.

In a fully controlled experiment, subjects need to be educated in how to use the specification languages to write a specification. If every subject has learned both RT-UML and SCR, then this should be indicated as a part of the sample's characteristics.

In this study, the ideal population are industrial practitioners, especially those who define requirements and communicate them to members of a software team. If this population can not be sampled from, then another population, such as students, may have to be used.

Choosing a Problem Appropriate for use with the Specification Languages

To help the fully controlled experiment run smoothly, we want to choose a problem that is appropriate for use with the specification languages. Although a requirements engineer working on a real-world system usually does not get to choose the problems that he is allowed to work on, he has significantly more control over which formal specification notation to use. If the requirements specification language does not match the problem, the language can inhibit the specification process.

For a formal method to be useful, it must have a limited domain of applicability. It is pointless to look for a "universal specification technique" [43]. Just as a population can be defined for the subjects that an experiment applies to, it should be possible to define the population of problems that the experiment applies to. If this is the case, then a

technique shown to work well for a *representative sample* of the problems that represent the population should work well for every problem in this population. By limiting the problems used in the fully controlled experiment, the experiment may lose some external validity, but this is why a population of problems should be explicitly defined; defining a sample of problems avoids misrepresenting the experiment. However, once one population of problems has been defined, it may be worth considering if a specification language can be applied to different populations of problems, as well.

Based on the subject's experience with the pilot study, it is extraordinarily frustrating to specify a non-event driven system using languages built for event-driven systems. Such frustration would likely demotivate subjects in a fully controlled experiment. By providing an appropriate problem to specify using the selected specification languages, the experiment becomes less obstructive for the subjects, and increases a subject's satisfaction with the experiment's execution. A trade-off must be made between disgruntled subjects and external validity; it is the author's opinion that for a fully controlled experiment, it is more important to ensure that the majority of the subjects are not dissatisfied with the way the experiment is going.

Evaluating Specification Quality

As mentioned in Subsection 2.1, evaluating quality was difficult in the pilot study because the number of defects in each URS was not known before the experiment started. With a list of defects, the experimenters can determine each specification's quality, and consider this in the analysis. For instance, the experimenters should compare the quality of a specification with the time required to modify the same specification to see if there is a correlation.

To evaluate specification quality, assemble a master list of known defects for each URS, and kept this list secret from the subjects. When every subject finishes the specification, each specification is analyzed and the defects discovered in the URS are compared with the master list. An estimation of the specification's quality can be made based on the number of defects discovered, the types of defects discovered, and the number remaining in the system. An evaluation framework for defects that is worth investigating is the one used by Kamsties, et al [29].

Evaluating Subject Efficiency

A metric that better indicates efficiency than the DC or the DRT is the percentage of the total time that is spent repairing defects during a specification. The DC and the DRT will still need to be measured to compute this number, but the hypothesis should be adjusted to such that the criteria for evaluation is the DRT divided by the time required to complete the specification. Another metric to consider using is the average of the DRTs for all defects, which measures the average cost required to repair each defect.

Improving Time Tracking

Time-tracking is exposed to a large amount of bias, or differences between subjects. One subject may consider monitoring only active time used to write a specification, whereas another may consider monitoring both active time and **passive time**, such as thinking about a specification in the shower. The problem with time tracking is not necessarily how accurately a single person can track his or her time among multiple sessions, but rather, how two different people track their time such that they are measuring the same tasks consistently.

Another issue with time tracking is that the tools are not very supportive of the personal time-tracking process. It was easy to make errors when recording defects and the time used for specification. A tool to help one monitor and analyze time would go a long way in helping developers track time, which is not only useful for controlled experiments, but also useful for cost tracking in general.

The following recommendations may help time tracking to be more consistent:

- **Measure active time only, not passive time.** The reasons for this are twofold. First, it is virtually impossible to measure accurately the amount of passive time devoted to a specification. Second, time spent on passive time is usually translated into active time. For instance, if a subject is sitting in her office thinking about a specification with a pencil in her hand, she would mark that as time that she is devoting to the specification. If she later takes a shower and thinks about the problem she is doing, and comes up with a new idea, then she will probably want to rush to her desk to write down her idea. In this case, as soon as the subject reaches her desk, she would start measuring the time that she is devoting to a specification.

  Another possibility is to have the subjects log passive time and active time separately.

- **Remove nested defects.** Due to limitations in the *pplog* tool, the subject must cope with nested defects. A **nested defect** is a defect that is discovered while the subject is already working on an existing defect. When using *pplog*, only one defect can be repaired at a time, and defects must be repaired in a last-in, first-out order. Instead, a subject should be free to enter a defect into his or her log as soon as it is discovered, and then decide to resume repairing a defect that is still open in the system. A subject should also be able to choose to not work on a defect, and instead decide to work on adding new functionality to the system. To solve this issue, better tracking tools are needed.

- **Record all interruptions.** Each duration of time that the subject spends away from his or her specification work should be logged as an interruption. If someone interrupts specification work at the subject's desk, if a subject get up to use the washroom, or if someone calls a subject about a topic unrelated to the specification, the event should be marked as an interruption.

If each subject receives identical guidelines, then measurement errors due to bias will be reduced. However, even if these guidelines are followed, there will still be measurement errors due to a lack of precision by each subject. No experimental design can completely remove measurement errors, but randomization can help reduce their effect.

Reducing the Experimenters' Influence

The experimenter's influence has a profound impact on an experiment [11]. One way to reduce experimenter influence is to keep communication between the experimenters and the subjects to a minimum. One way to reduce analysis bias is to use multiple analysts who independently examine the data and report their results. Automated analysis can also reduce experimenter bias.

4.3 Alternate Experiment Designs

Conducting a Fully Controlled Experiment in a Short Session

To circumvent time-tracking issues and external influences, a fully controlled experiment can be executed in a single session, during which subjects execute the experiment's tasks in one sitting. A fully controlled experiment that involves a single session or small number of sessions is easier to control than an experiment that permits subjects to work on his or her own time.

The external validity of a fully controlled experiment executed in a single session is far less than the external validity of an experiment executed over a longer period of time. A requirements specification that is small enough to do in a few hours is not considered representative of industrial systems. Although it can also be argued that the specifications completed in the pilot study were also not representative of industrial systems, a system that takes 33 hours to complete—as was the case for the BC IS written in SCR—better represents a large software system than a system that can be completed in a few hours. In addition, most requirements engineers do not write software specifications in such a controlled environment.

Another issue with a fully controlled experiment in a short session is that the time required to complete the experiment is bounded. The time can never be longer than the total time allocated to the length of the session, and therefore, each specification written by a subject must be analyzed for quality. The fully controlled experiment ends up measuring the degree to which a specification language helps a subject detect defects.

Finally, if time is not tracked during a short session, it is impossible to determine the amount of time each subject spends repairing defects during a specification task.

Despite the limitations of conducting a fully controlled experiment to compare the costs of modifying different specifications in a short session, it is still in the experimenter's interest not to have an experiment run for too long. Measurement errors would be reduced if the fully controlled experiment were conducted in one session. External influences from

peers or other sources would also be reduced. The experimenters may have an easier time finding subjects who can commit to a short session as opposed to finding subjects willing to give up twenty or thirty hours to do a fully controlled experiment across a longer time frame.

Reducing the Time Required to Run a Fully Controlled Experiment

Whether a fully controlled experiment is performed in a single session or not, it is in the best interest of the experimenters and the subjects not to let the experiment go on for too long. An experiment that is too long is expensive, and is difficult to control. The following suggestions may reduce the amount of time required to run a fully controlled experiment.

Provide no tool support. Another method of reducing required time is not to provide tools to subjects. This reduces the required amount of time required to complete a specification, but will increase the number of errors found. The specifications are manually verified against test cases designed before the experiment. Much of the time in the pilot study involved using the CASE tools.

Such an experiment would resemble an exam for a software engineering course. A threat to internal validity would be the ability of the experimenters to manually verify the specifications in a method that is fair. A threat to external validity is the fact that few formal specifications are written without the aid of tools.

Ask subjects to complete specification modifications only. Instead of asking each subject to complete an initial specification based on original requirements, and then asking each subject to modify the original specification that he wrote, instead, give half of the subjects a specification written in one treatment, and give the other half a specification written in the other treatment. Both specifications should be functionally identical. Each subject would also receive a copy of the URS. There should be no conflicts between the specification and the URS.

Every subject should be allowed a fixed amount of reading time so he can familiarize himself with the provided specification and the URS. Once this reading phase ends, provide every subject with an updated natural-language description of the problem, and instruct the subjects to update their specification to match the new natural-language description.

The reading phase should separate much of the time spent learning the specification and the URS from the time required to do modifications. Tt would be logical to believe that providing some time beforehand to let subjects read the IS would reduce the time they spend learning the IS during the modification phase.

This style of experiment has the benefit of reducing the time required to run the experiment because the emphasis is placed on modifying the specification only. The

analysis is simplified, because the experimenters will not need to analyze both an IS and an SM. However, if the SCR IS and the RT-UML IS are not identical, then one potential threat to validity in this design is the influence of the IS on the subjects' performance.

4.4 Recommended Design for a Future Fully Controlled Experiment

The author recommends that a fully controlled experiment be executed over multiple days to increase external validity.

The alternative hypotheses would be adjusted to read as follows:

$H_\alpha 1$: The time required to modify a specification of a system written using a tabular notation to reflect a requirements change is less than the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_\alpha 2$: The percentage of time spent repairing defects when modifying a specification of a system written in a tabular notation to reflect a requirements change is less than the percentage of time spent repairing defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_\alpha 3$: The average defect-repair time when modifying a specification of a system written in a tabular notation to reflect a requirements change is less than the average defect-repair time of the same system written in a state-of-the-practice notation to reflect the same requirements change.

Given the above alternative hypotheses, a fully controlled experiment would test the following null hypotheses.

$H_0 1$: There is no difference in the time required to modify a specification of a system written using a tabular notation to reflect a requirements change and the time required to modify a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_0 2$: There is no difference in the percentage of time spent repairing defects when modifying a specification of a system written in a tabular notation to reflect a requirements change and the percentage of time spent repairing defects when modifying a specification of the same system written in a state-of-the-practice notation to reflect the same requirements change.

$H_0 3$: There is no difference in the average defect-repair time when modifying a specification of a system written in a tabular notation to reflect a requirements change and

the average defect-repair time of the same system written in a state-of-the-practice notation to reflect the same requirements change.

Before subjects are provide with a specification, they are educated in the time-tracking methods and the specification languages used in the experiment.

When the experiment starts, half of the subjects would be randomly assigned to the control group, and would use RT-UML as the control language. The other half of the subjects would be assigned to the experimental group, and would use SCR as the experimental language. Every subject would get one copy of an identical URS, along with a digital copy of the initial specification written in the language assigned to his or her group. The subjects are given a period of time to read and understand the URS and the specification. During this time, subjects must track the time they use when reading and understanding the artifacts.

After the reading phase deadline, subjects would then receive a new URS that proposes a requirements change to the existing URS, and each would be ordered to modify the specification received to reflect the updates using the tool appropriate for his or her group. When the specification's modifications are complete, the experimenters would collect the specifications and the time logs for analysis.

This experiment design increases internal validity by controlling for variation in tracking methods. Because there are replications of each specification, statistical tests can be used to compare the performances of the control group and the experimental group. By using this design, we can give a URS with higher external validity to the subjects, in addition to allowing them to use tools.

Chapter 5

Conclusions

Data on the cost of writing a formal software-requirements specification can be found within literature, but there is little work done on the cost of modifying a formal software-requirements specification in response to requirements changes. The pilot study described in this thesis measures the cost of writing and maintaining software-requirements specifications in two different languages, and provides a comparative analysis of two different specification languages, Software Cost Reduction and Real-time Unified Modeling Language.

5.1 Results of the Pilot Study

Some preliminary conclusions can be drawn from the pilot study. These trends are not intended to be conclusive statements because of the limitations of the pilot study and the threats to validity. All of the results in this experiment must be confirmed with further research.

- There is no conclusive evidence that shows that there is an effect due to the treatment language on the cost of specification maintenance.
- There is evidence that suggests that the fact that one language worked better than the other when writing the initial specifications for a problem does not imply that the same language will work better than the other when writing the specification modifications for changes to the same problem.
- There is evidence that suggests that the fact that one language worked better than the other when writing the specification modifications for a problem does not imply that the same language will work better than the other when writing the specification modifications for changes to the same problem.
- There is evidence that suggests that because one language worked better than the other when writing the specification modifications for a problem does not imply

that the same language will work better than the other when writing the initial specification for changes to the same problem.

- Using RT-UML to write an initial specification appears to be more efficient than using SCR for the problems used in this research. The percentage of the total specification time spent on repairing defects when writing the initial specification in RT-UML was less than the percentage of the total specification time spent on repairing defects when writing the initial specification in SCR.

- Defects require more time on average to repair when modifying specifications than when writing initial specifications.

- When specifying a large, complex problem a second time, using a different language, the requirements engineer is more likely to learn about functionality that was omitted in the first specification. As a result, the time required to write the second specification may be greater than the time required to write the first specification, but the second specification may be of higher quality because it contains functionality omitted from the first specification.

- When specifying a small, simple problem for the second time, using a different language, the requirements engineer is not as likely to learn about functionality that was missed when writing the first specification. As a result, the time required to specify the second specification may be less than the the time required to specify the first, and the two specifications are less likely to vary.

5.2 Improvements for a Fully Controlled Experiment

If an experimenter were to do a fully controlled experiment to explore the effect of specification languages on the cost of modifying specifications, the following advice should be taken into consideration.

- The problem's effect is strong. The problem's effect on the time is likely stronger than the specification language's effect on the time, so an appropriate problem should be chosen for a fully controlled experiment.

- Time tracking and defect tracking depend heavily on the subject, and will have to be controlled. If a short session is not used for tracking the time in an experiment, then a presentation of time-tracking guidelines is recommended. Appropriate tool support for tracking will also benefit a fully controlled experiment.

- If a short session is used for an experiment, and if each subject does not track time, then the experiment will not be able to track the amount of time required to repair defects.

- A URS should not have defects seeded into it after the URS is written. Instead, defects that would be unknowingly injected into the URS during the writing should not be removed. This URS will be provided to the students. A master list of every defect in the URS must be created before the fully controlled experiment begins. This list is used to compare the quality of the specifications written and modified by the subjects.

- To keep the duration of a fully controlled experiment manageable, each subject should be provided with a URS and a formal specification in his assigned treatment. Each subject receives a fixed amount of time during which he must familiarize himself with the URS and the specification. After the time expires, each subject modifies the formal specification in his or her assigned treatment.

- There is no known estimation technique that can predict, based on a URS, the size of a required change. However, the amount of change made to a formal specification should be measurable. A metric to measure the size of a specification change may be able to provide a foundation on which to compare replications of the fully controlled experiment.

It is of the author's opinion that a fully controlled experiment should be conducted not as a single-session experiment, but instead over the course of a longer period. The reasons are as follows:

1. The result will be more interesting because of the fully controlled experiment's increased external validity.

2. Steps can be taken to reduce the threat of potentially inaccurate time tracking and defect tracking.

5.3 Experience with the Single Subject Experiment

In addition to reporting the results of the empirical work, this study also explores the use of a single-subject experiment and its role in software development. A single-subject experiment may at first glance seem deficient, but in fact it has a number of benefits.

- A single-subject experiment can explore a research question in a methodical way in an attempt to determine if there are effects worth investigating, and provide preliminary results.

- A single-subject experiment can allow a subject to work on research that otherwise would be difficult to do in a fully controlled experiment, like process-improvement techniques.

- A single-subject experiment can allow the use and creation of large artifacts that may not be feasable in a multiple-subject experiment, resulting in a conclusion that has higher external validity.

- A single-subject allows a user to track time and defects to determine the cost of the task and subject efficiency, thus providing more than anecdotal evidence of the effectiveness of a technique.

- A single-subject experiment allows one to be able to debug experiment designs.

- A single-subject experiment is inexpensive and involves less risk than a fully controlled experiment.

One cannot generalize the results of one single-subject experiment. However, if a database of empirical software-engineering literature is assembled, multiple single-subject experiments may be able to highlight important observations and trends that can be later explored further using fully controlled experiments.

5.4 Future Work

To confirm that the effects observed are in fact reproducible, replications of this study are encouraged. One improvement on the accuracy of this pilot study would be to run replications.

To confirm the results of the pilot study, we would conduct a fully controlled experiment with multiple subjects. This can help determine more clearly if the choice of specification language has an effect on the cost of modifying a specification.

Another option is to explore instead the learning effect. The paradoxical learning effect in this experiment was discovered accidentally and was a side-effect of this experiment, but it provides some more evidence that there is a second-system effect, as explained by Brooks [10]. Brooks explained that after developing one system, the software engineer confidently develops a large second system with extra features. In the author's opinion, the BDF problem was not gold-plated, but rather, simply too complex to specify correctly the first time. It was obvious after revisiting the problem that a large portion of the functionality was missing. However, for the BC problem, the second implementation of the same system resulted in a faster implementation. The first step is to confirm that the phenomenon observed in this pilot study can be reproduced. If the learning effect can be reproduced, then steps should be taken to detect which outcome will occur if a second specification is written—that is, whether the second specification will result in a specification of higher quality, but take more time, or whether the second specification will result in a specification that is functionally the same.

The initial results from this pilot study also pose the following questions:

- How can one measure the size of a requirements change, especially when given only a URS? Before one can compare the costs of modifications between two different problems, or multiple changes to one problem, the sizes need to be the same.
- It has been shown that if a language works well when writing an IS, it does not mean the same language will work well when writing an SM. Are the times required to write future SMs independent? That is, is it possible for one SM to work well with the language, and then for a second SM not to work well with the language?
- It has been shown that if a language works well when writing an IS, it does not mean the same language will work well when writing an SM. It appears that the cost of writing an IS and the cost of writing an SM are independent. However, the concepts of modularity, decomposition, and other similar strategies were developed explicitly to reduce the cost of modifications. Is it possible to reduce the cost of modifications by introducing different design mechanisms and styles?
- The experiment suggests that for the SM, the cost of repairing defects is more expensive than the cost of repairing defects during the IS. Is it the case that for each subsequent SM, the cost of repairing defects continues to increase beyond the cost of repairing defects for the previous SM?
- Would it be cheaper to write a new a formal specification for a problem when the requirements change, than to modify the existing specification? What criteria would one use to determine whether one should modify the existing specification, or write a new specification?

Exploring these questions in more detail may provide valuable insight into the nature of formal specification, especially under conditions of requirements change.

Appendix A

User Requirements Specifications

A.1 Bicycle Computer User Requirements Specification

A simple bicycle computer system consists of a sensor, a magnet, a wire, and a control unit. A magnet is attached to a spoke of the bicycle's wheel. The sensor is attached to the front fork, so that it detects when the magnet passes the fork, designating one complete rotation of the wheel. A wire connects the sensor and the computer's control unit, which is attached to the bicycle's handle bar. The wheel sensor sends a signal to the control unit with every rotation of the wheel.

The control unit displays the bicycle's current speed in the upper right-hand corner of the display. In addition, one other value, either elapsed time, maximum speed, distance, or average speed, is also displayed. This value is called the mode-function value, and which value is displayed depends on the current display mode. A mode symbol (TM, MXS, DST, AVS) indicates the current display mode, so that the user knows what mode-function value is being displayed. The mode symbol appears along the left side of the display, and the mode-function value appears in the lower right-hand corner of the display.

The control unit recalculates the following values every second, and updates the display for any displayed value:

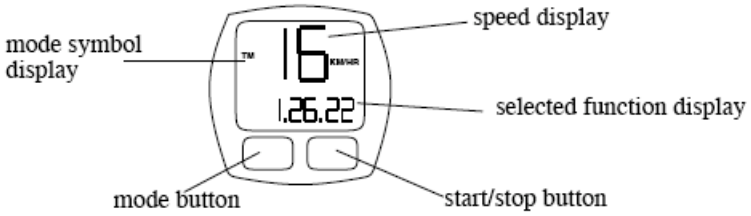


Figure A.1: Bicycle Computer User Interface

Speed The current speed, km/hr, is calculated using the previously stored value of the wheel size (the wheel circumference in cm) and the number of wheel rotations made in the last second.

Elapsed Time The elapsed time, in seconds, is measured from the start time (when the control unit was last RESET) to the current time, and is displayed in units of hours, minutes, and seconds. The display range is 0:00:00 to 9:59:59. When 10 hours have elapsed, the counter returns to 0.

Max Speed The maximum speed, km/hr, is the maximum of the current speed and the previously stored maximum speed.

Distance The trip distance, in 0.01 km increments, is measured from the starting point (when the control unit was last RESET) to the current point, using wheel size and the number of wheel rotations made in every second since the starting point. The display range is 0.00 to 999.9 km. When 1000 km is reached, the trip-distance counter returns to 0.

Average Speed The average speed, km/hr, is calculated on the basis of the elapsed time and the trip distance from the starting point to the current point. The average speed is measurable up to 9 hours 59 minutes 59 seconds for the elapsed time and up to 999.9 km for the trip distance. If either is exceeded, "E" is displayed and calculation ceases.

The bicyclist can interact with the bicycle computer by pressing two buttons on the control unit:

Mode Button The mode button is used to select which of the mode functions (elapsed time, maximum speed, trip distance, or average speed) will be displayed in the lower right-hand corner of the display. The initial mode (and the initial value displayed) is the elapsed time. Each time the mode button is pressed, the mode advances in the sequence: maximum speed, trip distance, average speed, elapsed time, maximum speed, etc.

Start/Stop Button The start/stop button is used to temporarily suspend and to resume calculation of the mode functions. Each time the start/stop button is pressed, the behaviour toggles between "computation active" and "computation suspended". When the bicycle computer is active, it updates the current speed and all mode functions every second. When computation is suspended, the mode functions' last values are saved in memory and are not updated. When computation is resumed, all of the functions are again updated every second, starting with the stored values.

When the mode and start/stop buttons are pressed at the same time, the control unit is RESET: all mode-function values are cleared and are reset to 0.

A.2 Bicycle Computer Updated User Requirements Specification

To upgrade the Bicycle Computer, so that it competes favorably against other products, the input to the computer will based on GPS coordinates rather than wheel rotations.

GPS works by:

- Identifying four reference satellites and their locations in space
- Calculating the distance to each satellite by computing how long it takes to receive a radio signal from each of them
- Using the locations of the satellites and the current distances from each of them to determine your (the receiver's) location on earth.

Fortunately, much of the above functionality is provided by other off-the-shelf components, which are responsible for finding the reference satellites, computing the distances to the reference satellites, and transforming this information into coordinates on Earth, i.e., longitude, latitude, and altitude. Thus, the bicycle computer receives, once every second, new coordinates representing the current location of the computer. Use this input to update the various functions: speed, distance travelled, time elapsed, average speed, and max speed. The mode button and start/stop button work as before. There is no AC button.

A.3 Bidirectional Formatting User Requirements Specification

Please see the following page for the Bidirectional Formatting URS. The page number at the bottom of each page corresponds to the page numbers of the original document provided to the subject. They have been retained because the URS refers itself using page numbers.

Bidirectional Formatting

Customers:
Daniel M. Berry דניאל ברי دانيال بيري
Dana Mohaplova Дана Мохаплова

Background

The property that the Arabic, Hebrew, Persian, and Urdu languages have in common is that they are written from right to left. Nearly all* of the languages of Europe are written from left to right. It is not too much of a simplification to consider each language to have its own alphabet. Even when two languages, e.g., English and French, seem to share the same alphabet, i.e., Latin, one may not use all the letters, and they may have differences in the use of accents and diacriticals. For example, French does not use "w" except for foreign words. French uses the accents "´", "`", "^", and English does not, except when it has borrowed words from other languages. One can regard the French alphabet as being the set of all possibly accented letters, i.e., including "a", "á", "à", "o", and "ô". Therefore, instead of talking about languages, we shall talk about alphabets. The Arabic, Hebrew, Persian, and Urdu (AHPU) alphabets are called *right-to-left* (RL) alphabets and their characters are called RL characters. The European alphabets are called *left-to-right* (LR) alphabets and their characters are called LR characters.

Mixed text is text consisting of both LR and RL characters from both LR and RL alphabets. The treatment of mixed text is interesting. Consider the mixed text line,

(1v) Dana said, "سلام دانيال ، שלום דניאל" to Daniel.

The label "(1v)" is not part of the line. For those of you that cannot read AHPU, Line 1v with the AHPU characters rendered phonetically in script-like Latin characters is

(2t) Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

Each AHPU character is represented by potentially several consecutive Latin letters, the first of which is upper case and the rest of which are lower case. Each AHPU letter is a consonant, a vowel, or a consonant *with* a vowel. Thus "س" is rendered as "*Sa*".

What is interesting about the mixed text is that the AHPU characters are read from right to left. However, since the line is embedded in an LR document, the general flow of the line is from left to right. A *chunk* is a maximal length subsequence of characters all of the same direction. Thus, Line 1v can be regarded as having three chunks.

1. Dana said, "
2. سلام دانيال ، שלום דניאל
3. " to Daniel.

Observe that the first chunk comma, the two quotation marks, and the period are LR characters. The second chunk comma, that appears to be backwards and upsidedown with respect to the normal Latin comma, is an RL character. Note also that by the maximal length property of the chunks, consecutive chunks of a single line of mixed text are in alternating directions. For the example line, the directions of the chunks are LR, RL, and LR, in that order.

* While we know of no exceptions, we are not 100% certain that there are no exceptions, so we are being safe and saying "Nearly all"

The rule for reading a line of mixed text is that the line is broken into its chunks. If the document is an LR document, then the chunks are read from left to right. Thus, the Line 1v chunks are read in numerical order. Each chunk is then read in its own direction. Therefore, in Line 1v,

1. chunk 1, an LR chunk, is read from left to right,

2. chunk 2, an RL chunk, is read from right to left, and

3. chunk 3, an LR chunk, is read from left to right.

As a result of this reading rule, the order in which the characters are read is captured by the following rendition of Line 1v.

(1t) Dana said, "לאינד סולש ،لايناد مالاس" to Daniel.

Line 1t is said to be in *time order*, while Line 1v is said to be in *visual order*. In the time-ordered rendition, each character is laid out from left to right in the order that one hears them spoken as someone is reading the visual order rendition according to the reading rule.

Thus, if we use the phonetic representation of AHPU characters, the time-ordered Line 1t corresponds to

(2t) Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

while the visual-ordered Line 1v corresponds to

(2v) Dana said, "*LEYNaD MOLahS ,LAYNAD MaaLaS*" to Daniel.

From now on, whenever we show RL characters in the script-like Latin characters, we will print them in visual order from right to left, unless we are trying to show the time ordering of the characters.

Processing Mixed Text

Have you ever wondered how web sites with AHPU text work? Clearly, the input to create a file should come in time order, because time order is the order in which one hears or thinks the characters, and if one is typing as one hears or thinks of the text, it is helpful if he or she can type the characters as he or she hears or thinks them. Equally clearly, when the contents of a file are displayed or printed, the characters should be displayed or printed in visual order, because if the characters are in visual order, then a reader following the reading rule will reconstruct the time ordering of the characters as he or she pronounces or thinks what he or she has read. The question is, "When is the best time to convert from time order to visual order?" The choices are "during input" or "during output". After years of trying different options, there is near universal agreement that the conversion should be done during output. After we have seen the algorithm, we shall explain why conversion should be done during output.

Consider now Lines 1t and 2t formatted to a shorter line length.

(1vs) Dana said, "שלום ، دانيال سلام
דניאל" to Daniel.

(2vs) Dana said, "*,LAYNAD MaaLaS*
LEYNaD MOLahS" to Daniel.

These are the desired visual ordered outputs. The lines to be put into visual order are the time-ordered

(1t) Dana said, "سلام دانیال، שלום לאינד" to Daniel.

(2t) Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

If we format these time-ordered lines into the desired line length, we get

(1ts) Dana said, "سلام دانیال، שלום
לאינד" to Daniel.

(2ts) Dana said, "*SaLaaM DANYAL,*
ShaLOM DaNYEL" to Daniel.

If you compare the formatted visual-ordered lines with the formatted time-ordered lines, you will see that for any line number *n*, line *n* of the formatted visual-ordered lines has exactly the same characters as line *n* of the formatted time-ordered lines, albeit in a different order! Permuting the characters on a line does not change the width of the line, because the sums of the widths of the characters in different permutations of the characters are the same. Within each line, the way to get from the time-ordered line to the visual-ordered line is to take each RL chunk in the line and reverse the order of its characters while preserving the order of the chunks. (Please verify from the two lists of formatted lines that this reversal of RL chunks has the desired effect.)

This particular trick of reversing the contents of the RL chunks works because the lines in question form an LR document. That is, the direction of the whole text is from left to right. Suppose we had an RL document. Consider

(3vs) Hello Daniel, bonjour" ,דנה אמרה
Daniel„ لدانيال.

(4vs) Hello Daniel," *,HaRMA HaNaD*
.LAYNADL " bonjour Daniel

These lines are right justified because they constitute an RL document. The time-ordered input for these lines is

(3t) הרמא הנד ,"Hello Daniel, bonjour Daniel„ لدانيال.

(4t) *DaNaH AMRaH, "* Hello Daniel, bonjour Daniel*" LDANYAL.*

This input formatted to the same line length as the visual-ordered output above is

(3ts) הרמא הנד ,"Hello Daniel, bonjour
Daniel„ لدانيال.

(4ts) *DaNaH AMRaH, "* Hello Daniel,
bonjour Daniel *" LDANYAL.*

Here again, for any line number *n*, line *n* of the formatted visual-ordered lines has exactly the same characters as line *n* of the formatted time-ordered lines, albeit in a different order. Within each line, the way to get from the time-ordered line to the visual-ordered line is to first reverse all the characters in the line. Then, in the reversed line, take each LR chunk in the line and reverse the order of its characters while preserving the order of the chunks. (Please verify from the two lists of formatted lines that this reversal of whole lines and then reversal of LR chunks

has the desired effect.)

Why Convert During Output

The main reason that we have found it best to convert from time order to visual order during output is flexibility for varying line lengths. Obesrver the strange effect of differing line lengths. We have seen Lines 1t and 2t formatted to one line length.

(1vs) Dana said, "שלום ، دانيال سلام
דניאל" to Daniel.

(2vs) Dana said, "*,LAYNAD MaaLaS*
LEYNaD MOLahS" to Daniel.

Here are the same lines formatted to a slightly longer line length.

(1vm) Dana said, "שלום דניאל ، دانيال سلام" to
Daniel.

(2vm) Dana said, "*MOLahS ,LAYNAD MaaLaS*
LEYNaD" to Daniel.

Compare Lines 1vs and 1vm. When the line length grew long enough to accommodate the entire RL chunk the word דניאל moved from the beginning, relative to the document's LR direction, of the second line to the end, relative to the RL chunk's RL direction, of the RL chunk in the first line, and that end of the RL chunk is at the left hand side of the chunk. This seemingly counter-intuitive move makes perfect sense when one considers the reading rule; that word דניאל is the last word of the RL AHPU chunk. The same observation can be made about the word *MOLahS* in Lines 2vs and 2vm.

Now suppose the lines were stored in visual order. It must be in visual order at some line length, because visual order depends on having lines of some length within which to permute the characters. In order to move text to its proper place when the output line length changes, it is effectively necessary to reconstruct the time ordering of the text in order to construct the correct visual ordering at the new line length. Time order is independent of line length, because we know that the beginning character of time-ordered Line *n*+1 immediately follows the last character of time-ordered Line *n* in the time ordering. Therefore, we store all time-ordered input in time order and convert to visual order only during output.

Another advantage of storing all text in time order is that it makes sorting easier. Regardless of the characters' visual order directions, the most significant character of each line with respect to the sort is at the same end of the line. The sorting algorithm does not have to take into account character directions, and it does not have to reverse text before comparing.

Thus, in conclusion, input is in time order, text is stored in files in time order, and conversion to visual order occurs during output.

Basic Algorithm

Based on these examples, it is possible to write a general algorithm that works on each line of a time-ordered file to convert that line into visual order. The algorithm needs to know the current document direction, which can be LR or RL. It needs to know also the direction of each character it finds so that it can break the line into single-directional chunks, each consisting of maximal length sequences of characters all of the same direction. At the top level of abstraction, the algorithm reads the characters of each line, and permutes these characters so that the single-directional chunks of a line flow in the current document's direction and the characters in each chunk flow in the chunk's single direction.

for each line in the file **do**
  if the current document direction is LR **then**
    reverse each contiguous sequence of RL characters in the line
  else (the current document direction is RL)
    reverse the whole line;
    reverse each contiguous sequence of LR characters in the line
  fi
od

This algorithm assumes that each line is in only one document direction. Thus, it will be necessary to assure either that document direction changes occur only at line boundaries or to decide that the document direction that applies for a line is that in effect at one well-defined point in the line, e.g., at the end, as measured by time order. We shall do the latter, taking as an output line's document direction the document direction that is current at the time-ordered end of its input.

This simple algorithm falls flat on its face when presented with an embedded LR numeral inside RL text inside an LR document, e.g., inside an English document, an AHPU address containing a Latin house number; the LR numeral splits the RL text into two pieces and the two pieces end up being in the document's LR order relative to each other rather than the required RL order relative to each other. This anomaly is prevented by applying the algorithm recursively on the RL text. To be concrete, without this provision, the time-ordered input

(5t) Daniel lives at سלام 4915 שלום in a beautiful house.

(6t) Daniel lives at *SaLaaM* 4915 *ShaLOM* in a beautiful house.

would appear as

(5i) Daniel lives at سلام 4915 שלום in a beautiful house.

(6i) Daniel lives at *MaaLaS* 4915 *MOLahS* in a beautiful house.

instead of the correct

(5v) Daniel lives at שלום 4915 سلام in a beautiful house.

(6v) Daniel lives at *MOLahS* 4915 *MaaLaS* in a beautiful house.

Note that the logical ordering of the house number is "4-9-1-5", and that this number must come *after* the name of the street, سلام and *before* the the name of the city שלום in the RL flow of the AHPU text that is embedded in an English sentence in an LR document. In the incorrect version, the LR number in the midst of the RL address has the effect in an LR document of causing the address not to be treated as a single RL unit, but to be treated as two RL

chunks embedded inside an LR document and to be printed in LR order with the first RL chunk, سلام or *MaaLaS*, to the left of the second RL chunk, שלום or *MOLahS*.

This effect is exacerbated if inside the LR numeral is some RL text, e.g., as to give an address number, a building name, and an apartment number.

(7t) Daniel lives at سلام 49בא15 שולש in a beautiful house.

(8t) Daniel lives at *SaLaaM* 49*AB*15 *ShaLOM* in a beautiful house.

would appear as

(7i) Daniel lives at سلام 49אב15 שלום in a beautiful house.

(8i) Daniel lives at *MaaLaS* 49*BA*15 *MOLahS* in a beautiful house.

instead of the correct

(7v) Daniel lives at שלום 15אב49 سلام in a beautiful house.

(8v) Daniel lives at *MOLahS* 15*BA*49 *MaaLaS* in a beautiful house.

The logical ordering of the address number, building name, and apartment number is "4-9-*alef-bet*-1-5". It must be printed as 15אב49 because it is part of an AHPU address whose flow is right to left. Furthermore, this address number, building name, and apartment number must come *after* the name of the street, سلام and *before* the the name of the city שלום in the RL flow of the AHPU text that is embedded in an English sentence in an LR document. In the incorrect printing, the fact that 49 and 15 are two LR chunks embedded within three RL chunks in an LR document causes the 49 and 15 to be printed in LR order instead of the correct RL order.

An analogous problem would happend with an embedded RL numeral inside LR text inside an RL document. However, in practice, this problem does not arise because in AHPU, with the exception of an ancient form of numerals in Hebrew, numerals are written with the most significant digit to the left, as in English (and other LR languages).

There are two solutions to this anomaly.

1. Consider the entire AHPU address to be an RL document embedded within the enclosing RL document and apply the algorithm recursively on the address to put it into visual order as a unit. Then treat this visual-ordered unit as an RL chunk in the embedding lines. However, this approach violates the requirement stated earlier that each line have only one document direction.

2. Treat each LR numeral embedded within RL text differently, that is, put it into LR order, but consider it after setting its printing order as RL text.

You will provide only the second solution.

The last section of this document gives all the example lines formatted to a variety of line lengths to allow you to see all the nuances of the effect of the conversion algorithm.

Finally, note that this document was typeset with software written by the author and his students that implements the algorithm described above within the context of a full-function formatter. This software is described by the documents referenced in item 6 of the sources of information listed below.

Your Assignment

For the basic assignment, you will use lower case Latin letters as LR characters and upper case Latin letters as RL characters. Punctuation, digits, and whitespace will be in both; which direction an instance of these is in will be made clear from the markup, described below, embedded in the text.

We will use a variant of SGML markup (similar to HTML markup) to give semantic information in the files. This markup is case sensitive.

Begin	End	What
<LRD>	nothing	begin LR document region
<RLD>	nothing	begin RL document region
<lr>	nothing	begin LR text region
<rl>	nothing	begin RL text region
<lrn>	</lrn>	surround LR numeral for use inside RL text
<rln>	</rln>	surround RL numeral for use inside LR text

Each of the markups <LRD>, <RLD>, <lr>, and <rl> applies from where it appears until it is overriden. Each of <LRD> and <RLD> overrides the other. Each of <lr> and <rl> overrides the other. It is assumed that there is a
 before each occurrence of <LRD> or <RLD>. See below for a description of
.

The initial document direction is LR, and the initial text direction is LR. That is, your program may assume that the entire input is preceeded by <LRD><lr>.

<lrn> and </lrn> come in pairs and <rln> and </rln> come in pairs. None of these pairs may be nested inside another. When the program finds a violation, it should stop outputting, issue an appropriate error message, and terminate.

The program does not check that between <lrn> and </lrn> and that between <rln> and </rln> are only numerals. In other words, these markups can be used to achieve their effect on any text. <lrn> and </lrn> have no effect when they are nested inside LR text. Conversely, <rln> and </rln> have no effect when they are nested inside RL text.

In addition, some basic HTML-like markup may be used to control formatting:

Begin	End	What
		text to be centered, each contained output line is centered on output; there is an assumed before and after
 	nothing	break, i.e., end current output line and start a new output line
<sp>	nothing	space, i.e., end current output line, output a blank line, and then start a new output line
<p>	nothing	paragraph, i.e., end current output line and start a new output line indented 5 spaces

No and pair may be nested inside another and pair.

Using all these conventions one could put Lines 2t, 4t, 6t, and 8t in one file as follows:

(9t)
```
<LRD><lr>
dana said, “<rl>SALAAM DANYAL, SHALOM DANYEL<lr>” to daniel.
<sp>
<RLD>
<rl>DANAH AMRAH, “<lr>hello daniel, bonjour daniel<rl>” LDANYAL.
<LRD>
<sp>
<lr>daniel lives at <rl>SALAAM <lrn>4915</lrn> SHALOM<lr> in a
beautiful house.
<sp>
<lr>daniel lives at <rl>SALAAM <lrn>49</lrn>AB<lrn>15</lrn> SHALOM<lr>
in a beautiful house.
```

The first of these input lines could be omitted.

After conversion to visual order, these three lines would appear as

(9v)
```
dana said, “LEYNAD MOLAHS ,LAYNAD MAALAS” to daniel.

                .LAYNADL ”hello daniel, bonjour daniel“ ,HARMA HANAD

daniel lives at MOLAHS 4915 MAALAS in a beautiful house.

daniel lives at MOLAHS 15BA49 MAALAS in a beautiful house.
```

The program is to be called bidifmt and it should have the command line interface of

bidifmt [**-l** *length*] [*filename* ...]

The optional -l *length* is to provide an output line length. If none is provided, the output line length is 72. The optional list of *filename*s is one way to specify the input. If these are provided the input is the concatenation, in order, of the contents of the files named by the list of *filename*s. If no list of *filename*s is provided, the input is whatever is supplied through the standard input. The output of the program is sent to the standard output.

The output is a version of the input with output lines filled as close to the output line length as possible and with all

sequences of consecutive white space characters (blank, tab, and newline) between two consecutive words on an output line replaced by one space.

If a single word is longer than the output line length, it is put on an output line by itself and that line is as long as it needs to be to accommodate the word.

In preparing the output, the embedded markup is obeyed but not output. Thus, the markup does not contribute towards determining the length of text that fits on an output line.

If the four lines of Example (9t) were stored in a file named input, then invoking the command

bidifmt -l 30 input

would yield the output

(9vs)
```
dana said, ",LAYNAD MAALAS
LEYNAD MOLAHS" to daniel.

                                    hello daniel," ,HARMA HANAD
                                       .LAYNADL "bonjour daniel

daniel lives at 4915 MAALAS
MOLAHS in a beautiful house.

daniel lives at 15BA49 MAALAS
MOLAHS in a beautiful house.
```

Note that the RL document lines are right justified in the output, and this right justification is exaggerated to make it more visible.

Sources of information:

1. A website with Arabic text: http://www.aljazeera.net/

2. A website with Hebrew text: http://www.haaretz.co.il/

3. How to view an Arabic Web site: http://www.aljazeera.net/help/arabic-instr-e.htm

4. How to view a Hebrew Web site: http://www.haaretz.co.il/hasite/pages/ShArt.jhtml?itemNo=56806

5. A website about the UNICODE standard for bidirectional text: http://www.unicode.org/reports/tr9/

6. Papers by Berry *et al* on bi-directional formatting:
   http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/buchman.berry.gonczarowski.pdf
   http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/arabic.journal.paper.pdf
   http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/keshide.journal.paper.pdf

7. Paper by Berry *et al* on bidirectional editing with vi.iv:
   http://se.uwaterloo.ca/~dberry/FTP_SITE/reprints.journals.conferences/vi.iv.journal.paper.pdf

8. Site about a bidirectional editor, Yudit: http://www.yudit.org/bidi/userguide.html

9. The UNIX fmt program: do man fmt

Examples

Below are the three examples presented in a variety of line widths so that you can see the effects of line breaks at different points on the reversing algorithm. In each case, the example is presented formatted to the given line length in time order and then it is presented in the same line length in visual order. First, we give the LR documents.

Dana said, "سלام دانیال، שלום לאינד" to Daniel.

Dana said, "سلام دانیال ، שלום דניאל" to Daniel.

Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

Dana said, "*,LAYNAD MaaLaS LEYNaD MOLahS*" to Daniel.

Daniel lives at سلام 4915 שלום in a beautiful house.

Daniel lives at سلام 4915 שלום in a beautiful house.

Daniel lives at *SaLaaM* 4915 *ShaLOM* in a beautiful house.

Daniel lives at 4915 *MaaLaS MOLahS* in a beautiful house.

Daniel lives at سلام 49בא15 שלום in a beautiful house.

Daniel lives at 15אב49 سلام שלום in a beautiful house.

Daniel lives at *SaLaaM* 49*AB*15 *ShaLOM* in a beautiful house.

Daniel lives at 15*BA*49 *MaaLaS MOLahS* in a beautiful house.

Dana said, "سلام دانیال، שלום לאינד" to

Daniel.

Dana said, "سلام دانيال ، שלום דניאל" to Daniel.

Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

Dana said, "*MOLahS ,LAYNAD MaaLaS LEYNaD*" to Daniel.

Daniel lives at سلام 4915 שלום in a beautiful house.

Daniel lives at שלום 4915 سلام in a beautiful house.

Daniel lives at *SaLaaM* 4915 *ShaLOM* in a beautiful house.

Daniel lives at *MOLahS* 4915 *MaaLaS* in a beautiful house.

Daniel lives at سلام 49בא15 שלום in a beautiful house.

Daniel lives at שלום 15אב49 سلام in a beautiful house.

Daniel lives at *SaLaaM* 49*AB*15 *ShaLOM* in a beautiful house.

Dana said, "سلام ،لاينداد ،לאינד שלום" to Daniel.

Dana said, "سلام دانيال ، שלום דניאל" to Daniel.

Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

Dana said, "*LEYNaD MOLahS ,LAYNAD MaaLaS*" to Daniel.

Daniel lives at سלام 4915 שולש in a beautiful house.

Daniel lives at שלום 4915 سلام in a beautiful house.

Daniel lives at *SaLaaM* 4915 *ShaLOM* in a beautiful house.

Daniel lives at *MOLahS* 4915 *MaaLaS* in a beautiful house.

Daniel lives at سلام 49בא15 שולש in a beautiful house.

Daniel lives at שלום 15אב49 سلام in a beautiful house.

Daniel lives at *SaLaaM* 49*AB*15 *ShaLOM* in a beautiful house.

Daniel lives at *MOLahS* 15*BA*49 *MaaLaS* in a beautiful house.

Dana said, "سلام دانیال، שולש לאינד" to Daniel.

Dana said, "سلام دانیال ، שלום דניאל" to Daniel.

Dana said, "*SaLaaM DANYAL, ShaLOM DaNYEL*" to Daniel.

Dana said, "*LEYNaD MOLahS ,LAYNAD MaaLaS*" to Daniel.

Daniel lives at سلام 4915 שולש in a beautiful house.

Daniel lives at שלום 4915 سلام in a beautiful house.

Daniel lives at *SaLaaM* 4915 *ShaLOM* in a beautiful house.

Daniel lives at *MOLahS* 4915 *MaaLaS* in a beautiful house.

Daniel lives at سلام 49בא15 שולש in a beautiful house.

Daniel lives at שלום 15אב49 سلام in a beautiful house.

Daniel lives at *SaLaaM* 49*AB*15 *ShaLOM* in a beautiful house.

Daniel lives at *MOLahS* 15*BA*49 *MaaLaS* in a beautiful house.

Then, we give the RL documents. Note that each time-ordered input is really an LR document. Thus the time-ordered inputs are left justified, but the visual-ordered outputs are right justified.

הנד הרמא, "Hello Daniel, bonjour Daniel" لاينادل.

דנה אמרה, "Hello Daniel, bonjour Daniel" لدانيال.

DaNaH AMRaH, "Hello Daniel, bonjour Daniel" *LDANYAL.*

,HaRMA HaNaD "Hello Daniel, bonjour Daniel" *.LAYNADL*

הנד הרמא, "Hello Daniel, bonjour Daniel" لاينادل.

דנה אמרה, "Hello Daniel, bonjour Daniel" لدانيال.

DaNaH AMRaH, "Hello Daniel, bonjour Daniel" *LDANYAL.*

,HaRMA HaNaD "Hello Daniel, bonjour Daniel" *.LAYNADL*

הנד הרמא, "Hello Daniel, bonjour Daniel" لاينادل.

דנה אמרה, "Hello Daniel, bonjour Daniel" لدانيال.

DaNaH AMRaH, "Hello Daniel, bonjour Daniel" *LDANYAL.*

,HaRMA HaNaD "Hello Daniel, bonjour Daniel"

.*LAYNADL*

הרמא הנד, "Hello Daniel, bonjour Daniel„ لدانيال.

דנה אמרה, "Hello Daniel, bonjour Daniel„ لدانيال.

DaNaH AMRaH, "Hello Daniel, bonjour Daniel"
LDANYAL.

"Hello Daniel, bonjour Daniel" ,*HaRMA HaNaD*
.*LAYNADL*

A.4 Bidirectional Formatting Updated User Requirements Specification

Please see the following page for the Bidirectional Formatting enhancements URS.

Bidirectional Formatting

Enhancement

Customers:
Daniel M. Berry דניאל ברי دانيال بيري
Dana Mohaplova Дана Мохаплова

Background

A few terms must be defined.

1. The *minimum text length* of an output line is the number of characters including spaces in the output line when the interword white space is exactly one space. Note that after the processing of an output line implied by the original specification of bidifmt is done (see first lines of page 9), the output line is at its minimum text length. This minumum text length is less than or equal to the output line's length unless the output line consists of one word whose length is longer than the output line length.

2. The *excess* of an output line is the difference between the output line's length and the minimum text length of the output line. This excess is positive unless the output line consists of one word whose length is longer than the output line's length.

Enhancement Behavior

The enhancement is to add to the bi-directional formatting program, bidifmt, the ability to achieve justification to the end of an output line in two possible manners:

1. *spreading* the words in the output line and

2. *stretching* the last stretchable character in the output line.

Justification happens to an output line only if the output line is generated by input that is in a region in which justification of either kind is requested. Justification can happen to an output line only if it has a positive excess. Therefore, in the discussion below, it is assumed that the output line at hand is generated by input in a region in which justification is requested and the output line has a positive excess. Moreover, only a line that has been filled and not ended with a, possibly non-explicit, `<br>` is justified. A line that is ended with a, possibly non-explicit, `<br>` is left unchanged.

In the first option, the words in an output line to be justified are spread with extra spaces so that (1) the lengths of the sequences of spaces between words differ by no more than 1 and (2) the last character of the last word on the output line is positioned at the end of the output line. That is, the white space between each pair of words is the single space extended by the excess divided by the number of interword gaps rounded to the resolution of the output device and possibly padded by 1 to make the sum of the extending spaces equal to the excess.

In the second option, the last, in the sense of the current document direction, stretchable letter in an output line to be justified is stretched by an amount equal to the excess. For each alphabet, there is a table listing its stretchable letters. Stretching a letter is achieved by passing the stretch amount as a parameter to the letter. If a letter is not stretchable, passing a stretch amount to it has no effect. If there is no stretchable letter anywhere in an output line, then the output line is justified with the spreading option.

Not doing any justification to an output lines is called leaving the output line *ragged*.

New Commands

The new commands are described in the table below:

Begin	End	What
`<Spread>`	nothing	begin region in which each output line is spread
`<Stretch>`	nothing	begin region in which the last character in each output line is stretched
`<Ragged>`	nothing	begin region in which output lines are not justified

Each of the markups `<Spread>`, `<Stretch>`, and `<Ragged>` applies from where it appears until it is overriden. Each of `<Spread>`, `<Stretch>`, and `<Ragged>` overrides any of the other. There is an implicit `<br>` before each occurrence of `<Spread>`, `<Stretch>`, or `<Ragged>`. The default is `<Ragged>`; that is, the input should be processed as if there were a `<Ragged>` before the beginning of the input.

Appendix B

System Specifications

The formal specifications written for this experiment are available at the following URL.
`http://www.cs.uwaterloo.ca/~ihkwan/`

- Bidirectional Formatter Initial Specification in SCR
- Bidirectional Formatter Specification Modifications in SCR
- Bicycle Computer Initial Specification in SCR
- Bicycle Computer Specification Modifications in SCR
- Bidirectional Formatter Initial Specification in RT-UML
- Bidirectional Formatter Specification Modifications in RT-UML
- Bicycle Computer Initial Specification in RT-UML
- Bicycle Computer Specification Modifications in RT-UML

Appendix C

Tables

This appendix presents the data that was computed from logs gathered during the specification process. The phases correspond to the phases presented in Section 2.8.1.

Each table, except for tables C.1, C.2, C.11, and C.12, report the data broken down by phase. The last row of the table reports the data for the specification, and is the sum of the other rows.

Table Headings

Run The run number in the pilot study.

Problem The problem specified in this task. It can be either the BDF problem, or the BC problem.

Task The writing, or the modifying of a specification. It can be either initial specification (IS), or specification modification (SM).

Treatment The treatment variable. It can be either SCR, or RT-UML.

Phase The phase. A list of phases is in Section 2.8.1.

Time The total time required to complete a phase or task.

Number of Phases The number of times this phases has been entered.

Average Time The average time duration of a phase or run. This is calculated by dividing the time by the number of phases.

Percentage of time Spent in Phase The percentage of time spent in this phase. It is calculated by dividing the time by the total time required for the specification task.

Defect Count The number of defects discovered in this task or phase.

Defect Repair Time The total amount of time required to repair every defect discovered in this task or phase.

Percentage of Time Repairing Defects The percentage of the total time spent repairing defects discovered during the task. It is calculated by dividing the defect-repair time over the total time required to complete the specification.

Average Defect-Repair Time The average defect-repair time in this task or phase. It is calculated by dividing the defect-repair time by the defect count.

Run	**Problem**	**Task**	**Treatment**	**Phases**	**Time (hours)**	**Time (minutes)**
1	BDF	IS	SCR	10	**55:25**	3325
2	BDF	IS	RT-UML	28	**114:44**	6884
3	BC	IS	RT-UML	13	**36:21**	2997
4	BC	IS	SCR	5	**33:09**	2758
5	BDF	SM	RT-UML	21	**45:58**	2181
6	BDF	SM	SCR	15	**49:57**	2997
7	BC	SM	SCR	6	**21:52**	1312
8	BC	SM	RT-UML	5	**10:00**	600

Table C.1: Time Required for Requirements Specification

Problem	**Treatment**	**Total Specification Time**	**Modification Time**	**Percentage of Modification Time**
BDF	SCR	105:22	49:58	**47%**
BDF	RT-UML	144:02	45:57	**28%**
BC	SCR	55:01	21:52	**40%**
BC	RT-UML	46:21	10:00	**23%**

Table C.2: Time Required for Complete Development Cycle

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	2	879	439.50	26.44
Specification	3	815	271.67	24.51
Test Design	2	553	276.50	16.63
Compilation	1	1	1.00	0.03
Testing	2	1077	538.50	32.39
Total	10	3325	332.50	100.00

Table C.3: Duration of Phases: BDF Initial Specification written in SCR

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	1	19	19.00	0.63
Test Design	4	632	158.00	21.09
Specification	3	878	292.67	29.30
Compilation	2	12	6.00	0.40
Testing	5	1456	291.20	48.58
Total	15	2997	199.80	100.00

Table C.4: Duration of Phases: BDF Specification Modifications written in SCR

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	4	1526	381.50	22.17
Specification	10	1745	174.50	25.35
Compilation	6	115	19.17	1.67
Inspection	2	184	92.00	2.67
Test Design	3	1114	371.33	16.18
Testing	3	2200	733.33	31.96
Total	28	6884	245.86	100.00

Table C.5: Duration of Phases: BDF Initial Specification written in UML

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	1	146	146.00	5.29
Specification	7	575	82.14	20.85
Compilation	5	62	12.40	2.25
Testing	5	1776	355.20	64.39
Test Design	3	199	66.33	7.22
Total	21	2758	131.33	100.00

Table C.6: Duration of Phases: BDF Specification Modifications written in UML

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	1	78	78.00	3.92
Specification	1	281	281.00	14.13
Compilation	1	58	58.00	2.92
Test Design	1	361	361.00	18.15
Testing	1	1211	1211.00	60.88
Total	5	1989	397.80	100.00

Table C.7: Duration of Phases: BC Initial Specification written in SCR

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	1	69	69.00	5.26
Specification	1	228	228.00	17.38
Test Design	2	356	178.00	27.13
Testing	2	659	329.50	50.23
Total	6	1312	218.67	100.00

Table C.8: Duration of Phases: BC Specification Modifications written in SCR

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	2	271	135.50	12.43
Specification	3	595	198.33	27.28
Compilation	3	25	8.33	1.15
Test Design	2	466	233.00	21.37
Inspection	1	157	157.00	7.20
Testing	2	667	333.50	30.58
Total	13	2181	167.77	100.00

Table C.9: Duration of Phases: BC Initial Specification written in UML

Phase	**Number of Phases**	**Time (minutes)**	**Average Time (minutes)**	**Percentage of Time Spent in Phase**
Modelling	1	24	24.00	4.00
Specification	1	147	147.00	24.50
Compilation	1	29	29.00	4.83
Test Design	1	88	88.00	14.67
Testing	1	312	312.00	52.00
Total	5	600	120.00	100.00

Table C.10: Duration of Phases: BC Specification Modifications written in UML

Run	Problem	Task	Treatment	Specification Time (minutes)	Defect Count	Defect Repair Time (minutes)
1	BDF	IS	SCR	3325	**111**	**956**
2	BDF	IS	RT-UML	6884	**176**	**1676**
3	BC	IS	RT-UML	2181	**107**	**793**
4	BC	IS	SCR	1989	**58**	**910**
5	BDF	SM	RT-UML	2758	**72**	**1450**
6	BDF	SM	SCR	2997	**99**	**1066**
7	BC	SM	SCR	1312	**36**	**523**
8	BC	SM	RT-UML	600	**19**	**247**

Table C.11: Defect Counts and Repair Times During Specification

Run	Problem	Task	Treatment	Percentage of Time Repairing Defects	Average Defect-Repair Time
1	BDF	IS	SCR	**29%**	**9.8**
2	BDF	IS	RT-UML	**24%**	**10.8**
3	BC	IS	RT-UML	**36%**	**20.1**
4	BC	IS	SCR	**46%**	**11.9**
5	BDF	SM	RT-UML	**53%**	**13.6**
6	BDF	SM	SCR	**36%**	**13.0**
7	BC	SM	SCR	**40%**	**12.0**
8	BC	SM	RT-UML	**41%**	**13.6**

Table C.12: Percentage of Time Repairing Defects and Average Defect-Repair Times During Specification

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Specification	24	24.00	87.63	3.65
Test Design	12	12.00	329.50	27.46
Testing	64	64.00	497.58	7.77
Total	100	100.00	914.72	9.15

Table C.13: Defects: BDF Initial Specification written in SCR

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Specification	1	1.02	14.37	14.37
Test Design	2	2.04	4.03	2.02
Testing	95	96.94	1036.68	10.91
Total	98	100.00	1055.08	10.77

Table C.14: Defects: BDF Specification Modifications written in SCR

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Compilation	32	18.93	46.17	1.44
Modelling	3	1.78	2.47	0.82
Inspection	4	2.37	68.53	17.13
Specification	16	9.47	440.92	27.56
Test Design	3	1.78	31.27	10.42
Testing	111	65.68	1083.68	9.76
Total	169	100.00	1673.03	9.90

Table C.15: Defects: BDF Initial Specification written in UML

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Compilation	7	9.86	5.92	0.85
Specification	7	9.86	199.80	28.54
Test Design	1	1.41	14.47	14.47
Testing	56	78.87	1229.58	21.96
Total	71	100.00	1449.77	20.42

Table C.16: Defects: BDF Specification Modifications written in UML

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Compilation	4	6.90	50.75	12.69
Test Design	3	5.17	10.67	3.56
Testing	51	87.93	848.60	16.64
Total	58	100.00	910.02	15.69

Table C.17: Defects: BC Initial Specification written in SCR

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Test Design	3	11.54	123.67	41.22
Testing	23	88.46	398.57	17.33
Total	26	100.00	522.23	20.09

Table C.18: Defects: BC Specification Modifications written in SCR

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Compilation	12	13.64	16.07	1.34
Modelling	4	4.55	75.17	18.79
Inspection	10	11.36	15.32	1.53
Specification	21	23.86	144.55	6.88
Test Design	13	14.77	136.50	10.50
Testing	28	31.82	393.50	14.05
Total	88	100.00	781.10	8.88

Table C.19: Defects: BDF Initial Specification written in UML

Phase	**Defect Count**	**Percentage of Defects In Phase**	**Defect Repair Time (minutes)**	**Average Defect-Repair Time (minutes)**
Compilation	1	5.26	27.95	27.95
Testing	18	94.74	219.08	12.17
Total	19	100.00	247.03	13.00

Table C.20: Defects: BDF Specification Modifications written in UML

Appendix D

Defect Classification

D.1 Defect Classification

Defects in this table have six components: the defect-repair time, the phase in which the defect is injected, the phase in which the defect is removed, the classification of the defect, the cause of the defect, the component that this defect was a part of, and a comment describing the defect.

The PSP encourages classifying defects. Humphrey's recommended classification, however, is specific to programming and is not suitable for requirements specification. Prechelt [36] recommends an alternate system for defect classifications which focus on interface usage, missing elements of code, and wrong elements of code. Prechelt's classification system is a suitable classification for coding, but is not appropriate for use in requirements specification. The tracking in this experiment uses a customized classification system based Prechelt's defect classification system. The classification system used is shown in Table D.1.

Type	**Defect Classification**	**Abbr**	**Description**
Requirements	Missing Requirement	mr	Required software functionality is missing in the requirements document.
	Ambiguous Requirement	ar	The software functionality described in the documentation has multiple interpretations.
Continued on next page			

Table D.1: Defect Classifications

Table D.1: Continued from previous page

Type	Defect Classification	Abbr	Description
	Inconsistent Requirement	ir	The software functionality described in the documentation contradicts another requirement in the documentation.
	Wrong Requirement	wr	The software functionality described in the documentation is not actually what the customer intended for the system.
Modelling	Missing Model	mm	Additions need to be added to the model to accomodate a requirement. (A requirement was omitted.)
	Wrong Model	wm	Large changes are required to the model to accomodate a requirement. (A requirement was misunderstood.)
	Bad Model	bm	The requirements and code are correct, but the model is too complex, non-modular, or is otherwise unsatisfactory and needs to be modified.
Specification	Missing Variable	mv	Input variable, output variable, or signal not present.
	Wrong Variable	wv	Wrong variables or bad names.
	Wrong Type	wt	Bad type, bad type-checking, bad range, or bad cast.
	Wrong Condition	wc	A bad condition in transitions.
	Missing Assignment	ma	A required assignment was missing.
	Wrong Assignment	wa	An incorrect value was assigned or an assignment occured when it should not have.

Continued on next page

Table D.1: Defect Classifications

Table D.1: Continued from previous page

Type	**Defect Classification**	**Abbr**	**Description**
	Missing statement	ms	A statement or keyword is missing.
Test case	Wrong Test Case	wtc	Test case parameters changed for clarity.
Test case	Wrong Test Case Output	wto	Expected output in the test case is incorrect.
	Wrong Test Case Input	wti	Input step in test case script is incorrect.
	Missing Test Case	mt	Test case that should be present is missing.
Miscellaneous	Clarification	cl	The change clarified existing information in the specification.
	No Error	ne	A defect was recorded when in fact there was none.
	Won't fix	nf	The defect will not be repaired.
	External	ex	The cause of the defect was external and beyond the subject's control.

Table D.1: Defect Classifications

D.1.1 Causes of Defects

In addition to marking the phase when the defect was injected, and the defect type, the PSP encourages that developers record why defects occur. Prechelt recommends the following defect causes [36] in Table D.2.

Classification	**Abbr.**	**Description**
Omission	om	I did not do something I knew I had to do.
Ignorance	ig	I did not do something because I did not know I had to do it.
Commission	cm	I did something incorrectly, even though I knew how to do it.
Typo	ty	I did something incorrectly because of a typing error.
Knowledge	kn	I did something incorrectly because did not know how to do it.
Information	in	I did something incorrectly because I was misinformed on what to do, or how to do it.
External	ex	I did not cause the defect.

Table D.2: Reasons for Injecting Defects

D.2 Defect Lists

D.2.1 Defect Lists

Full defect lists are available from `http://www.cs.uwaterloo.ca/~ihkwan/`.

D.2.2 Table Headings

Due to limitations in the scripts that output the defect lists, the table headings are missing on the defect list pages above. However, the table headings that should be attached to the defect list tables are described below. For a description of phases, please refer to Section 2.8.1.

No. The defect identifier. The number is assigned in the order that the defects are repaired.

Dur The time required to repair this defect, in minutes.

Inj The phase during which the defect was believed to be injected into the system.

Rem The phase during which the defect was removed.

Cls The classification of the defect (Table D.1).

Re The reason that the defect was injected (Table D.2).

Comment An optional comment describing the defect.

Bibliography

[1] Joanne M. Atlee. Personal correspondence, 2004.

[2] Victor R. Basili. The role of experimentation in software engineering: Past, current, and future. In *ICSE*, pages 442–449, 1996.

[3] Daniel M. Berry. Formal methods: the very idea - Some thoughts about why they work when they work. *Sci. Comput. Program.*, 42(1):11–27, 2002.

[4] Daniel M. Berry. The inevitable pain of software development: Why there is no silver bullet. In Martin Wirsing, Alexander Knapp, and Simonetta Balsamo, editors, *RISSEF*, volume 2941 of *Lecture Notes in Computer Science*, pages 50–74. Springer, 2002.

[5] Daniel M. Berry. Personal coresondence, 2004.

[6] Daniel M. Berry. Personal correspondence, 2005.

[7] "Daniel M. Berry and Jeanette M. Wing". "specification and prototyping: Sme thoughts on why they are successful". In *Proceedings of TAPSOFT, Vol. 2*, pages 117–128, Berlin, Germany, March 1985. Springer.

[8] Jonathan P. Bowen and Michael G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12:34–41, July 1995.

[9] Michael Breen. Experience of using a lightweight formal specification method for a commercial embedded system product line. *Requir. Eng.*, 10(2):161–172, 2005.

[10] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1978.

[11] Larry B. Christensen. *Experimental Methodology.* Allyn and Bacon, Inc., Boston, 3rd edition, 1985.

[12] Dan Craigen, Susan L. Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. Software Eng.*, 21(2):90–98, 1995.

[13] Arnis Daugulis. Time aspects in requirements engineering: Or 'every cloud has a silver lining'. *Requir. Eng.*, 5(3):137–143, 2000.

[14] Steve M. Easterbrook, Robyn R. Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. Software Eng.*, 24(1):4–14, 1998.

[15] S. R. Faulk, L. Finneran, J. Kirby Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the lockheed C-130J software requirements. In *COMPASS'94*, June 1994.

[16] John S. Fitzgerald, Peter Gorm Larsen, Tom Brookes, and Michael Green. Developing a security-critical system using formal and conventional methods. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, chapter 14, pages 333–356. Prentice Hall, London, 1995.

[17] Kevin Forsberg and Harold Mooz. System engineering overview. In Richard H. Thayer and Merlin Dorfman, editors, *Software Requirements Engineering*, pages 44–72. IEEE Computer Society Press, Los Alamitos, California, 1997.

[18] Donald C. Gause. Comparative evaluation in requirements engineering: Is it too much too soon, too little too late or just the right amount at just the right time? In *Second International Workshop on Comparative Evaluation in Requirements Engineering, 7 September 2004, Kyoto, Japan*, 2004.

[19] Susan L. Gerhart, Dan Craigen, and Ted Ralston. Case Study: Darlington Nuclear Generating Station. *IEEE Software*, 11(1):30–39, 28, 1994.

[20] James A. Hager. Software cost reduction methods in practice. *IEEE Transactions on Software Engineering*, 15(12):1638–1644, December 1989.

[21] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7:11–19, September 1990.

[22] Warren Harrison. Skinner wasn't a software engineer. *IEEE Software*, pages 5–7, May/June 2005.

[23] C. L. Heitmeyer and R. D. Jeffords. The SCR tabular notation: A formal foundation. Technical Report NRL/MR/5546-03-8678, Naval Research Laboratory, 2003.

[24] Constance Heitmeyer and Ramesh Bharadwaj. Applying the SCR requirements method to the light control case study. *Journal of Universal Computer Science (JUCS)*, 6(7):650–678, July 2000.

[25] Constance L. Heitmeyer, James Kirby, and Bruce G. Labaw. The scr method for formally specifying, verifying, and validating requirements: Tool support. In *ICSE*, pages 610–611, 1997.

[26] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Software Eng.*, 24(11):927–948, 1998.

[27] Johann Hörl and Bernhard K. Alchernig. Validating voice communication requirements using lighweight formal methods. *IEEE Software*, 17(3):21–27, May/June 2000.

[28] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, Reading, Massachusetts, 1995.

[29] Erik Kamsties, Antje von Knethen, Jan Phillips, and Bernhard Schatz. An emperical investigation of the defect detection capabilities of requirements specification languages. In *Intl. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*, June 2001.

[30] Steve King, Jonathan Hammond, Rod Chapman, and Andy Pryor. Is proof more cost-effective than testing? *IEEE Transactions on Software Engineering*, 26(8):675–686, August 2000.

[31] Matthias Müller, Oliver Gramberg, and Lutz Prechelt. PSP Resources Page. `http://www.ipd.uka.de/mitarbeiter/muellerm/PSP/`, 2003.

[32] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11(3):259–266, March 1985.

[33] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Trans. Software Eng.*, 11(3):259–266, 1985.

[34] Shari Lawrence Pfleeger. Albert Einstein and Empirical Software Engineering. *IEEE Computer*, 32(10):32–37, 1999.

[35] Shari Lawrence Pfleeger and Les Hatton. Investigating the influence of formal methods. *IEEE Computer*, 30(2), February 1997.

[36] Lutz Prechelt. Defect classification. `http://www.ipd.uka.de/mitarbeiter/muellerm/PSP/Dokumente/DefTyp/defecttypes.html`, 1999.

[37] Bran Selic and Jim Rimbaugh. Using UML for modeling complex real-time systems. Technical report, IBM, March 1998.

[38] Susan Elliott Sim, Steve M. Easterbrook, and Richard C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE*, pages 74–83. IEEE Computer Society, 2003.

[39] Dag Sjøberg, Jo E. Hannay, Ove Hansen, Vigdis By Kampenes, Amela Karahasanović, Nils-Kristian Liborg, and Anette C. Rekdal. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Software Eng.*, 31(9):733–753, 2005.

[40] A. E. Kelley Sobel and M. R. Clarkson. Formal methods application: An empirical tale of software development. *IEEE Trans. Softw. Eng.*, 28(3):308–320, 2002.

[41] Software Standards Commitee of the IEEE. IEEE Standard Glossary of Software Engineering Terminology, December 1990.

[42] Walter F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, 1998.

[43] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 147–159, New York, NY, USA, 2000. ACM Press.

[44] Jeanette M. Wing. A study of 12 specifications of the library problem. *IEEE Software*, 5(4):66–76, July 1988.

[45] Marc K. Zimmerman, Kristina Lundqvist, and Nancy Leveson. Investigating the readability of state-based formal requirements specification languages. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 33–43, New York, NY, USA, 2002. ACM Press.