Université d'Ottawa · University of Ottawa

Determining the Reuse Worthiness of a Component: Empirical and Analytical Approaches

Rym Mili

University of Ottawa, Canada

December 1996

To Chédly and Mélika
for their precious lessons

Acknowledgments

This work has taken the collaboration of a number of people, who I hereby wish to gratefully acknowledge.

First and foremost, I wish to thank my supervisor, Professor Jacques Raymond; his guidance, encouragements and unfailing support have been indispensable to the completion of this work. Professor Raymond's persistence has enabled me to continue my progress despite his sabbatical at the University of Singapore; his dedication and patience are greatly appreciated. I am very honored that a world renown authority in software reuse, Dr Jeffrey Poulin, from *Lockheed Martin Federal Systems, Inc.* has accepted to review my work; the extensive discussions he and I had on my thesis and related research have given me precious new insights and greatly improved my dissertation. Professor Jean-Pierre Corriveau, from Carleton University, has shown a great deal of interest in my work from its beginning, and has profoundly influenced its orientation by his continuous feedback. Professor Louis Birta, Professor Emeritus of Computer Science at the University of Ottawa, has graciously accepted to serve on my doctoral committee; he provided me with insightful guidance throughout the various stages of the doctoral program. Finally, Professor Robert Probert, from the University of Ottawa, head of the Software Engineering and Telecommunications Research Group, has offered me the privilege of serving on my examination committee; his feedback on an earlier version of my dissertation has led to substantial improvements.

This work has been reviewed by a number of software reuse researchers, including: Professor Victor Basili, from University of Maryland; Mr James Baldo, from MITRE Corporation; Dr John Bailey, from the US's Institute for Defense Analysis; Professor Anestis Topstis, from York University. We thank Prof. Basili, Mr Baldo, Dr Bailey and Prof. Topstis for their time, interest, and feedback.

On a personal level, I wish to thank my husband and daughter for tolerating my part time status while I was working on my thesis. I also wish to thank my parents Chédly and Mélika and my siblings Zyed and Selma for their encouragement and continued support.

Preface

Despite a large body of theoretical knowledge and practical experience, the software industry is still characterized to some extent by low productivity and persistent questionable quality. A number of solutions have been proposed in the past to deal with the difficulties of software development: high level languages, structured design, structured programming, formal methods, fourth generation languages, programming team organizations, object oriented software engineering, etc. Despite their apparent promise, these methods did not deliver outstanding gains in productivity and quality; rather they only added small increments to the practice of software development.

We have reasons to believe that software reuse will do more to the practice of software development than add a small increment:

- The concept of *reuse* is central to all successful industries, and it is fair to consider that the software industry will not achieve any level of maturity unless it improves its reuse technology. When a new automobile model is designed, it is not designed from scratch from the ground up; rather it reuses all sorts of existing work products (parts, designs, design ideas) and improves on them.

- Unlike many other technologies (structured programming, programming team organizations), which focus on a single aspect of software development, the technology of software reuse attempts to consider all three aspects of software engineering, namely technical aspects, managerial aspects and organizational aspects.

- Current empirical evidence is showing a great deal of gains in programmer productivity and product quality, much larger than any previous technology has provided. Further, we expect these gains to grow as the software reuse technology matures.

- Unlike many other technologies, the concepts of software reuse are consistent with today's emphasis on large scale software development (mega-programming) and on software architectures. Hence the difficulty of scaling up, which has caused most other technologies to fade away, will not prevent the widespread use of the software reuse technology.

This dissertation deals with managerial aspects of software reuse; in particular, it attempts to define and compute a measure of reusability, whose purpose is to predict the savings potential of a software component prior to storing it in a reuse library.

In Part I, titled *Background*, we give an overview of the state of the art and state of the practice of software reuse. Then we present a survey of past proposals of reusability measures and motivate the need for a new measure. Finally, we present some mathematical background that we use throughout the dissertation to carry out our work.

One of the key premises of our work is a sharp distinction between defining a measure and figuring out how to compute it; in order to highlight this distinction, we deal with these two questions in separate parts. In Part II, titled *Component Level ROI Model for*

Software Reuse, we define a basic measure of component reusability for two separate reuse lifecycle (synchronoous lifecycle and asynchronous lifecycle) then we discuss a number of auxiliary measures that can be derived from the basic measure.

In Part III, titled *Estimating The Return on Investment*, we discuss means to estimate the reusability of a component using the formulas presented in the previous part. To this effect we use empirical data as well as existing software cost estimation models, and present a software package that assists in computing the desired reusability measures.

So far we have concerned ourselves with how to measure component reusability, without concern with understanding the mechanics of reusability (e.g. what features drive the reusability of a component up). In the remainder of the dissertation we discuss a formal basis of software reuse, whose purpose is to capture reuse related activities in a unified mathematical framework. In Part IV, titled *A Formal Framework for Software Reuse*, we discuss a formal basis for three crucial activities of software reuse, namely: the specification of user queries and component functionalities; the storage and retrieval of software components in a reuse library; the correctness preserving modification of a component to satisfy a user query that, presumably, the component *almost* satisfies.

In Part V, titled *Analytical Measures of Reusability*, we discuss two analytical measures of distance between specifications, which can be used to measure the distance between a user query and an available software component. These measures of distance are: *functional distance*, which reflects the level of commonality between the functional properties of the specifications; *structural distance*, which reflects the amount of effort that it takes to modify a software component (represented by one specification) to satisfy a user query (represented by the other specification). Using these two distances, we attempt to define an analytical measure of component modifiability, which reflects the ease with which the component can be modified to satisfy a *neighboring* specification.

In the conclusion, we summarize our results, evaluate them, and sketch directions of future research.

Rym Mili

December 1996

Contents

List of Figures

Part I

Background

In this part, we introduce the background of our study and present its main premises. In chapter 1 we present an overview of software reuse and discuss how our work fits in the overall landscape. In chapter 2 we present a survey of existing proposals for measuring software reusability then we define the premises of our work and contrast them with existing work. Finally, in chapter 3 we offer some mathematical background that is used throughout our dissertation; this includes some mathematics for engineering economics, as well as elements of measurement theory.

Chapter 1

Overview of Software Reuse

1.1 Software Reuse: The Potential and The Pitfalls

Manufacturing industries would never have achieved the level of efficiency and product quality that they are enjoying nowadays if it were not for the systematic application of reuse technologies. When a new car model is designed today, designers do not reinvent the car from the ground up —rather they make extensive reuse of existing parts, existing designs, and existing processes (e.g., production lines).

Similarly, the software industry cannot possibly hope to achieve any level of productivity and quality unless it has perfected the art of reusing existing software assets, such as components, designs, test suites, development processes, and the like. Yet observation of the state of the practice in the software industry indicates that this industry is far from achieving the full potential of software reuse. Among the details that we have observed (chapter 7) we mention the following:

- Many organizations do not have the necessary infrastructures to support reuse; as a result reuse is practiced on an individual level or a project level. Yet at this level, reuse benefits are known to be marginal at best.

- Many organizations that have reuse infrastructures do not necessarily take adequate steps to ensure that the organization takes the best advantage of these infrastructures.

- Many organizations that have adequate reuse infrastructures and are trying to make the best use of them do not quantify the benefits of reuse and match them against the benefits/costs of traditional development methods.

In order to be effective, a reuse venture must be carefully planned, adequately supported by the management hierarchy, carefully executed, and adequately monitored in its implementation.

Even though it is widely recognized that software reuse offers great potentials in terms of productivity and quality, it is not widely practiced in industry. Some of the reasons for this situation are the following:

- *A software reuse program involves significant investments.* In order set up a software reuse program, an organization must provide computer tools (hardware and software) to manage the storage and retrieval of software assets, as well as manpower to operate these tools and monitor the reuse impacts. Unless the managerial hierarchy has a clear understanding of the potential benefits of the reuse technology, they are reluctant to agree to the necessary investments.

- *A software reuse program does not pay off on the short run.* There are two primary benefits to software reuse: gains in productivity stem from the opportunity that programmers have to reuse an existing software asset rather than have to develop it from scratch; gains in quality stem from the opportunity to use an existing software asset that has been duly validated by its many uses in the past, rather than use a new asset that may have many faults. Both of these gains require time to materialize, and they are nearly linear as a function of time. Given the emphasis that today's managers place on short term gains, it may be difficult to impress upon them the benefits of a software reuse program.

- *A software reuse program involves technical changes.* In order to have a successful reuse program, it is not sufficient to set up reuse infrastructures and make software assets available to programmers and analysts; it is necessary to integrate reuse activities into the development processes. Technical personnel may resist the proposed changes, thereby endangering the reuse program or reducing its benefits.

- *A software reuse program involves managerial changes.* In a traditional programming environment, the productivity of a programmer can be measured by the number of lines of code produced by unit of time. In a programming environment that includes reuse, one must revise the formula for determining the productivity of a programmer; the new formula must prorate the count of lines that are reused verbatim, the count of lines that are reused after modification and the count of lines that are developed from scratch. Also, managers must redefine the reward structure in the organization so as to encourage reuse. All these changes are potential sources of resistance.

In order to be successful, a software reuse program must meet the following criteria:

- *Management commitment.* Because of the scale of upfront investments, the length of the investment cycle, and the profoundness of changes that are required in the operational procedures of the organization, a software reuse program cannot possibly succeed unless it enjoys the full support of the organization's upper management.

- *Technical Personnel Commitment.* Programmers and analysts are greatly affected by the existence of a reuse program within their organization, because the program has a great impact on their day to day work. Instead of developing code exactly as they see fit, they must now try to reuse code developed by other programmers. This does not sit well with much of the empirical evidence available nowadays on the psychology of programmers [31, 72, 73, 75, 74].

- *Adequate Operational Procedures.* In order to take most advantage of a reuse venture and ensure that reuse investment costs are recovered as quickly as possible, an organization must apply reuse as extensively as possible. This means in particular that the option to reuse existing assets (rather than develop from scratch) should not be left entirely to the discretion of the programmer. Rather, organizational development procedures should encourage programmers to investigate reuse possibilities extensively as part of the software development lifecycle.

- *Adequate Technical Support.* It is possible, with today's technology, to provide adequate computer support to programmers and analysts in identifying and adapting reusable assets. There are two broad families of tools: those that are based on library science retrieval procedures and use hypertext to identify potential reuse candidates [63]; and those that are based on formal specification ideas and use refinement relations to identify potential reuse candidates [46, 77, 59]. The first family of products is widely used nowadays, whereas the second family is at the stage of experimental prototypes. While there is no evidence to the effect that the second family of products save retrieval time (presumably because they are automated), there is reason to believe that they offer a better retrieval precision (all components that are retrieved are relevant).

- *Adequate Measurement and Control.* In order to keep the reuse effort on track, a software organization must maintain information on its level of activity and its level of benefit, as measured by such features as: the percentage of reused code, the frequency of reuse, the amount of manpower saved per unit of time, the gain in product quality, etc. This measurement aspect is an integral part of the reuse program, and serves as its main justification.

- *Adequate Reward Structure.* In order to achieve high levels of software reuse, and adequate return on investment in its reuse program, an organization must set up adequate reward structures to encourage software reuse. Traditionally, programmers resist using other people's code, and consider modifying code as a form of maintenance (hence something they are usually reluctant to do, as they consider it a second rate activity). A reward structure must make provisions for rewarding the author of a reusable asset (hence promoting the concern for quality) as well as the user of the reusable asset (hence promoting concern for productivity).

1.2 Software Reuse Aspects

As with all other software engineering problems, software reuse can be seen from three distinct viewpoints: we can consider its managerial/ financial aspects, its technical aspects, and its organizational/ human aspects. We briefly review these aspects in this section, so as to better define the context of our dissertation.

1.2.1 Software Reuse: Managerial Aspects

The managerial aspect of software reuse is probably the most important aspect, because it is that which has the greatest impact on the success or failure of a software reuse experiment. Managerial considerations are present in all decisions pertaining to software reuse, and serve as the primary motivation for setting up a software reuse program. In fact many software reuse decisions are nothing but investment decisions, which must be dealt with in the same manner as any investment decision. We have identified three investment cycles, which we describe briefly below.

1.2.1.1 Organization Wide Investment Cycle

This is the cycle whereby an organization decides whether or not to launch a software reuse program. In this cycle, investment costs include: the cost of domain analysis, to determine that/whether the application domain may benefit from reuse; the cost of hiring staff dedicated to running the reuse database and monitoring its level of usage; the cost of acquiring the necessary hardware and software support to run the reuse database; the cost of modifying the organization's operational procedures, and gaining organization-wide acceptance of the new procedures; the cost of training the software development personnel on reuse technologies.

Once the reuse program has been set up, operating on a day to day basis involves costs as well as benefits. Benefits include savings in development effort and gains in product quality. Costs include the cost of operating the reuse database such as staff salaries, maintaining computer hardware and software, measuring and monitoring the reuse activity.

As with any investment decision, both options carry risks: if we fail to launch a reuse program when the opportunity of an efficient reuse program exists, we miss out on potential savings and technological advantage; if we do launch a reuse program only to find that the level of activity within the organization does not justify it, we waste resources and create confusion (e.g., by changing operational procedures in the organization then reverting back).

The most crucial activity in this decision cycle is that of *domain engineering*, which includes three activities [2, 3]: *domain analysis*, which is the identification, acquisition and evolution of reusable information on a problem domain to be reused in software specification and construction [63]; *infrastructure specification*, which is the selection and organization of reusable information in the model to fit the patterns of reuse in the environment of a reuser; *infrastructure implementation*, which the design and encoding of the pieces resulting from the specification process using particular representations required by the technology of the reuser.

1.2.1.2 Project Wide Investment Cycle

This is the cycle whereby a project manager decides to develop a project with or without software reuse. The terms of the investment decision vary widely with whether a reuse program exists in the organization or not. If a reuse program exists then the manager

is under pressure to apply reuse, so as to help amortize its cost (design with reuse) and provide reusable assets (design for reuse); then project level investment costs are minimal, since typically project personnel would be familiar with reuse related operational procedures. If no reuse program is in effect when the project is launched, then the manager must contemplate the investment costs which are a small scale version of those that we have discussed above for organization wide investment cycle.

The periodic costs incurred if the reuse option is chosen include the extra effort and complexity involved in incorporating reuse in the development activity. Benefits include the usual gains in productivity and quality, as well as the production of reusable assets that can be made available to subsequent development projects.

As for the risks involved in this decision, they can be summarized as follows: if we choose the *with-reuse* option, we carry the risk that our software personnel are distracted by complex procedures that ultimately lead to little reuse; if we choose the *without-reuse* option, we run the risk of missing out on gains in personnel productivity, product quality as well as the opportunity to produce reusable assets for subsequent projects.

1.2.1.3 Component Level Investment Cycle

Given that we are working in an organization that has a reuse program, for a project that is applying reuse, and given a software asset, we are pondering the question of whether this asset should be stored in the reuse database for future reuse. An asset is a standard building unit in an organization that is used to develop applications [39]. The terms of the investment decision vary greatly with whether the asset at hand was designed specifically for reuse, or whether it was designed for single use as part of an existing project and we are contemplating (as a second thought) to include it for future reuse. If the asset at hand was developed for single use, then investment costs include: the cost of reengineering the asset for the purpose of reuse, the cost of verifying it (up to the standards of the reuse library) and the cost of storing it in the reuse library. If the asset at hand is due to be developed for reuse, then investment costs include primarily the effort to develop the asset *for reuse* (with due concern to quality) as well as the costs of storing it in the reuse library.

The periodic costs incurred if the option to store the asset is chosen include: the effort to run the reuse library (prorated to the number of assets in the library) as well as the impact that the presence of the asset has on the performance of the reuse library (any additional asset has the potential to slow down retrievals and affect precision, by turning out this component in cases when it is not relevant); also, the costs incurred to retrieve the asset at hand, evaluate it, and instantiate it or modify it. Benefits include savings in future development efforts as well as gains in product quality and reliability.

As for the risks associated with the decision to store or not to store the asset at hand in a reuse library, they can be summarized as follows: the decision to store the asset carry the risk that the asset will slow down retrievals, drive down retrieval precision (by showing up in retrievals where it is not relevant) and distracting programmers (who would then have to evaluate it and exclude it); the decision to forego the asset (and not store it) carry the risk of missed reuse opportunities.

The purpose of this thesis is to derive a component-specific measure of software

reusability. Instead of analyzing the qualities that promote reusability and discussing means to quantify them and measure them, we have chosen to derive our measure of reusability from an economic perspective. We define the reusability of a component as the return on investment that we stand to reap by including this component in our reuse library. Hence the main focus of our work is on the *component-level* investment cycle, rather than the organization-wide cycle or the project-wide cycle.

1.2.2 Software Reuse: Technical Aspects

There are three families of technical aspects in software reuse (ref: STARS Conceptual Framework): those that are related to *developing* reusable assets; those that are related to *managing* reusable assets; and those that are related to *using* reusable assets. We review them briefly below.

1.2.2.1 Design for Reuse

The development of software components for the purpose of software reuse differs from the development for single use in a number of significant ways, which we review below.

- *Specification.* A component that is designed for single use must fit the specific requirements of the system where it is to be included. By contrast, a component that is developed for reuse must fit a wide range of requirements, which are not even known at design time but can only be imagined. In order to maximize the chances that a component will be subsequently (re)used, the component developer must make the component as generic as possible.

- *Documentation.* A component that is designed for single use must be inspected by its designer and possibly later by the analyst responsible for its maintenance. By contrast, a component that is designed to be reused is expected to be inspected and evaluated by a larger number of potential users, with different backgrounds. Hence it is essential that components designed for reuse be properly documented; this includes documentation of their function (for the purpose of those who will use the component without modification) and documentation of their design/structure (for the purpose of those who will modify the component to meet their needs).

- *Design.* Design integrity is a desirable feature in all software. For reusable software, it is an essential feature, for two reasons: first design integrity is a prerequisite to reliability; and second, design integrity is an excellent form of documentation, and makes the task of documenting the component much easier.

- *Validation.* Ideally, all software that is developed must be validated to the point where we can ensure its correctness. This is often costly in practice, so that in general a tradeoff must be struck between the need to ensure the correctness of the component and the need to keep development costs under control. The terms of this tradeoff are radically altered when we are dealing with reusable components, because of the impact of a failure of the component. A single use component that fails brings

down with it the single system in which it is running. A reusable component that fails brings down as many systems as there are that include it. Hence when we are validating reusable component, we may be willing to invest a great deal more effort than when validating a single-use component.

In chapter 7 we discuss the impact that design for reuse has on the cost of developing a software component.

1.2.2.2 Managing Reusable Assets

The key issue in managing reusable assets is that of defining a storage structure for reusable assets; as libraries of reusable assets grow in size, this issue becomes increasingly critical. The main criterion used in designing a storage structure for software assets is the performance of retrievals on the structure. Retrievals are evaluated on the basis of two criteria [33]: the *precision* of the retrieval, which reflects to what extent all retrieved assets are relevant; the *recall* of the retrieval, which reflects to what extent all relevant assets are retrieved. A number of storage structures have been proposed in the past; they range from keyword searches (inspired from library science) to signature matching to formal specification matching. A proposal for specification based retrieval will be discussed in chapter 11.

1.2.2.3 Design with Reuse

The three technical issues that we have identified with *design with reuse* are: component retrieval, component instantiation, and component modification. We review these issues in turn below.

- *Component Storage and Retrieval.* A reuse library typically consists of a database of component descriptors, where each descriptor contains information about the component at hand as well as a reference to the source or object code of the component. The information recorded about each component deals primarily with the functional features of the component and is typically represented in natural language. Retrieval from such a library typically proceeds by mere inspection of the component's record as well as (sometimes) the source code; the retrieval is typically assisted by computer based methods inspired by library science, consisting primarily of matching keywords or applying hypertext tools.

  Some attempts have been made to improve on this basic technology; chapter 11 shows an example of such an attempt, where component specifications and user queries are represented by formal specifications and matching is defined by the refinement relationship. Such an approach, which is currently in the mere prototype stage, ensures perfect precision (all retrieved components satisfy the user query) but does not ensure perfect recall (some correct components may fail to be retrieved). Also, this approach (as well others that are based on formal specifications [59, 77]) is dependent upon theorem proving technology to produce results; because theorem provers are still fairly unpredictable in their performance, this approach is still in the experimental stage.

- *Instantiation.* Instantiation is the process whereby a component whose specification is generic is instantiated to a specific function by setting some of its parameters, flags, options and arguments. For example, a general purpose sorting routine can be instantiated by specifying the record structure (of the file to be sorted), the key on which sorting is to be based, the order of the sorting (increasing, decreasing) e.g., a maximum file size.

  Instantiation is usually a simple process that takes a small manpower effort, and that can be carried out by inspection of the requirements specification of the component (as opposed to the source code).

- *Modification.* Modification is the activity of modifying a reusable software component to make it meet user requirements it is thought to *almost* satisfy. The recognition that a component *almost* satisfies user requirements is in fact a retrieval issue; it is discussed in chapters 11 and 14. Unlike component instantiation, component modification is not planned for by the component's designer, except possibly that the designer has made provisions for making the component well structured and well documented. Also, unlike component instantiation, component modification is carried out by inspection of both the requirements specification of the component as well as its source code. Finally, unlike component instantiation, component modification can be fairly complex, arbitrarily in the details of the component at hand, and arbitrarily costly in terms of manpower (costing as much possibly more than the cost of developing the component from scratch). These matters are discussed in more detail in chapter 7.

1.2.3 Software Reuse: Organizational Aspects

In [20] Caldiera and Basili propose a software factory-like organization of software reuse, which accommodates both design for reuse and design with reuse. According to this organization, the personnel of a software development institution is divided into two groups: the *project organization* and the *experience factory.* We discuss them in turn below.

1.2.3.1 Project Organization

This group is responsible for producing software products, and is encouraged to make use of the software reuse facilities that are made available by the experience factory. This group cycles through the following phases:

- Specifications: This consists of the traditional requirements specifications phase [14].
- Design: This consists of the traditional product design phase [14].
- Component Lookup: Once the product design has reached the phase where we have specifications of software components ready to be developed, we formulate queries to the experience factory and await its feedback. If the experience factory finds exact matches, it instantiates them and returns them; if it finds approximate matches, it modifies them and returns them; else it develops a component from scratch and

returns. In this last case, if the component proves to have reuse potential, it is reengineered and stored in the reuse library.

- Integration and Test: Whether the components have been retrieved from the experience factory by instantiation, modification, or development from scratch, they are integrated and the system is tested.

1.2.3.2 Experience Factory

The experience factory manages the reuse library by maintaining existing assets, including project specific assets that prove to have reuse potential, and producing assets for the purpose of reuse. This group carries out two concurrent activities, which we discuss in turn below:

- *Synchronous Activities.* Synchronous activities are the activities that are triggered by requests of the project organization. Whenever the project organization submits a request that cannot be satisfied with existing reusable assets the experience factory develops the necessary asset. Once an asset is developed, the experience factory investigates the question of whether this asset has reuse potential. If the answer is affirmative then the experience factory proceeds with reengineering the asset and validating it, then storing it for future reuse.

- *Asynchronous Activities.* Whereas synchronous activities are triggered by the project organization, asynchronous activities are started spontaneously whenever the experience factory has idle manpower. These activities consist of identifying potentially reusable assets, proceeding with developing them then storing them into the reuse library. The cycle of these activities is the following:

  - Plan: This is the phase when a need is identified, and it is determined that this need can be met with a specific asset. Plans are then made for the development of this asset.
  - Produce: This is the phase when the asset in question is developed and duly validated.
  - Package: This is the phase when the asset is packaged for the purpose of being stored in the reuse library. This includes such steps as documenting it according to the standards of the library, and building its library descriptor.
  - Store: This is the phase when the component is stored in the reuse library; this phase includes whatever operational procedures the organization has defined as part of storing a component (e.g., peer review, reliability estimation, functional validation, etc.).

Chapter 2

Measuring Software Reusability

In this chapter we review the main proposals that have been put forward in the past for measuring component reusability. On the basis of this overview we discuss the premises of our measure of reusability and motivate our approach.

2.1 A Survey of Reusability Measures

In this section we briefly overview the main proposals that have been presented in the past for measuring software reusability; we will review in turn qualitative approaches (in section 2.1.1 then quantitative approaches (in section 2.1.2).

2.1.1 Qualitative Approaches

2.1.1.1 Caldiera and Basili

In [20], Caldiera and Basili consider that the three basic attributes that make a component reusable are its *functional usefulness* in the context of the application domain, its *reuse cost* and its *quality*. Each one of the basic attributes is further decomposed into a set of factors. *Usefulness* is decomposed into *commonality of function* (overall, within a domain, or within a system) and *variety of function* (processing, transfer, control). The *Cost* factor is decomposed into *extraction* (identification, qualification), *use in new systems* (retrieval, modification, integration), and *packaging*. *Quality* is decomposed into *performance* (time, space), *readability*, *testability*, *correctness* and *ease of modification*. All these factors are represented by a *fish-bone diagram*.

The proposal of Caldiera and Basili is primarily qualitative; it consists in cataloging features that are positively correlated with good reusability, but does not quantify reusability. Specifically, Caldiera and Basili submit that good reusability can be predicted on the basis of the following quantitative software measures: Halstead's *volume*; McCabe's *cyclomatic complexity*; as well as *regularity* and *reuse frequency*.

Regularity is defined on the basis of Halstead's software science metrics by the following formula:

$$r = \frac{N'}{N}$$

where
N is the actual software science length, and N' is the estimated software science length.

Reuse frequency is defined as

$$v_\sigma(C) = \frac{n(C)}{\frac{1}{M}\Sigma_{i=0}^{M} n(S_i)}$$

where
$n(C)$ corresponds to the number of calls addressed to the user defined component C, and $\frac{1}{M}\Sigma_{i=0}^{M} n(S_i)$ corresponds to the average number of calls addressed to a standard component.

2.1.1.2 Dynamic Research Corporation

Dynamic research corporation [22] defines general standards for deciding on the reusability of a software component on the basis of six selected attributes: being *complete and exact*, being *well-defined and deterministic*, being *portable*, being *general and flexible*, being *human readable*, and being *efficient*. For each such attribute the authors define a set of language-specific guidelines that programmers must follow in order to maximize compliance with the attribute; in particular, the guidelines deal with C, C^{++}, COBOL and FORTRAN.

Two features of this work are noteworthy: first, it gives explicit concrete guidelines that programmers can apply on a routine basis to enhance the reusability of software components; second, it remains, nevertheless, quite qualitative and give no basis for measuring reuse worthiness per se.

2.1.1.3 Raymond Paul

In [57], Raymond Paul defines reusability in terms of sixteen *high-level metrics*:

1. *portability*: operating environment independence.
2. *reliability*: probability of error/fault occurrence (MTBF).
3. *adaptability*: capability to modify software component to meet requirements.
4. *complexity*: indicator of the structure of the software
5. *completeness*: extent to which the component implements all the required capabilities,
6. *coupling*: degree of connectivity between different components (static and dymanic aspects),
7. *modularity*: the way the component is decomposed into subcomponents,

8. *expandability/augmentability*: ability of the software to support expansion of data-storage requirements.
9. *performance/efficiency*: empirical execution data, including history of performance,
10. *fault tolerance*: ability to produce correct results, despite input errors/faults,
11. *size*: measured software size by source statements or lines of code,
12. *understandability*: ease of comprehending the meaning of the software,
13. *resource requirements*: requirements for memory, storage and communication power,
14. *testability*: if equipped with test plans, regression testing,
15. *maturity*: degree of readiness for reuse (how long ago was the component created? how many faults have been detected in the code?, ...),
16. *usability*: ease of use and embedded training.

While it is conceivable that reusability is indeed correlated to these sixteen software attributes, it is not clear how each attribute influences the reusability of the component, nor is it clear how to quantify these attributes in the first place.

2.1.1.4 REBOOT

The ESPRIT-2 project called REBOOT (Reuse Based on Object-Oriented Techniques) defines reusability in terms of four factors:

1. *portability*: the ease with which someone can transfer the software from one computer to another; this includes the criteria *modularity* and *environment independence*.
2. *flexibility*: the number of choices a programmer has in determining the use of a component; this includes *generality* and *modularity*.
3. *understandability*: this refers to the ease with which a programmer can understand the component; this includes *code complexity*, *self descriptiveness*, *documentation quality*, and *module complexity*.
4. *confidence*: this is the subjective probability that a component will perform without failure over a specified time in a new environment; this includes *module complexity*, *observed reliability*, and *error tolerance*.

A metric is associated to each one of the criteria as follows:

- *generality*: generality checklist,
- *modularity*: code/number of methods,
- *environment independence*: machine-dependent code/executable code, system dependent code/ executable code,

- *code complexity*: cyclomatic complexity,
- *self descriptiveness*: comments/ source code, self descriptiveness checklist,
- *documentation quality*: documentation/source code, documentation checklist,
- *module complexity*: fan-in, fan-out, cyclomatic complexity,
- *reliability*: total number of tests, number of observed errors,
- *error tolerance*: error tolerance checklist.

By assigning quantities (between 0 and 10) to each one of these software qualities, one can define the reusability of a component as the sum of the quantities that are assigned.

2.1.1.5 STARS

In [67], a list of general attributes of reusable software is given as follows:

- *ease of understanding*,
- *functional completeness*,
- *reliability*,
- *good error and exception handling*,
- *information hiding*,
- *high cohesion and low coupling*,
- *portability*.

This list appears to be the synthesis of earlier proposals; like other aggregate measures, this measure allows one to make comparisons but does not allow one to make reuse decisions.

2.1.1.6 Prieto-Diaz and Freeman

In [63] Prieto-Diaz and Freeman identify five program attributes and associated metrics for evaluating reusability. These are:

1. *program size*: lines of source code,
2. *program structure*: low coupling and low cyclomatic complexity,
3. *program documentation*: rating on a scale of 1 to 10,
4. *programming language*: degree of similarity of the target language and the language used in the module.
5. *reuse experience*: experience of the reuser in the programming language and in application domain.

These metrics seem to be geared towards assessing one particular aspect of software reuse: adaptation costs. As such it is incomplete because it does not capture reuse aspects such as pervasiveness in the application domain or generality of the functional properties of the component, e.g., which are both crucial factors of reusability.

2.1.1.7 Selby

In his work, Selby [66] studies twenty five moderate and large size software systems selected from a NASA software production environment. The systems size range from 3000 to 112,000 lines of Fortran source code. There were from 22 to 853 modules (subroutines, utility functions, main programs, macro and data blocks). The modules were classified into four categories:

1. complete reuse without revision,
2. reuse with slight revision (< 25 % changes)
3. reuse with major revision (≥ 25 % changes)
4. complete new development.

The conclusions drawn from this study are:

- when we compare completely reused modules without revision to newly developed, extensively revised or slightly revised modules, we find that completely reused modules without revision have:
  - less interaction with other system modules, in terms of the number of module calls per source line;
  - simpler interfaces, in terms of number of input/output parameters per source line;
  - less interaction with human users;
  - higher ratios of commentary;
  - more interaction with utility functions (when compared to newly developed modules);
  - smaller size, in terms of number of source lines;
  - less total development effort;
  - fewer changes in terms of versions per source line;
  - more assignment statements, (when compared to newly developed modules).

While this data correlates some observable characteristics with reuse potential, it does not constitute a definition of reusability. Some of this data will, however, be used in later chapter to derive estimation models for our measure of reusability.

2.1.2 Quantitative Approaches

2.1.2.1 Bailey

Bailey measures the reuse worthiness of a component by means of the cost savings achieved by each individual occurrence of software reuse. He proposes the following formula for the cost savings per occurrence of reuse:

$$So = C_a - (C_i + C_c + \frac{G}{N} + M)$$

where
So = savings by reuse occurrence,
C_a = cost avoided by this reuse,
C_i = cost to instantiate or configure the reusable component,
C_c = cost to constrain the problem to allow this reuse,
G = cost to generalize the component,
N = number of reuse expected during the useful life of the component,
M = cost to manage the reuse process, prorated by reuse client.

The reuse savings by reusable component is given by the following formula:

$$S_c = N * (Ca_{ave} - (Ci_{ave} + Cc_{ave})) - G - Mc$$

where
S_c = savings realized from a given component in the repository,
Ca_{ave} = average cost avoided by reuse of this component,
Ci_{ave} = average cost to instantiate this reusable component,
Cc_{ave} = average cost to constrain problem to allow reuse of component.
N = number of reuses of this component during amortization period,
G = cost to generalize this component,
Mc = prorated cost to manage the reuse process, prorated by component.

This formula does not recognize that some of the terms of this equation are upfront investment costs, whereas others are incurred (or saved) on a periodic basis over a long period of time; also this formula does not properly equate the role that time plays with some of these terms. Finally, this formula does not give any insight as to what drives the various cost factors listed here, nor how to control them.

2.1.2.2 Poulin and Caruso

In [61], Poulin and Caruso present ROI models for the *project* level investment cycle and the *corporate* level investment cycle. At the project level, the formula is

$$ROI = RCA + ORCA - ADC$$

where

- ROI represents the Return On Investment that would occur in infinite time.

- *RCA* is the Reuse Cost Avoidance for the initiating project.
- *ORCA* is the Reuse Cost Avoidance for other projects benefiting from the reusable code written by the initiating project.
- *ADC* is the Additional Development Cost of writing reusable code to the initiating project.

The corporate level ROI is given by

$$NPV = -C_0 + \frac{R_1 - C_1}{(1+k_0)} + \frac{R_2 - C_2}{(1+k_0)^2} + .. + \frac{R_n - C_n}{(1+k_0)^n}$$

where

- C_0 is the corporate reuse start-up costs.
- R_i is the savings in year i.
- C_i is the costs in year i.
- n is the number of years for which revenues are to be considered.
- k is the discount rate.

Poulin and Caruso's project level model takes an infinite investment cycle and does not take into account the time value of money. Also, while the authors do give a formula for corporate level ROI, they do not discuss how to compute the costs and savings listed in the formula.

2.1.2.3 Lim

In [42], Lim proposes an NPV model whose purpose is to assess the benefits of introducing reuse in a corporation on the basis of a cost-benefit argument. This model quantifies the domain engineering decision, which is: is it worthwhile to introduce software reuse in an organization? The net present worth of the reuse initiative is given by the following formula

$$NPV = \frac{[C_{c,s}(k) + C_{c,a}(k) + pi(k) - C_p(k)] * P(k)}{(1+i)^k}$$

where,

- $C_{c,s}(k)$ is the consumer costs saved at time period k.
- $C_{c,a}(k)$ is the consumer costs avoided at time period k.
- $pi(k)$ is the increased consumer profit at time period k.
- $C_p(k)$ is the producer costs at time period k.
- $P(k)$ is the probability of receiving net cash flow in time period k.
- i is the interest rate by which cash flows are discounted.

Lim does not discuss how to estimate the different factors that are involved in this formula.

2.1.2.4 Gaffney and Cruickshank

Gaffney and Cruickshank [37] recognize two key decisions in the lifecycle of software reuse: the first is the decision to launch an organization-wide program of software reuse, along with the attending management of a library of reusable assets; the second is the decision to integrate reuse processes in the development lifecycle of a software project. Both involve costs and potential benefits, and both can be modeled as investment decisions. Gaffney and Cruickshank define the terms of these investment decisions and propose *return on investment* formulae thereof.

The first model, *the basic economics model* covers the case where domain engineering is done up front, all at once. It is given by the following formula:

$$C_{US} = C_{VN} - (C_{VN} - C_{VR} - \frac{C_{DE}}{N})R$$

Where

- C_{US} = Unit cost of the application system (LM/KLOC),
- C_{VN} = Unit cost of new code developed for this application system (LM/KLOC),
- C_{VR} = Unit cost of reusing code from the reuse library in this application system. It represents the unit cost of reused code in the case where the library components can be instantiated directly into the application system with no modification (LM/KLOC),
- C_{DE} = Unit cost of domain engineering (LM/KLOC),
- N = Expected number of application systems
- R = Proportion of reuse

The return on investment is defined as the difference in costs between the cost of N application systems in which there is no reuse and the cost of N application systems in which there is an average reuse of R. It is given by the formula

$$ROI = [N \times E \times (C_{VN} - C_{VR})/C_{DE} - 1] \times 100$$

where

- N = number of application systems,
- E = efficiency factor: ratio of the amount of reused code in the
- application system to the available reusable code,
- C_{VN}= cost of new software,
- C_{VR}= cost of reused software,
- C_{DE}= cost of domain engineering.

The second model deals with the case where domain engineering is done incrementally over the time period during which the application systems are created. It generalizes the basic reuse economics model.

The authors make some hypotheses regarding the costs of producing, using and reusing individual assets in the library but do not discuss how these costs are determined.

2.1.2.5 Malan and Wentzel

In [45], Malan and Wentzel introduce a cost-benefit analysis framework for software reuse; they outline the major reuse-related cost factors associated with the development and maintenance phases, and incorporate them into a long term, multi-product net benefit model. The basic development model takes a consumer perspective and assesses the savings achieved at the corporate level if corporate software development projects decide to integrate reuse into their development lifecycle (turning it into a development *with reuse* lifecycle). In order to account for savings that are achieved at the corporate level, Malan and Wentzel include a factor that counts the number of projects that use available components, as well as the development cost differential that stems from developing with reuse. On the other hand, in order to account for the costs that are incurred at the corporate level, the authors incorporate factors that reflect the cost of producing reusable components and maintaining them in a reuse library. In order for the operation to be cost effective, the corporation must provide incentives to the individual projects to use available components. The basic model is given by the following formula:

$$S = \left[\sum_{i=1}^{n}(C_{N_i} - C_{CR_i})\right] - [C_{PR} + A]$$

where

- n = number of products sharing the reusable components
- C_{N_i} = cost to develop product i without reuse
- C_{CR_i} = cost of creating product i with reuse
- $C_{N_i} - C_{CR_i}$ = expected consumer cost saved for product i
- C_{PR} = expected cost that the producer incurs in producing the reusable component
- A = reuse specific overhead and setup costs incurred by the family of products

From the basic cost benefit model, Malan and Wentzel produce a number of extensions: first, they integrate the time value of money, which provides that a monetary unit decreases with time, hence the benefit that returns in the future must be greater than the present investment in order for the investment to be worthwhile. The second extension deals with the uncertainties of future reuse frequencies. Part of the domain analysis activity consists in identifying components that are thought to be reusable frequently, and estimating their reuse frequency (or equivalently, their probability of use per unit of time).

Malan and Wentzel illustrate their economic model on an extensive example, which involves many projects developed with reuse over many years, and shows the impact of the time value of money, the uncertainty of reuse instances, and the evolution of cost benefit with time, as the initial investment costs are amortized.

2.2 Premises and Orientation

In light of the survey that we have presented above, we will identify opportunities for a new measure of reusability by considering the main premises and orientations of our work.

2.2.1 Defining Component Reusability

Whereas past proposals for a measure of reusability define reusability by association with other software qualities, we choose to define it in terms of its cost benefit analysis: the reusability of a component is the return on investment achieved by storing the component in a reuse library for future reuse; this return on investment is computed as a function of a discount rate and for a specific investment cycle (1 year, 3 years, 5 years, indefinite).

We feel that by proceeding in this manner we get a better focus on the actual feature that we are measuring, at the same time as we provide a uniform basis for decision making. For example, a reuse library manager should not consider storing a component unless its return on investment is positive over the chosen investment cycle. Also, a manager may define a threshold value (greater than zero) under which he would not consider storing a component (e.g., we do not store a component in this library unless its return on investment is at least 0.7).

> **Definition 2.1** *The* reusability *of a software asset is the return on investment, with respect to a given present cost, periodic cost/benefit, investment cycle and discount rate, associated with the decision to store the asset in a software library for the purpose of software reuse.*

We feel that research efforts that define reusability by associating it with other software qualities create two weak links in their definition: first, it is not clear in what manner does reusability depend on these qualities; second, these other qualities are difficult to quantify in a meaningful manner (if quantifying the other qualities is as difficult as quantifying reusability, we have hardly made any progress). In addition, such informal definitions of reusability provide no tangible basis for decision making.

On the other hand, our component-level return on investment model is different from all other return on investment models that were developed for software reuse. Some of the most salient differences are:

- Whereas our model focuses on the component-level investment decision and addresses the question

  Is it worthwhile to store a given component in a software library for the purpose of reuse?

  the other models deal with the corporate-level investment decision or the project-level investment decision.

- Unlike our model, the earlier return on investment models produce an ROI formula in terms of predefined cost factors, but do not discuss means of estimating the identified cost factors.

- Whereas other models quantify the investment decision in terms of net present worth, we quantify it in terms of return on investment (which is the net present worth divided by the present cost); we feel that the return on investment is a better indicator than the net present worth, because it takes into account the investment risks.

2.2.2 Estimating Component Reusability

We view the problem of defining reusability and that of estimating it as totally separate problems, and we treat them quite separately. While we are defining reusability we do not concern ourselves with how to measure it, not even with whether it can be measured. Once we have identified a formal measure of reusability, investigated its form, identified its cost drivers, then we focus on estimating the various cost factors that are involved in the formula at the time when reusability must be assessed. For the synchronous lifecycle reusability must be estimated after the component has been developed and before the decision to store it is taken. For the asynchronous lifecycle reusability must be assessed once the requirements specification of the component has been written and before the decision to develop the component is taken. In chapter 7 we will see how existing software cost estimation models and existing empirical data about software management can be used to estimate all the cost drivers that intervene in the evaluation of our reusability measure.

Chapter 3

Mathematics for Software Engineering

In this chapter we survey some of the mathematics that we need for the purpose of our work. In section 3.1, we briefly discuss some elements of engineering economics, so as to lay the ground for chapter 4. In section 3.2 we present some arithmetic identities that we will use to derive closed forms (forms that can be computed algorithmically in a finite number of steps) for our measures of reusability. In section 3.3 we introduce some background on measurement theory.

3.1 Engineering Economics

Our main reference for this section is [64]; However we will interpret the discussions not in terms of financial investments, but rather in terms of management of human resources (person months).

3.1.1 Present and Future Worth

Generally, it is not meaningful to add and/or compare monetary values when these exist at different points in time. Indeed, nobody would exchange ten thousand dollars today for ten thousand dollars in, say, three years, for several reasons: first, we value present consumption more than future consumption; second, having the funds now opens investment opportunities during the next three years that will be missed if we were to wait; third, due to inflation, the same amount of money may be worth less in three years than it is worth now. In terms of software developers person months (PM), we would not want to trade one person month today for one person month in three years, because: one person month today can be used to produce a software product (or part thereof) which can be sold in the meantime; one person month in three years may cost more (due to salary increases) than one person month today.

In order to take this variance into account, we introduce the notion of *discount rate* or *interest rate*. The discount rate represents the smallest value of d for which I accept to exchange 1 PM today for $(1+d)PM$ in a year. Given the discount rate d, which we

consider the same from one year to the next, we expect that whenever we invest a person month today, it will earn us the equivalent of $(1+d)^n$ after n years. Conversely, the present value of a person month that we are due to earn in n years is $\frac{1}{(1+d)^n}$. Generally, the *present worth* at time 0 of an amount A_n at time n for discount rate d is given by the formula

$$P_{0,n} = \frac{A_n}{(1+d)^n}.$$

If n amounts are involved over n consecutive years (benefits or costs), their cumulative present worth is given by the formula:

$$P_0 = \sum_{k=1}^{n} \frac{A_n}{(1+d)^k}.$$

Interestingly, if A were constant and n were infinite, this formula would become $P_0 = \frac{A}{d}$. Should the discount rate vary from year to year, the cumulative present worth of these amounts is given by the formula:

$$P_0 = \sum_{k=1}^{n} \frac{A_n}{\prod_{j=0}^{k-1}(1+d_j)},$$

where d_j is the discount rate from year j to year $j+1$. If we wish to take into account other factors, such as the rate of inflation, then we get the following formula, where r_j is the rate of inflation at year j:

$$P_0 = \sum_{k=1}^{n} \frac{A_n}{\prod_{j=0}^{k-1}(1+d_j)(1-r_j)},$$

3.1.2 Economic Appraisals

In this section we use the net present worth and the return on investment formula to appraise and compare investment solutions. We consider an investment scheme over n years that involves a cost function C_k (i.e., C_k is the cost that we must incur at year k) and benefit function B_k (i.e., B_k is the cost that we must incur at year k). In order to appraise this investment scheme (i.e., assess its worthiness, compare it to other investment options), we compute its *net present worth* by subtracting the present worth of the costs from the present worth of the benefits. We find:

$$NPW = \sum_{k=1}^{n} \frac{B_k - C_k}{\prod_{j=0}^{k-1}(1+d_j)}.$$

As a special example of application, we consider the case where the cost is incurred once (at time 0) and the discount rate d is constant from year to year; the formula of net present worth then becomes:

$$NPW = \sum_{k=1}^{n} \frac{B_k}{(1+d_k)} - C_0.$$

The *return on investment* is computed as the net present worth divided by the present worth of the costs. It is given by the formula

$$ROI = \frac{NPW}{\sum_{k=1}^{n} \frac{C_k}{\prod_{j=0}^{k-1}(1+d_j)}}.$$

In case the cost is incurred only once (at time 0) and the discount rate d is constant from year to year, the formula of return on investment becomes:

$$ROI = \frac{1}{C_0} \times \left(\sum_{k=1}^{n} \frac{B_k}{(1+d_k)} - C_0 \right).$$

3.2 Mathematics for Engineering Economics

Many of the formulas that we use to derive reusability measures and associated metrics include terms of the form

$$\sum \frac{1}{(1+d)^y}$$

where y is the variable of the sum. Variable d represents the discount rate (reflects the time-value of resources) and factor $\frac{1}{(1+d)^y}$ represents the depreciation of today's resources as a function of the year y. We consider in turn some of the recurrent patterns of such a formula and derive closed forms (forms that can be computed algorithmically in a finite number of steps) for them. We assume throughout this section that d is non zero (it is typically positive and small, in the neighborhood of 0.1).

This material will be presented without proofs; the interested reader is referred to sources on engineering mathematics for proofs.

Proposition 3.1
$$\sum_{y \geq 1} \frac{1}{(1+d)^y} = \frac{1}{d}.$$

Now we consider a finite sum, ranging from 1 to Y. We have the following proposition.

Proposition 3.2
$$\sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} = \frac{(1+d)^Y - 1}{d \times (1+d)^Y}.$$

We turn our attention to the sum of terms of the form $\frac{y}{(1+d)^y}$. We consider, in turn, infinite sums, then finite sums.

Proposition 3.3
$$\sum_{y \geq 1} \frac{y}{(1+d)^y} = \frac{1+d}{d^2}.$$

Proposition 3.4
$$\sum_{y=1}^{y=Y} \frac{y}{(1+d)^y} = \frac{(1+d)^{Y+1} - d \times Y - (d+1)}{d^2 \times (1+d)^Y}.$$

3.3 A Background on Measurement

3.3.1 Measurement Theory

Measurement is the process by which entities in a given sample are ordered according to a set of attributes. Typically, we consider a single attribute for each entity, but we may, occasionally, consider more than one attribute.

Examples of measurements include: measuring the height of players in a basketball team; measuring the performance of students in a class by means of their *grade point average* over two consecutive terms. The first example involves one attribute, whereas the second example involves two attributes.

The most natural way to define a measure —but not necessarily the only way— is to assign numbers to the attributes of interest; in doing so, we say that we have defined a *measure* or a *measurement mapping.*

To illustrate the definition of a measure, we consider that in order to measure the height of players we assign a quantity to each player, which represents the height of the players in some unit, e.g., feet and inches. Likewise, in order to measure the performance of students we assign to each student two numbers, one for each term being considered, where each number represents the score of the student for a term. Note that in the first example the measure defines a total ordering whereas in the second example the measure defines a partial ordering.

It is customary [30, 29] to distinguish between *direct measurement* and *indirect measurement.* The former deal with attributes for which measurement mappings can be defined, whereas the latter deal with attributes whose measurement mappings are defined in terms of other measurement mappings.

Measurements can be used for two distinct purposes: *assessment* and *prediction.* Predictive measurement of an attribute A depends on a mathematical model relating A to existing measures of attributes A_0, A_1, .. A_n. It is also dependent on procedures for determining the parameters of the model, as well as procedures for interpreting the results.

Hence, e.g., B. Boehm's predictive measurement of development effort is defined by the following features:

- A mathematical model of the form

  $$E = a \times S^b,$$

  which provides that the development effort (E) is the product of some constant (a) by the software size (S) raised to some exponent (b).

- Procedures for determining the parameters of the mathematical model, in this case a and b.

- Procedures for interpreting the results of applying the model.

There has been a great deal of interest, in the past, in measuring features of the software product and the software process. This discipline is known under the general name of

software metrics. In light of the above discussion, we will, henceforth, distinguish between *software metrics*, which refer to direct measurement of software product and process, and *software measures*, which refer to indirect measurement of software product and process. Hence, e.g., we talk about a *size metric* (a direct measurement of the product) and a *readability measure* (an indirect measurement of the product).

3.3.2 Representational Theory of Measurement

Direct measurement of a particular attribute of a set of entities must reflect the intuitive use of empirical relations related to the attribute. Hence, e.g., if we are interested in measuring the height of a set of players, we must have means to represent notions such as: "is tall", "is taller than", "is much taller than"; these are called *empirical relations*. An *empirical relations system* is defined whenever we are given: a set C of entities; a set R of empirical relations. The system is then represented as (C, R).

Representation Condition. Given an empirical relations system (C, R), we wish to define a measurement mapping M from C to $\mathcal{R}$, in such a way that empirical relations of R can be defined conveniently in terms of M. For example, if C is the set of players in a basketball team and R contains the empirical relations *is tall*, *is taller than*, *is much taller than*, *is as tall as*, then we define function M as: $M(c)$ = the height of c in meters, and we let the empirical relations be defined as follows.

- *Is tall*(c): $M(c) \geq 2.05$.
- *Is taller than*(c, c'): $M(c) > M(c')$.
- *Is much taller than*(c, c'): $M(c) > M(c') + 0.15$.
- *Is as tall as*(c, c'): $M(c) = M(c')$.

Note that the representation conditions may change if we change the set of entities; hence, e.g., if C were the population at large rather a basketball team, the definition of *Is tall* may be changed to a more moderate value.

Scale Types and Meaningfulness. For a given empirical relation system, we can find several measurements which satisfy the representation condition (M measure in inches or cm). When we can transform one valid representation into another, we call this transformation an *admissible transformation*. The class of admissible transformations determines the *scale type* for an attribute with respect to a fixed empirical relation system. In increasing order of sophistication, the best known scale types are: *nominal*, *ordinal*, *interval*, *ratio* and *absolute*.

Part II

Component Level ROI Model for Software Reuse

In this part we focus on defining measures of component reusability. To this effect, we analyze the lifecycles of software reuse, investigate the cost factors that intervene in each of the phases and activities of the lifecycle, then derive a return on investment model that involves these costs. We define our measure of reusability to be precisely this return on investment quantity. Chapters 4 and 5 follow this pattern, as it pertains to design for single use (synchronous cycle) and design for reuse (asynchronous cycle). Chapter 6 discusses a number of derived reusability measures, including long term reusability, intrinsic reusability, and break-even frequency.

Chapter 4

Design for Single Use

4.1 A Synchronous Reuse Process

The *synchronous* reuse lifecycle deals with software components that have been developed for a single use, as part of a larger system (say S), and are being considered for possible storage in a software reuse repository [4, 6, 5]. For the sake of argument, we assume that S has been developed following Boehm's *waterfall lifecycle* [14]; our study can be changed trivially to accommodate other lifecycles.

Prior to the decision of whether the component should be stored in the software repository, it proceeds through the following phases:

- **Component specification.** This takes place during the *Product Design* phase of system S, as defined by the waterfall lifecycle [14].
- **Component Design.** This takes place during the *Detailed Design* phase of system S, as defined by the waterfall lifecycle [14].
- **Coding.** This takes place during the *Coding* phase of system S, as defined by the waterfall lifecycle [14].

Following the *Coding* phase, a decision must be made as to whether the component should be stored in the repository for future reuse. If the decision is that it must be stored, then the component proceeds through the following phases:

- **Reengineering** . Typically, a component that is developed for a particular system cannot be stored as-is in a reuse library. Rather, the component must be generalized; the components' interface details that are specific to its original host system must be abstracted away; and it must be parameterized. We refer to all these activities as *reengineering.*
- **Verification.** Once it is reengineered, the component must be validated against its new (more general) requirements, and must be checked according to the quality standards of the host reuse library.

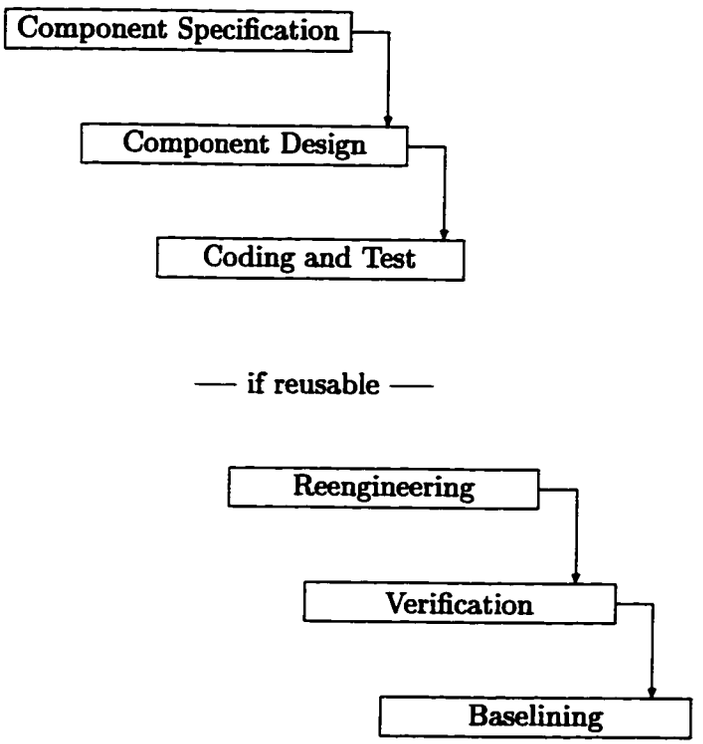

Figure 4.1: A Component's Lifecycle

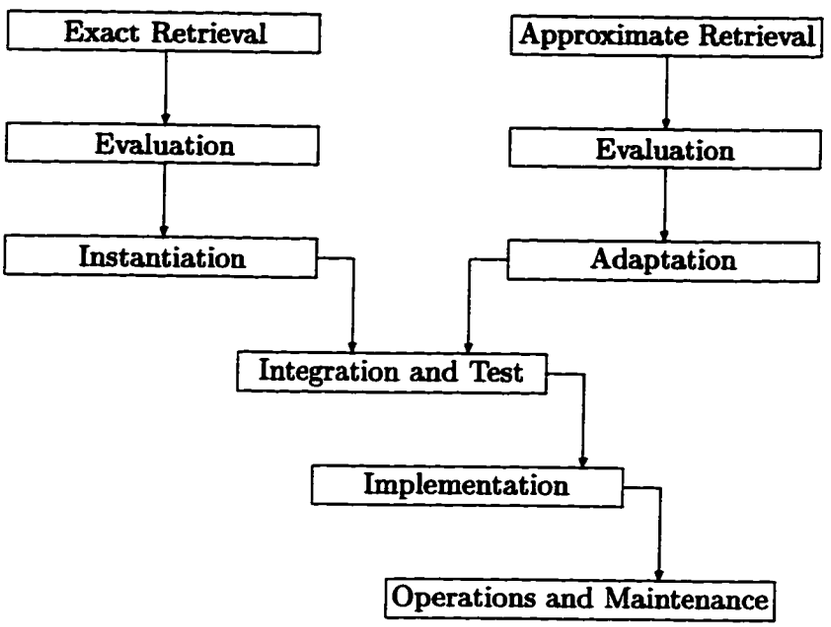

Figure 4.2: Lifecycle of a Retrieved Component

- **Baselining.** Once it is validated, the component is stored in the reuse database, following local baselining procedures.

Once it is stored, the component is available for retrieval and reuse; whenever it is selected for reuse, the component proceeds through the following phases:

- **Retrieval** . We distinguish between *exact retrieval* , when the component is used as-is, and *approximate retrieval* , when the component does not match the exact requirements submitted in the query, and must be adapted before reuse.[1] For the sake of completeness, we consider that even when a component is used as-is, some effort is still required for such endeavors as parameterizing the component, specializing it, tuning its interfaces, etc. We refer to this step as the *instantiation* of the component.

- **Integration and Test.** Once it is ready, the retrieved component is integrated into its host system, and the whole system is tested.

- **Implementation.** Once the system is ready, it must be implemented on the host computing environment [14].

- **Operations and Maintenance.** Following the implementation phase, the system is put in operation [14].

[1] What we refer to as exact retrieval is also known under the name *black box* or *verbatim*; what we refer to as *approximate retrieval* is also known under the name *white box*.

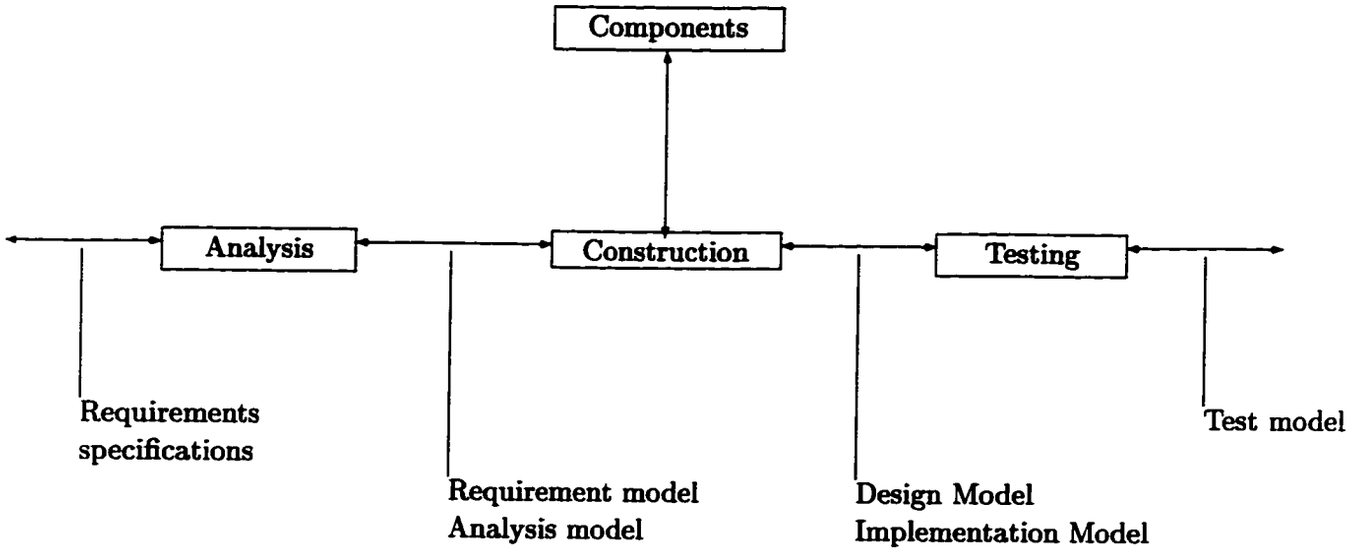

Figure 4.3: Objectory Lifecycle

4.2 An OOP Lifecycle

Because it embodies principles of good software design, object oriented programming promotes reusability. In this section we take a close look at object oriented programming, and discuss whether, how and to what extent adoption of this programming discipline affects the validity of our study of reusability measures.

4.2.1 Objectory: An Object Oriented Lifecycle

The Objectory (Object Factory for Software Development) lifecycle [39] consists of four communicating processes: *analysis*, *construction*, *testing* and *components* (see figure 4.3). Each process produces models of the system. The lifecycle consists of a transformation, by the processes, of one or several of these models into other models. The final model is a complete and tested description of the system.

The analysis process creates a conceptual picture of the system. Two models are built during this phase: the *requirement model* and the *analysis model*. These models are application oriented and no consideration of the implementation environment is taken. The requirement model captures the functional requirements of the system in terms of a description of the use of the system. In this model, the system is described as a number of *use cases* that are performed by a number of *actors*. Use cases represent complete

courses of events initiated by actors. Actors represent roles that the user can play. The analysis model gives a conceptual configuration of the system consisting of *control objects*, *entity objects* and *interface objects*. In this model we assume an ideal implementation environment.

The construction process develops the system and implements it. It produces two models: the *design model* and the *implementation model*. The design model adapts the analysis model to real implementation environments. Design objects are represented by *blocks*, and interaction diagrams provide information about blocks protocol (and therefore blocks interface). Finally, an intermediate level description (such as state transition graphs) is used to represent block behavior. The implementation model consists of code.

The component development process develops and maintains components to be used mainly during construction.

The testing process integrates the system and verifies it. During this process, a *test model* is developed to support the verification of the developed system. This model includes test cases.

4.2.2 Development and Storage Lifecycle

In [39], Jacobson proposes the following lifecycle for component development (chapter 6) and storage (chapter 11):

1. *Analysis.* This process produces the specification of the component as part of the *analysis model* [39].

2. *Construction.* This process includes two activities, which are *design* and *implementation* of the component. The design is part of the *design model* [39], and the implementation is part of the *implementation model* [39].

3. *Components.* This process includes all the steps that pertain to the evaluation and storage of the component in a reuse library. For our purposes, we consider that this process can be divided into three phases: *reengineering*, whereby the component is generalized and packaged; *verification*, whereby the modifications of the previous phase are validated; *baselining*, whereby the component is classified and stored.

Figure 4.4 reflects Jacobson's lifecycle, in which we superimpose Jacobson's terminology with our own.

4.2.3 Retrieval and Reuse Lifecycle

In [39] (section 11.3.2, page 300), Jacobson distinguishes between two patterns of component retrieval: black box reuse, when components are reused verbatim; and white box reuse, when components are adapted or specialized. Jacobson recognizes the difficulties and costs associated with component retrieval, as well as the necessity to evaluate components after retrieval and prior to their integration in their host system. All these activities are discussed under a process that Jacobson designates as the *components* process. Figure 4.5 illustrates the details of this process, and superimposes Jacobson's terminology with ours.

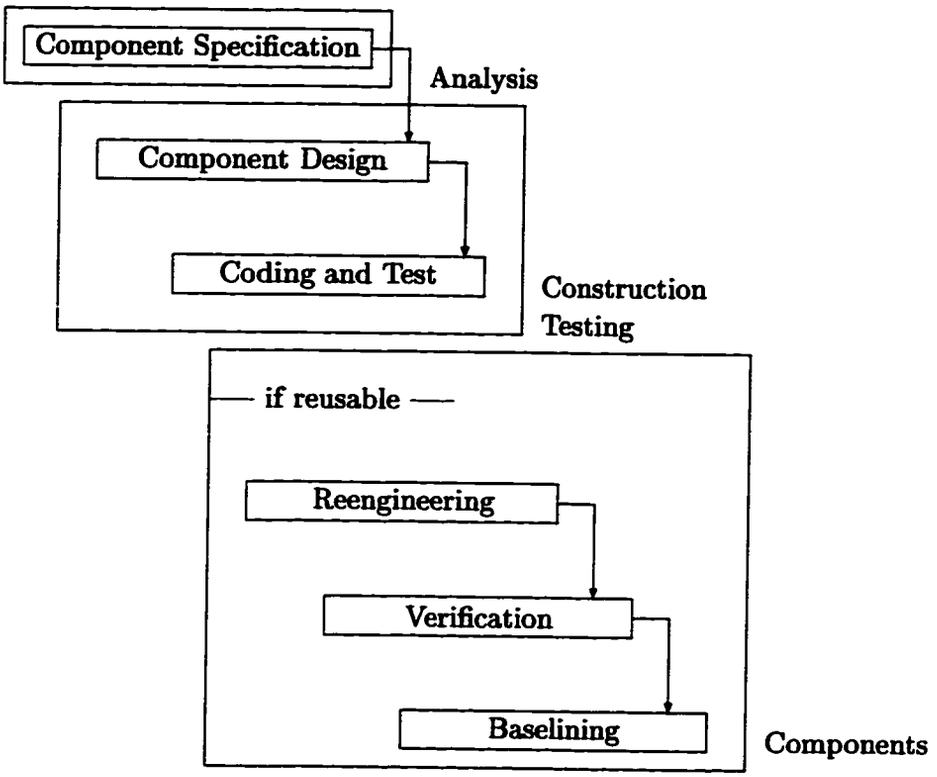

Figure 4.4: An OOP Component's Lifecycle

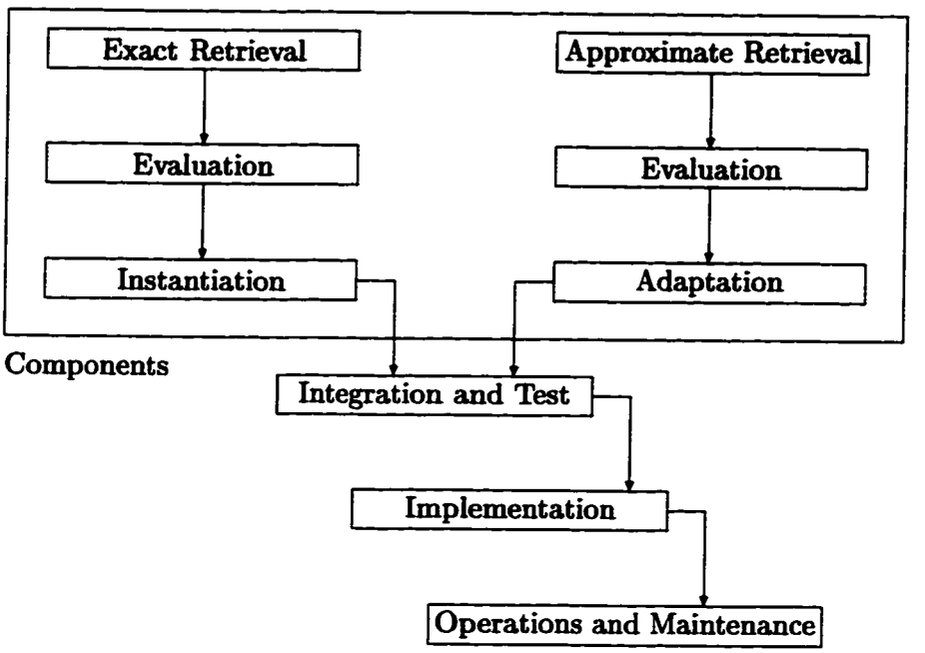

Figure 4.5: Lifecycle of a Retrieved Object Oriented Component

4.2.4 Impact of Object Oriented Paradigm

From the foregoing discussion, it appears that the object oriented paradigm does not have a great impact on the lifecycle of a reusable component. The processes that are discussed in object oriented sources ([39]) are similar to those that we have discussed in section 4.1. This seems to suggest that the same model can be used to assess the reusability of object oriented components, and that of traditional software components. There may be differences in the way we compute the factors that intervene in estimating reusability; these will be discussed in chapter 7.

4.3 Reuse Costs

We consider the lifecycle presented above and endeavor to identify the costs associated with all its phases; these costs will be used, in the sequel, to define a *return on investment* formula, which we define as our measure of reusability. In order to derive the ROI formula, we must review in turn the costs and benefits associated with each option (the option to store the component, and the option to forego it); for the sake of this discussion, we refer to both costs and benefits as *costs*, adjusting with negative signs as necessary.

Definition 4.1 *The* reusability *of a software asset is the return on investment, with respect to a given present cost, periodic cost/benefit, investment cycle and discount rate, associated with the decision to store the asset in a software library for the purpose of software reuse.*

4.3.1 Option: Storing the Component

We consider in turn the *investment costs*, which are incurred once when the option is selected, and the *operational costs*, which are incurred periodically (we let the period in question be a year).

- **Investment costs** include the *reengineering* cost (*Reeng*) and the *verification* cost (*Verif*); and the *baselining* cost (*Base*). We write:

$$IC_S = Reeng + Verif + Base.$$

The subscript $_S$ stands for the fact that we are dealing with the option to *store* the component.

  - *Reeng* is the cost of reengineering the component in order to prepare it for reuse
  - *Verif* is the cost of verifying the component to validate the reengineering step, and to ensure that the component meets reuse related quality standards
  - *Base* is the cost of storing the component in the reuse database following local baselining procedures

- **Operational costs** involve the following factors:
  - *Store*: the yearly cost of maintaining a component in storage. This cost includes two terms: the cost *per component* of operating the library; the impact that the presence of this component has on the performance of the retrieval algorithm (e.g., the component may be looked up but not retrieved), as well as on its precision (e.g., the component may be retrieved but subsequently found to be irrelevant).
  - $Freq_X$: the yearly frequency with which the component is reused verbatim (after instantiation). $Freq_A$: the yearly frequency with which the component is reused after adaptation. These two frequency factors reflect to what extent the component at hand is needed within the given application domain.

- *Exact*: the average/typical cost of an exact retrieval. This term reflects the retrieval procedure used in the organization at hand.

- *Instance*: The cost of instantiating a component to make it satisfy the specific requirements of the query at hand. This term reflects the complexity of the component's parameter structure.

- *Eval*: The cost of assessing the component and making the decision of whether to (re)use it or not. For the sake of simplicity, we assume that this cost is the same for exact retrieval and approximate retrieval, although it may be argued that in practice it is different.

- *Aprox*: The average/typical cost of an approximate retrieval. This term reflects the approximate retrieval procedure used in the organization at hand.

- *Adapt*: The cost of adapting a retrieved software component to make it satisfy the requirements of the query at hand; presumably, the query is sufficiently close to the functional specification of the component to make this adaptation step cost effective. This term reflects the structural properties of the component (such as: complexity, modularity, structure, documentation, and readability).

- *Rest*: the cost of completing the remainder of the lifecycle (i.e., *Integration and test*, *Implementation*) for the component at hand, in a typical/average host system. This cost factor depends primarily on the structural features of the component at hand, since these determine how easy it is to integrate this component in a host system.

- *Oper*: the yearly cost of operating and maintaining a retrieved instance of this component. This factor depends primarily on the quality of the component —a good quality component costs less to maintain and operate.

Given the cost factors defined above, we can define factor *Retrieve*, which is the cost of retrieving components and instantiating them (in the case of exact retrieval) or adapting them (in the case of approximate retrieval); factor *Retrieve* is given by the following formula:

$$\begin{aligned} Retrieve &= Freq_X \times (Exact + Eval + Instance + Rest) + \\ &\quad Freq_A \times (Aprox + Eval + Adapt + Rest). \end{aligned}$$

- $Freq_X$ is the yearly frequency with which the component is reused verbatim (after instantiation)
- *Exact* is the average/typical cost of an exact retrieval
- *Eval* is the cost of assessing the component and making the decision of whether to (re)use it or not.
- *Instance* is the cost of instantiating a component to make it satisfy the specific requirements of the query at hand.
- *Rest* is the cost of completing the remainder of the lifecycle (i.e., *Integration and test*, *Implementation*)
- $Freq_A$ is the yearly frequency with which the component is reused after adaptation
- *Aprox* is the average/typical cost of an approximate retrieval This term reflects the approximate retrieval procedure used in the organization at hand.
- *Adapt* is the cost of adapting a retrieved software component to make it satisfy the requirements of the query at hand

Also, we are interested in factor *Maintain*, which measures the yearly effort of maintaining instances of component C, which have been retrieved in past years and are being used in current applications; the reason why we introduce this factor is to take into account the difference in quality that we may observe between a reused component and a component developed from scratch. Because this factor depends on the year y (counted since component C has been included in the library), we denote it by $Maintain(y)$, and define it as:

$$Maintain(y) = (y - 1) \times Freq \times Oper,$$

where $Freq = Freq_X + Freq_A$.

- y is the year counted since the component has been included in the library
- *Freq* is the total yearly frequency of reuse of the component
- *Oper* is the yearly cost of operating and maintaining a retrieved instance of the component

Now, the yearly operational cost of component C at year y (counted from the time the component is included in the software base), if we choose to store component C

in the software library, is given by the following formula:

$$OC_S(y) = Store + Retrieve + Maintain(y).$$

4.3.2 Option: Foregoing the Component

If we decide to forego component C, then we incur no investment costs; hence IC_F (where the subscript $_F$ stands for the option to *forego* the component) is given by:

$$IC_F = 0.$$

On the other hand, component $OC_F(y)$ no longer has a *Store* term, and no *Retrieve* term, since no instances of C will be stored and retrieved. Rather, instead of retrieval cost, we will have a term to measure the cost of development. We find:

$$OC_F(y) = Develop + Maintain_F(y),$$

where

- *Develop*, the cost of developing components from scratch to fulfill the function of C, is given by the following formula:

  $$Develop = Freq \times (First + Rest).$$

  First, the cost of carrying out the first phases of component C's development lifecycle, includes the cost of *component specification*, *component design* and *component coding*.

  - *Freq* is the total yearly frequency of reuse of the component
  - *First* is the cost of developing the component (i.e., *component specification*, *component design* and *component coding*)
  - *Rest* is is the cost of completing the remainder of the lifecycle (i.e., *Integration and test*, *Implementation*)

- $Maintain_F(y)$, the cost of maintaining those components that, instead of being retrieved (as instances or variations of C), were developed from scratch, is given by the formula:

  $$Maintain_F(y) = (y-1) \times Freq \times Oper'.$$

- y is the year counted since the component has been included in the library
- $Freq$ is the total yearly frequency of reuse of the component
- $Oper'$ is the yearly cost of operating and maintaining a retrieved instance of the component that was developed from scratch

Typically, $Oper'$ is greater than $Oper$, to account for the difference in quality between a reused component and a component that is developed from scratch.

4.4 The Net Present Worth

Given that we have analyzed the costs associated with the option of storing component C and the option of foregoing it, we now have a quantitative basis for making the decision. For the sake of generality and abstraction, we will not use a monetary unit (e.g., the US dollar) as our unit of cost; rather we use the person-month. Also, we recognize that the cost formula has two kinds of factors: *investment costs*, i.e., costs that must be incurred immediately; *operational costs*, i.e., costs that must be incurred periodically in the future. In order to make comparisons meaningful, we must formulate these costs in terms of a common unit, say today's person-month. To this effect, we introduce two additional factors:

- **The discount rate.** If investing 1 person-month will save me $(1+x)$ person-months in a year, what is the minimal value of x for which I agree to invest? That minimal value of x is the discount rate, which we denote by d.
- **The investment period**, which we denote by Y and measure in number of years. This is the period during which we wish to assess the cost effectiveness of our investment decision: we do not agree to invest in storing a component today unless we can retrieve our investment cost within a period of Y years.

Faced with the two options, to store a component or to forego it, a software manager assesses the costs/benefits associated with each, which he expresses in terms of today's person-months, then takes his decision accordingly. To provide a basis for this decision, we introduce the following factors:

- **Investment Cost Differential**, defined as: $IC = |IC_F - IC_S|$.
- **Operational Cost Differential**, defined as: $OC(y) = |OC_F(y) - OC_S(y)|$.

Using factors IC and $OC(y)$, it is possible to derive the expected cost/benefit associated with the decision to store component C; this is known in the *engineering economics literature* as the *net present worth* (NPW) [64] and is given by the following formula

$$NPW = \Sigma_{y=1}^{y=Y} \frac{OC(y)}{(1+d)^y} - IC.$$

If NPW is positive, it is *in principle* advantageous to store the component; NPW then represents the savings achieved over Y years by storing C. If NPW is negative, it is *in principle* advantageous not to store component C; the absolute value of NPW represents the loss incurred over Y years by keeping C in the software library.

We consider two components C and C', whose net present worth is say, 3 person-months; further we assume that storing C requires an upfront investment of 2 person-months whereas storing C' requires an upfront investment of 20 person-months. Even though C and C' both have the same net present worth, we cannot consider that they are equally reusable; surely, C is a great deal more reusable than C', because of the risks involved in investing 20 person months for the sake of saving 3 person-months (a budget overrun of fifteen percent over the investment effort, not far-fetched in software engineering, will wipe out all the expected savings). Hence, despite its benefits [28], we will not adopt the net present worth as our measure of reusability; rather, we choose a return on investment formula.

4.5 A Return On Investment Model

The *return on investment* (ROI), defined as the ratio between the net present worth and the upfront investment, is more faithful to our intuition regarding a measure of reusability. Hence the following definition.

Return on Investment, synchronous process.

Definition 4.2 *Given a component C whose investment cost differential is IC, and whose operational cost differential is $OC(y)$, the* return on investment (ROI) *of C for discount rate d and investment period Y is:*

$$Reu = \frac{1}{IC} \times \left(\Sigma_{y=1}^{y=Y} \frac{OC(y)}{(1+d)^y} - IC \right).$$

In the example above, we find that the return on investment of C is 1.5 (an abstract number, without unit) whereas the return on investment of C' is 0.15.

Chapter 5

Design for Reuse

5.1 An Asynchronous Reuse Process

5.1.1 Component Lifecycle

The *asynchronous* reuse process models the situation where the experience factory decides to create a software component and store it, on the belief that its reuse potential justifies the investment cost. The development of such a component proceeds in the same manner as the development of a *single use* component, except for some details, such as:

- A greater emphasis on reliability, on the premise that any effort it may take to enhance the quality of the component can be justified by the number of uses of this component.
- A greater emphasis on design integrity, on the premise that a large number of potential reusers are going to inspect the code as part of an adaptation process.
- A greater emphasis on generality, on the premise that a large number of potential reusers are going to inspect the component' specification as part of an instantiation process.

The difference in development costs that these concerns generate is widely recognized in the literature, and fairly well documented [43].

5.2 Reuse Costs

In the asynchronous reuse process, the decision we have to ponder is not whether to store or not to store a component that was already developed; rather it is whether to develop a component for reuse or not. An inspection of the cost model presented in section 4.1 shows that all the factors introduced for the synchronous process apply equally well to the asynchronous process.

If we decide to develop a component for reuse, we incur the following investment costs:

$$IC = DR$$

. where DR represents the cost of development for reuse.
The operational cost $OC(y)$ is given by the formula

$$OC(y) = Store + Retrieve + Maintain(y),$$

- *Store* is the yearly cost of maintaining a component in storage

where

$$\begin{aligned} Retrieve &= Freq_X \times (Exact + Eval + Instance + Rest) + \\ &\quad Freq_A \times (Aprox + Eval + Adapt + Rest) \end{aligned}$$

- $Freq_X$ is the yearly frequency with which the component is reused verbatim (after instantiation)
- *Exact* is the average/typical cost of an exact retrieval
- *Eval* is the cost of assessing the component and making the decision of whether to (re)use it or not.
- *Instance* is the cost of instantiating a component to make it satisfy the specific requirements of the query at hand.
- *Rest* is the cost of completing the remainder of the lifecycle (i.e., *Integration and test*, *Implementation*)
- $Freq_A$ is the yearly frequency with which the component is reused after adaptation
- *Aprox* is the average/typical cost of an approximate retrieval This term reflects the approximate retrieval procedure used in the organization at hand.
- *Adapt* is the cost of adapting a retrieved software component to make it satisfy the requirements of the query at hand

and

$$Maintain(y) = (y - 1) \times Freq \times Oper_R.$$

- y is the year counted since the component has been included in the library
- $Freq$ is the total yearly frequency of reuse of the component
- $Oper_R$ is the yearly cost of operating and maintaining a retrieved instance of the component that was developed for reuse

5.3 A Return On Investment Model

In light of the discussions above, we propose the following definition for the reusability measure in the context of the asynchronous reuse process.
Reusability, asynchronous process.

Definition 5.1 *Given a component C whose development for reuse cost is DR, and whose operational cost differential is $OC(y)$, the* return on investment (ROI) *of C for discount rate d and investment period Y is:*

$$Reu = \frac{1}{DR} \times \left(\Sigma_{y=1}^{y=Y} \frac{OC(y)}{(1+d)^y} - DR \right).$$

While this definition is formally quite similar to that of synchronous processes, its application in practice differs significantly from the application of the synchronous definition. In the synchronous case, when the decision to store/ not to store arises, part of the development cost is known ($First$) and the other part can be estimated with little risk of error (IC). By contrast, in the asynchronous case, the development cost (DR) is not known; in chapter 7 we discuss means to estimate this cost.

Chapter 6

Extensions: Derived Measures

In the previous two chapters, we have defined an ROI formula that quantifies reuse worthiness of a component using a specific investment cycle (Y) and specific component frequencies ($Freq_X$ and $Freq_A$). In this chapter we discuss three auxiliary measures that can be derived from our original formula: the *long term ROI* (also called *indefinite term ROI*), which is the formula of ROI that we find for $Y = \infty$; the *intrinsic ROI* (also called *ROI gradient*) which is the derivative of the ROI formula with respect to the reuse frequency; finally, the *break even frequency*, which is the frequency for which the ROI formula is zero (or, more generally, the frequency for which the ROI takes a special value ρ).

6.1 Long Term ROI

6.1.1 A Closed Formula for Long Term ROI

If a company intends to operate a software reuse program on a long term basis, then it is necessary to consider that the benefits of storing a component will be cumulated over an indefinite number of years. Also, if a reuse library managers wishes to prove that a component is not reusable at all, one way is to prove that its ROI is negative even for $Y = \infty$ (in other words, we will not recover our investment costs even if we are infinitely patient!).

In order to accommodate these situations, we consider the ROI formula of Reu, where we let Y be infinite: we let this be denoted by Reu_∞ and rewrite it as:

$$Reu_\infty = \left(\frac{1}{IC} \times \sum_{y \geq 1} \frac{OC(y)}{(1+d)^y} \right) - 1.$$

In order to derive a closed formula for Reu_∞, we consider the formula of operational cost differential, as given in the previous section:

$$\begin{aligned} OC(y) &= OC_F(y) - OC_S(y) \\ &= Freq_x \times (First - Exact - Eval - Instance) + \end{aligned}$$

$$\begin{aligned} &Freq_A \times (First - Aprox - Eval - Adapt) + \\ &Freq \times (y-1) \times (Oper' - Oper) - \\ &Store. \end{aligned} \tag{6.1}$$

We note that some terms (the first, second and last) are independent of y, and can be assumed to be constant from year to year. On the other hand, we find a term (viz. the third), which is a linear function of y (if we assume the other factors to be constant from year to year, which is quite reasonable). In order to reflect this feature we rewrite the equation of ROI as:

$$Reu_\infty = \left(\frac{1}{IC} \times \sum_{y \geq 1} \frac{A + B \times y}{(1+d)^y} \right) - 1.$$

We obtain this equation by consolidating all the constant terms (that do not involve factor y) into A and factoring y in all the other terms to obtain $B \times y$. Because both A and B are independent of y, they can be factored out of the sum formulas; this yields the following equation.

$$Reu_\infty = A \times \sum_{y \geq 1} \frac{1}{(1+d)^y} + B \sum_{y \geq 1} \frac{y}{(1+d)^y} - 1.$$

The developments of chapter 3 provide closed forms for the terms

$$\sum_{y \geq 1} \frac{1}{(1+d)^y}$$

and

$$\sum_{y \geq 1} \frac{y}{(1+d)^y}.$$

Proposition 3.1 provides:

$$\sum_{y \geq 1} \frac{1}{(1+d)^y} = \frac{1}{d}.$$

On the other hand, proposition 3.3 provides:

$$\sum_{y \geq 1} \frac{y}{(1+d)^y} = \frac{1+d}{d^2}.$$

In the sequel, we analyze the expression of $OC(y)$, highlight factors A and B, factor them out of the sums, and replace the sums by their value; this yields the following development.

$$
\begin{aligned}
& Reu_{\infty} \\
= & \quad \{ \text{ definition } \} \\
& \left(\frac{1}{IC} \times \sum_{y \geq 1} \frac{OC(y)}{(1+d)^y}\right) - 1 \\
= & \quad \{ \text{ substituting } OC(y) \} \\
& (\frac{1}{IC} \times \sum_{y \geq 1} \frac{1}{(1+d)^y} \times \\
& (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (y-1) \times (Oper' - Oper) - \\
& Store)) - 1 \\
= & \quad \{ \text{ we isolate constant terms from } y \text{ terms in the sum } \} \\
& (\frac{1}{IC} \times \sum_{y \geq 1} \frac{1}{(1+d)^y} \times \\
& (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (Oper - Oper') - \\
& Store + Freq \times y \times (Oper' - Oper))) - 1 \\
= & \quad \{ \text{ we distribute the sum over the constant term and the linear term } \} \\
& \frac{1}{IC} \times (\sum_{y \geq 1} \frac{1}{(1+d)^y} \times \\
& (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (Oper - Oper') - \\
& Store + \sum_{y \geq 1} \frac{y}{(1+d)^y} \times Freq \times (Oper' - Oper)) - 1 \\
= & \quad \{ \text{ factoring out the constants and subsituting the sums } \} \\
& \frac{1}{IC} \times \frac{1}{d} \times (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (Oper - Oper') - Store) + \\
& \frac{1}{IC} \times \frac{1+d}{d^2} \times Freq \times (Oper' - Oper)) - 1 \\
= & \quad \{ \text{ simplification } \} \\
& \frac{1}{IC \times d} \times (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (Oper - Oper') - Store) + \\
& \frac{1+d}{IC \times d^2} \times Freq \times (Oper' - Oper)) - 1.
\end{aligned}
$$

This yields the following definition.

Definition 6.1 *The measure of* long term ROI *of a component with respect to discount rate d is given by:*

$$\begin{aligned} Reu_{\infty} &= (Freq_X \times (First - Exact - Eval - Instance) + \\ &\quad Freq_A \times (First - Aprox - Adapt)) \times \frac{1}{IC \times d} \\ &\quad (Freq \times (Oper' - Oper)) \times \frac{1+d}{IC * d^2} \times -1. \end{aligned}$$

In fact the measure of long term ROI can be understood as an upper bound on the savings achieved by storing the component for software reuse, or can be understood as the exact savings if the perspective of the decision maker is indeed long term (the higher order terms tend to be small as y grows larger, as $\frac{1}{(1+d)^y}$ grows smaller).

6.1.2 Examples of Long Term ROI

Example. We consider the example of a component C that has the following characteristics, and we attempt to compute its long term ROI.

$d = 0.15$.
$First = 6.7PM$.
$Reeng = 0.6 \times First$
$Verif = 0.45 \times First$
$Base = 0.2 \times First$.
$Stor = 3PD$[1].
$Freq_X = 0.6$
$Freq_A = 1.0$.
$Exact = 1PD$.
$Eval = 1PD$.
$Instance = 0.25 \times First$.
$Aprox = 3PD$.
$Adapt = 0.70 \times First$.
$Oper' = 0.15 \times First$.
$Oper = 0.80 \times Oper'$.

With these assumptions, we find that the long term ROI of component C with discount rate of 15 percent is:

$$Reu_{\infty} = 4.64.$$

In other words, the saving that the organization achieves in the long run by storing this component for future reuse is 4.64 times today's investment. □

[1] Person-Day.

Example. For the sake of experimentation, we change the cost factors as follows (we reduce the frequency of reuse, increase the investment costs, and reduce the benefits per instance of reuse), and attempt to reevaluate the long term ROI:

$Freq_X = 0.3$
$Freq_A = 0.5$
$Reeng = 0.7 \times First$
$Verif = 0.5 \times First$
$Base = 0.2 \times First$
$Exact = 1PD.$
$Eval = 1PD.$
$Instance = 0.3 \times First$
$Aprox = 3PD.$
$Adapt = 0.75 \times First.$

We find that the long term ROI of this component is now:

$$Reu_{\infty} = 1.28PM.$$

In other words, the savings that the organization achieves in the long run by storing this component for future reuse is only 1.28 times today's investment. The incentive to store this component is much smaller now than in the previous example. □

6.2 Intrinsic ROI

6.2.1 Intrinsic Limited Term ROI

In the previous two chapters, we have observed that our ROI depends on two families of factors: components specific factors, such as reengineering costs, verification costs, operational costs; and domain specific costs, such as the frequency of exact retrieval, the frequency of approximate retrieval, and the frequency of retrieval (exact or approximate). In this section, we wish to investigate an ROI that reflects exclusively the features of the component; we call it the measure of *intrinsic ROI*.[2] Hence we may find that a component has a low ROI measure for a particular reuse library, and if it has a high intrinsic ROI, consider it for inclusion in another library. Also, the measure of intrinsic ROI can be used to compare components in cases where we have no information on their frequency of reuse: then we compare them on the basis of intrinsic ROI.

The basic idea of intrinsic ROI is to compute the derivative of the ROI with respect to the frequency of reuse. Because we have three frequency parameters, we need to establish relationships between them to bring them down to one parameter. We already have the equation:

[2]Intrinsic ROI reflects properties of the component that are known to promote reusability, such as: design integrity, clarity, modifiability, generality, reliability, etc. Unlike reuse frequency, which reflects the application domain and the development environment (in addition to being dependent on the component), intrinsic ROI depends exclusively on properties of the component.

$$Freq = Freq_X + Freq_A.$$

We define the *frequency ratio* of a component C as the ratio between the frequency of exact reuse and the total frequency of reuse. From these two equations we derive:

$$Freq_X = \lambda \times Freq.$$
$$Freq_A = (1 - \lambda) \times Freq.$$

Using these equations, we can now consider the ROI formula again and formulate as a function of variable $Freq$. We find:

$$\begin{aligned}
& Reu \\
= & \quad \{ \text{ definition } \} \\
& \left(\frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{OC(y)}{(1+d)^y}\right) - 1 \\
= & \quad \{ \text{ expanding } OC(y) \} \\
& OC_F(y) - OC_S(y) \\
= & \quad \{ \text{ expanding } OC_F(y) \text{ and } OC_S(y) \} \\
& -1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times \\
& (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (y-1) \times (Oper' - Oper) - \\
& Store). \\
= & \quad \{ \text{ substituting } Freq_X \text{ and } Freq_A \} \\
& -1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times \\
& (\lambda \times Freq \times (First - Exact - Eval - Instance) + \\
& (1-\lambda) \times Freq \times (First - Aprox - Eval - Adapt) + \\
& Freq \times (y-1) \times (Oper' - Oper) - \\
& Store). \\
= & \quad \{ \text{ factoring } Freq \text{ wherever it applies } \} \\
& \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Freq \times \\
& (\lambda \times (First - Exact - Eval - Instance) + \\
& (1-\lambda) \times (First - Aprox - Eval - Adapt) + \\
& (y-1) \times (Oper' - Oper)) \\
& -(1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store). \\
= & \quad \{ \text{ factoring } Freq \text{ out of the sum } \} \\
& Freq \times \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times \\
& (\lambda \times (First - Exact - Eval - Instance) + \\
& (1-\lambda) \times (First - Aprox - Eval - Adapt) + \\
& (y-1) \times (Oper' - Oper))
\end{aligned}$$

$$-(1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store).$$

This formula is of the form

$$Reu = A \times Freq + B.$$

The derivative of this function with respect to $Freq$ is factor A. We get the following definition.

Definition 6.2 *The* intrinsic ROI *of component C is given by the formula:*

$$\begin{aligned} IR &= ((Aprox + Adapt - Exact - Eval - Instance) \times \lambda + \\ &\quad (First - Aprox - Adapt - Oper' + Oper)) \times \frac{(1+d)^Y - 1}{IC \times d \times (1+d)^Y} + \\ &\quad (Oper' - Oper) \times \frac{(1+d)^{Y+1} - d \times Y - (d+1)}{IC \times d^2 \times (1+d)^Y}. \end{aligned}$$

IR can be rewritten as

$$\begin{aligned} IR &= C \times (\lambda - 1) \times Aprox + C \times (\lambda - 1) \times Adapt - \lambda \times C \times Exact - \\ &\quad \lambda \times C \times Instance - \lambda \times C \times Eval + C \times First + \\ &\quad (D - C) \times Oper' - (D - C) \times Oper \end{aligned}$$

where

$$C = \frac{(1+d)^Y - 1}{IC \times d \times (1+d)^Y}$$

$$D = \frac{(1+d)^{Y+1} - dY - (1+d)}{IC \times d^2 \times (1+d)^Y}$$

and

$$D - C = \frac{(1+d)^Y - (1+dY)}{IC \times d^2 \times (1+d)^Y}$$

We observe that in order for IR to be positive, the following inequality must hold:

$$\begin{aligned} C \times First + (D - C) \times Oper' \geq\ & C \times (1 - \lambda) \times Aprox + C \times (1 - \lambda) \times Adapt + \\ & C \times \lambda \times Exact + C \times \lambda \times Instance + \\ & C \times \lambda \times Eval + (D - C) \times Oper \end{aligned}$$

After simplification we find:

$$First + \frac{(1+d)^Y - (1+dY)}{d \times ((1+d)^Y - 1))} \times Oper' \geq (1-\lambda) \times (Aprox + Adapt) + \lambda \times (Exact + Eval + Instance) + \frac{(1+d)^Y - (1+dY)}{d \times ((1+d)^Y - 1))} \times Oper$$

This result can be interpreted as follows: if the cost of of finding a component (either by approximate retrieval or exact retrieval), adapting it or instantiating it and maintaining it is smaller than the cost of developing a similar component from scratch and maintaining it then the intrinsic reusability of this component is positive.

Intuitively, intrinsic ROI represents the gain offered by this component for each instance of reuse per year, as a ratio of the investment cost. For example, if the investment cost is 10 person days and the intrinsic ROI is found to be 2.3, we conclude that each increase of one to the yearly frequency of reuse increases our manpower savings by 23 person days over the whole investment cycle.

Example. We reconsider the example of component C, whose long term ROI we had computed in section 6.1.2; we compute now the intrinsic ROI of the same component. To this effect, we no longer need reuse frequency data; rather, we only need to know the frequency ratio. For the sake of this example, we take this ratio to be $\lambda = 0.66$: this reflects a situation where we have two exact retrievals for each approximate retrieval. Also, we take the investment cycle $Y = 3$. We find:

$d = 0.15$.
$Y = 3$.
$First = 6.7PM$.
$Reeng = 0.6 \times First$
$Verif = 0.45 \times First$
$Base = 0.2 \times First$.
$Stor = 3PD$.
$\lambda = 0.66$
$Exact = 1PD$.
$Exact = 1PD$.
$Instance = 0.25 \times First$.
$Aprox = 3PD$.
$Adapt = 0.70 \times First$.
$Oper' = 0.15 \times First$.
$Oper = 0.80 \times Oper'$.

We find,

$$IR = 1.1.$$

This means that for each increase of 1 in the annual reuse frequency, we stand to gain a saving of 1.1 times the investment costs (which is $IC = 1.25 \times First = 8.37$) over the

investment period of 3 years. Unless the frequency of this component is known to be very low (under 1) and we are certain of our estimate, this component is a good candidate for reuse. □

6.2.2 Intrinsic Long Term ROI

We could, in fact, follow the same argument for long term ROI, to define a measure of intrinsic long term ROI. To this effect, we consider again the definition of long term ROI, viewed as a function of the single parameter $Freq$, then we derive the expression with respect to $Freq$.

$$
\begin{aligned}
& Reu_{\infty} \\
= & \quad \{ \text{ definition 6.1 } \} \\
& \frac{Freq_X \times (First - Exact - Eval - Instance) + Freq_A \times (First - Aprox - Adapt) - Stor}{d \times (Reeng + Verif + Base)} + \\
& \frac{(1+d) \times (Freq) \times (Oper' - Oper)}{d^2 \times (Reeng + Verif + Base)} - 1. \\
= & \quad \{ \text{ substituting } Freq_X,\ Freq_A, \text{ factoring } \} \\
& Freq \times \left(\frac{\lambda \times (First - Exact - Eval - Instance) + (1-\lambda) \times (First - Aprox - Adapt) - Stor}{d \times (Reeng + Verif + Base)} + \right. \\
& \left. \frac{(1+d) \times (Oper' - Oper)}{d^2 \times (Reeng + Verif + Base)} \right) - 1.
\end{aligned}
$$

If we note that this is a linear function of $Freq$ and we derive it with respect to variable $Freq$, we find the formula given in the following definition.

Definition 6.3 *The* intrinsic long term ROI *of a component C is denoted by IR_{∞} and given by the following formula:*

$$
\begin{aligned}
IR_{\infty} \quad = \quad & ((Aprox + Adapt - Exact - Eval - Instance) \times \lambda + \\
& (First - Aprox - Adapt - Oper' + Oper)) \times \frac{1}{IC \times d} + \\
& (Oper' - Oper) \frac{1+d}{IC \times d^2}.
\end{aligned}
$$

The intrinsic long term ROI represents the amount of effort saved over the long term by each increase of one in the reuse frequency, as a proportion of the investment cost.

Example. Now we consider the same component, and propose to derive its intrinsic long term ROI. We find,

$$IR_{\infty} = 4.46.$$

An increase of 1 in the reuse frequency yields savings of 1.1 times the investment cost over a period of three years, and yields the investment costs times 4.46 over the long

(indefinite) term. □

6.3 Break-even Frequencies

6.3.1 Limited Term Break-even Frequency

We consider a situation where we are interested in evaluating the ROI of a component, but find it difficult to estimate the reuse frequency. One way to deal with this situation is to simplify the question of frequency: instead of asking the question

> How often do you estimate that this component will be used?

we wish to derive a privileged value of frequency, and ask the following question

> Do you suppose this component will be used more often or more seldom than this figure?

The privileged frequency we are interested in is that for which the ROI takes a special value, say ρ. Imagine for example that, as a corporate or departmental policy, no component is stored in the reuse library unless its ROI over the period Y (finite or infinite) is greater than or equal to ρ. Then the break-even frequency is that for which the ROI is indeed equal to ρ. In order to derive the break-even frequency for value ρ we solve the equation

$$Reu = \rho$$

in variable $Freq$. We find:

$$
\begin{array}{ll}
 & Reu = \rho \\
\Leftrightarrow & \quad \{ \text{ substituting } Reu \} \\
 & \rho = Freq \times \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times \\
 & (\lambda \times (First - Exact - Eval - Instance) + \\
 & (1-\lambda) \times (First - Aprox - Eval - Adapt) + \\
 & (y-1) \times (Oper' - Oper)) \\
 & -(1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store). \\
\Leftrightarrow & \quad \{ \text{ equivalence } \} \\
 & \rho + 1 + \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store \\
 & = \quad Freq \times \frac{1}{IC} \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times \\
 & (\lambda \times (First - Exact - Eval - Instance) + \\
 & (1-\lambda) \times (First - Aprox - Eval - Adapt) + \\
 & (y-1) \times (Oper' - Oper)) \\
\Leftrightarrow & \quad \{ \text{ multiplying by } IC \text{ on both sides } \}
\end{array}
$$

$$IC \times (\rho + 1) + \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store$$
$$= \quad Freq \times \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times$$
$$(\lambda \times (First - Exact - Eval - Instance) +$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) +$$
$$(y - 1) \times (Oper' - Oper))$$

$\Leftrightarrow$ { breaking down the right hand sum }

$$IC \times (\rho + 1) + \sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times Store$$
$$= \quad Freq \times (\sum_{y=1}^{y=Y} \frac{1}{(1+d)^y} \times$$
$$(\lambda \times (First - Exact - Eval - Instance) +$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) +$$
$$\sum_{y=1} y = Y \frac{y}{(1+d)^y} \times (Oper' - Oper))$$

$\Leftrightarrow$ { substituting the sum of $\frac{1}{(1+d)^y}$, proposition 3.2 }

$$IC \times (\rho + 1) + \frac{(1+d)^Y - 1}{d \times (1+d)^Y} \times Store$$
$$= \quad Freq \times (\frac{(1+d)^Y - 1}{d \times (1+d)^Y} \times$$
$$(\lambda \times (First - Exact - Eval - Instance) +$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) +$$
$$\sum_{y=1} y = Y \frac{y}{(1+d)^y} \times (Oper' - Oper))$$

$\Leftrightarrow$ { substituting the sum of $\frac{y}{(1+d)^y}$, proposition 3.4 }

$$IC \times (\rho + 1) + \frac{(1+d)^Y - 1}{d \times (1+d)^Y} \times Store$$
$$= \quad Freq \times (\frac{(1+d)^Y - 1}{d \times (1+d)^Y} \times$$
$$(\lambda \times (First - Exact - Eval - Instance) +$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) +$$
$$\frac{((1+d)^Y - 1)(d+1)}{(1+d)^Y \times d^2} \times (Oper' - Oper))$$

$\Leftrightarrow$ { We multiply both sides by $(1 + d)^Y \times d^2$ }

$$(1 + d)^Y \times d^2 \times IC \times (\rho + 1) + d \times ((1 + d)^Y - 1) \times Store$$
$$= \quad Freq \times ((1 + d)^Y - 1) \times d \times$$
$$(\lambda \times (First - Exact - Eval - Instance) +$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) +$$
$$((1 + d)^Y - 1) \times (d + 1) \times (Oper' - Oper))$$

$\Leftrightarrow$ { solving the equation in $Freq$ }

$$Freq =$$
$$\frac{(1 + d)^Y \times d^2 \times IC \times (\rho + 1) + d \times ((1 + d)^Y - 1) \times Store)}{((1 + d)^Y - 1) \times d \times (\lambda \times (First - Exact - Eval - Instance) +}$$
$$(1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) +$$
$$((1 + d)^Y - 1) \times (d + 1) \times (Oper' - Oper)).$$

Hence the following definition.

Definition 6.4 *The* break-even frequency *of component C for ROI value ρ is the value of annual reuse frequency for which the ROI takes value ρ. It is denoted by $Bef(\rho)$ and defined by:*

$$Bef(\rho) = \left((\rho+1) + \frac{Store}{IC} \times \left(\frac{(1+d)^Y - 1}{d \times (1+d)^Y}\right)\right) \times \frac{1}{IR}.$$

Of particular interest is the break-even frequency of a component for value zero of the ROI: this is the frequency at which the return equals the investment. We get the following definition.

Definition 6.5 *The* normal break-even frequency *of component C is the value of annual reuse frequency for which the ROI takes value 0. It is denoted by Nbf and defined by:*

$$Nbf = \frac{((1+d)^Y \times d^2 \times IC + d \times ((1+d)^Y - 1) \times Store)}{((1+d)^Y - 1) \times d \times (\lambda \times (First - Exact - Eval - Instance)} + (1-\lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper')) + ((1+d)^Y - 1) \times (d+1) \times (Oper' - Oper)).$$

Example. As an illustration of this definition, we propose, in the sequel, to compute the normal break-even frequency of component C whose pertinent information is given below:

$d = 0.15$.
$Y = 3$ years.
$First = 6.7PM$.
$Reeng = 0.6 \times First$
$Verif = 0.45 \times First$
$Base = 0.2 \times First$.
$Stor = 3PD$.
$\lambda = 0.66$.
$Freq_A = 1.0$.
$Exact = 1PD$.
$Eval = 1PD$.
$Instance = 0.25 \times First$.
$Aprox = 3PD$.
$Adapt = 0.70 \times First$.
$Oper' = 0.15 \times First$.
$Oper = 0.80 \times Oper'$.

We find,

$$Nbf = 0.73.$$

As soon as we know that this component is used with frequency $Freq = 0.73$ (once every sixteen months) we know that this component will recover its cost over three years. □

6.3.2 Long Term Break Even Frequency

We are interested in deriving the break even reuse frequency under the hypothesis that the investment cycle is infinite; this is what we call the *long term break even frequency.* To derive this frequency, we consider the formula of long term ROI and solve the equation $\rho = Reu_{\infty}$ for variable $Freq$. We find:

$$
\begin{array}{ll}
& Reu_{\infty} = \rho \\
\Leftrightarrow & \quad \{ \text{ definition of long term ROI } \} \\
& \frac{1}{IC \times d} \times (Freq_x \times (First - Exact - Eval - Instance) + \\
& Freq_A \times (First - Aprox - Eval - Adapt) + Freq \times (Oper - Oper') - Store) + \\
& \frac{1+d}{IC \times d^2} \times Freq \times (Oper' - Oper)) - 1 = \rho \\
\Leftrightarrow & \quad \{ \text{ substituting } Freq_X \text{ and } Freq_A \} \\
& \frac{1}{IC \times d} \times (Freq \times \lambda \times (First - Exact - Eval - Instance) + \\
& Freq \times (1 - \lambda) \times (First - Aprox - Eval - Adapt) + Freq \times (Oper - Oper') - Store) + \\
& \frac{1+d}{IC \times d^2} \times Freq \times (Oper' - Oper)) - 1 = \rho \\
\Leftrightarrow & \quad \{ \text{ factoring out } Freq, \text{ passing constant on opposite side } \} \\
& Freq \times \frac{1}{IC \times d} \times (\lambda \times (First - Exact - Eval - Instance) + \\
& (1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper') + \\
& \frac{1+d}{IC \times d^2} \times (Oper' - Oper)) \\
& = 1 + \rho + \frac{Store}{IC \times d} \\
\Leftrightarrow & \quad \{ \text{ we multiply on both sides by } IC \times d^2 \} \\
& Freq \times d \times (\lambda \times (First - Exact - Eval - Instance) + \\
& (1 - \lambda) \times (First - Aprox - Eval - Adapt) + (Oper - Oper') + \\
& (1 + d) \times (Oper' - Oper)) \\
& = IC \times d^2 \times (1 + \rho) + d \times Store \\
\Leftrightarrow & \quad \{ \text{ whence we derive the frequency } \}
\end{array}
$$

$$
Freq = \frac{IC \times d^2 \times (1 + \rho) + d \times Store}{d \times (\lambda \times (EG) + (1 - \lambda) \times (AG) + (Oper - Oper')) + (1 + d) \times (Oper' - Oper)}.
$$

where,

$$
EG = (First - Exact - Eval - Instance)
$$

and

$$AG = (First - Aprox - Eval - Adapt).$$

Whence the following definition.

Definition 6.6 *The* long term break even frequency *of component C for ROI value ρ is the annual reuse frequency for which long term ROI takes value ρ. It is given by the following formula:*

$$\frac{IC \times d^2 \times (1+\rho) + d \times Store}{d \times (\lambda \times (EG) + (1-\lambda) \times (AG) + (Oper - Oper'))} + (1+d) \times (Oper' - Oper).$$

A privileged value of ROI for which we want to determine the annual reuse frequency is $Reu_\infty = 0$. We get the following definition:

Definition 6.7 *The* normal long term break even frequency *of component C for ROI value ρ is the annual reuse frequency for which long term ROI takes value 0. It is given by the following formula:*

$$\frac{IC \times d^2 + d \times Store}{d \times (\lambda \times (EG) + (1-\lambda) \times (AG) + (Oper - Oper'))} + (1+d) \times (Oper' - Oper).$$

We illustrate this definition with an example.

Example. As an illustration of this definition, we propose, in the sequel, to compute the normal long term break even frequency of component C whose pertinent information is given below:

$d = 0.15$.
$Y = 3$ years.
$First = 6.7PM$.
$Reeng = 0.6 \times First$
$Verif = 0.45 \times First$
$Base = 0.2 \times First$.
$Stor = 3PD$.
$\lambda = 0.66$.
$Exact = 1PD$.
$Eval = 1PD$.
$Instance = 0.25 \times First$.
$Aprox = 3PD$.

$Adapt = 0.70 \times First.$
$Oper' = 0.15 \times First.$
$Oper = 0.80 \times Oper'.$
We find,

0.27.

In other words, if the annual reuse frequency is 0.27, we will break even at time infinity. Note that this frequency is significantly smaller than that which we found earlier for an investment cycle of 3 years (which was 0.73).

$Nbf = 0.73.$

Part III

Estimating the Return on Investment

Whereas the previous part deals with defining measures of reusability, this part deals with estimating reusability. In order for our reusability measure to be useful, we need to be able to estimate it at the time when the decision to store the component in a reuse library must be taken. To this effect, we have conducted a worldwide survey on the practice of software reuse in industry and on the associated costs. The original verbatim survey form is given in appendix A, and its rationale is discussed in appendix B. Because the survey was not conclusive we used existing costs models and controlled experimental studies to build our ROI model. Chapter 7 reports on our data collection effort then discusses all the cost models that are involved in estimating the reusability of a component prior to storing it. Chapter 8 presents an automated tool for estimating the reusability of a component based on the formulas introduced in chapter 7. Finally, chapter 9 discusses the validation of our ROI model by presenting analytical arguments to support the model.

Chapter 7

Reusability Cost Factors

The formulae of reusability we have presented in part 4.5 cannot be useful in practice unless we devise means to estimate the costs involved at the time when the reuse decision must be taken (before development, for the asynchronous process, and before storage, for the synchronous process). In this chapter, we use some of the results we have collected in our survey as well as results of other empirical studies of software reuse to derive means to estimate the cost factors involved in our formula of reusability. We will consider in turn strategic cost factors, domain cost factors, as well as component specific cost factors. First, we report briefly on our data collection effort, in the next section.

7.1 Data Collection and Analysis

Armed with our survey form (see appendices A and B), we have set out to conduct a worldwide survey on the practice of software in software development organizations. To this effect we have contacted governmental organizations and industries in the Ottawa area which are known to develop and reuse software (Statistics Canada, Revenue Canada, NRC's Institute for Information Technologies, Bell Northern Research, Mitel, Newbridge), industries in the Dallas area (Bell Northern Research, E-systems, Hewlett Packard), federal agencies of the US government that are known to practice reuse on a large scale (DoD, STARS, ASSET, IDA, NASA), large industrial organizations which are known to champion software reuse (Hewlett Packard in USA, Mitsubishi in Japan, Q-labs in Sweden, ECRC in Germany), a wide audience of reuse practitioners, researchers and developers that attend reuse conferences (WISR, REUSE) and subscribe to reuse mailing lists (reuse@wustl.edu).

Our data collection effort lasted for about a year, but produced little data. Most people that we contact find the study interesting, express interest in seeing its outcome, but decline to provide data, for a variety of reasons:

- Academic respondents cite that their experiments are very small scale, hence are not representative of industrial conditions; further, they are generally interested in functional aspects of their experiments, and have kept no trace of the manpower aspects (which is an important focus of our survey form).

- Industrial respondents cite that much of the information we request is proprietary, and much of it is unavailable anyway; the practice of software reuse in industry is not so advanced that a precise accounting is maintained of all the costs involved in software reuse.

- Governmental respondents cite that some of the information we request is confidential, some of it is unavailable, and that some of the hypotheses that we make in our work (and that appear in the survey) are not applicable in a governmental context (factors such as discount rate and investment cycle do not have the same meaning in government and in industry).

- Finally all respondents, whether they are academic, industrial or governmental, cite the usual reasons of lack of time, lack of data, and lack of accurate data.

To encourage participants to complete survey forms, we have mentioned that the form is useful even if it is partly filled out; the reason is that we are interested in deriving a number of independent cost models, and a partially filled out form may be useful for some models and not for others.

After a year of searching potential respondents, soliciting them, encouraging them, reminding them, and begging them, we have collected but a few completed forms. If we consider the volume of data we have *per cost model* (rather than the total number of forms which are partially completed) the number is even smaller, and falls short of providing data for statistical analysis. At the same time, we have found other statistical studies that were conducted on software reuse for various purposes and that produce statistical data that is useful for our purposes. Among these, we mention in particular:

- A study of W. Lim conducted in 1994 at Hewlett Packard to investigate the impact of software reuse on programmer productivity and software quality and involving a total of 685 reusable work products (software components, software designs, software test plans, etc.) [43].

- A study by R. Selby conducted at NASA Langley laboratory and involving nearly 3000 software components written in Fortran; the objective of Selby's study is to characterize reusable components and relate their features to their reuse frequency [66].

- A study conducted by Lientz and Swanson in 1979 on software maintenance, and believed to reflect current maintenance practice in most of its aspects; we use this study, which is based on 487 software projects, to assess the portion of software maintenance costs that is due to software quality, as measured by fault ratios [41].

- An ongoing study of software costs by B. Boehm [14, 9, 13, 11, 12]; the original cost model is based on 63 projects, and has been expanded to include Ada projects and reuse activities. The COCOMO 2.0 model provides equations that capture reuse costs and view maintenance processes from a reuse perspective.

In the next chapter we use these sources to derive our cost models; we occasionally refer to our own collected data, but only as an indication of value ranges and overall cost trends.

7.2 Component Specific Factors

We divide component specific factors into three categories: *development for reuse* cost, which is the cost of developing a component with the premeditated intention to store it in a reuse library; *investment factors*, which are the factors that are involved in preparing a software component to be stored in a reuse library, given that the component was originally designed for single use; *customization factors*, which are the costs involved in preparing a component that has been retrieved to be integrated into its host system. We review these cost factors in turn, below.

7.2.1 Development for Reuse Factor

In this subsection, we consider factor *DR* (*development for reuse*) and discuss means to estimate this factor prior to the design of the component at hand. Note that in the asynchronous lifecycle, the question of whether to proceed with the development and storage of a component for the sake of software reuse is raised before the component is developed. Our measure of reusability attempts to quantify this very decision.

A number of cost estimation models are known nowadays; these range from COCOMO-like models that estimate the cost of a software product on the basis of its expected size, to Function Point-like models that estimate the cost of a software product by focusing on the functionality that the product is required to deliver. These cost models cannot be used for our purposes, though, because they are intended to estimate the cost of software developed for single use. When software is developed for multiple uses, a number of issues arise and drive the development cost up. These are:

- *Generality.* If a component is to be developed for a single use, then it must satisfy the specific requirements of the software system that will invoke. If, however, we wish to develop a component for multiple uses, then the component must satisfy a multitude of requirements that are not known individually, but are captured by a general specification. Clearly, a general component takes more effort to develop, because it must be able to interact with a wider variety of software systems.

- *Reliability.* If a component is to be developed for single use, it must be as reliable as the required reliability of the system in which it will fit. If the same component must be developed for multiple uses, then its required reliability must be as high as the highest required reliability of any system where it must be integrated; this generally means a higher reliability rating, hence a higher development cost.

- *Validation.* If a component is to be developed for a single use, then whenever it fails, it may bring down its host system with it. If the same component is to be developed for reuse, then whenever it fails it may bring down all the systems where it is reused. This means that it is important for the component to be validated properly. On the other hand, if the component is to be reused several times, the effort spent validating it will be amortized over the many uses. This means that the validation of reusable components is not only more important (than that of single use components) it is

also better amortized. Hence in practice more effort is spent validating a reusable component than a single use component, further increasing the development cost.

In estimating the cost of development for reuse, we will consider two factors in turn, which we estimate separately:

- *Cost of development for single use.* To derive an estimate for this particular factor, one may use any of the wide range of cost estimation techniques available nowadays.
- *The development for reuse surcharge.* A number of statistical studies have been conducted in the past on this particular ratio. Ada COCOMO [9] estimates that cost multipliers due to required reusability range from 1.0 to 1.5 depending on the scope of reusability. Also, AT& T has experienced a cost escalation factor of 2.25 in developing software for broad-based reuse (across product lines)[12]. We denote this surcharge by $REUS$.

We use a COCOMO 2.0 method to estimate $RUSE$: to this effect, we proceed in two steps, whereby we build a table of effort ratings, then a table of effort multipliers that correspond to these ratings. The table for effort ratings is given below [12]:

	very low	low	nominal	high	very high	extra high
RUSE		none	across project	across program	across product line	across many product lines
RELY	slight inconvenience	low, easily recoverable losses	moderate, easily recoverable losses	high financial loss	risk to human life	

In order to build the effort multipliers table, we take Ada COCOMO's table for $RUSE$ [9], which has the following entries for the four rightmost ratings (nominal to extra high):

1.0, 1.1, 1.3, 1.5

We shift this table of entries to the left and pad it with a value on the right end, which we choose to smooth the progression of values.We also take COCOMO's table for $RELY$. Hence we get:

	very low	low	nominal	high	very high	extra high
RUSE		1.0	1.1	1.3	1.5	1.75
RELY	0.75	0.88	1.00	1.15	1.40	

Then we multiply the first entry of *RUSE* with the nominal value of *RELY* (which is 1.0); we multiply the next two entries of *RUSE* with the value of *RELY* for the rating high (which is 1.15); and we multiply the last two entries of *RUSE* with the value of *RELY* for rating very high (which is 1.40). The rationale for this decision is that required reliability increases with the scope of required reusability. This yields the following effort multiplier table for factor *REUS*.

	very low	low	nominal	high	very high	extra high
REUS		1.0	1.26	1.49	2.1	2.45

Hence, to summarize: Our model requests that the user provides an estimate of the development effort (for single use) for the component at hand (using the user's favorite cost estimation model). Then the model enquires about the required scope of reusability (across the project, the program, the product line, other product lines) and derives the corresponding effort multiplier. We write

$$DR = REUS \times DE$$

where DE is the development effort of the component for a single use.

7.2.2 Investment Factors

In chapter 4, we have identified three investment factors in the synchronous lifecycle. These are: *Reeng*, the cost of reengineering the product to generalize it and prepare it for reuse; $Verif$, the cost of verifying the component to bring it to the reliability standards of the software reuse library; *Base*, the cost of baselining the component into the reuse library. We discuss these factors in turn, below.

7.2.2.1 Reengineering Costs

To derive the reengineering costs, we proceed in two steps:

- First, we estimate the equivalent source lines of code for the component at hand; this is done using a formula of COCOMO 2.0 [12], which we will discuss below.
- Then apply the cost estimation equation to the equivalent source lines of code figure. An economical way to this, however, is to prorate $ESLOC$ to the actual size of the component (say, $SLOC$) and to consider that the reengineering effort, *Reeng* is in the same ratio with respect to the development effort, DE. This step is based on the premise that the development effort is proportional to the product size, a premise which is highly appropriate for the kind of sizes we are interested in: reusable components are usually small products, having typically a few hundred lines of source code [55]. Hence we pose:

$$Reeng = \frac{ESLOC}{SLOC} \times DE.$$

We turn our attention now to computing $ESLOC$. The reengineering equations of COCOMO 2.0 [12] provide the following formula for $ESLOC$:

$$ESLOC = ASLOC \times (AA + SU + 0.4 \times DM + 0.3 \times CM + 0.3 \times IM),$$

where:

$ASLOC$: Adapted Source Lines of Code.
AA: Assessment and Assimilation increment.
SU: Software Understanding increment.
DM: portion of design modified.
CM: portion of code modified.
IM: portion of integration and testing modified.

In the circumstances that are of interest to us, we are dealing with a small component, which must be inspected in whole before the modification is carried out (presumably because it has high cohesion, hence we must read all of it before modifying any part of it), hence $ASLOC = SLOC$. Also, we are preparing this component to be placed in a reuse library, not to be integrated into a software system, hence $IM = 0$. For the same reason, we do not have to assimilate the component, hence $AA = 0$. Our formula becomes:

$$ESLOC = SLOC \times (SU + 0.4 \times DM + 0.3 \times CM).$$

Whence we derive the equation for *Reeng* as follows:

$$Reeng = (SU + 0.4 \times DM + 0.3 \times CM) \times DE.$$

Further, we provide the default value of 0 for DM (typically reengineering a component, in our sense of the word, does not involve design changes) and the default value of 1 for CM (because of the small size of components, this may be a reasonable default value if no precise value is available). As for the value of SU, it is derived by using table 7.1.

7.2.2.2 Verification Costs

Given that the component has been duly generalized and parameterized, it must now be verified according to the standards of quality that are in force in the reuse library. In order to derive the verification of this component, we proceed in two steps:

- First we evaluate the effort required to derive an equivalent component for reuse (as opposed to developing the component for single use then deciding to store it for reuse). The developments of section 7.2.1 give an explicit formula for this effort.
- Then we prorate the total development effort to determine how much of it is devoted to verification.

	very low	low	nominal	high	very high
Structure	very low cohe sion, high cou pling,spaghetti code	moderately low cohesion high coup ling	reasonably well struc tured;some weak areas	high cohe sion, low coupling	strong modula rity, informa tion hiding in structures
Application Clarity	no match bet ween program and application and world views	some correla tion between program and application	moderate cor relation bet ween pro gram and application	good cor relation bet ween program and application	clear match between pro gram and application world views
Self Descriptiveness	obscure code documenta tion missing obscure or obsolete	some code commentary and headers some useful documenta tion	moderate level of code commentary headers documenta tions	good code commentary and headers useful docu mentations weak areas	self descriptive code; documen- tation up to date, well orga nized, with de sign rationale
SU	0.50	0.40	0.30	0.20	0.10

Figure 7.1: Rating Table for SU increment

The rationale behind this procedure is that, once it is reengineered, the component is in the same status (of the same quality, degree of integrity, etc.) as a component that would have been developed for reuse.

The COCOMO cost model [14] provides figures for the distribution of development effort by phase. For small size software products (most representative of reusable software components), COCOMO tables provide that 42 % of total development effort is spent on *coding and unit test*. If we assume that half of this effort is spent on verification, we get a ratio of 0.21. This ratio is consistent with the general figures of $0.20 - 0.25$ cited in the literature ([65, 7]). Hence we find, for factor $Verif$:

$$Verif = 0.21 \times REUS \times DE.$$

7.2.2.3 Baselining Costs

Baselining costs include the cost of classifying a component according to some taxonomy, and the cost to store the component in the library. Additional documentation should be provided to the library along with the reusable asset. The *STARS Reusability Guidebook* suggests the following documents:

- **Part Description**: title, type of component, type of function, purpose of function, interface requirements.
- **Submitter Data**: name, address/network address, phone.
- **Component Constituents**: abstract, requirement specification, functional specification, design, algorithm, source code, object code, test specification, test/data results, maintenance/operations manuals.
- **Component History**: reason for component development, date of completion, description of applications used, frequency of use, description of development standards, version number.
- **Component Relationships**: name of parent, name of children.
- **Component Attributes**: keywords, development language, host environment, target environment.
- **Restrictions**: government, developer, reusability metrics.
- **Disclaimers**: warnings, problems, limitations, lack of tests.
- **Software Support**: support organization or person, qualification, frequency of update.
- **Miscellaneous Instructions**: fees, warranties.
- **Releases**: transfer and/or assignment of copyright.
- **Deliverable Media Description**: media.

- **Media**: electronic, magnetic, optical, paper, type of format (ASCII record, etc.).

According to Morrison [55], Japanese reuse librarians do not include a component in their libraries unless it satisfies the following criteria:

- Installation instructions.
- Conditions for operation.
- Operating instructions.
- Description of the application function.
- Program author.
- Development history.
- External specification.
- Coding specification.
- Management administrative information.

Other experimental data on the Japanese reuse factories is collected by Cusumano [23].

It is usually straightforward for a user (of our reusability model) to determine the baselining costs at their organization; this can be determined by inspection of the baselining procedures, by a straightforward estimation of the number of people involved, and the length of time involved. Hence our model expects the user to provide this figure. We do, however, define a default value, in case the user cannot provide a reliable figure: we take the default value of 3 person days, on the basis of following assumptions:

- Two persons are involved in the baselining process, namely: the reuse library manager and the person who produced the component at hand.
- We count that on average the library manager spends two days reviewing the component.
- We count that the two parties meet for half a day to proceed through the steps of baselining and quality assurance associated with the reuse library at hand, for a total of 1 person day.

In summary we propose the following formulas for deriving investment costs, which are $Reeng$, $Verif$, and $Base$.

- Reengineering costs:

$$Reeng = (SU + 0.4 \times DM + 0.3 \times CM) \times DE,$$

where

SU: Software Understanding, provided using table 7.1.
DM: portion of design modified, provided by user, default is 0.
CM: portion of code modified, provided by user, default is 1.
DE: development effort, provided by the user.

- Verification costs:

$$Verif = 0.21 \times REUS \times DE.$$

where

$REUS$: the required reusability cost driver, computed as per section 7.2.1.
DE: development effort, provided by the user.

- Baselining costs: We assume that this cost is provided by the user, reflecting the baselining procedure that are in force in the user's organization. If no value is provided, we take the default value of $Base = 3PD$.

7.2.3 Customization Factors

Customization Factors include instantiation costs for instances when component C is used verbatim (factor *Instance*), modification costs for instances when component C is used after modification (factor *Adapt*), as well as the cost to integrate component C (after instantiation or modification) into the host system (factor *Rest*). We discuss the evaluation of these three factors, in turn.

7.2.3.1 Instantiation Costs

Citing Selby's study of 2954 Fortran modules, Boehm et al. [12, 11] plot a curve which shows how the adaptation cost of a component varies according to the amount of code modified. Contrary to naive assumptions, this curve is not linear –in at least two ways:

- The curve does not start from the origin of the plot. When the amount of modified code is zero, the reuse effort is 0.046 times the development effort.

- The curve starts with a steep slope, then increases at lower and lower rates. This reflects the phenomenon whereby to modify any portion of the code, one must understand a large portion of the code, possibly all of it. So that the difference between modifying a small portion of the component or a large portion is relatively small, because in both cases we must understand a large portion of the code.

This curve is reproduced in figure 7.2. On this curve, we observe that when the component is merely instantiated (is not modified), we must still spend some effort, which is 0.046

times the development effort. We take this to be exactly the instantiation effort, hence we get [1]

$$Instance = 0.046 \times DE.$$

7.2.3.2 Adaptation Costs

The cost of adaptation is the cost of adapting a component that was deemed worthy of the effort to make it satisfy the requirements of the user query. Figure 7.2 [66] shows how the adaptation cost varies according to the amount of code modified. At the time when reusability must be evaluated, we are not dealing with a specific adaptation; rather we are interested in the average adaptation cost. We let t, $0.0 \leq t \leq 1.0$, be the modification amount, and we let $Rc(t)$ (relative cost) be the function plotted on figure 7.2. Let $P(t)$ be the probability distribution of the modified amount t over all possible adaptations of the component at hand; i.e., the probability that the modification amount be included between a and b, where $0.0 \leq a \leq b \leq 1.0$ is given by:

$$\int_a^b P(t)dt.$$

Then the average adaptation effort for the component at hand is given by:

$$Adapt = DE \times \int_{0.0}^{1.0} P(t) \times Rc(t)dt.$$

As a simplifying assumption, we consider that all modification amounts between 0.0 and 1.0 are equally likely in practice. Of course one may argue that modification amounts near 0.0 are more likely than amounts near 1.0. Yet we maintain our assumption because quite often amounts near 1.0 occur unexpectedly: before proceeding with the modification, the programmer may well have estimated the the modification amount to be significantly less than 1.0, but it proved to be near (sometimes above) 1.0. Under our assumption, the adaptation effort takes on the simpler form:

$$Adapt = DE \times \int_{0.0}^{1.0} Rc(t)dt.$$

Now the second term of this formula can be computed quite easily, as it represents the area under the curve of figure 7.2. We find,

$$\begin{aligned} & \textstyle\int_{0.0}^{1.0} Rc(t)dt \\ = & \quad \{ \text{ property of integrals } \} \\ & \textstyle\int_0^{0.125} Rc(t)dt + \int_{0.125}^{0.625} Rc(t)dt + \int_{0.625}^{1} Rc(t)dt \\ = & \quad \{ \text{ computing areas } \} \end{aligned}$$

[1] It is possible that the figure of 0.046 cited by Boehm actually covers not only the instantiation costs (factor *Instance*) but also the retrieval costs (factor *Exact*); if such is the case our estimate may be slightly excessive. Because the amounts involved are fairly small, we will not mind this approximation.

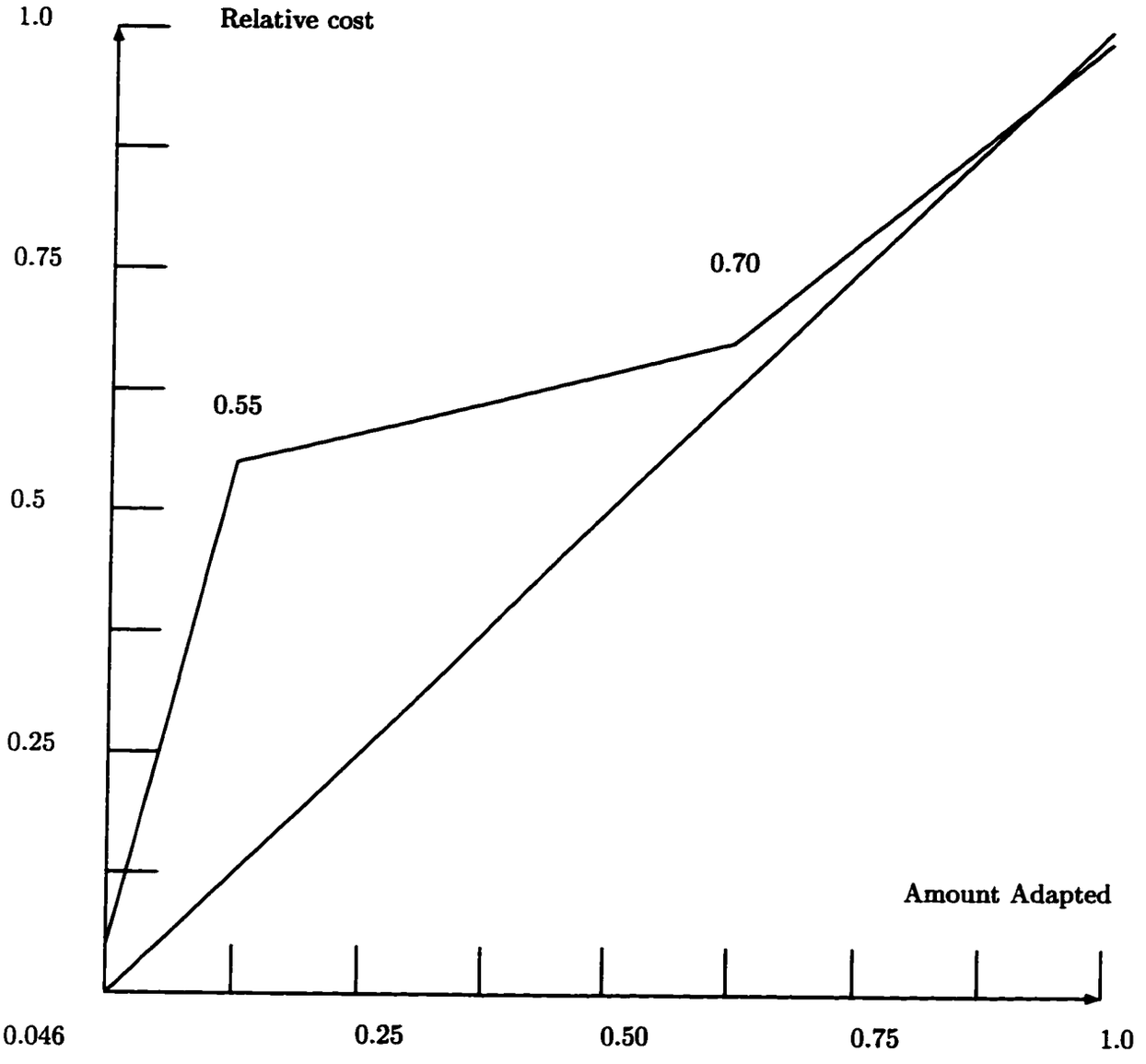

Figure 7.2: Non Linear Reuse Effects

$$
\begin{aligned}
& 0.125 \times \tfrac{0.046+0.55}{2} + 0.5 \times \tfrac{0.70+0.55}{2} + 0.375 \times \tfrac{0.70+1.0}{2} \\
= & \quad \{ \text{ evaluation } \} \\
& 0.67.
\end{aligned}
$$

[2] Hence we write:

$$Adapt = 0.67 \times DE.$$

7.2.3.3 Integration and Test

Factor *Rest* represents the costs pertaining to the remainder of the lifecycle (that was interrupted when the component was placed in the reuse library), i.e., the costs of integration and test. These costs are widely documented [65, 7]. In [14] Boehm provides a figure for the distribution of development effort by phases. The portion of development effort devoted to integration and testing varies according to the size of the product as well as the development mode. For our purposes, we take the size *small* and an intermediate development mode (*semi-detached* mode [14]). This yields:

$$Rest = 0.19 \times DE.$$

Summarizing, we find that all customization efforts are linear with respect to the development effort (DE), and are given by the following formulas:

$$Instance = 0.046 \times DE.$$
$$Adapt = 0.67 \times DE.$$
$$Rest = 0.19 \times DE.$$

7.2.4 Quality Factors

Factors *Oper* and *Oper'* are called *quality factors* because they reflect the quality of the component: the higher the quality of the component the less it costs to operate it and maintain it.

7.2.4.1 Operational Cost: Single Use Component

Factor *Oper'* represents the annual cost of operating (maintaining) component C within a host system. The rationale for involving this factor in our equations is that we expect that reusable components cost less to maintain than components developed for single use, and we wish to include this gain in quality into our computations. One would expect that the cost to maintain a component be a linear function of the cost to develop it. The original version of COCOMO computes maintenance costs by multiplying the development cost by factor ACT, called the *Annual Change Traffic*[3]. In the original COCOMO document

[2] Had the adaptation effort been linear with respect to the modification amount, we would have found a value of 0.50 rather than 0.67.

[3] COCOMO 2.0 computes maintenance effort using the reuse model.

[14], it is reported that the median value of *ACT* across the projects of the COCOMO database is 0.08. Recent estimates of the average/typical *ACT* place this value at 0.1 [13]. We take:

$$Oper' = 0.1 \times DE.$$

7.2.4.2 Operational Cost: Reusable Component

Factor *Oper* represents the cost of maintaining a software component that was developed for reuse, or was developed for single use and then reengineered for reuse. We attempt to derive it using empirical studies that compare the gains in product quality that stem from software reuse. This gain in quality results from the cumulative effect of debugging the reused component as it is being reused in more and more applications in the organization. In [43], Lim compares the defect densities of software that was developed for single use with the defect density of software that is a mixture of new code and reused code. Lim finds that code that contains reused software shows drops of defect density that vary between 24% and 51%. If we focus on the reusable code contribution to the drop in defect density, we find that reusable code shows a defect density reduction of 76%.

One may be tempted to consider that this causes a commensurate reduction in operating costs, but studies conducted on the economics of software maintenance show that defect removal accounts for a small fraction of total maintenance costs. In [41], it is found that the maintenance effort of a typical organization is distributed as follows:

- Emergency program fixes: 12.4%.
- Routine debugging: 9.3 %.
- Accommodating Changes in data, input files: 17.4 %.
- Accommodating changes to HW, OS: 6.2%.
- Enhancements for users: 41.8 %.
- Improve documentation: 5.5 %.
- Improve code efficiency: 4.0 %.
- Other: 3.4 %.

While it is clear that a reduction in defect density will directly impact the first two items of this list, namely *emergency program fixes* and *routine debugging*, it is not clear how or to what extent defect density affects the other activities. Yet it is quite conceivable that a reusable component, especially if it was designed for reuse in the first place, would have a better structure and a more coherent design than a component developed for single use. The maintenance equations of COCOMO ([14], chapter 30, table 30.2) provide a table which illustrates the impact of structured design (by use of modern programming

practices) on the cost of maintenance. For components of small size, it is found that a very high concern for design integrity (rating *very high* for factor $MODP$) yields an effort multiplier of 0.81. Whereas the reduction in defect density applies to the first two entries above, the gains from structured design applies to all the entries. This yields the following multiplier:

$$\begin{aligned} & 0.81 \times (0.24 \times (12.4 + 9.3) + 78.3) \\ = & \quad \{ \text{ evaluation } \} \\ & 0.67. \end{aligned}$$

This factor represents the ratio between $Oper$ and $Oper'$. Hence we find, for $Oper$,

$$Oper = 0.067 \times DE.$$

7.3 Strategic Cost Factors

We have identified two factors in our reusability formula that depend on the corporate strategy of the software organization: These are the investment cycle and the discount rate.

7.3.1 Investment Cycle

The *investment cycle* is the length of time (number of years) over which the organization expects to retrieve its investment cost (and benefits). Of course this question matters only for the case of limited term reusability: for long-term reusability the investment cycle is taken to be infinite.

There does not appear to be a standard figure that can safely be taken as a default value. A survey of our respondents as well as enquiries with software managers and practitioners lead to a wide variety of answers, including:

- Some (automobile industry, in the U.K) feel that unless the storage of a component for the purpose of reuse can pay off within a year, it is not considered worthwhile.
- Others (a federal agency of the U.S. federal government) find that their organization has been able to collect benefits after three to four years from the time the component is placed in a reuse library, and consider that a component is reuse worthy only if it pays off within this time frame.
- Still others (software development firm, Dallas area) adopt a five year term as part of their corporate strategy.

There are a number of ways to deal with this uncertainty:

- The first option is to consider that the investment cycle is a parameter of the reusability formula, and to expect that users of the formula provide a figure for this parameter (by consulting those that are responsible for corporate strategy within their organization).

- The second option considers in asking the user to provide a figure for the investment cycle, but making provisions for a default value in case the user cannot offer a figure. A possible default value may be 3 years.

- The third option consists in defining reusability, not by means of a single value, but rather by means of an array of size Y, which gives the return on investment values for all y between 1 and Y. A good value of Y may then be $Y = 5$ or $Y = 6$. This issue will be discussed in chapter 8.

7.3.2 Discount Rate

The *discount rate* is another parameter that is strategic (rather than technical) in nature and that is best defined by those responsible for corporate strategy within the software organization. Most of our respondent were unable to quote their corporate rate of discount, presumably because they make reuse decisions without regard to quantifiable economic impact.

The problem of determining the discount rate is well known in the engineering economics literature (e.g., [64]), and several approaches are known to address this problem.

- *Corporate Discount Rate.* Corporate discount rates include a premium for risk, a provision for corporate taxes on extra income, and a profit to be returned to shareholders.

- *Marginal Time Preference Rate.* Whereas the previous approach is corporate-wide, this approach is project-wide and is carried out by the project manager. In this approach, the project manager determines rate d by asking the following question: *what is the smallest value of d for which I would consent to spend 1 person month today to save $(1 + d)$ person month in a year*?

- *Minimum Attractive Rate.* Whenever all other parameters are known, we may evade the question of deciding the discount rate by turning the question around: We ask what is the minimum discount rate for which the return on investment takes a given value (e.g., zero). Then, using this discount rate we now ask the question: How does your discount rate compare with the break-even rate?

In practice, it may be useful to solicit the user to provide a tentative discount rate; and to make provisions for a default value in case the user cannot. The default value will be chosen to reflect typical values in the context of software reuse, which are lower than typical *corporate discount rates*: indeed, discount rates are usually applied to financial resources, which can be invested or disposed of much more easily than manpower resources. For example, if we have a person who happens to be idle (or under occupied) between two software projects (has completed first project and is awaiting second project that has not started yet) then we have an incentive to assign this person to the task of storing a component even if the payoff of this task may be low. We will discuss this matter further in chapter 8.

7.3.3 Library Factors

We have identified four cost factors that depend on the operation of the reuse library. These are: the cost of storing a component in the library (*Stor*); the cost of an exact retrieval from the library (*Exact*); the cost of an approximate retrieval from the library (*Aprox*), and the cost of evaluating retrieved components to decide whether to reuse them (*Eval*). We review them in turn below.

7.3.3.1 Storage Costs

Storage costs include the cost to maintain a given component in the reuse library, as well as the overhead that the component creates by being in the library:

$$Stor = RL + OV,$$

where RL is the annual cost of running the reuse library and OV is the annual overhead of running the library.

The annual cost of maintaining a component in the reuse library can be computed as the total annual effort spent on maintaining the reuse library, divided by the number of components in the library. Now, the total annual effort, in person months, spent on maintaining the reuse library is quite easy to determine: it is 12 times the number of full time persons assigned to the task of running the reuse library.

$$RL = 12 \times RLS \times \frac{1}{N+1},$$

where RLS represents the full time equivalent of the personnel assigned to maintain the reuse library and N is the current size of the reuse library (before storage).

There are two forms of overhead that the component may cause by its presence in the library:

- The component has the potential to slow down retrieval algorithms, because it gives them one more entry to look up in any search operation.
- The component may inadvertently be retrieved in the search for another component, thereby distracting the user.

We let ov_1 and ov_2 be the two overhead costs cited above; we find

$$OV = ov_1 + ov_2.$$

Let SR (SeaRch) be the total person month effort spent annually in the organization searching for reusable component, taking into account all the organization's software personnel. It is fair to consider that, on average, the cost of a search is linear in the size of the library. Hence the overhead involved in adding a new component to a database of size N can be evaluated as follows:

$$
\begin{aligned}
& ov_1 \\
= & \quad \{ \text{ search effort difference } \} \\
& \frac{N+1}{N} \times SR - SR \\
= & \quad \{ \text{ simplification } \} \\
& \frac{SR}{N}.
\end{aligned}
$$

As for factor SR, it can be evaluated either by the time sheets of software personnel, or by keeping track of the logon time on the software reuse library.

The overhead that is due to components being inadvertently retrieved when they are not needed varies greatly with the precision of the retrieval algorithm. A retrieval algorithm is precise if and only if all the components it retrieves are relevant. Under this condition, the second overhead is zero. We will, henceforth, take the hypothesis of perfect precision, and neglect the factor ov_2. Note that perfect precision characterizes all retrieval algorithms that are based on formal specifications and formal matching criteria (see, e.g., the algorithm of chapter 11).

Summarizing, we find that the annual cost of storage is given by the following formula:

$$Stor = 12 \times RLS \times \frac{1}{N+1} + \frac{SR}{N},$$

where

RLS: monthly effort to maintain the library.
SR: annual effort spent searching the library.
N: current size of the library.

7.3.3.2 Retrieval Costs

We have discussed two retrieval costs, namely exact retrieval cost and approximate retrieval cost. These costs vary according to two criteria:

- First, whether specifications and queries are written formally (using formal specification methods [46, 59, 77]) or informally (using library science techniques [63]).
- Second, whether the matching criterion is a formal theorem (see, e.g., formulas of chapter 11) or a mere comparison of keyword lists [63].

Most of the respondents to our survey, as well as the practicing programmers we have surveyed verbally, seem to be using informal representations of components and queries, and seem to perform matches by comparing keywords. It is difficult to make estimates of such efforts when their process is so informally defined.

It is not difficult, however, for a practicing programmer or manager to make an estimate of the time it takes to perform a retrieval, especially that this effort appears to be quite orthogonal to the structure and complexity of the components at hand. Typically, components are represented by means of uniform entries, which contain a number of predefined, component independent fields.

In practice, we will ask for an estimate of these quantities, and provide default value if no estimate is available. Our estimate of the default values is one person-day for exact retrieval and two person days for approximate retrievals. These default values may be revised upwards if the reuse library is uncommonly large, and may be revised downwards if automated retrieval tools are available.

7.3.3.3 Evaluation Costs

Given that a component has been retrieved, either by exact retrieval or by approximate retrieval, it must further be evaluated to determine whether it is worthwhile to reuse in this particular instance. The *evaluation cost* is the cost of carrying out this decision process; we view it as a library cost because its primary function is to deal with the insufficiency of the retrieval step. If exact retrieval was known to retrieve only relevant components (perfect precision) then no evaluation is required after exact retrieval; on the other hand, if approximate retrieval was known to retrieve only components that are close enough to the user query to be reuse worthy then no evaluation is required after approximate retrieval.

We consider that the software library manager knows about evaluation costs as part of his knowledge of the retrieval processes and costs: Knowing how precise his retrieval algorithms are, he can determine how much evaluation work must be completed downstream of the retrieval process.

7.4 Domain Cost Factors

7.4.1 Reuse Frequency Variance

We have identified three domain cost factors, which are: the frequency of exact retrieval, the frequency of approximate retrieval, and the total retrieval frequency, which the sum of the first two factors. The problem of estimating domain cost factors is probably the most crucial decision in the application of our reusability formula, for two reasons:

- Retrieval frequencies vary a great deal from one instance to the other, depending on such issues as the functional specification of the component at hand, the application domain under consideration, the level of activity of the software organization, all of which are hard to quantify. Because of this wide variance, it is virtually impossible to provide a default value.

- At the same time as it is difficult to predict, the frequency of reuse is also very crucial in the computation of the reusability, because the benefit factor of the reusability formula is a linear function of frequency.

Some of the factors that are known to influence retrieval frequencies from one instance to another are the following:

- *The design integrity of the component.* In a study of 2954 Fortran modules that are used in a NASA production environment, Selby [66, 8] finds that high frequency of

reuse is significantly correlated with the following features, which characterize good modular design:

  - Lower interaction with other system modules in terms of the number of module calls per source line.
  - Simpler interfaces in terms of the number of input output parameters per source line.
  - Less interaction with human users in terms of the number of read-write statements per source line.
  - Higher ratios of commentary to eventual implementation size in terms of the number of comments per source line.

- *The Pervasiveness of the component within the application domain.* How often a component may be invoked in a production environment for the purpose of being reused depends not only on the features of the component (which we have reviewed above), but also on the pervasiveness of the component's function within the application domain at hand: a sort routine, e.g., will likely be in high demand in a data processing production environment, but will not be as solicited in a numerical analysis environment.

- *The level of activity of the production environment.* For a given level of design integrity, and for a given level of pervasiveness, the reuse frequency of a component varies linearly with the level of activity of the production organization at hand. The more software is produced by the software organization, the more often a given component will be solicited.

- *The recall of retrieval algorithms.* A component may well be needed in a given production environment, but seldom be retrieved, due to a lack of recall in the retrieval algorithm: because of poor recall, the algorithm fails to retrieve the component even when it is queried. This may be due to a weakness in the documentation standards (what information is recorded about a given component) or a weakness in the matching procedures (how a component is matched against a query to determine whether the component satisfies the query).

These factors explain why reuse frequencies may vary a great deal from one instance to another, and emphasize the need to deal with this particular reusability factor with special care and attention.

7.4.2 Assessing Frequencies

In order to deal with the wide variance of reuse frequencies, we have taken the following steps:

- *Linking reuse frequencies.* While it is difficult to predict the reuse frequency of a component, it is fairly easy to predict a ratio between the various frequency factors of a given component. Consequently, we tackle the problem of reuse frequency

estimation in two steps: first, estimate the ratio between exact reuse frequency ($Freq_X$) and approximate reuse frequency ($Freq_A$); second estimate the overall reuse frequency, using the formula

$$Freq = Freq_X + Freq_A.$$

- *Dealing with a single variable.* Given that the ratio between $Freq_X$ and $Freq_A$ is determined, and given that $Freq$ is known to be the sum of $Freq_X$ and $Freq_A$, we can express the reusability of a component as a function of a single reuse frequency variable, namely $Freq$. This opens a wide range of possibilities, including that reusability is now given as an array, for various values of frequency, rather than a single value.

- *Factoring frequency out.* Given that reusability can now be expressed in terms of a single reuse frequency factor, we can compute the derivative of reusability as a function of frequency, thereby obtaining a definition of reusability that is independent of frequency; such a value is intrinsic to the component, in the sense that it reflects the intrinsic properties of the component, rather than how often the component is reused.

- *Reasoning by analogy.* As an alternative to factoring the reuse frequency out of the reusability formula, one may want to derive the reuse frequency by analogy with similar instances where the reuse frequency is known. An example of reasoning by analogy may proceed as follows:

  > We currently have a sort routine in our reuse library, that sorts employee records in increasing order. We have inspected its reuse log and have found that it was invoked 20 times last year. This new component sorts employee records in increasing order and decreasing order, by setting a simple parameter. Our analysts estimate that they need decreasing sorting routines 20% as often as increasing sorting routines. Hence $Freq = 24$.

  Another example may proceed as follows:

  > Component C, which we are considering to include in the reuse library, performs function F, which can also be provided by component C', after a minor modification. Last year, component C' was retrieved and modified 5 times. On the other hand last year there were 7 retrievals for function G, which is also provided by component C, that failed. We take: $Freq = 12$.

Our reliability estimation model calls for requesting the user to provide an estimate of the reuse frequency. The model does not provide default values, because it would be very difficult to justify any value as a default. The model remains useful even in the exceptional case when the user does not provide a value for reuse frequency; this matter is discussed in section 7.4.4.

7.4.3 Assessing Frequency Ratios

In our reusability model, the user is asked to provide a value for the ratio between $Freq_X$ and $Freq$. While we have no basis for offering a default value for the total reuse frequency ($Freq = Freq_X + Freq_A$), we have a default value for the ratio between exact frequency and approximate frequency, which we discuss below. If we let λ be the value of this ratio, then we find:

$$Freq_X = \lambda \times Freq.$$
$$Freq_A = (1 - \lambda) \times Freq.$$

Also, the cost of retrieving components and instantiating them (in the case of exact retrieval) or adapting them (in the case of approximate retrieval) can then be written as:

$$\begin{aligned}
& Retrieve \\
= & \quad \{ \text{ formula, chapter 4 } \} \\
& Freq_X \times (Exact + Eval + Instance + Rest) + \\
& Freq_A \times (Aprox + Eval + Adapt + Rest) \\
= & \quad \{ \text{ substitutions, factoring } \} \\
& Freq \times (\lambda \times (Exact + Eval + Instance + Rest) + \\
& (1 - \lambda) \times (Aprox + Eval + Adapt + Rest)) \\
= & \quad \{ \text{ simplification } \} \\
& Freq \times (Rest + \lambda \times (Exact + Eval + Instance) + \\
& (1 - \lambda) \times (Aprox + Eval + Adapt)).
\end{aligned}$$

Whence the formula of the annual operational cost, under the hypothesis that the component at hand is stored, becomes:

$$\begin{aligned}
& OC_S(y) \\
= & \quad \{ \text{ formula, chapter 4 } \} \\
& Store + Retrieve + Maintain(y) \\
= & \quad \{ \text{ substitution } \} \\
& Store + Freq \times (Rest + \lambda \times (Exact + Eval + Instance) \\
& + (1 - \lambda) \times (Aprox + Eval + Adapt)) + (y - 1) \times Freq \times Oper \\
= & \quad \{ \text{ factoring } \} \\
& Store + Freq \times (Rest + \lambda \times (Exact + Eval + Instance) \\
& + (1 - \lambda) \times (Aprox + Eval + Adapt) + (y - 1) \times Oper).
\end{aligned}$$

Now we focus on estimating the default value of factor λ. An empirical study conducted by Selby in a NASA programming environment [66, 8] and involving 2954 software components, finds a value of λ that is about 0.62. Since the cost of software adaptation

increases sharply with the amount of code that must be modified, reuse is most effective whenever adaptation is not involved. The default value we propose for λ provides in effect that on average a component is reused without modification nearly twice as often as it used with modification.

Summarizing, if no value of λ is provided by the user we take:

$$\lambda = 0.62.$$

7.4.4 Frequency as a Parameter

The fact that reusability can be formulated as a function of a single variable has a number of implications, which we discuss in this section.

If all the other parameters are fixed, then our reusability equation can be used to determine the break-even reuse frequency, i.e., the reuse frequency for which the reusability is positive, or greater than a given threshold value. The advantage of this approach is that we no longer ask the question:

> What is the expected reuse frequency of this component in this production environment?

Rather, we ask the question:

> Do we expect the reuse frequency of this component to exceed this break-even value?

The second question can often be solved easily, and is always easier to solve than the first.

It is conceivable that our reusability estimation model be used not only to estimate the reusability of a component before it is stored, but rather to audit a reuse program that is under way. Under such conditions, the frequency of reuse is known at the time when we are interested in determining the reusability of our components. We may consider an example where we are auditing a reuse program to determine which of the following premises applies to the programming environment at hand:

- The reuse library is cluttered with components that are not reuse worthy; these have a negative impact on the speed, precision and recall of the retrieval algorithms, and distract the programmers and analysts who try to work with the library.
- Most of the components in the reuse library have a high reusability measure; in addition, they are properly documented. But the retrieval algorithms are inappropriate, due to poor precision (or poor recall).
- Most of the components in the reuse library have a high reusability measure and the retrieval algorithms are adequate. But due to poor documentation of the components, the algorithms fail to retrieve them.
- The average reusability of components in this reuse library is very high, and they are well documented; in addition the retrieval algorithms are adequate. However, programmers and analysts do not use this library as much as they should.

It is easy to see how evaluating the reusability of the components of the library is important in making a diagnosis on this situation. Also, it is clear that in this situation, the frequency of reuse can be estimated from the data collected about the operation of the library.

Finally, we point out that in chapter 6 we have discussed the definition and use of an auxiliary measure of reusability, namely the *intrinsic reusability*. Because the original formula of reusability is linear in the frequency factor, the intrinsic reusability, which is the derivative of reusability with respect to frequency, is a constant, and does not depend on $Freq$. Whereas the original reusability measure depends critically on reuse frequency, intrinsic reusability does not hence it can be evaluated even when $freq$ is not known.

7.5 Estimating the Reusability of Object Oriented Components

In chapter 4 we had discussed the synchronous reuse lifecycles of object oriented components and traditional software components and have observed that, with the exception of name changes, these lifecycles are essentially the same. While the phases are the same, their costs may be different depending on whether the software is object oriented or not; furthermore, the methods that we use to estimate these costs may differ. In this section, we review the various cost factors that we have discussed above and discuss whether, and to what extent, the use of object oriented programming influences the value of these costs or the method to estimate these costs.

- *Development for Reuse.* Because of its focus on modularity and information hiding, the discipline of object oriented programming in geared towards producing reusable code; consequently, the distinction of *development for reuse* and *development for single use* is not as sharp in object oriented programming as it is in traditional programming. The COCOMO 2.0 cost model derives the cost of a development project using an original measure of size, which is the *object points count* [12].

- *Investment Factors.* Investment costs include costs of reengineering, verification and baselining. We expect that on average the cost of reengineering an object oriented module is less than that of reengineering a traditional software module of equal functionality, because the object oriented discipline promotes good engineering in the first place. As far as the other factors are concerned, we have no basis for estimating how they compare with respect to traditional programming paradigms.

- *Customization Factors.* Customization factors include the costs of instantiation, adaptation and integration testing. We expect adaptation costs of object oriented software to be lower than those of traditional software, because of the emphasis of object oriented discipline on modularity and information hiding, two features that promote adaptability by making software easier to understand. As for the other two factors, we have no basis for considering that they are any different under the object oriented discipline than under traditional disciplines.

- *Quality Factors.* Quality factors include the costs of maintaining software developed for single use ($Oper'$) and software developed for reuse ($Oper$). These factors do not appear individually in our reusability equations; rather, they appear only as part of the term ($Oper' - Oper$). Because of the emphasis of the object oriented discipline on developing reusable software, we expect a smaller difference between ($Oper$) and ($Oper'$) under object oriented programming than under traditional programming. We know of no empirical data to quantify this claim, however.

- *Strategic Factors.* Strategic factors include investment cycle and discount rate. Both of these factors are organization-wide factors that are defined by the organization's management, irrespective of the programming discipline that is used for software development.

- *Domain Factors.* Domain factors include the frequency of exact retrieval and the frequency of approximate retrieval. These frequencies characterize the application domain and the functional properties of the component at hand (given that the component has these functional features, how often is it used in this application domain?). By definition, these frequencies are independent of the language in which the component is implemented.

Chapter 8

Automated Reusability Evaluation

The formulas proposed in chapter 7 cannot be reasonably used in practice without computer assistance. In this chapter we envisage this possibility and discuss the features of a system that we have developed to compute the ROI of a component; we also discuss future expanded versions of this system and consider a sample application on a number of Ada components that have been selected from Grady Booch's library of components [15].

8.1 Requirements Specification

We envisage three versions of our proposed system for reusability evaluation:

- *version 1.0*, which does the basic calculation of ROI, intrinsic ROI and break even frequency (the frequency at which the ROI is zero).
- *version 1.1*, which provides for the case when the user does not know some of the parameters (frequency, discount rate, investment cycle, e.g, and gives ROI tables instead of ROI values.
- *version 1.2*, which has a database facility, whereby it not only computes current ROI values but also keeps track of ROI values for the whole reuse library, and can highlight outliers (component with very low ROI, that may be candidates for the wastebasket).

We discuss these three versions in turn below. For the sake of discussion, we call this system the *ARES* system, to stand for *Automated Reusability Evaluation System.*

8.1.1 Version 1.0: Deterministic Reusability Evaluation

8.1.1.1 Version 1.0 Inputs

System ARES proceeds through a set of data entry screens requesting data as it goes and offering default values when the user is unable to provide accurate or reliable data.

First, the system requests an estimate of the investment cycle and proposes 3 as a default value if the user wishes to fall back on the default value. Also the system informs

the user that a long term ROI will be given, which takes an infinite value for the investment cycle. Version 1.1 will make provisions for the case when the investment cycle is not known precisely.

Second, the system requests an estimate of the corporate discount rate; it is highly unlikely that the management of an organization does not have a known value for this parameter, since most strategic decision making in the organization is dependent on it. Nevertheless, if no such value is available, the system provides the default value of 0.10; this may be revised downwards if manpower is currently in surplus, and revised upwards if manpower is scarce. Version 1.1 will make provisions for the case when the discount rate is not known precisely.

Storage costs are known to be function of three parameters: RLS, the number of full time persons (or equivalent) responsible for maintaining the library; SR, the annual effort spent searching the library; and N, the current size of the library. System ARES requests these three data items from the user.

System ARES enquires whether the user knows the average/typical retrieval cost in the organization at hand. If not, the system enquires about the use of automated retrieval tools and about the size of the reuse library, and derives default values according to the table given in figure 8.1. We have derived these values by interviewing reuse practitioners and interpolating their estimates; a user of our model would do well to check these values for the particular organization that wishes to apply the model. Also, this table clearly depends on current storage and retrieval technology, and must be updated accordingly.

For all the reasons mentioned in chapter 7, the estimation of frequency figures is probably the most difficult decision that the user of ARES has to make. In version 1.1, we allow the user to give a range of values for the frequency, rather than a single accurate value. In this version, however, we request that the user propose a tentative value, possibly by consulting domain experts. System ARES prompts the user to supply a value for $Freq_X$ and $Freq_A$. If the user cannot, the systems takes the frequency default ratio of chapter 7 ($\lambda = 0.62$) and requests an estimate of the overall reuse frequency. The system also informs the user that it will compute the break-even frequency, and that it will provide a measure of intrinsic ROI, which is independent of frequency.

In the context of the asynchronous reuse process, the system must determine the cost of development for reuse. To this effect, the system prompts the user to produce an estimate of the development effort (for single use) of the component at hand, using the user's favorite cost estimation method. Then the system submits the table of ratings of the $REUS$ factor (see chapter 7) and asks the user to choose an entry; the system derives the corresponding effort multiplier, which it multiplies by the estimate of development effort.

In order to estimate the reengineering costs, system ARES must estimate the following factors: SU, the software understanding increment; DM, the portion of design modified; CM, the portion of code modified; DE, the development (for single use) effort. System ARES displays table 7.1 and prompts the user to select entries that correspond to the component at hand; and it derives factor SU according to the user's choice. Then it prompts the user to provide estimates of factor DM (for which it takes default value 0.0 if the user provides no value) and factor CM (for which it takes default value 1.0 if

Exact Retrieval

library size	*Retrieval Method*		
	by hand	hypertext	automatic
small ($\leq$ 100)	1.0 PD	0.8 PD	0.5 PD
medium (100 $\leq\leq$ 500)	1.5 PD	1.0 PD	0.6 PD
large ($\geq$ 500)	2.0 PD	1.2 PD	0.7 PD

Approximate Retrieval

library size	*Retrieval Method*		
	by hand	hypertext	automatic
small ($\leq$ 100)	2.0 PD	1.6 PD	1.0 PD
medium (100 $\leq\leq$ 500)	3.0 PD	2.0 PD	1.2 PD
large ($\geq$ 500)	4.0 PD	2.4 PD	1.4 PD

Figure 8.1: Default Retrieval Costs

the user provides no value). Finally the user is prompted to provide an estimate of the development effort (in the case of the synchronous lifecycle); because this step occurs at a time when the component has been totally developed, its development cost should in principle be known.

Verification costs are dependent on two factors that have previously been provided, namely $REUS$ and DE; hence they can be derived without user intervention.

System ARES prompts the user to provide an estimate of baselining costs by inspection of the procedures that are in force in the user's organization. In case the user is unable to provide such an estimate, the system takes $Base = 3PD$.

All remaining factors in our cost model (namely instantiation costs, adaptation costs, integration and testing costs, and operational costs with and without reuse) can be derived without user intervention because they depend on factors that have previously been provided or computed. This ends the sequence of inputs that the user must provide.

8.1.1.2 Version 1.0 Outputs

On output, system ARES provides the following data:

- A table summarizing all the input data, indicating which data was provided by the user and which data was taken from default values.
- The ROI of the component at hand.
- The intrinsic ROI of the component at hand.
- The long term ROI of the component at hand.
- The break even frequency (the value of $Freq$ for which ROI is zero).

8.1.2 Version 1.1: Non Deterministic Reusability Evaluation

8.1.2.1 Version 1.1 Inputs

No matter how carefully we derive default values, there are cases where the default values are significantly off the mark. Version 1.1 of ARES makes provisions for this possibility by allowing the user to provide, not a specific value for a given parameter, but rather a range of values.

We have identified three factors which are prime candidates for being described by their range of values rather than a single value. These are:

- *The investment cycle.* Version 1.1. does not prompt the user to provide a figure for the investment cycle. Instead, it prompts the user to provide a lower bound, an upper bound and an increment. hence for example if the lower bound is 1, the upper bound is 10 and the increment is 1, we get the following set of values for the investment cycle:

$$Y \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \infty\}.$$

Note that $Y = \infty$ is included because it can be computed by a closed formula and it can be used as an upper bound of the value of ROI —all other arguments being fixed.

- *The discount rate.* Version 1.1 asks the user to provide a lower bound, a higher bound, and an increment for the discount rate. Hence for example if the lower bound is 0.08, the upper bound is 0.12 and the increment is 0.01 then the range of values to be considered is:

$$d \in \{0.08, 0.09, 0.10, 0.11, 0.12\}.$$

- *The reuse frequency.* Unlike the other two factors, reuse frequency may vary widely; so that a single range (with fixed increment may prove inadequate. Version 1.1 allows the user to specify up to three consecutive ranges, with different increments. A possible set of values may be, for example:

$$Freq \in \{5, 6, 7, 8, 9, 10, 20, 30, 40, 50\}.$$

8.1.2.2 Version 1.1 Outputs

On output, system ARES Version 1.1 provides the following data:

- A table summarizing all the input data, indicating which data was provided by the user and which data was taken from default values. Ranges are represented by their lower bounds, upper bounds and increments.
- For each value d in the range of discount rates Version 1.1 provides a bidimensional table that includes the component's ROI for all values of the investment cycle Y and all values of the overall reuse frequency $Freq$.
- Further, Version 1.1 provides a bidimensional table that includes the component's intrinsic ROI for all values of the investment cycle Y and all values of the discount rate d.
- Finally, Version 1.1 provides a bidimensional table that includes the component's break even frequencies (the value of $Freq$ for which ROI is zero) for all values of the investment cycle Y and all values of the discount rate d.

8.1.3 Version 1.2: Historic Reusability Evaluation

8.1.3.1 Version 1.2 Inputs

Whereas versions 1.0 and 1.1 focus on a single component, version 1.2 is concerned with the set of reusable components that are currently stored in the library. Version 1.2 can be used in conjunction with the features of 1.0 (based on accurate data) or the features of 1.1 (based on value ranges).

8.1.3.2 Version 1.2 Outputs

In addition to providing the outputs of version 1.0 or version 1.1, version 1.2 returns the following information:

- The average ROI of all the components in the database, as well as the standard deviation.
- A Histogram of the ROI distribution of all the components in the reuse library, as well as an indication of where the current component fits on the scale of the histogram.
- A ranking of the component at hand with respect to the components in the reuse library, as well as ROI information pertaining on the components that rank near the component of interest.

8.2 Reusability of Booch Components

In order to illustrate a concrete use of our ROI measure, and relate it to actual conditions, we compute the ROI of a number of Ada components that have been selected from Grady Booch's library of components [15]. The components in question are: a list, a queue, a dequeue, a ring, and a string. While this experiment does not validate the model per se, it shows that the measures provided by the model reflect some meaningful aspects of the components at hand.

8.2.1 First Experiment: Constant Retrieval Costs

We consider a hypothetical library of reusable software components, and we ponder the question of whether to include in it selected Booch components. We consider that the library contains 40 components, that it is managed on a part-time basis (to an average manpower of 0.1), and that the yearly effort spent searching the library is 1 person month.

In this experiment, we assume that the effort to retrieve a component (by exact or approximate retrieval) is constant. We estimate that the manpower effort required to perform an exact search of a component in this library is 0.1 person days, and the effort required for an approximate search is 0.15 person days. In practice, the retrieval effort depends to a large extent on the component in question (complex components require more effort), but, for the sake of simplicity, we will assume a uniform effort across components for this experiment.

Further, for the sake of this experiment, we assume that all candidate components have the same expected frequencies of reuse (in fact more general components have greater frequencies, and more specialized components have lower frequencies). Because we are dealing with components that are highly packaged, that perform standardized functions, we expect their frequency of exact reuse to be much greater than their frequency of approximate reuse. We take, for the sake of this experiment, a frequency of 10 for exact retrieval and a frequency of 1 for approximate retrieval. In addition, we will focus our

attention on intrinsic (frequency-independent) ROI and on break-even frequency —hence we will downplay the impact of our choice of frequency values.

Because these components have been designed for reuse, we will use the cost model presented in chapter 5, for which we consider an investment cycle of five (5) years and a discount rate of 0.15. The cost model introduced in chapter 5 relies heavily on our estimate of the development for reuse cost of the component at hand. We could not use COCOMO-like models to this effect because such models apply to larger software products. Hence we resolve to proceed as follows:

- Use PC-Metric[1] to estimate the *Software Science* development effort of the component at hand; because this effort is estimated in mental discriminations (rather than person months), we need a conversion factor to bring it to person months.

- We acknowledge that the Software Science measure reflects the programming effort per se but does not reflect all the software engineering activities (feasibility study, requirements phase, manuals activity, testing), that surround the programming effort. In order to take this extra effort into account, we consider the effort distribution by phase and by activity, and we derive the ratio of the programming effort as a fraction of the overall software engineering effort for small size products.

- Because these components are meant to be reused on a large scale, across many product lines, we assign the reuse factor ($RUSE$) the rating extra high.

- We divide the effort figure so obtained by Stroud's constant (20, for 20 mental discriminations per second), then by 3600 (seconds per hour) then by 152 (hours per month).

For all remaining costs, we take the default values provided by ARES (see chapter 7).

Table 8.2 summarizes the results of this experiment. For each component, we give the short term (5-year) ROI, intrinsic ROI and break even frequency; then the long term ROI, intrinsic ROI and break even frequency. For the complete log file produced by ARES, consult appendix D.

The figures given in table 8.2 seem to indicate that the ROI of these components increases with their size: As we go from the list to the string, the intrinsic ROI increases, and the break even frequency decreases; for a given frequency of reuse (which we have taken as 11 in this experiment) the absolute ROI increases. Note that for a given frequency ratio (λ) the ROI of a component can be written as a linear function of the overall frequency as per the following formula:

$$Reu = IR \times (Freq - Bef).$$

This explains why, as IR increases and Bef decreases, Reu increases for a uniform value

[1] ©Set Laboratories, Mulino, OR.

Measure	List	Queue	Dequeue	Ring	String
5-yr Reusability	-0.33	1.42	2.58	5.78	8.15
5-yr Intr. Reus.	0.14	0.28	0.38	0.65	0.85
5-yr Br. Ev. Freq.	13.39	5.99	4.23	2.10	1.39
L.T. Reusability	0.59	1.63	2.31	4.22	5.63
L.T. Intr. Reus.	0.36	0.71	0.94	1.58	2.05
L.T. Br. Ev. Freq.	7.30	3.35	2.35	1.09	0.66

Figure 8.2: Experimental Results: Constant Retrieval Effort

Measures of Size	List	Queue	Dequeue	Ring	String
Lines of Code	191	192	231	293	604
Software Science Length	247	305	382	603	977
Software Science Volume	1356	1645	2098	3470	5542

Figure 8.3: Booch Components: Measures of Size

of $Freq$.

- $Freq$ is the total yearly frequency of reuse of a component
- IR is the *intrinsic ROI* of a component; it is computed as the derivative of Reu with respect to $Freq$
- Bef is the *break-even frequency* of a component for ROI value ρ; it corresponds to the value of annual reuse frequency for which the ROI takes value ρ.

Table 8.3 shows the various measures of size as provided by PC-Metric; clearly, these measures rank the components by increasing values of ROI.

8.2.2 Second Experiment: Variable Retrieval Costs

In light of the results obtained in the previous experiment, one may be tempted to believe that ROI merely increases with size. In order to contradict this belief, we alter the conditions of the experiment and show that the ROI becomes non monotonic; in particular

Measure	List	Queue	Dequeue	Ring	String
5-yr Reusability	-1.186	-0.778	-0.527	-0.225	-0.288
5-yr Intr. Reus.	0.059	0.084	0.099	0.104	0.081
5-yr Br. Ev. Freq.	31.104	20.274	16.349	13.170	14.548

Figure 8.4: Experimental Results: Constant Retrieval Effort

it may decrease as the size increases.

The reason why ROI increases with size in the first experiment is the following: as the size of a component increases, the savings achieved by reusing it also increase. In practice, as the size of a component increases, so does the cost of retrieving it, assessing it, and modifying it. Yet, the first experiment assumed, for the sake of simplicity, that retrieval and assessment costs are constant; further by taking 0.1 persons to manage the library and 1.0 person months spent annually on component retrieval, this experiment has reduced overheads —letting development costs play against instantiation costs and modification costs. Under these conditions, one would always find that it is advantageous to reuse, since instantiation costs and modification costs are lower than development costs.

In this second experiment we wish to depart from these conditions and choose retrieval and assessment costs that increase with the development effort of the component. We have also altered the frequency ratio (six exact retrievals, five approximate retrievals) for the same overall frequency (eleven) in order to increase the impact of the modification costs. The results of our experiment are given in table 8.4; detailed ARES log files are given in appendix D.

We observe that the limited (5 year) ROI of these components is not monotonic, since it reaches its maximum for the Ring component then decreases for the string, as the size of the component increases.

Chapter 9

Analytical Validation

Our chapter defines the reusability of a software component as a return on investment pertaining to the decision to store the component in a reuse library. In this chapter we validate our approach by addressing in turn the following questions: the rationale of defining reusability by means of the return on investment; the validity of the return on investment model.

9.1 Rationale for the Return on Investment Approach

Traditional approaches to defining the reusability of a component attempt to identify all the qualities that promote the component's reusability, and let reusability be reflected by the aggregate of these qualities. It is possible to distinguish between two families of software qualities that have a relationship to software reusability.

- **Black Box Reuse Qualities.** Black box reuse qualities are those that enhance the likelihood that the component be selected in a retrieval operation. These include such features as: generality, abstraction, relevance to the application domain, pervasiveness, usefulness.

- **White Box Reuse Qualities.** White box reuse qualities are those that enhance the likelihood that, once selected in an approximate retrieval operation, the component is easily adapted to the requirements at hand. These include such features as: understandability, modularity, usability.

Although it is intuitively appealing, this approach has a number of weaknesses, which our model attempts to address. Among these weaknesses, we mention:

- *Quantifying qualities.* Most of the qualities presented above do not lead themselves to quantification.

- *Linking qualities to reusability.* Even if all the relevant software qualities were quantified, it is not clear how these qualities affect the measure of reusability. Hence such a measure of reusability is, at best, a vector of several quantities (one for each

quality), which can be compared row by row; such a measure defines a *very* partial ordering, hence many components cannot be compared. Therefore, such a measure cannot be used for decision making.

- *Recognizing environmental factors.* Software qualities are only part of what makes a software reusable. Other considerations include strategic factors and domain factors.

Our approach tackles these weaknesses by producing a quantitative measure of reusability that offers the following features:

- *Quantifiability.* Our measure of reusability amounts to a single abstract number, with no unit. This number can be used for comparisons, for establishing reusability standards, and for defining reusability requirements.

- *Relevance.* The number that we compute as our reusability is nothing but the return on investment associated with storing the component at hand in a reuse library. Such a quantity is quite relevant to a decision maker.

- *Exhaustiveness.* It is well known that the main benefits of software reuse include gains in programmer productivity and program quality. Our model integrates these two aspects by expressing both of them in terms of a single unit: the person-month; we are able to express quality gains in person-months by equating them with the difference between maintenance costs of a reusable component and maintenance cost of a component developed for single use. The production environment is also taken into account in our equations, by means of the reuse frequency factors of our model.

- *Versatility.* Our model provides not only a return on investment figure to reflect reusability, it also returns auxiliary reusability measures such as break-even frequency, intrinsic reusability, and long term reusability, along with combinations of these.

- *Automation.* Along with our reusability model, we have developed an automated system that aids in the collection of reusability data and the derivation of the various reusability measures.

- *Generality.* Although we have developed this model in the context of the traditional waterfall lifecycle, we consider that it is applicable to other lifecycles —as we have shown in our discussions of object oriented programming (chapters 4 and 7).

As a return on investment model pertaining to a reuse lifecycle, our model has some similarities of form with other models [37, 43, 61, 45]. Our model differs from others by the fact that it deals with the component-level investment cycle, whereas others deal with the organization-level investment cycle or the project-level investment cycle. Also, unlike other models, ours analyzes the various cost factors that are involved in the return on investment formula, and discusses means to estimate these factors.

9.2 Analytical Validation of the Return on Investment Formula

We argue that our reusability measure is a valid return on investment model for our reuse lifecycle, on the basis of the following premises:

- *It is based on a well defined reuse lifecycle.* The lifecycle we have defined for our reuse activities is a straightforward extension of existing lifecycles, such as the waterfall lifecycle [14], in which we have carefully integrated reuse objectives [20].

- *It reflects all the activities of the lifecycle.* In order to build our return on investment model we have analyzed the activities of our reuse lifecycle and have investigated the costs and benefits associated with each activity [47, 48, 52, 49].

- *It uses traditional engineering economics formulas.* Given that investment costs and periodic costs and benefits have been identified, the net present worth associated with the investment decision can be derived in a straightforward manner, using traditional sources of engineering economics [64]. The activities that are carried out once at the beginning of the cycle are taken into account as part of the investment set-up costs. The periodic activities that are carried out at each retrieval are taken into account as part of the periodic operational costs.

- *It reflects all the costs and benefits associated with the reuse activities.* The formula of the return on investment includes a number of cost factors and benefit factors that pertain to various activities undertaken (or skipped) during the reuse lifecycle. To estimate these costs and benefits at component storage time, we have used a number of existing cost models [9, 12, 10, 14], as well as existing statistical data from controlled experiments [41, 43, 66]. For the cost of development for reuse, we have used results from the original COCOMO model [14], which we have combined with the reuse equation of COCOMO 2.0 [12] and statistical results from a study carried out at AT& T. For reengineering costs, we have used COCOMO 2.0's equations for estimating the size equivalent of adapted software. For verification costs, we have used traditional figures for development effort distribution by phase [14, 7, 65]. For library factors we prorate a user provided person month figure with the current size of the software library. To estimate baselining costs, we have provided baselining guidelines produced by STARS [67] and by investigations of Japanese reuse practice [55]. Instantiation costs and adaptation costs are derived on the basis of a statistical study carried out by Selby on 2954 software components. The costs of integration and test are derived from traditional figures for development effort distribution by phase [14, 7, 65]. To derive quality costs, we have used recent observations on the typical values of *annual change traffic* [13], statistical data from reuse practice at Hewlett Packard [43], and maintenance effort distribution figures that are derived from observations of 487 maintenance projects [41].

It is noteworthy that a statistical validation of this model by means of experiments is beyond the scope of our study, for many reasons, of which we cite the following:

- To be faithful to the terms of our model, we need to run each experiment for the duration of the investment cycle, for which typical values are 3 years or 5 years.
- Many of the cost factors involved in our reusability model (instantiation costs, modification costs, reengineering costs, verification costs, etc..) cannot be guessed ahead of time, because they depend critically on the individual instances of retrieval of the component. As a result, we propose in chapter 7 to produce an average value for these costs; any individual experiment may deviate arbitrarily from the average, making the testing of the model quite difficult.
- As a consequence of the above observation, an estimation of the reusability of a component may deviate arbitrarily from the actual return on investment of the component without putting into question the validity of the model.

It is noteworthy that none of the existing return on investment models that have been derived for software reuse have been empirically validated. Hence we depend primarily on our analytical argument to support the validity of our model.

Part IV

A Formal Framework for Software Reuse

In addition to the empirical study of reusability that we have conducted in the previous chapters, we wish to discuss, in the sequel, some analytical measures. Whereas empirical measures are based on external observations of costs and benefits, analytical measures are based on a detailed analysis of the structure and design of the components at hand. Consequently, whereas empirical measures take observations of costs and benefits without concern for what drives these costs and benefits, analytical measures are, as their name indicates, based on an analysis of how features of the component lead to costs, and how they lead to benefits. Also, whereas empirical measures can include environmental factors (such as reuse frequency, or library management characteristics), analytical measures are essentially intrinsic to the component.

In order for us to carry out the analysis that is required for our analytical approach, we need a framework in which the activities of software reuse can be defined. In this part we introduce a mathematical foundation whose purpose is to capture important phases and activities in the software reuse lifecycle. This foundation will be used in the next part to define some basic measures that are relevant to software reuse phases and activities, namely: measures of semantic (functional) distance between specifications, which reflect how much functional features two specifications (or a specification and a component) have in common; measures of structural (syntactic) distance between specifications, which reflect how much effort is required to refine one specification into the other; a measure of modifiability, which reflects how much functional distance can be covered by unit of structural distance.

At the same time as it serves as a basis for our analytical approach to software reuse measurement, this part fulfills a function of its own: It serves as a basis for a mathematical discipline of software reuse; as we will see in this part, this discipline enables us to

- Formulate queries to software reuse libraries in a formal, arbitrarily abstract manner.
- Represent reusable components by means of formal specifications, which capture the most salient functional properties of the component, and abstract away irrelevant functional detail.
- Define formal, automatable, matching criteria that ensure a perfect retrieval precision.
- Give a mathematical foundation for program modification, whereby we must modify a component to satisfy a query. The calculus that we provide to this effect ensures correctness with respect to the given specification while it attempts to take the best advantage of the available component.

Hence in addition to its role as a foundation for the analytical measures that we take in part V, this part serves as a foundation for a disciplined approach to software reuse, and will be pursued as a research direction in its own right.

In chapter 10, we discuss how to represent specifications of software components and software requirements using relations; this unified framework allows us to represent reusable components and user queries and to define the conditions under which a component satisfies a query. Also, we will see that it allows us to represent specifications

of static software components (that compute a simple input output function) as well as specifications of dynamic software component (that maintain an internal state) using a single, simple, mathematical model, which is set theory. In chapter 11, we discuss the design and implementation of a system for component storage and retrieval that relies on formal specifications of components and queries; this system, which has been implemented using a theorem prover, offers a perfect retrieval precision (all retrieved components are relevant) —although at the expense of some loss of recall (some relevant components may fail to be retrieved). Finally, in chapter 12 we define a formal framework for component modification; this framework will be used in part V to define a measure of structural distance. Component modification is a vital step in the software reuse lifecycle, because very often retrieved components do not satisfy the user query precisely and must be adapted; it is important to monitor the cost of this adaptation process, and to ensure its correctness preservation.

Chapter 10

Specifying Components and Queries

In this chapter we discuss the use of relations for program specification. In the context of software reuse, specifications can be used for two different purposes: first to represent user queries (for software components that meet some functional properties); second to capture the important functional features of reusable software components.

10.1 Properties of Specifications

We distinguish between the *specification product*, which is the representation of the program specification, and the *specification process*, which the process by which a specification is derived from user requirements. We have identified three desirable properties of the specification product, which are:

- *Simplicity.* A specification must be simple to enable or enhance reading, analysis and comprehension. To achieve simplicity, a specification must be properly structured and its structure must reflect the structure of the problem at hand.
- *Formality.* A specification must be formal to enable/facilitate specification validation and software product verification.
- *Abstraction.* A specification must represent what functional properties are expected from candidate systems rather than how to achieve these properties.

Also, we have identified two properties of the specification process, which are:

- *Completeness.* A specification must reflect all the functional features of the software requirements.
- *Minimality.* A specification must reflect nothing but the functional features of the software requirements.

Liskov and Zilles [44] distinguish between three kinds of software component specifications:

- *Procedural specifications*, which describe the behavior of programs that map input to outputs and have no *internal states.*
- *Data Type Specifications*, which describe the behavior of data types and other data based modules that carry an *internal state* and export access to their data by means of procedures and functions.
- *Continuous Process Specifications*, which describe the behavior of stimulus response systems (such as process control systems, operating systems).

In this chapter, we discuss briefly how relations, which we introduce in section 10.2, can be used to represent all three types of software specifications. In section 10.3 we present a simple relational model that represents procedural specifications, which we call, following Davis [25], *static specifications.* In section 10.4 we present a relational model that represents data type specifications and continuous process specifications, which we refer to, following Davis [25] by the generic name of *dynamic specifications.* Finally in section 10.5 we present the ordering relation of refinement and discuss its lattice properties.

10.2 Mathematics for Structured Specifications

In this section we present briefly some elements of mathematics that will be needed subsequently to discuss foundations for software reuse. Most of the material in this section can be found in traditional sources of discrete mathematics ([68, 19] although some of it may be specific for our purposes [16]. We discuss in turn, relations, relational operators, and relational properties.

10.2.1 Sets and Relations

In this dissertation, we use sets to represent program spaces, we use functions to represent program functions, and we use relations to represent program specifications. Because functions are (special types of) relations, our notation rests in fact on the use of sets and relations. Sets are usually represented by Pascal-like declarations, and are structured as cartesian products of elementary sets. Among the elementary sets, we mention:

- **natural** , the set of natural numbers,
- **integer** , the set of integers,
- **real** , the set of real numbers,
- and **boolean** , the set of boolean values.

Hence for example, if we declare S as follows

```
x: integer;
y, z: real;
```

then S is defined as the cartesian product $S = \textbf{integer} \times \textbf{real} \times \textbf{real}$. Given an arbitrary element s of S, we denote its components by $x(s)$, $y(s)$ and $z(s)$.

We use the traditional set theoretic operations of *union*, denoted by $S \cup S'$, *intersection*, denoted by $S \cap S'$, *complement*, denoted by $\overline{S}$, *cartesian product*, denoted by $S \times S'$, and *power set*, denoted by $\mathcal{P}(S)$.

A *relation* on a set S is a subset of the cartesian product $S \times S$. Given a pair (s, s') in a relation R on space S, we say that s is an *argument* in R, that s' is an *image* in R, and that it is an *image* of s by R. Given an element s in S, the *image set* of s by R is the set denoted by $s.R$ and defined by

$$s.R = \{s' | (s, s') \in R\}.$$

The *antecedent set* of s' by R is the set denoted by $R.s'$ and defined by

$$R.s' = \{s | (s, s') \in R\}.$$

Among the constant relations on a set S, we mention: the *universal relation*, denoted by $L(S)$ (or L when S is implicit) and defined by $L = S \times S$; the *identity relation*, denoted by $I(S)$ (or I if S is implicit) and defined by

$$I = \{(s, s') | s' = s\};$$

the *diversity relation*, devoted by $V(S)$ (or V if S is implicit) and defined as $\overline{I}$; and the *empty relation*, which contains no pairs and is denoted by $\emptyset$.

Given a space S and a variable x that is not part of the definition of S. We denote by $S + x$ the space obtained by the declaration of all the variables of S and variable x. The *projection* of space $S + x$ on space S is the relation Π defined by:

$$\Pi = \{(s, s') | \forall a \overline{\in} S : a(s) = a(s')\}.$$

By $a \overline{\in} S$ we mean that variable a appears in the definition of space S. This definition provides that all variables in S are preserved by the projection and variable x is ignored. Projections allow us to capture the meaning of variable declarations, by reflecting the effect of a block entry and a block exit.

10.2.2 Operations on Relations

Because relations are sets, all set theoretic operations that we have discussed above can be applied to them. In addition, we can apply the following operations.

Among the binary operations on relations we mention the following: The *product* of relation R by relation R' is the relation denoted by $R \circ R'$ and defined by:

$$R \circ R' = \{(s, s') | \exists t : (s, t) \in R \wedge (t, s') \in R'\}.$$

We may sometimes write the product of R by R' merely as RR'.

Among the unary operations on a relation R, we mention the following: The *domain* of relation R is the set denoted by $dom(R)$ and defined by:

$$dom(R) = \{s | \exists s' : (s, s') \in R\}.$$

The *range* of a relation R is the set denoted by $rng(R)$ and defined by:

$$rng(R) = \{s | \exists t : (t, s) \in R\}.$$

The *inverse* of relation R is the relation denoted by $\widehat{R}$ and defined by

$$\widehat{R} = \{(s, s') | (s', s) \in R\}.$$

The *nucleus* of relation R is the relation denoted by $\nu(R)$ and defined by

$$\nu(R) = \{(s, s') | \exists t : (s, t) \in R \wedge (s', t) \in R\}.$$

Equivalently, we can write $\nu(R) = R\widehat{R}$.
The *prerestriction* of relation R to set A is the relation denoted by ${}_{A\backslash}R$ and defined by

$${}_{A\backslash}R = \{(s, s') | s \in A \wedge (s, s') \in R\}.$$

The *postrestriction* of relation R to set A is the relation denoted by $R_{/A}$ and defined by

$$R_{/A} = \{(s, s') | (s, s') \in R \wedge s' \in A\}.$$

The *kernel* of relation R is the relation denoted by $\kappa(R)$ and defined by:

$$\kappa(R) = \{(s, s') | \emptyset \neq s'.R \subseteq s.R\}.$$

The i^{th} *power* of relation R is the relation denoted by R^i and defined as the product of R by itself i times, with $R^0 = I$. The *transitive closure* of relation R is the relation denoted by R^+ and defined as the union of R^i for all $i \geq 1$. The *reflexive transitive closure* of relation R is the relation denoted by R^* and defined as $R^* = R^+ \cup I$.

10.2.3 Properties of Relations

10.2.3.1 Totality Properties

Given a relation R on set S, and given that L is the universal relation on S, we compute the products RL and LR. We find:

$$RL = \{(s, s') | s \in dom(R)\}$$
$$LR = \{(s, s') | s' \in rng(R)\}.$$

A relation R is said to be *total* if and only if $RL = L$ (equivalently, $dom(R) = S$). A relation R is said to be *surjective* if and only if $LR = L$ (equivalently, $rng(R) = S$). A relation R is said to be *rectangular* if and only if $R = RL \cap LR$ (equivalently, $R = A \times B$, for $\emptyset \subset A \subseteq S$ and $\emptyset \subset B \subseteq S$).

10.2.3.2 Equivalence Properties

Relation R is said to *symmetric* if and only if $\widehat{R} = R$. Relation R is said to be *reflexive* if and only if $I \subseteq R$. Relation R is said to be *transitive* if and only if $R^2 \subseteq R$. Relation R is said to be an *equivalence* if and only if it is symmetric, reflexive and transitive.

10.2.3.3 Ordering Properties

Relation R is said to be *antisymmetric* if and only if $R \cap \widehat{R} \subseteq I$ (i.e., if $(s, s') \in R$ and $(s', s) \in R$ then $s' = s$). Relation R is said to be *asymmetric* if and only if $R \cap \widehat{R} = \phi$ (i.e., if $(s, s') \in R$ then $(s', s) \notin R$). Relation R is said to be *connected* if and only if $V \subseteq R \cup \widehat{R}$ (i.e., whenever s and s' are distinct, then either $(s, s') \in R$ or $(s', s) \in R$). Relation R is said to be *strongly connected* if and only if $L \subseteq R \cup \widehat{R}$ (i.e., for any s and s', either $(s, s') \in R$ or $(s', s) \in R$).

Relation R is said to be a *partial ordering* if and only if it is reflexive, antisymmetric, and transitive (example: relation *divides* on the set of natural numbers). Relation R is said to be a *total ordering* if and only if it is a partial ordering and is strongly connected (example: relation *less than or equal* on the set of natural numbers). Relation R is said to be a *strict partial ordering* if and only if it is asymmetric and transitive (example: relation *divides and is distinct of* on the set of natural numbers). Relation R is said to be a *strict total ordering* if and only if it is a strict partial ordering and is connected (example: relation *less than* on the set of natural numbers).

10.2.3.4 Lattice Properties

Let R be a partial ordering (to be interpreted as *less-than*) on set S and let A be a subset of S.

An element a in A is said to be *minimal* (with respect to R) if and only if $R.a \cap A = \emptyset$ (in other words, a has no antecedent by R in A). Given two elements a and b of S, an *upper bound* of a and b is an element of $a.R \cap b.R$. A *least upper bound* of a and b is a minimal element of the set $a.R \cap b.R$. Equivalently, we can define the notions of *maximal* elements, *lower bounds* and *greatest lower bounds.*

> **Definition 10.1** *Given a set S and a partial ordering R, we say that (S, R) is a* lattice *if and only if for any pair of elements a and b there exists a unique least upper bound and a unique greatest lower bound.*

We usually represent the least upper bound of a and b as $a \sqcup b$ and the greatest lower bound of a and b as $a \sqcap b$. Traditionally, a greatest lower bound may be referred to as a *meet* and a least upper bound may be referred to as a *join.*

Example. Let S be **natural** and let R be the relation defined by: $R = \{(s, s') | s' \textbf{ mod } s = 0\}$. In other words, s' is a multiple of s. We claim without proof that this relation is a partial ordering and that the least upper bound is the smallest common multiple while the greatest lower bound is the greatest common divisor. Figure 10.1 presents the lattice defined by relation R on the set $S = \{1, 2, 3, 5, 6, 10, 15, 30\}$. We call this the *divide lattice.* □

An element $\top$ of S is said to be a *universal upper bound* if and only if $R.\top = S$; an

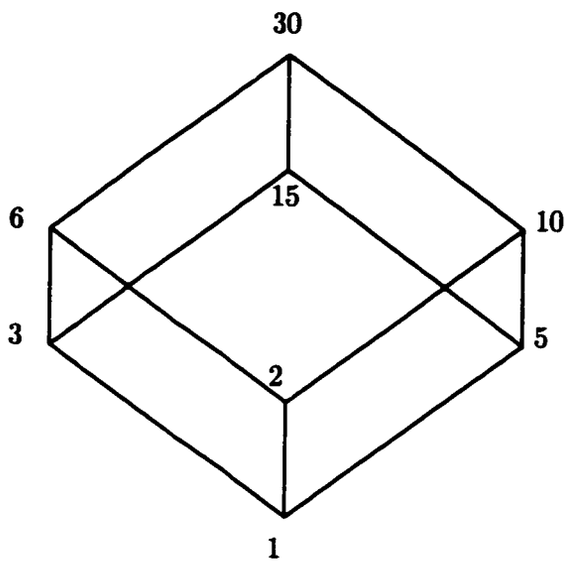

Figure 10.1: A Sample Lattice

element $\perp$ of S is said to be a *universal lower bound* if and only if $\perp.R = S$.

Example. We consider the lattice introduced above on a subset of natural numbers and represented in figure 10.1. The universal lower bound is 1; the universal upper bound is 30. □

10.2.3.5 Determinacy

Given two relations R and R' on S, we say that R is *more-deterministic than* R' if and only if

$$\hat{R}R \subseteq \widehat{R'}R'.$$

A relation R is said to be *deterministic* if and only if it is more-deterministic than the identity relation I (equivalently: $\hat{R}R \subseteq I$). If R is deterministic, we say that it is a *function.*

10.2.3.6 Regularity

Relation R is said to be regular if and only if

$$R = R\hat{R}R.$$

Because $R \subseteq R\hat{R}R$ is a tautology, this definition is in fact equivalent to $R \supseteq R\hat{R}R$.

For the purposes of our work, the interest of regular relations is summarized in the following proposition.

Proposition 10.1 *If R is regular then the kernel of R is identical to the nucleus of R.*

The significance of this proposition is that in practice, nuclei are much easier to compute than kernels.

10.3 Static Specifications

We propose to represent static specifications with two parameters:

- A set S, called the *space* of the specification. Typically, S is constructed as discussed in section 10.2, using the variables that the requirements specification deals with.
- A relation R on S, called the *relation* of the specification.

The space of the specification includes all the *states* that may appear as initial states, or final states of the specification. A state, in this context, is understood to be a function from variable names to variable values [51]. Given a space S defined by, e.g., variables a, b and c, where we suppose that a is of type **natural**, b is of type **integer** and c is of type **real**. A state, says s, is a function from the set $\{a, b, c\}$ to the set **natural**×**integer**×**real**, which we write

$$s : \{a, b, c\} \rightarrow (\textbf{natural} \times \textbf{integer} \times \textbf{real}.$$

Given this definition, we can write (according to [51]), for example:

$$s(a) = 4 \wedge s(b) = -3 \wedge s(c) = 3.14.$$

For the sake of simplicity, we consider that the variable names are ordered (e.g., a then b then c); then we write the state as the triplet of values that the state takes for a, b and c. the state given above can be written as:

$$s = \langle 4, -3, 3.14 \rangle.$$

For the sake of simplicity, we will, in the future, equate a state with the set of values that it assigns to the (ordered) set of variables in our space.[1]

The relation of the specification includes all the pairs that are considered correct input/output pairs. For the sake of minimality, we consider that relations can be arbitrarily partial (with arbitrarily small domains) and arbitrarily non-deterministic (with arbitrarily many outputs for each input). Also, because they focus on inputs and outputs (rather than structural or design details), our relations satisfy the criterion of abstraction. Whenever the space S is implicit, or whenever the exact description of S is not needed for

[1]Note that, according to the notation introduced in section 10.2.1, we then refer to the components of s by $a(s)$, $b(s)$ and $c(s)$, which is the inverse of Mills' notation. This should not, However, cause confusion, because we do not use Mills' notation in this document.

our purposes, we may omit S from the description of our specifications, and represent specifications merely by relations.

Example. We consider the space $S = \mathbf{real}$ and we consider the following simple user requirement:

"Specifying a square root program".

For the sake of illustration, we present below a number of possible interpretations of this simple, straightforward user requirement, and present a relational representation for each such interpretation. The multiplicity of these interpretations shows, incidentally, the need for formal specifications as a means to weed out sources of ambiguity.

1. Only non-negative arguments are submitted; any square root is accepted.

$$R_1 = \{(s, s') | s = s'^2\}.$$

2. Only non-negative arguments are submitted; the non-negative square root is required.

$$R_2 = \{(s, s') | s = s'^2 \wedge s' \geq 0\}.$$

3. For non-negative arguments, return the non-negative square root; negative arguments may be submitted.

$$R_3 = \{(s, s') | s = s'^2 \wedge s' \geq 0\} \cup \{(s, s') | s < 0\}.$$

4. For non-negative arguments, return the non-negative square root; for negative arguments, return 0.

$$R_4 = \{(s, s') | s = s'^2 \wedge s' \geq 0\} \cup \{(s, s') | s < 0 \wedge s' = 0\}.$$

Note that as we go from R_1 to R_4, we obtain specifications that are increasingly stronger, i.e., impose increasingly tighter requirements. We will review this feature in section 10.5. □

10.4 Specifying Dynamic Software

The model presented in section 10.3 is adequate for the representation of static specifications, where the output of the software systems depends exclusively on the current input; but it is not sufficient to represent dynamic specifications, where the output depends not only on the current input but also on past inputs.

In order to accommodate the specification of dynamic systems, we introduce a mathematical model that is based on relations, and constitutes an elaboration of the previous model. We say that we have defined the *specification of a dynamic system* if and only if we have provided the following parameters:

- A set X, called the *input space*. From the input space, we derive the *set of input histories*, which we denote by H and define as $H = X^*$ (i.e., H is the set of sequences of elements of X).
- A set Y, called the *output space*.
- A relation R from H to Y, i.e., a subset of $H \times Y$.

As an illustration of this model, we consider the following user requirements:

> The *counter* system is a dynamic software system that takes three inputs: *reset*, which resets the system, *pulse*, which registers an incoming signal (to increment the counter), and *display*, which causes the counter to display the number of pulses received since the most recent *reset*.

We represent the formal specification of this system as follows:

- Input space, $X = \{reset, pulse, display\}$. We pose: $H = X^*$.
- Output space, $Y =$ **natural** .
- Relation R can be written in closed form as follows:

$$R = \{(h, y) | last(h) \neq display\} \cup \{(h, y) | \exists h'', h' : h = h''.reset.h'.display \wedge reset \notin h' \wedge y = nbpulse(h')\}.$$

The first term of relation R provides that if the most recent operation of h is not *display* then the output is arbitrary; the second term provides that if the most recent operation is *display* then y equals the number of occurrences of operation *pulse* since the most recent *reset*; the occurrence of *reset* highlighted in

$$h''.reset.h'.display$$

is the most recent because of the conjunct $reset \notin h'$.

For more complex specifications, it is not feasible to write a closed form representation such as that given above. We propose then to write an axiomatic representation of this relation, as illustrated below. We will provide in turn some axioms and some rules, along with brief explanations; more information about this representation can be found in [50, 18].

The following are some of the axioms that define relation R given above in closed form:

- **Reset axiom.** $(h.reset, y) \in R$.
  This axiom provides that the output that corresponds to a reset operation is arbitrary.
- **Pulse axiom.** $(h'.reset.h.pulse, y) \in R$.
  This axiom provides that the output that corresponds to a pulse operation is arbitrary; the structure $h'.reset.h$ means that a reset operation must necessarily appear in the input sequence in order for the output to be defined.

- **Display axiom.** $(h.reset.display, 0) \in R$.
  This axiom provides that a display that occurs immediately after reset causes an output of zero (0). A rule will provide an inductive argument to the effect that display counts the number of pulses after the most recent reset.

The following are some of the rules that define relation R given above in closed form:

- **Reset rule.** $\frac{(h'.reset.h,y)\in R}{(h''.reset.h,y)\in R}$.
  This rule provides that operation *reset* reinitializes the system by making everything that has occurred prior to it immaterial (the output is the same for any future sequence h, irrespective of whether the past was h' or h'').

- **Inductive pulse rule.** $\frac{(h.display,y)\in R}{(h.pulse.display,y+1)\in R}$.
  This rule provides that an additional occurrence of operation *pulse* causes the result of *display* to be incremented. This rule, along with the *display axiom*, provide that *display* returns the number of pulses.

- **Idle display rule.** $\frac{(h'.h,y)\in R \wedge h\neq()}{(h'.display.h,y)\in R}$.
  This rule provides that as soon as a display operation has been serviced, it is ignored: whether it has occurred or has not occurred has no impact on the future behavior of the counter.

□

The key conclusion we wish to draw from our discussion of dynamic systems specification is that the specification of all software systems, whether they are static or dynamic, is amenable to a relation. Hence our subsequent discussions will equate specifications with relations.

10.5 The Refinement Ordering

10.5.1 Refinement Properties

In this section we introduce an ordering between specification whose general meaning is that a specification carries more requirements information, or that it is stronger than another. For the sake of readability, we introduce it by means of examples.

We consider specifications R_0 and R_1 on space S=**natural**.

- $R_0 = \{(s, s') | s - 1 \leq s' \leq s + 1\}$.
- $R_1 = \{(s, s') | s - 2 \leq s' \leq s + 2\}$.

We ask the question: *which of these two specifications is stronger.* Intuitively, it is easy to see that R_0 is stronger, because it defines a narrower range for possible output values. Note that in this case, stronger is synonymous with being a subset (as we have: $R_0 \subseteq R_1$).

We consider the following specifications on the same space:

- $R_0 = \{(s, s') | s' = s + 1\}$.
- $R_1 = \{(s, s') | s \leq 200 \wedge s' = s + 1\}$.

We ask the same question. Intuitively, it appears that R_0 is again stronger than R_1: whereas R_1 requires to compute $s+1$ in s' for s between 0 and 200, R_0 requires this result for all s in **natural**. Note that in this case, stronger is synonymous with being a superset (as we have: $R_0 \supseteq R_1$).

We consider the following specifications on the same space:

- $R_0 = \{(s, s') | s \leq s' \leq s + 2\}$.
- $R_1 = \{(s, s') | s \leq 200 \wedge s - 1 \leq s' \leq s + 3\}$.

We ask the same question. Intuitively, it appears that R_0 is yet again stronger than R_1 for a combination of the reasons mentioned above: R_0 deals with more inputs, and imposes a stricter condition on outputs. Note that in this case R_0 is neither a subset nor a superset of R_1.

The following definition captures the intuitive notion of strength of a specification.

Definition 10.2 *Specification R is said to be* more-defined *than specification R' if and only if:*

$$R'L \subseteq RL \wedge R \cap R'L \subseteq R'.$$

When R is more defined than R', we also say that R *refines* R', or that it *is a refinement* of R'. The importance of this ordering relation for our purposes is highlighted in the following definition.

Definition 10.3 *Program P is said to be* correct *with respect to R if and only if the function of P is more-defined than R.*

The significance of the refinement ordering is further highlighted in the following proposition, which we present without proof.

Proposition 10.2 *R is more-defined than R' if and only if any program correct with respect to R is correct with respect to R'.*

Further we admit without proof that the refinement relation is reflexive, antisymmetric and transitive; hence it is a partial ordering.

10.5.2 Lattice Properties

Given that the refinement relation is a partial ordering, it is legitimate to ponder the question of whether it is a lattice. We will see in this section that even though it falls short of being a lattice, this relation has lattice-like properties.

As an intuitive introduction to lattice operators, we consider examples of pairs of relations, and we try to derive a single relation that captures all the requirements information of each.

- $R_0 = \{(s, s') | s' \leq s + 2\}$.
- $R_1 = \{(s, s') | s - 1 \leq s'\}$.

We ask the question: what relation captures all the requirements information of R_0 and all the requirements information of R_1 and nothing more? Intuitively, it is easy to see that the following relation does:

$$R = \{(s, s') | s - 1 \leq s' \leq s + 2\}.$$

Note that in this case the composite specification R is nothing but the intersection of R_0 and R_1.

We consider the following pair of relations, and we ask the same question:

- $R_0 = \{(s, s') | s \leq 200 \wedge s' = s + 2\}$.
- $R_1 = \{(s, s') | s > 200 \wedge s' = s + 2\}$.

Intuitively, it is easy to see that the following relation R captures all the requirements information of R_0 and all the requirements information of R_1.

$$R = \{(s, s') | s' = s + 2\}.$$

Note that in this case the composite specification R is the union of specifications R_0 and R_1. Hence adding up two specifications may sometimes be the union, and sometimes be the intersection.

As a third example, we consider the following pair of specifications:

- $R_0 = \{(s, s') | s' \leq s + 1\}$.
- $R_1 = \{(s, s') | s' \geq s + 3\}$.

We ask the question of what relation R captures all the requirements information of R_0 and all the requirements information of R_1 and nothing more. Clearly, the answer is that there is no such relation, since it is not possible to satisfy R_0 and R_1 simultaneously.

If we summarize the lessons of these examples, we find that the requirements sum of two specifications (relations) may or may not be defined; and when it is defined it may be the union, and may be the intersection (and may take other forms that we have not prospected). The following proposition, which we provide without proof, synthesizes our discussion.

Proposition 10.3 *Any two relations that satisfy*

$$RL \cap R'L = (R \cap R')L$$

have a least upper bound (join) given by

$$R \sqcup R' = R \cap \overline{R'L} \cup R' \cap \overline{RL} \cup R \cap R'.$$

The condition

$$RL \cap R'L = (R \cap R')L$$

checks in effect whether R and R' are consistent, i.e., whether they can be satisfied simultaneously; we call it the *consistency condition*. Also, the formula of $R \sqcup R'$ can be interpreted to mean: all the information of R *plus* all the information of R'.

The following proposition provides that any two specifications have a greatest lower bound (meet), and gives its formula.

Proposition 10.4 *Any two relations R and R' have a greatest lower bound, which is*

$$R \sqcap R' = RL \cap R'L \cap (R \cup R').$$

If we interpret the meaning of this formula, we find that it represents the information that R and R' have in common.

Whenever we discuss lattice structures, it is common to illustrate lattice properties by means of eight sample items arranged in a cube-like fashion by the ordering at hand. This is what we do in the following example. We leave it to the reader to check that the edges drawn in this cube-like structure correspond indeed to the definition of the refinement ordering. Also, the reader may check that the least upper bounds and greatest lower bounds that can be inferred by inspection of the cube do correspond to the formulae given in propositions 10.3 and 10.4.

Example. The perennial cube.

$$S = \textbf{real} \cup \{error\},$$

- $R0 = \{(s, s') | s \in \textbf{real} \wedge s \geq 0 \wedge s' \in \textbf{real}\}$,
- $R1 = \{(s, s') | s \in \textbf{real} \wedge s \geq 0 \wedge s' \in \textbf{real}\}$
  $\cup \{(s, s') | s \in \textbf{real} \wedge s < 0 \wedge s' = error\}$,
- $R2 = \{(s, s') | s \in \textbf{real} \wedge s'^2 \geq s \geq 0 \wedge s' \in \textbf{real}\}$,
- $R3 = \{(s, s') | s \in \textbf{real} \wedge s'^2 \leq s \wedge s' \in \textbf{real}\}$,
- $R4 = \{(s, s') | s \in \textbf{real} \wedge s'^2 \geq s \geq 0 \wedge s' \in \textbf{real}\}$
  $\cup \{(s, s') | s \in \textbf{real} \wedge s < 0 \wedge s' = error\}$,
- $R5 = \{(s, s') | s \in \textbf{real} \wedge s'^2 \leq s \wedge s' \in \textbf{real}\}$
  $\cup \{(s, s') | s \in \textbf{real} \wedge s < 0 \wedge s' = error\}$,

- $R6 = \{(s, s') | s \in \textbf{real} \ \wedge s'^2 = s \wedge s' \in \textbf{real} \}$,
- $R7 = \{(s, s') | s \in \textbf{real} \ \wedge s'^2 = s \wedge s' \in \textbf{real} \}$
  $\cup \{(s, s') | s \in \textbf{real} \ \wedge s < 0 \wedge s' = error\}$.

□
□

10.6 Concluding Remarks

The material presented in this chapter will be used in the remainder of this thesis for the purpose of defining a foundation for software reuse: we use formal specifications to represent user queries, and to represent functional properties of software components; we use the refinement lattice to structure a library of reusable software components; we use the refinement ordering as a basis for a calculus of refinement in the context of software adaptation; we use this calculus of refinement as a basis for defining structural (syntactic) distance between specifications; and we use lattice operators to define functional (semantic) distance between specifications.

The pertinence of formal specifications in software reuse has long been recognized. The *Resolve* project [70] research effort is geared primarily towards the specification and design of reusable software components. *Resolve* combines a specification language, a specification discipline, and a discipline of software development for reuse and with reuse. *Resolve* uses some programming language ideas (such as generality, parameterization), which it carries further than programming languages: *Resolve* modules can be parameterized with respect to a wide range of parameters. Our proposal shares some of its goals and premises with the Resolve methodology: the use of formal specifications and the concern for correctness preservation. The specification language used by Resolve is quite different from ours, since it is *model-based* (whereas ours is behavioral); it uses programming language theory to describe the model (whereas ours uses relations); and it emphasizes component representations (whereas ours is focused primarily on semantic properties of specifications, rather how they are represented). Other projects that use formal specifications to perform reuse-related activities (such as component matching and retrieval) will be discussed in chapter 11.

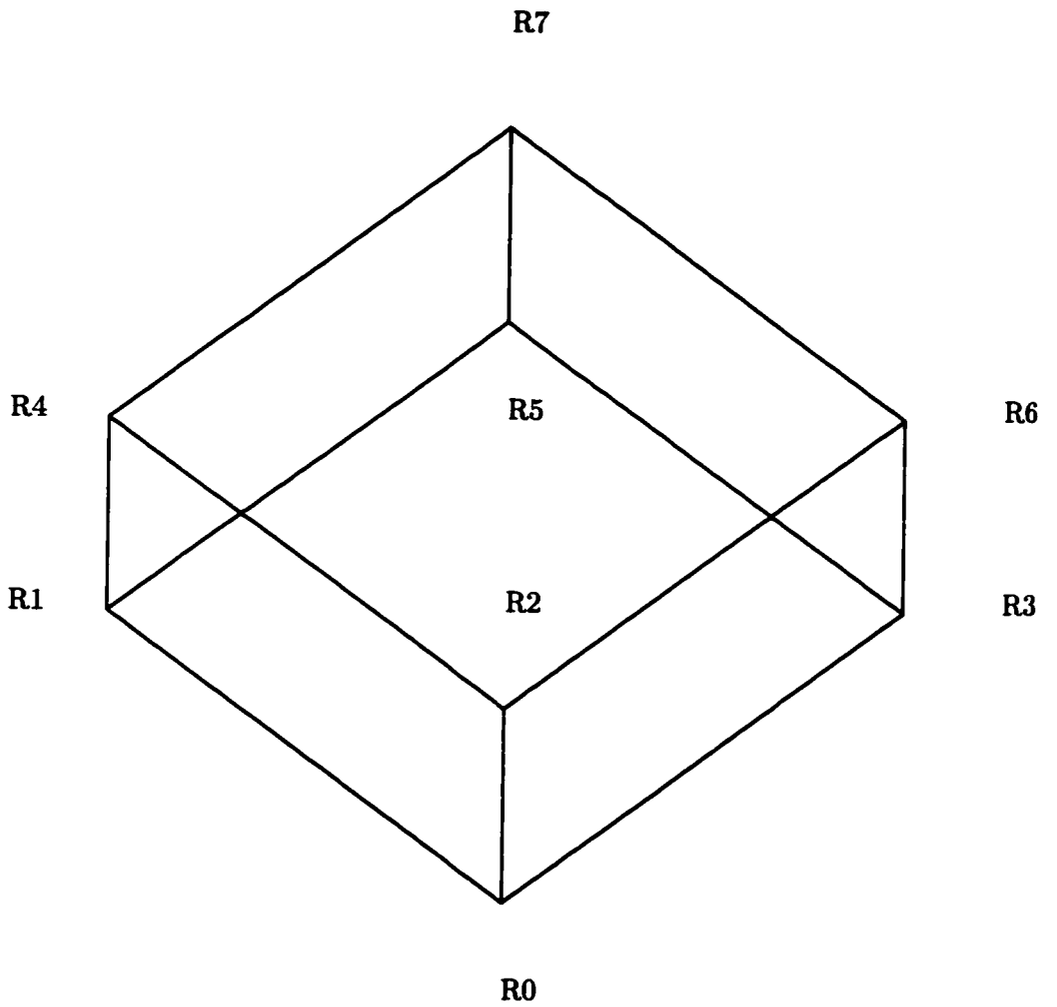

Figure 10.2: A Specifications Cube

Chapter 11

A Formal Approach to Component Retrieval

In this chapter we present a proposal for a formal specification based database structure to support the storage and retrieval of software components. We discus the overall storage structure, then we investigate algorithms for exact retrieval and approximate retrieval of software components stored in the library.

11.1 Background: Storing and Retrieving Reusable Components

The problem of storing and retrieving software components for the purpose of software reuse has attracted a great deal of attention in the past. Given that most companies that practice software reuse operate small-scale libraries (in the hundreds of components) they usually do not need advanced tools for storage and retrieval. Hence much of the research so far has been primarily academic. But as software reuse libraries grow in size, and as library components grow arbitrarily close to each other (for example, several versions of the same component, that have been slightly modified from the original), it becomes increasingly difficult to depend on informal methods for the storage and retrieval of reusable components. Before we present our specification-based solution to component storage and retrieval, we briefly review some earlier work.

Retrieval algorithms can be divided into four broad families [33]: AI-based algorithms [26, 56]; hypertext-based algorithms [32]; library science/ information science algorithms [62, 63]; and formal specifications-based algorithms [27, 46, 58, 71, 78, 79]. In practice, library science/information science techniques appear to be most popular with organizations that practice software reuse [33, 69]. Specification based storage and retrieval procedures include the *Inquire* system of Perry and Popovitch [58] and the Paris system of Katz et al. [40], as well as the works of Chen et al. [21], Zaremski and Wing [79], and Moineau and Gaudel [54].

In [78], Zaremski and Wing discuss a tool for retrieving software components in a software library based on *signature matching*. Signature matching is the process of de-

termining which library components match a query signature. The authors consider two kinds of components *functions* and *modules*, and two kinds of matching *function matching* and *module matching*. They also consider two kinds of match predicates, *exact match* and *relaxed match*. Function matching is based on *function types* matching. Two function types match *exactly* if they are equal modulo variable renaming. The authors present two types of relaxed function match, the *generalized* match and the *specialized* match. Generalized matching is used when the user has difficulties determining the most general type of the desired function but can give an example of what is desired. Conversely, specialized match is used when the user asks for a general type that does not match any function in the library; in this case, the tool returns more specific components. A module is defined as a collection of functions. Module matching is based on *module interface* matching. A module interface is a pair $\langle I_T, I_F \rangle$ where I_T represents a multiset of user-defined types, and I_F a multiset of function types. For a library interface $\langle I_{LT}, I_{LF} \rangle$ to match exactly a query interface $\langle I_{QT}, I_{QF} \rangle$ there must be a total mapping from I_{QF} and I_{LF} (exact function match between the function types in I_{QF} and the corresponding function types in I_{LF}). Since the exact module match is rather restrictive, a relaxed match is defined to allow the querier to specify a subset of the functions and match a module that is more general (may contain additional functions) or more specialized.

In [60], Podgurski and Pierce propose a technique for component storage and retrieval called *behavior sampling*: The user submits information on the signature of a function s/he wishes to retrieve, and the system returns random samples that match the signature and requests the corresponding desired outputs. With the sample inputs and outputs the system executes all the components of the library and selects those that match the chosen sample. In [38], Hall extends Podgurski and Pierce's work by making the signature specification more flexible, by improving the sampling technique, and by running side-effect-free *functional models* of the components — rather than to run the components themselves and undergo their side effects. Further, Hall's technique concerns itself not only with finding components that satisfy the specified behavior, but also with components that may be used to build an aggregate solution to the specified behavior.

Our approach, is based on formal specifications and on the refinement ordering between specifications [17]. In [17] a design is proposed for a repository of software components. The most salient feature of this design is its use of the partial ordering between specifications. This allows a better retrieval precision than algorithms that are based on signature matching [53, 77]. This chapter uses the design proposed in [17] to produce a working prototype based on theorem proving technology. The proposed implementation can be characterized by the following premises:

- Software components are represented in the database by a specification that describes their most important functional properties. Because such specifications may be arbitrarily abstract, they allow us to focus the description on those properties of the component that are most relevant in a retrieval operation. This improves the chances of a match, as it recognizes the relationship between two components that have the same important properties, but differ in minor details.

- A crucial feature for efficient retrieval in traditional databases is the ordering between

keys: when such a feature is available, it becomes possible to perform logarithmic search, and even linear search can be speeded up by a factor of two. There is no total ordering between specifications; however, the refinement ordering, which is partial, affords us some chances to improve retrieval efficiency. In our implementation, this ordering is defined by means of a first order theorem that we submit to a theorem prover for verification.

- Given a *search argument* (the specification for which we are seeking components) and a *key* to a stored component (its specification), we do not require that the key be identical to the search argument; rather, we consider that there is a match as soon as the key *refines* the argument. Then, by definition, any program that is known to be correct with respect to the stored key is correct with respect to the search argument and hence can be returned as a result of the retrieval operation. This weaker condition is sufficient for the purpose of correctness preservation, while dramatically enhancing the chances of a match.

- It may well happen that no component in the database matches (i.e., refines) a given search argument, but that some components satisfy parts of the requirements it expresses. Then we wish to identify software components in the database that satisfy the largest portion of the requirements of the search argument. Topologically, this amounts to identifying keys that minimize some measure of distance from the search argument. We call this *approximate retrieval* . In practice, approximate retrieval yields software components that do not necessarily satisfy the search key, but may be slightly transformed to produce a component that does.

Section 11.2 briefly covers the design of our database, following the outline given in [17]. Section 11.3 introduces the general features of our implementation of this database, and section 11.4 describes how we implement some of the operations of the database Finally, section 11.5 summarizes our conclusions and prospects.

11.2 Architecture of the Library

We consider designing this repository of software components as an exercise in database design. Consequently we consider in turn the external view, followed by the logical view, then the physical view of the database. The external view and the logical view are presented in this section, while the physical view is presented in section 11.3.

11.2.1 The External View

The items stored in our database are *software components*. From a reuse perspective, a software component is an aggregate made up of executable code, source code, requirements specification, user manuals, operations manuals, design documents, operational procedures, etc. In database design (see e.g., [24]), one selects one or more attributes of an entity as its *key* to uniquely identify database entries. Among the fields listed above, the source code offers this property and hence could be considered a candidate. However,

such a key is far too bulky and far too detailed, so that the likelihood of a match with a search argument is virtually null. Further, such a key contains a great deal of information that is irrelevant to any retrieval. Hence we use an aggregate key, of the form $\langle R, p \rangle$, where p is a program source code (or a reference thereof) and R is a formal specification such that p is correct with respect to R. While p is typically very detailed, R can be both arbitrarily abstract, and arbitrarily focused on the important functional properties of program p. Following database texts again, when referring to an entry, we just discuss its key; i.e., in the sequel we consider as an entry merely the $\langle R, p \rangle$ pair.

Example. As an example, let S be the set of natural numbers. Consider the following specifications:

$R_0 = \phi$
$R_1 = \{(s, s') | s' \geq 4s\}$
$R_2 = \{(s, s') | s' \leq 5s\}$
$R_3 = \{(s, s') | 4s \leq s' \leq 5s\}$,

and the following programs,

$p_0 =$ **begin** $s := s + 5$ **end**
$p_1 =$ **begin** $s := 8 * s$ **end**
$p_2 =$ **begin** $s := 2 * s$ **end**
$p_3 =$ **begin** $s := 4 * s$ **end**
$p_3' =$ **begin** $s := 5 * s$ **end**

The following set defines a database of software components, since for each pair the program part is correct with respect to the specification part.

$\Delta = \{\langle R_3, p_3 \rangle, \langle R_3, p_3' \rangle,$
$\langle R_2, p_2 \rangle, \langle R_2, p_3 \rangle, \langle R_2, p_3' \rangle,$
$\langle R_1, p_1 \rangle, \langle R_1, p_3 \rangle, \langle R_1, p_3' \rangle,$
$\langle R_0, p_0 \rangle, \langle R_0, p_1 \rangle, \langle R_0, p_2 \rangle, \langle R_0, p_3 \rangle, \langle R_0, p_3' \rangle\}$.

Operations on the database include, in addition to initialization, the addition and deletion of entries. Further, we introduce the operation of search, which consists in submitting a specification K (the *search argument* or *search key*) for the purpose of seeking all the entries of the database whose specification component refines K (and whose program component is, consequently, correct with respect to K).

11.2.2 The Logical View

At the logical level, we split the overall entry into two distinct *entities* related to each other. These are: the set of *programs*, denoted by Π; and the set of *specifications*, denoted by Σ. Further, we define two relationships on the logical level: the *refinement* relation between specifications, denoted by Θ; and the *correctness* relation, linking specifications to programs that are correct with respect to them, denoted by Γ.

The following observations yield two integrity constraints:

- The refinement relation is transitive. Hence, we need not represent all of its arcs; it suffices to represent an irreducible transitive root thereof. So, we adopt the integrity constraint: The representation of relation Θ does not contain transitive arcs (e.g., if Θ contains $\langle R_0, R_1 \rangle$ and $\langle R_1, R_2 \rangle$ then it does not contain $\langle R_0, R_2 \rangle$).

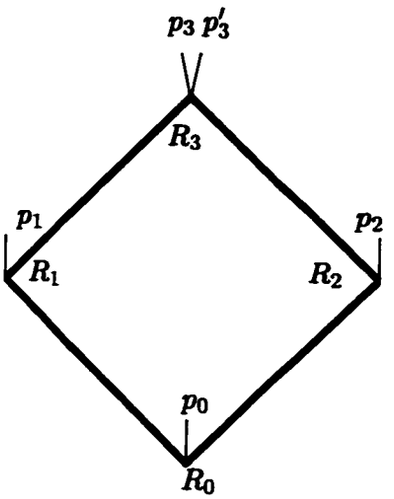

Figure 11.1: A Logical View

- Correctness is transitive, in the following sense: whenever p is correct with respect to R, and R is a refinement of R', we conclude (by transitivity of the refinement ordering, and by definition of correctness) that p is correct with respect to R'. Hence we adopt the integrity constraint: Each program is attached (by relation Γ) to the most refined specification with respect to which it is correct.

The external view (as defined by relation Δ) can be derived from the logical view (as defined by relations Θ and Γ) using the following formula: $\Delta = \Theta^* \circ \Gamma$. In other words, a pair $\langle R, p \rangle$ is in Δ if and only if there exists a specification R' such that $\langle R', p \rangle \in \Gamma$, and that R' refines R (whence $\langle R, R' \rangle \in \Theta^*$). Conversely, the logical view can be derived from the external view by recognizing refinement relationships among specifications, and by not explicitly storing any redundant transitive arcs. Each program is attached to the most refined specification with respect to which it is correct.

As an illustration, we may check that the database defined by the following logical view is the same as that defined by the external view given previously.

$\Pi = \{p_0, p_1, p_2, p_3, p'_3\}$
$\Sigma = \{R_0, R_1, R_2, R_3\}$
$\Gamma = \{\langle R_0, p_0 \rangle, \langle R_1, p_1 \rangle, \langle R_2, p_2 \rangle, \langle R_3, p_3 \rangle, \langle R_3, p'_3 \rangle\}$
$\Theta = \{\langle R_0, R_1 \rangle, \langle R_0, R_2 \rangle, \langle R_1, R_3 \rangle, \langle R_2, R_3 \rangle\}$.

This logical view is illustrated in Figure 11.1. Relation Θ is represented by the arcs that connect specifications, whereas relation Γ is represented by the arcs that connect programs to specifications. We may check that this representation satisfies the integrity constraints given above. The key defined for the external view is split into the specification itself serving as *key for specification entities* and the program serving as *key for program entities.*

The operations of initialization, addition, deletion and retrieval, which we have defined on the external view, must be recast in terms of the logical view; these operations prove

to be graph traversals that can be implemented by traditional graph algorithms. Interestingly, the logical structure we have introduced enables us to define other operations that were not visible in the conceptual view: it is now possible, and indeed meaningful [17], to add and delete specifications (without associated programs) and programs (without associated specifications).

11.3 The Physical View

11.3.1 Basic Premises

The logical view of our database presents two kinds of nodes: program nodes and specification nodes. The basic premises of our physical representation are the following:

- Each specification is described in a file containing a formal definition of the specification (using first order logic) as well as references to other specifications and programs.
- Each program is described in a file containing the text of the program (or a reference thereof), as well as references to specifications (maximal specifications with respect to which it is correct).
- Operations on the database involve comparing specifications (e.g., a search argument against the specification's key) and computing lattice operators between specifications (e.g., computing the meet of a search argument and a key in order to assess their proximity). These operations are carried out by means of a theorem prover. We have chosen *Otter* [1], to support our operations [76].
- The physical view is organized into directories, each containing a family of files: a directory where we store specification files, a directory where we store program files, and a directory where the system stores the results of queries.

11.3.2 File Structures

A specification node is represented by a file containing the following information:

- A section with references to the program nodes attached to this specification by relation Γ.
- A section with references to the specifications this node immediately refines.[2]
- A section with references to the specifications that are immediately refined by the current node.
- A section containing a binary predicate of first order logic, written in Otter's syntax, to define the relation of the current specification.

[1] ©*Argonne National Laboratory*

[2] Reminder: we represent an irreducible transitive root of the refinement ordering, hence we only refer to those specifications the current node refines immediately.

A program node is represented by a file made up of the following four sections:

- A section listing the specification nodes to which this program is attached by relation Γ.
- A section indicating the programming language used for this program.
- A section containing the text of the program or a reference to its source file.
- A section with a description of the program's function written in first order logic according to Otter's syntax.

Inclusion of this last section stems from a compromise we have to make with the integrity of our logical view: given that it is not possible to determine automatically whether a program is correct with respect to a specification by inspection of the program's text, the user must provide this additional information. We are currently investigating means to alleviate this burden on the user.

11.3.3 Example: Retrieving Compilers

To illustrate the physical view, consider the example of Pascal compilers, which was introduced in [17]. This example deals with a family of compilers that map *Pascal* programs into various versions of P-code.

We define three possible input spaces:

- *Simple*: the set of all *Pascal* programs that do not include records, pointers, and user-defined files.
- *Standard*: the set of all syntactically correct *Pascal* programs that are consistent with the ISO standard.
- *Full*: the set of all the strings that can be composed with *Pascal* symbols (including correct *Pascal* programs).

For each set defined above, we represent the predicate defining it with respect to a set containing an arbitrary sequence of characters by its name written in lower case letters (e.g., predicate *simple* defines set *Simple*).

Likewise, we define three possible output spaces:

- *Reduced*: the set of all P-code programs that are restricted to eight registers ($r_0..r_7$) and do not include *inc* instructions (which is instead achieved by addition of 1 through the ALU).
- *Medium*: the set of all P-code programs that use sixteen registers ($r_0..r_{15}$) and do not include *inc* instructions.
- *Complete*: the set of all P-code programs.

From these sets we derive the predicates *reduced*, *medium* and *complete* in the same manner as above. Further, we define two predicates that are applicable on P-code programs: *peephole*(y) means that program y is peephole optimized; and *globopt*(y) means that program y is globally optimized.

We let our database contain the following twelve compilers:

C0: accepting simple *Pascal* programs, yielding complete P-code;

C1: accepting simple *Pascal* programs, yielding reduced P-code;

C2: accepting simple *Pascal* programs, yielding peepholed medium P-code;

C3: accepting standard *Pascal* programs, yielding complete P-code;

C4: accepting standard *Pascal* programs, yielding peepholed medium P-code if the input is in simple *Pascal*, and reporting an unavailable feature otherwise;

C5: accepting any string of *Pascal* terminal symbols, yielding peepholed medium P-code if the input is in simple *Pascal*, reporting an unavailable feature if the input is in standard *Pascal*, and an error message otherwise;

C6: accepting standard *Pascal*, yielding reduced P-code;

C7: accepting simple *Pascal*, yielding peepholed reduced P-code;

C8: accepting standard *Pascal*, yielding peepholed medium P-code as output;

C9: accepting standard *Pascal*, yielding globally optimized reduced P-code;

C10: accepting standard *Pascal*, yielding peepholed reduced code if the input is in simple *Pascal*, reporting an unavailable feature otherwise;

C11: finally, accepting any string of *Pascal* terminal symbols, yielding peepholed reduced code if the input is in simple *Pascal*, reporting an unavailable feature if the input is in standard *Pascal*, and an error message if the input is not in standard *Pascal*.

If we let the specifications of these compilers be represented by relations $R_0..R_{11}$, we get the following definitions:

$$R_0 = \{(x,y)|simple(x) \wedge correct(x,y) \wedge complete(y)\}$$
$$R_1 = \{(x,y)|simple(x) \wedge correct(x,y) \wedge reduced(y)\}$$
$$R_2 = \{(x,y)|simple(x) \wedge correct(x,y) \wedge peephole(y) \wedge medium(y)\}$$
$$R_3 = \{(x,y)|standard(x) \wedge correct(x,y) \wedge complete(y)\}$$
$$R_4 = R_2 \cup \{(x,y)|standard(x) \wedge \neg simple(x) \wedge y = unavailable\}$$
$$R_5 = R_4 \cup \{(x,y)|full(x) \wedge \neg standard(x) \wedge y = incorrect\}$$
$$R_6 = \{(x,y)|standard(x) \wedge correct(x,y) \wedge reduced(y)\}$$
$$R_7 = \{(x,y)|simple(x) \wedge correct(x,y) \wedge peephole(y) \wedge reduced(y)\}$$

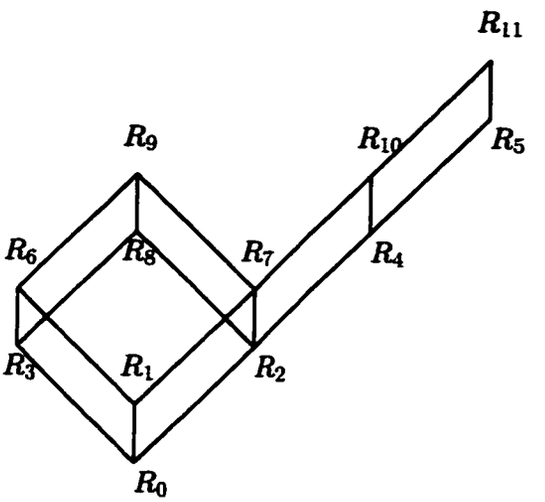

Figure 11.2: A Database of Pascal Compilers

$$R_8 = \{(x,y)|standard(x) \wedge correct(x,y) \wedge peephole(y) \wedge medium(y)\}$$
$$R_9 = \{(x,y)|standard(x) \wedge correct(x,y) \wedge globopt(y) \wedge reduced(y)\}$$
$$R_{10} = \{(x,y)|simple(x) \wedge correct(x,y) \wedge peephole(y) \wedge reduced(y)\}$$
$$\cup\{(x,y)|standard(x) \wedge \neg simple(x) \wedge y = unavailable\}$$
$$R_{11} = R_{10} \cup \{(x,y)|full(x) \wedge \neg standard(x) \wedge correct(x,y) \wedge y = incorrect\}.$$

Figure 11.2 shows these relations as they are ordered by the more-defined ordering; we may imagine that a program is attached to each of these specifications. Hence this figure can be interpreted as the logical view of our database.

Below, we give a brief description of the files that represent specification R_4, and the program that is attached to it, say p_4. The former is stored in directory **spe** under the name **r4.spe**; the latter is stored in directory **prg** under the name **p4.prg**. First, we consider the file **spe/r4.spe**:

```
%programs%     p4.prg
%refines%      r2.spe
%refined by%   r5.spe
               r10.spe
%logical formula%
 (simple(x) & correct(x,y)
            & peephole(y) & medium(y)) |
 (standard(x)&-simple(x)&unavailable(y))
%end%
```

Now, the file **prg/p4.prg**:

```
%specifications%  r4.spe
%language%        C
%program%         p4.c
%functional abstraction%
 (simple(x) & correct(x,y)
  &peephole(y) & medium(y))
 |(standard(x)&-simple(x)&unavailable(y))
%end%
```

In this example, the functional abstraction of program p_4 is identical to the logical formula of specification R_4. This is not always the case; in general, in order for p_4 to be attached to R_4, it suffices that the former be a refinement of the latter.

11.4 Implementing Database Operations

The operations implemented for the database are: initializing the database, inserting a specification, inserting a program, retrieving programs that are correct with respect to a given key, performing an approximate retrieval of programs with respect to a key when the exact retrieval fails; deletion of specifications or of programs is also defined. In this chapter, we discuss only exact retrieval and approximate retrieval.

11.4.1 Exact Retrieval

Given a specification K (the *search argument*), we must retrieve all programs that are correct with respect to K. Our retrieval algorithm proceeds by comparing K against the specification nodes of the database to identify those nodes that refine K. Given that the network of specifications is structured in a lattice-like fashion, the specification nodes are visited in consecutive layers starting from the top, as long as they refine the search argument. The argument is first matched against the maximal nodes of the lattice to see if it refines them. Whenever it is refined by a node R, that node is stored in a set of possible answers; whenever K is found to also refine a descendant of R then R is deleted from the set and replaced by its descendant. Whenever it is found that K refines all the elements of the answer set but does not refine any of their descendants, the search terminates. The answer set contains the minimal specifications that refine the search argument. All the programs attached to these specifications (either directly or by transitivity of the refinement relation Θ) are correct solutions for the search.

Matching of the search argument against a current specification node is formulated as a first order theorem and submitted to the *Otter* theorem prover. During the course of an exact retrieval, our system generates a number of such theorems, submits them to *Otter*, then looks up *Otter*'s output file to determine whether the theorem has been proven. It proceeds according to the outcome of the proof. The general format of such theorems is the following:

```
% checks whether a refines b.
%
% set parameters of inference,
 assign(max_mem,1500).
 assign(max_seconds,360).
 set(free_all_mem).
% set resolution strategy
set(hyper_res).
formula_list(usable).
% in this section we store domain knowledge.
end_of_list.
%
formula_list(sos).
% definition of relations a and b:
%
% here we formulate the theorem that
% provides that A refines B.
 ((domclause&imageclause)<->refines).
 (all x ((exists y A(x,y))<->doma(x))).
 (all x ((exists y B(x,y))<->domb(x))).
 ((all x (domb(x)->doma(x)))<->domclause).
 ((all x all y ((domb(x)&A(x,y))->B(x,y)))
  <-> imageclause).
% goal clause: does A refine B?
   - refines.
end_of_list.
```

As illustration of exact retrieval, consider the following query:

> *We are looking for a compiler that takes standard* Pascal *as input, and produces peepholed medium P-code on output if the input is in simple* Pascal.

Note that the query does not specify what happens if the input is in standard *Pascal* but not in simple *Pascal*; presumably, this implies that the compiler may do anything in such cases. On the basis of this query, we generate a file containing the following information.

```
(simple(x)&correct(x,y)&peephole(y)
&medium(y)) | (standard(x)&-simple(x))
```

In formulating theorems to be submitted to *Otter*, our system places this code as the definition of relation B.

The search proceeds as follows (follow on Figure 11.2): The search argument is matched

1. against node R_{11}, and the system declares a success (R_{11} refines the key);
2. against node R_9, and the system declares a failure (R_9 does not refine the key);

3. against node R_{10}, yielding success;
4. against node R_5, yielding success;
5. against node R_4, yielding success;
6. against node R_2, yielding failure.
7. against node R_7, yielding failure.
8. The program terminates and exits. Upon termination, the result directory contains specification **r4.spe** and its associated program **p4.prg**.

It may be instructive to consider, e.g., the Otter file in which we record the theorem that is generated by the comparison of the search key with specification R_4:

```
 assign(max_mem,1500).
 assign(max_seconds,360).
 set(free_all_mem).
 set(hyper_res).
 clear(print_given).
 clear(print_kept).
 clear(print_back_sub).
formula_list(usable).
% axiomatization of the domain space
 (all x (simple(x) -> standard(x))).
 (all x (standard(x) -> full(x))).
% axiomatization of the range space
 (all y (reduced(y) -> medium(y))).
 (all y (medium(y) -> complete(y))).
 (all y (globopt(y) -> peephole(y))).
% output conditions do not reduce domain
 (all x ((exists y correct(x,y))  ->
        (exists y (correct(x,y)
         & globopt(y) & reduced(y))))).
end_of_list.
formula_list(sos).
% definition of relations a and b
 (all x all y (A(x,y) <->
      (simple(x) & correct(x,y)
        & peephole(y) & medium(y))
       | (standard(x) & -simple(x)
        & unavailable(y)))).
 (all x all y (B(x,y) <->
      (simple(x) & correct(x,y)
        & peephole(y) & medium(y))
       | (standard(x) & -simple(x) ))).
```

```
% code for refinement, as given above
% goal clause: does A refine B?
   - refines.
end_of_list.
```

Particularly noteworthy about this description is how little domain knowledge is required. (See e.g., the clauses listed under `formula_list(usable)`). The few clauses that are written under this section are all that is needed to carry out any proof that deals with the predicates of our domain (i.e., *simple*, *standard*, *full*, etc.).

11.4.2 Approximate Retrieval

Given two specification nodes R and R', and a search argument K, which of R or R' is closer to K in terms of its functional properties? Our approach to approximate retrieval is based on the following answer: We compute $G = R \sqcap K$ and $G' = R' \sqcap K$; if G is more-defined than G' then K is closest to R than to R'. This answer can be justified by the following premises: the *meet* of two relations measures the amount of information that the relations have in common; on the other hand, if a relation refines another then it contains more information.

Example. We use a simple example to illustrate our proposed solution: Let K be defined on some space S by the following formula:

$K = \{(s, s') | a(s, s') \wedge b(s, s') \wedge c(s, s')\}$

and let R and R' be defined as follows:

$R = \{(s, s') | a(s, s') \wedge d(s, s')\}$

$R' = \{(s, s') | a(s, s') \wedge b(s, s')\}$.

Note that neither R nor R' is a refinement of K; hence exact retrieval with key K fails to select R or R'. Note also that R and R' are not comparable by the refinement ordering, hence neither is vacuously a better solution than the other. Our approximate retrieval algorithm will recognize that R' has more in common with K (properties $a(s, s')$ and $b(s, s')$) than does R (property $a(s, s')$), and will therefore select R' as an optimal approximation of K. For the sake of simplicity, we assume that all three relations are total. First, we compute $G = R \sqcap K$. We find by substitution and definition of $\sqcap$:

$$
\begin{aligned}
& G \\
= & \quad \{ \text{ definition of } G \ \} \\
& R \sqcap K \\
= & \quad \{ \text{ definition of meet } \} \\
& (R \circ L) \cap (K \circ L) \cap (R \cup K) \\
= & \quad \{ \ R \text{ and } K \text{ are total, hence } R \circ L = L, K \circ L = L \ \} \\
& L \cap (R \cup K) \\
= & \quad \{ \ L \cap A = A \ \}
\end{aligned}
$$

$$
\begin{array}{ll}
 & (R \cup K) \\
= & \quad \{ \text{ substitution } \} \\
 & = \{(s,s') | a(s,s') \wedge (b(s,s') \wedge c(s,s') \vee d(s,s'))\}.
\end{array}
$$

Similarly, we compute $G' = R' \sqcap K$ and find in analogy to the above reasoning (and because $K \subseteq R'$):

$$
\begin{array}{ll}
 & G' \\
= & \quad \{ \text{ substitution, analogy } \} \\
 & R' \cup K \\
= & \quad \{ \text{ because } K \subseteq R' \} \\
 & R' \\
= & \quad \{ \text{ substitution } \} \\
 & \{(s,s') | a(s,s') \wedge b(s,s') \wedge c(s,s')\}.
\end{array}
$$

Relations G and G' have the same domain (both are total); on the other hand, G' is a subset of G. Hence G' is more-defined than G. Our approximate retrieval algorithm returns R' as an optimal approximation of key K (within the set $\{R, R'\}$).

Using this background, we now address the question: Given a database of software components and a search argument K for which exact retrieval has failed, how do we identify specifications which best approximate specification K? One possible answer would be to compute the *meet* of K with all the nodes in the database and identify those that maximize the *meet*. There are a number of reasons why this option is undesirable:

- It is clearly undesirable to visit all the nodes of the database. Rather, we take advantage of the refinement structure to search more efficiently.
- Because the *meet* operation is monotonic, higher specifications will give higher values for $R \sqcap K$. This yields systematically the maximal elements of the refinement structure, independent of K.
- In fact, more than one specification may maximize the *meet* with K. We should not be looking for any specification R that maximizes $R \sqcap K$; rather, we should be looking for minimal specifications among those that maximize $R \sqcap K$. Indeed, the lower a specification in the refinement structure, the more correct programs it has. (Of course, we aim for high recall).

Hence, we reformulate our solution as follows: identify all the specifications R that maximize $R \sqcap K$; select among them the minimal specifications. The programs that are attached to these specifications (directly or transitively) are solutions to the approximate retrieval query.

To transform this strategy into a working solution, consider the following property: if R and R' are two specifications of the database such that R refines R', then two conditions may arise when we apply the meet operation with K:

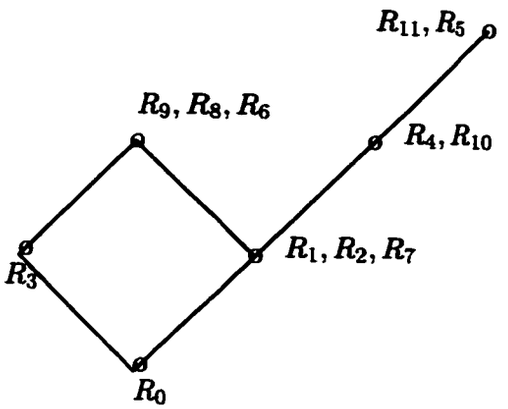

Figure 11.3: Graph derived from Compilers Graph (Fig.2), Collapsed with respect to K

- Either $R \sqcap K$ is the same as $R' \sqcap K$; this occurs whenever the difference between the functional properties of R and the functional properties of R' is totally orthogonal to K. As far as satisfying K is concerned, R is as good as R'.
- Or $R \sqcap K$ is strictly more-defined than $R' \sqcap K$; this occurs when R has more properties in common with K than does R'.

Hence, if we apply the meet operation with K to all the nodes of the refinement graph of the database, we obtain a *collapsed* version of this graph. In this collapsed graph some nodes that are distinguishable in the original graph (see Figure 11.2) have collapsed into a single node and can hence no longer be distinguished (see Figure 11.3).

To perform approximate retrieval with the argument K, our system starts from the maximal nodes in the refinement graph of the database and computes their *meet* with K; then it starts prospecting further and further down the graph as long as the nodes it encounters collapse with the maximal nodes. It returns the set of minimal nodes of the original graph that collapse in the derived graph.

To compute the meet of a specification R with the search argument K our system generates the following *Otter* definition:

```
% defines meet of current node R with arg. K
%
% defining R
 (all x all y (r(x,y) <->
  (definition of R, taken from R.spe))).
% defining K
 (all x all y (k(x,y) <->
  (definition of K, taken from key.spe))).
```

```
% defining meet
 (all x ((exists y r(x,y)) <-> domr(x))).
 (all x ((exists y k(x,y)) <-> domk(x))).
 (all x all y (meet(x,y) <->
  domr(x) & domk(x) & (r(x,y) | k(x,y)))).
```

If `max(x,y)` is the relation obtained by taking the *meet* of K with a maximal specification M, and given that R has been reached from specification M, then we generate the following theorem and submit it to *Otter*:

```
formula_list(usable).
  {here comes definition of meet,
              taken from file above}
  {here comes definition of max,
              taken from specialized file}
formula_list(sos).
% defining collapse
  ((all x all y (meet(x,y) <-> max(x,y)))
                              <-> collapse).
 % does R collapse with M
   - collapse.
```

If the proof is established, we include R in the answer set and delete its immediate ancestor; else we check other specifications in the answer set whose descendants have not all been checked.

To illustrate this operation, we consider the following query:

> *We are looking for a compiler that accepts any string of* Pascal *terminal symbols and returns medium P-code if the input is in standard* Pascal, *or an error message otherwise.*

The search key that this query defines is
$K = \{(x,y)|standard(x) \wedge correct(x,y) \wedge medium(y)\}$
$\cup \{(x,y)|full(x) \wedge \neg standard(x) \wedge y = incorrect\}$.
Execution of the approximate retrieval proceeds as follows:

- Specifications R_{11} and R_9 are put into the answer set.
- Specification R_5 collapses with R_{11}; hence R_5 is included in the answer set.
- Specification R_{10} does not collapse with R_{11}.
- Specification R_8 collapses with R_9; hence R_8 is included.
- Specification R_6 collapses with R_9, and is included.
- Specification R_7 does not to collapse with R_9.
- Specification R_3 does not to collapse with R_9.

Upon termination, the result directory contains specifications R_6, R_8 and R_5. (Specifications R_{11} and R_9 have been deleted because they are not minimal.) Observe that in the case of program p_8, it would suffice to add code to declare an error when the input is syntactically incorrect.

11.5 Concluding Remarks

In this chapter we have investigated the feasibility of using formal specifications to implement a database of software components for the purpose of software reuse. We have relied on plain C programming for the graph traversal aspects of our task, and on theorem proving for the logical aspects. Also, we have illustrated both the design and the implementation of our solution with an example which is not totally trivial: a set of *Pascal* compliers. The intent of this example is to show that even though there is a fair amount of information to deal with, the system does manage to find its way around and provide reasonable answers.

Among the extensions that we envisage for this work, we mention the following. First and foremost, we must address the question of performance: the search given as an example in section 11.4.1, which involves six comparisons of the key against stored specifications, takes about 3 minutes on a Sparc 10 workstation. It is conceivable that this may become prohibitively longer as predicates grow more complex. We are considering the option whereby theorems are preprocessed before they are passed on to *Otter*, e.g., to abstract recurring patterns of predicates.

A second extension we are considering is to improve the recall of our retrieval algorithm: as it is currently designed, the algorithm retrieves programs by considering the specifications to which they are attached; it is conceivable, e.g., that a program p be attached to a specification R in the database, that specification R does not refine the search argument K, and that the program p is correct with respect to K. Hence the exact search does not return p, even though p is correct with respect to K. This loss of recall appears to be a direct consequence of focusing on abstract specifications as opposed to detailed programs. It appears to depend to a large extent on how the database is used. (E.g., we may improve the recall by increasing the density of specifications.) A last extension we are contemplating is to provide the system with a user-friendly interface, which assists users in editing and checking incoming programs and specifications.

Chapter 12

Software Component Modifications

It is rather seldom that a software component retrieved from a reuse library meets the user's requirements precisely. What is far more typical (see chapter 7) is that components that are retrieved approximate the user's requirements and must be adapted to satisfy them precisely. In this chapter, we take a closer look at the mathematics that govern this stepwise program transformation; our results will be used in chapter 14 to define the notion of *structural distance* and in chapter 15 to define an analytical measure of *modifiability*.

In section 12.1, we consider how complex specifications can be structured as aggregates of simpler subspecifications by means of the lattice operators of the refinement lattice. This structuring is used as a basis for program construction by parts: to refine a complex specification into a program, we refine in turn the individual subspecifications, then we combine their solutions to derive a solution to the overall specification. In section 12.2 we briefly present the refinement rules that are used to map partial solutions of subspecifications into a global solution to the overall specification. We have found that this pattern of program construction by parts, whereby we perform refinements by adapting two subspecifications to each other, has much in common with the process of program modification where, given a specification that represents a user query and an available component, we modify the available component to satisfy the user query; in section 12.3 we investigate this analogy and derive a calculus of program refinement by parts. The section 12.4 summarizes our main findings and introduces extensions of these results.

12.1 Structuring Specifications

In [34, 35, 36], Frappier et al introduce a notation for structuring relational specifications, along with transformation rules that allow one to refine specifications into programs in a correctness preserving manner. In this chapter we use the proposed notation and refinement rules to formalize the process of component modification in the context of software reuse. From our viewpoint, the programming constructs proposed by Frappier et al have

three crucial properties:

- They reflect the natural structuring of complex specifications as aggregates of simpler specifications.
- They are monotonic with respect to the refinement ordering, i.e., whenever an operand (specification) is refined, the compound specification is refined.
- Traditional programming language constructs can be mapped into these constructs by means of systematic rules. This property is crucial for our purposes, since we want to represent software components (that are presumably written in traditional programming languages) using the proposed specification notation.

12.1.1 Elementary Statements

We use four elementary statements in the proposed notation, namely the *assignment* statement, the *skip* statement, the *establish* statement (*est*), and the *preserve* statement (*prs*) —which we define in turn below.

- Given a variable x (defined in the space S) and an expression E that can be evaluated for elements of S, the *assignment* statement $x := E$ places in x the value of expression E for the current state, while keeping other program variables intact. This statement is defined if E can be evaluated and if x can hold the value computed for E. We write:

  $$(x := E) = \{(s, s') | x(s') = E(s) \wedge \_(s) = \_(s')\},$$

  where $\_$ stands for all the program variables other than x.

- Given a predicate t, statement *skip(t)* is defined only on those states that satisfy t and returns the initial state intact. It is defined by:

  $$skip(t) = \{(s, s') | s' = s \wedge t(s)\}.$$

- Given a predicate t, statement *est(t)* is defined for all initial states and produces an arbitrary final state that satisfies t. It is defined by:

  $$est(t) = \{(s, s') | t(s')\}.$$

- Given a predicate t, statement *prs(t)* is defined for initial states that satisfy t and returns an arbitrary final state that satisfies t. It is defined by:

  $$prs(t) = \{(s, s') | t(s) \wedge t(s')\}.$$

In addition, we may sometimes, when we do not wish to highlight any particular structure, use closed form relations to represent specifications.

12.1.2 Compound Statements

While elementary statements represent relations, compound statements represent relational operators. In this section we present six compound statements, namely: *join*, *meet*, *monotonic composition*, *monotonic closure*, *restriction* and *parallel combination*. We define them in turn below.

- **Join.** If we have two requirements, that are captured by specifications R and R', and we wish to impose that both requirements be satisfied simultaneously, we take the *join* of relations R and R', as defined in chapter 10. As we have discussed in chapter 10, the join is defined only if R and R' satisfy the *consistency condition*:

  $$cs(R, R') \stackrel{\text{def}}{=} (RL \cap R'L = (R \cap R')L).$$

  Also, when the join is defined, it is given by the following formula:

  $$R \sqcup R' = (\overline{R'L} \cap R) \cup (\overline{RL} \cap R') \cup (R \cap R').$$

  Example. To illustrate the join construct, we consider the space S defined by variables a, b and g of type **natural**, where a and b are non-zero, and we consider the requirement that the greatest common divisor of a and b be stored in g. This can be defined in terms of the following clauses:

  - The (final) value of g divides (the initial value of) a and b.
  - The (final) value of g is maximal (any integer greater than g does not divide a and b).

  To reflect this decomposition, we write (CD stands for *common divisor*, and MX stands for *maximal*):

  $$CD = \{(s, s') | a(s) \textbf{ mod } g(s') = 0 \wedge b(s) \textbf{ mod } g(s') = 0\}.$$

  $$MX = \{(s, s') | \forall k : k > g(s') \Rightarrow (a(s) \textbf{ mod } k \neq 0 \vee b(s) \textbf{ mod } k \neq 0)\}.$$

  Whence we take

  $$GCD = CD \sqcup MX.$$

  We present a brief argument to the effect that these relations satisfy the consistency condition: CD and MX are both total, hence $CD \circ L = L$ and $MX \circ L = L$, hence $CD \circ L \cap MX \circ L = L$. On the other hand, $CD \cap MX = GCD$, where GCD is the gcd function, which known to be total; hence $GCD \circ L = L$. □

- **Meet.** Given two specifications R and R', if we want to capture the requirements information that they both have in common, we take their *meet*. Their meet is defined for all R and R', and is given by the following formula:

$$R \sqcap R' = (RL \cap R'L) \cap (R \cup R').$$

Example. To illustrate meets, we consider the following specifications defined on the space S of positive real numbers:

$$R = \{(s, s') | s \geq 1 \wedge s' = s^2\},$$

$$R' = \{(s, s') | s \leq 1 \wedge s' = s^{-2}\}.$$

We find,

$$\begin{array}{ll} & R \sqcap R' \\ = & \quad \{ \text{ definition } \} \\ & RL \cap R'L \cap (R \cup R') \\ = & \quad \{ \text{ substituting } RL \text{ and } R'L \} \\ & \{(s, s') | s \geq 1 \wedge s \leq 1\} \cap (R \cup R') \\ = & \quad \{ \text{ simplifying, distributing the prerestriction } \} \\ & \{(s, s') | s = 1 \wedge s' = s^2\} \cup \{(s, s') | s = 1 \wedge s' = s^{-2}\} \\ = & \quad \{ \text{ simplifying } \} \\ & \{(1, 1)\}. \end{array}$$

This relation does indeed capture the information that is common between R and R'. □

- **Monotonic Composition.** When we wish to apply two specifications R and R' in sequence, we combine them by the *monotonic composition*, which is defined as follows:

$$R \circ R' = RR' \cap \overline{\overline{RR'L}}.$$

This relation is a subset of RR' (the traditional relation composition) and equals RR' whenever R is deterministic or R' is total. Unlike the traditional composition operator, this operator is monotonic with respect to the refinement ordering, i.e., whenever R (or R') is refined, so is the product $(R \circ R')$.

Example. Given a specification *CodGen* for a code generator (from some internal program representation) and a specification *CodOpt* for a code optimizer, we can specify an optimizing code generator by writing:

$$OptCodGen = CodGen \,\square\, CodOpt.$$

□

- **Monotonic Closure.** Given a relation R that we wish to see applied an arbitrary number of times, we take its *monotonic closure*, which is defined by:

  $$R^\star = \sqcap_{i \geq 0} R^{\Delta i},$$

  where $R^{\Delta i}$, for $i \geq 1$, is the monotonic composition of R by itself i times, and $R^{\Delta 0} = I$.

  Example. Given relation R defined on the set of natural numbers by

  $$R = \{(s, s') | s' = s - 5\},$$

  we find that the monotonic closure of R is given by:

  $$R^\star = \{(s, s') | s \geq s' \wedge s \textbf{ mod } 5 = s' \textbf{ mod } 5\}.$$

  □

- **Prerestriction.** Given a relation R and a predicate t, if we wish to restrict the domain of R to those elements that satisfy t, we use the *prerestriction* operator, which is defined by:

  $$t \rightarrow R = \{(s, s') | t(s) \wedge (s, s') \in R\}.$$

  Example. We consider the space $S =$ **natural** and the relation R defined by:

  $$R = \{(s, s') | s' = s \textbf{ mod } 5\}.$$

  If we define predicate t on S by: $s < 5$ then we find, e.g.,

  $$t \rightarrow R = t \rightarrow I.$$

  □

- **Parallel Combination.** Given two relations R and R', if we wish to apply them sequentially (i.e., one after the other) but do not care in what order they are applied, we use the *parallel combination* operator. This operator is defined by:

$$R||R' = (R \circ R') \sqcap (R' \circ R).$$

Given that the meet reflects the information that is common between its arguments, the parallel combination of R and R', as defined here, captures the information that is common between the two statements: R has executed, followed by R'; and R' has executed, followed by R. The common information is that R and R' have both executed, in an arbitrary order.

Example. Given variables a and b of type **integer**, consider that we want to increment them but do not care in what order we do so (because it does not make any difference, since the statements are independent). We write:

$$a := a + 1||b := b + 1.$$

□

- **Variable introduction.** Given a specification R on space S, if we wish to introduce a fresh variable that is not already in space S, we use the **var** construct, as given in the following statement: **var** $x : T$ R. The semantics of this statement is given by

$$\textbf{var } x : TR = \widehat{\Pi} \circ R \circ \Pi,$$

where Π is the *projection relation* from set $S + x$ to set S (see the definition of projections on page 121, section 10.2.1). Among the properties of variable introduction, we mention that variable introduction is monotonic with respect to the refinement ordering, and that it refines its argument, R (i.e., (**var** $x : T$ R) refines R).

We admit without proof that variable introductions are commutative. Whenever more than one variable is introduced, we write the declarations in sequence following a single instance of **var**; also when the variables have the same type we may write the type only once. Hence e.g., if we wish to introduce variables x, y and z of type natural, we write: **var** x, y, z : **natural** . Usually we omit to write explicitly the declarations associated with a specification, unless we need to —for the purposes of our discussion or our computations.

12.1.3 Mapping Rules

In the sequel, we will often have to manipulate software components and software specifications simultaneously, for such tasks as: modifying the component to meet a given specification, measuring the modification effort, measuring the functional distance between the program and the specification, etc. For all these applications, it is convenient that both the program and the specification be represented in the same notation. In this section we discuss how Pascal-like programs can be refined by the specification notation introduced above.

To this effect we briefly review the main control structures of, e.g., Pascal, and show for each control structure what specification construct captures it. We give these mappings without formal justification; the interested reader is referred to [34, 35, 36].

1. **begin end.** We use parenthesized relational expressions to represent *begin-end* blocks.

2. **composition.** We use monotonic composition to represent Pascal's semi-colon. Note that when R is deterministic (which is the case of all Pascal statements) $R \,\square\, R'$ is equal to RR'.

3. **alternation.** Pascal's statement (**if** t **then** F **else** G) is mapped into

$$(t \rightarrow F) \sqcup (\neg t \rightarrow G).$$

4. **conditional.** Pascal's statement (**if** t **then** F) is mapped into

$$(t \rightarrow F) \sqcup (\neg t \rightarrow I).$$

5. **iteration.** Pascal's statement (**while** t **do** B) is mapped into

$$((t \rightarrow B) \sqcup (\neg t \rightarrow I))^{\star} \sqcup \mathbf{est}(\neg t).$$

12.2 Refinement Rules

Given a specification written using the notation proposed in the previous section, we are interested in transforming it into an executable Pascal-like program. Frappier et al [34, 35, 36] have derived a set of transformation rules which are used to support the stepwise refinement of specifications into programs in a correctness preserving fashion. We use these same rules for a different purpose: to carry out the stepwise transformation of a program (presumably a software component retrieved by *approximate retrieval*) to satisfy a new specification (which, presumably, the program *almost* satisfies initially). If we review the main compound statements discussed above we find that the statement which is most difficult to map into programming notation is the *join*: indeed, a meet of two (or more) specifications can be mapped into one of its arguments; a monotonic composition can be mapped into a traditional sequential composition; a monotonic closure can be mapped into an iteration (under some restrictive conditions); a prerestriction can be mapped into a conditional statement; and a parallel combination can be mapped into the sequential combination of its arguments in an arbitrary order. As a result of this observation, the transformation rules are geared towards the goal of getting rid of joins, in favor of programming-like constructs; these rules are presented below.

12.2.1 Eliminating Joins

In this section we survey a number of rules that map a join-structured specification into a specification/program that has no join (at least not at the upper level). These rules are presented without proof (of their correctness preservation); the interested reader is referred to [34, 35, 36] for details.

Refinement Rule 1 *A specification of the form $R \sqcup R$ is refined by R.*

Under some conditions, joins can be mapped into Pascal-like statements; these possibilities are explored below. These rules may seem to be redundant with the mapping rules presented in the previous section; in fact they play opposite roles, as we will see later in this chapter. Mapping rules transform programs into specifications prior to the modification step; whereas refinement rules map specifications into programs as part of the modification process.

Refinement Rule 2 *Given a space S with variables x_i, $1 \leq i \leq N$, and given an expression $E(s)$ on S, the specification*

$$\{(s, s') | x_1(s') = E(s) \wedge \forall i > 1 : x_i(s') = x_i(s)\}$$

refines and is refined by

$$x_1 := E.$$

Refinement Rule 3 *If specifications R and R' satisfy the condition $R \cap R'L \subseteq R'$ then their join is defined; further this join refines and is refined by*

if t **then** R **else** R',

where $t(s) \equiv s \in dom(R)$.

As a special case of application of this rule, we consider the case when $R'=\overline{RL} \cap I$. This case yields the conditional statement,

if t **then** R.

Refinement Rule 4 *When a specification of the form ((***if** t **then** R)*$\sqcup$**est**$(\neg t)$*) satisfies the following conditions*

- $t \rightarrow R$ *is progressively finite.*
- $t \subseteq RL$,

then the join is defined, and the specification refines and is refined by **while** t **do** R.

12.2.2 Propagating Joins

All the rules in this section stem from a single, general proposition, which we give without proof (the interested reader is referred to [34, 35, 36]).

Proposition 12.1 *Let ϕ be a monotonic (with respect to the refinement ordering) binary operator (this result generalizes trivially to larger arities). Then the specification $\phi(R, R') \sqcup \phi(Q, Q')$ is refined by $\phi(R \sqcup Q, R' \sqcup Q')$, provided all the joins are defined.*

All the refinement rules given below are instances of this general proposition, for various substitutions of operator ϕ.

Refinement Rule 5 *Specification $(R \circ R') \sqcup (Q \circ Q')$ is refined by $(R \sqcup Q) \circ (R' \sqcup Q')$.*

Refinement Rule 6 *Specification $(R||R') \sqcup (Q||Q')$ is refined by $(R \sqcup Q)||(R' \sqcup Q')$.*

Refinement Rule 7 *Specification $(R \sqcap R') \sqcup (Q \sqcap Q')$ is refined by $(R \sqcup Q) \sqcap (R' \sqcup Q')$.*

Refinement Rule 8 *Specification $R^* \sqcup Q^*$ is refined by $(R \sqcup Q)^*$.*

Refinement Rule 9 *Specification $(t \rightarrow R) \sqcup (t \rightarrow Q)$ is refined by $(t \rightarrow (R \sqcup Q))$.*

Refinement Rule 10 *Specification (**var** $x : T$ R)$\sqcup$ (**var** $x : T$ Q) is refined by (**var** $x : T$ $R \sqcup Q$).*

Refinement Rule 11 *Specification (**if** t **then** R **else** R')$\sqcup$ (**if** t **then** Q **else** Q') is refined by (**if** t **then** $R \sqcup Q$ **else** $R' \sqcup Q'$).*

Refinement Rule 12 *Specification (**while** t **do** R)$\sqcup$ (**while** t **do** Q) is refined by (**while** t **do** $R \sqcup Q$).*

12.3 Software Adaptation

The subject of this section is to introduce a mathematical model for software adaptation, particularly in the context of software reuse. We are given a software component C and a specification K, and we ponder how to derive from C a component that satisfies K. First, we give a set of refinement rules that are useful for our purposes.

12.3.1 Imposing Specification Structures

We give, without proof, rules where a specification is decomposed into a predefined form; we will consider in turn sequential decompositions then closure decompositions.

Refinement Rule 13 *Specification* $\mathbf{est}(t)$ *is refined by* $\mathbf{est}(t) \circ \mathbf{prs}(t)$.

One way to establish t is to establish it then preserve it.

Refinement Rule 14 *Specification* $\mathbf{est}(t)$ *is refined by* $\mathbf{est}(\mathbf{true}) \,\square\, \mathbf{est}(t)$.

One way to establish t is to postpone it then establish it.

Refinement Rule 15 *Specification* R *is refined by* $\kappa(R) \,\square\, R$.

This rule stems from a characterization of the kernel of relation R as a feasible solution to the equation $R \sqsubseteq X \,\square\, R$. The three rules above deal with sequence decompositions; we turn our attention now to closure decompositions.

Refinement Rule 16 *Specification* $\mathbf{prs}(t)$ *is refined by* $\mathbf{prs}(t)^\star$.

Refinement Rule 17 *Specification* $\kappa(R)$ *is refined by* $\kappa(R)^\star$.

Refinement Rule 18 *Specification* $R^\star$ *is refined by* $(\ \mathbf{if}\ t\ \mathbf{then}\ R)^\star$ *for all predicate* t.

12.3.2 Adaptation Model

We consider a specification K that a user/programmer submitted to a reuse library, and we assume that the exact search failed to turn up a correct component, and that the approximate search produced a component C (exact retrieval and approximate retrieval are defined in chapter 11). The question we ponder in this section is: how do we transform component C in a minimal number of steps so as to make it satisfy specification K? We observe that this consists primarily of deriving a program that satisfies K and has (as much as possible) the structure of C. Further, we note that the refinement rules presented above give us a formal framework for achieving two goals.

- Imposing a specific structure on a specification; this will be used to impose the structure of C (or its derived components) onto K (or its derived components).
- Once K and C have the same structure (namely that of C), this framework allows us to match components of K against their homologous components of C.

Using these features we model the process of modifying C to satisfy K as the refinement of

$$K \sqcup C,$$

but with an important qualification: The refined solution has to satisfy K but does not have to satisfy C; rather it has to have as much as possible the same structure as C. Hence in refining $K \sqcup C$ we extract syntactic information from C and semantic information from K. The idea is to force on specification K the structure of component C and to propagate the $\sqcup$ construct deeper and deeper into the structure of C, with smaller and smaller components of K.

12.3.3 Illustration

We consider the following user query K defined on space S=

x, y, z : **natural**

by

$$K = \{(s, s') | x(s') = x(s) \wedge y(s') = 0 \wedge z(s') = z(s) + y(s)\}$$

and we assume that the exact search performed on the reuse library has failed, and that the approximate search has produced the following software component defined on space T=

x, y, w : **natural**

by

```
C:
begin
w:=0;
while y<>0 do
   begin
   y:=y-1;  w:=w+x
   end
end.
```

Our goal is to modify component C to make it satisfy specification K. To this effect, we must first map this Pascal program to our specification notation. If we let relations N (for *iNit*), D (for *Decrement*) and A (for *Add*) be defined as follow (to capture the three assignment statements of the program):

$$N = \{(s, s') | w(s') = 0 \wedge y(s') = y(s) \wedge x(s') = x(s)\}$$
$$D = \{(s, s') | x(s') = x(s) \wedge y(s') = y(s) - 1 \wedge w(s') = w(s)\}$$
$$A = \{(s, s') | x(s') = x(s) \wedge y(s') = y(s) \wedge w(s') = w(s) + x(s)\},$$

then program C can be written as

$$C = \textbf{var } x, y, w : \textbf{natural } (N \,\square\, ((\textbf{ if } y \neq 0 \textbf{ then } (D \,\square\, A))^* \sqcup \textbf{est}(y = 0))).$$

Because D and A can be applied in an arbitrary order, we can actually rewrite C as follows:

$$C = \textbf{var } x, y, w : \textbf{natural } (N \,\square\, ((\textbf{ if } y \neq 0 \textbf{ then } (D || A))^* \sqcup \textbf{est}(y = 0))).$$

The first step is to bring the two specifications under the same space; to this effect, we invoke the refinement rule 10 and proceed as follows:

$$
\begin{array}{ll}
 & C \sqcup K \\
= & \quad \{ \text{ variable introduction } \} \\
 & \textbf{var } x, y, w : \textbf{natural } C \sqcup \textbf{var } x, y, z : \textbf{natural } K \\
= & \quad \{ \text{ commutativity of variable declarations } \} \\
 & \textbf{var } x, y, z, w : \textbf{natural } C' \sqcup \textbf{var } x, y, z, w : \textbf{natural } K' \\
= & \quad \{ C', K' \text{: bodies of } C, K \text{; rule 10 } \} \\
 & \textbf{var } x, y, z, w : \textbf{natural } C' \sqcup K'
\end{array}
$$

We now focus on $C' \sqcup K'$, where C' and K' are the bodies of C and K. We propose to incorporate the modification imposed by K deep into the structure of C. We observe that the outer structure of C is a sequence; hence we attempt to use refinement rule 15 to decompose K as a sequence. We find,

$$
\begin{array}{ll}
 & \kappa(K) \\
= & \quad \{ K \text{ is deterministic, proposition 10.1 } \} \\
 & K\widehat{K} \\
= & \quad \{ \text{ substitution } \} \\
 & \{(s, s') | x(s') = x(s) \wedge z(s') + y(s') = z(s) + y(s)\}.
\end{array}
$$

If we denote this relation by P, we know now that K is refined by $P \sqcap K$; if we denote the second component of C by W (for *While* statement), we can write:

$$
\begin{array}{ll}
 & C \sqcup K \\
\sqsubseteq & \quad \{ \text{ substitution of } C \text{, refinement of } K \} \\
 & (N \sqcap W) \sqcup (P \sqcap K) \\
\sqsubseteq & \quad \{ \text{ sequence refinement rule 5 } \} \\
 & (N \sqcup P) \sqcap (W \sqcup K).
\end{array}
$$

We consider the two components of this product in turn.

$$
\begin{array}{ll}
 & N \sqcup P \\
= & \quad \{ \text{ because } N \text{ and } P \text{ are total, and so is } N \cap P \} \\
 & N \cap P \\
= & \quad \{ \text{ substitution } \} \\
 & \{(s, s') | w(s') = 0 \wedge y(s') = y(s) \wedge x(s') = x(s) \wedge z(s') + y(s') = z(s) + y(s)\} \\
= & \quad \{ \text{ simplification } \} \\
 & \{(s, s') | w(s') = 0 \wedge y(s') = y(s) \wedge x(s') = x(s) \wedge z(s') = z(s)\} \\
\sqsubseteq & \quad \{ \text{ definition of assignment statement } \} \\
 & (w := 0).
\end{array}
$$

We now turn our attention to the term $(W \sqcup K)$. Because W has the structure of a while statement (join of an establish statement with the closure of a conditional), we attempt to impose the same structure on K.

$$
\begin{array}{ll}
 & K \\
= & \quad \{ \text{ substitution } \} \\
 & \{(s,s') | x(s') = x(s) \wedge y(s') = 0 \wedge z(s') = z(s) + y(s)\} \\
= & \quad \{ \text{ arithmetic } (y(s') = 0) \} \\
 & \{(s,s') | x(s') = x(s) \wedge y(s') = 0 \wedge z(s') + y(s') = z(s) + y(s)\} \\
= & \quad \{ \text{ set theory } \} \\
 & \{(s,s') | x(s') = x(s) \wedge z(s') + y(s') = z(s) + y(s)\} \cap \{(s,s') | y(s') = 0\} \\
= & \quad \{ \text{ definition of join } \} \\
 & \{(s,s') | x(s') = x(s) \wedge z(s') + y(s') = z(s) + y(s)\} \sqcup \{(s,s') | y(s') = 0\} \\
= & \quad \{ \text{ definition of establish } \} \\
 & \{(s,s') | x(s') = x(s) \wedge z(s') + y(s') = z(s) + y(s)\} \sqcup \mathbf{est}(y = 0) \\
= & \quad \{ \text{ observation: the first term is } \kappa(K) \} \\
 & \kappa(K) \sqcup \mathbf{est}(y = 0) \\
\sqsubseteq & \quad \{ \text{ by virtue of rule 17 } \} \\
 & \kappa(K)^* \sqcup \mathbf{est}(y = 0) \\
\sqsubseteq & \quad \{ \text{ by virtue of rule 18 } \} \\
 & (\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ \kappa(K))^* \sqcup \mathbf{est}(y = 0).
\end{array}
$$

Now that we have exhibited the same structure in W and K, we take their join and propagate the join to the components of W. We find

$$
\begin{array}{ll}
 & W \sqcup K \\
\sqsubseteq & \quad \{ \text{ substitution of } W, \text{ refinement of } K \} \\
 & ((\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ (D||A))^* \sqcup \mathbf{est}(y = 0)) \\
 & \sqcup (\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ \kappa(K))^* \sqcup \mathbf{est}(y = 0)) \\
= & \quad \{ \text{ commutativity, associativity, idempotence of join } \} \\
 & (\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ (D||A))^* \\
 & \sqcup (\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ \kappa(K))^* \sqcup \mathbf{est}(y = 0)) \\
= & \quad \{ \text{ associativity of join, monotonicity of conditional and closure } \} \\
 & (\ \mathbf{if}\ y \neq 0\ \mathbf{then}\ (D||A) \sqcup \kappa(K))^* \sqcup \mathbf{est}(y = 0)) \\
\sqsubseteq & \quad \{ \text{ rule 4, whose conditions are verified } \} \\
 & \mathbf{while}\ y \neq 0\ \mathbf{do}\ ((D||A) \sqcup \kappa(K)).
\end{array}
$$

We now focus our attention on the body of the while loop, namely $((D||A) \sqcup \kappa(K))$. We find,

$$
\begin{array}{ll}
 & ((D||A) \sqcup \kappa(K)) \\
\sqsubseteq & \quad \{ \text{ } || \text{ is refined by } \square \text{ in any order } \} \\
 & ((A \square D) \sqcup \kappa(K)) \\
\sqsubseteq & \quad \{ \kappa(K) \sqsubseteq F \square D, \text{ where } F = \{(s,s') | x' = x \wedge y' = y \wedge z' = z+1\} \} \\
 & ((A \square D) \sqcup (F \square D)) \\
\sqsubseteq & \quad \{ \text{ rule 5 } \} \\
 & ((A \sqcup F) \square (D \sqcup D)) \\
\sqsubseteq & \quad \{ \text{ rule 1 } \} \\
 & (A \sqcup F) \square D \\
= & \quad \{ A \text{ and } F \text{ are total, satisfy the consistency condition } \} \\
 & (A \cap F) \square D \\
= & \quad \{ \text{ substitution of } (A \cap F) \} \\
 & \{(s,s') | x(s') = x(s) \wedge y(s') = y(s) \wedge w(s') = w(s) + x(s) \wedge z(s') = z(s) + 1\} \square D \\
= & \quad \{ \text{ substitution of the first term } \} \\
 & \{(s,s') | x(s') = x(s) \wedge y(s') = y(s) \wedge w(s') = w(s) + x(s) \wedge z(s') = z(s)\} \\
 & \square \{(s,s') | x(s') = x(s) \wedge y(s') = y(s) \wedge w(s') = w(s) \wedge z(s') = z(s) + 1\} \square D \\
\sqsubseteq & \quad \{ \text{ rule 2, applied three times } \} \\
 & w := w + x; z := z + 1; y := y - 1.
\end{array}
$$

Putting all the pieces together, we find the following modified program.

```
var
   x, y, z, w: natural;
begin
w:=0;
while y<>0 do
   begin
   w:=w+x;
   z:=z+1;
   y:=y-1
   end
end;
```

This program does preserve x, set y to 0 and place $z + y$ in z. Hence it is correct with respect to K.

12.4 Concluding Remarks

In this chapter, we have considered the question of *program modification*, which can be formulated as follows: Given a specification K and a component C, how can we modify C to solve K? Alternatively, this question can also be formulated as: How can we solve K by making use of C? The assumption, in both cases, is that C *almost* satisfies K, hence it

takes presumably little effort to modify C to satisfy K—hopefully much less than it takes to solve K from scratch.

We have observed that the modification of a component C to satisfy a query K has similarities with the stepwise refinement by parts of the compound specification $K \sqcup C$, but with two crucial differences:

- Whereas in the refinement by parts of $R \sqcup R'$ specifications R and R' play symmetric roles, in the modification of C to satisfy K the terms C and K play distinct roles.
- In particular, in refinement by parts the bottom of recursion occurs whenever one of the terms refines the other, whereas in program modification the recursion ends whenever C refines K or C is irrelevant with respect to K (in which case K must be implemented from scratch).
- Whereas in the refinement by parts of $R \sqcup R'$ either R or R' may impose its structure on the other, in the modification of C to satisfy K the structure of C is imposed on K. This produces a situation where C provides syntactic information and K provides semantic information pertaining to the modified version of C.

Our approach to program modification is inductive, and proceeds in the following manner:

1. Consider the outer control structure of C. Using structure coercion rules, impose the structure of C on specification K.
2. Using Propagation rules, propagate the join operator inside the common structure that is identified between K and C.
3. Consider the subproblems generated by propagation rules, that have the form $K_i \sqcup C_i$, where K_i is a descendant of K and C_i is a descendant of C.

The induction ends whenever a C component refines its homologous K component (in which case no further modification is required), or whenever a C component proves to be of no use in solving the corresponding K component (in which case the K component must be solved from scratch).

With this chapter, we have concluded the presentation of our relation-based model for software reuse. The model we have presented captures such aspects of software reuse activities as: component specification; component storage; query specification; component retrieval; approximate retrieval; component adaptation. In the sequel, we use this background to define an analytical basis for software reuse measurement.

Part V

Analytical Measures of Reusability

The notion of distance between specifications pervades much of what a software engineer does, especially in the context of software reuse. Hence, e.g. approximate retrieval (chapter 11) consists in identifying a library component that minimizes some measure of distance with the search key; also, the assessment of a candidate component (to decide whether it is advantageous to modify it to solve a given query, or whether it is best to develop a new component from scratch) depends on comparing the distance between the component and the query against the effort it takes to develop a solution from scratch.

In this part we concern ourselves with defining measures of distance between specifications. We distinguish between two families of distance measures: *functional distance*, which reflects the functional features of the specifications at hand; and *structural distance*, which reflects the structural differences that exist between the specifications at hand. While the first measure is intrinsic to the specifications at hand, the second measure deals with their representation. Also, while the second measure takes numeric (quantifiable) values, the first measure ranges over a partially ordered set. These measures are the subject of chapters 13 and 14; in chapter 15 we use these measures to define an analytical measure of modifiability. We view modifiability as a key factor of intrinsic reusability, and we view this study as a first step towards an analytical approach to reusability measurement.

Chapter 13

Measures of Functional Distance

Much of the technical decision making that takes place in software reuse depends, in fact, on an intuitive perception of distance between two specifications, or between a specification and a component. In this chapter and the next, we attempt to give meaning to this intuition by formally defining measures of distance between specifications; because components can be represented by their specifications, our measure of distance between specifications can be used to derive the distance between two specifications or the distance between a specification and a component. As we discussed in the introduction to this part, we can distinguish between functional measures and structural measures of distance; functional measures are the focus of this chapter.

Intuitively, we can define the distance between two specifications in a number of ways: either we reflect the information that the specifications have in common, then we consider that two components are closer to each other if the common information is greater; or by the information that discriminates between the two specifications, then we consider that two components are closer to each other if the discriminating information is smaller; or by a combination of these two quantities, then we consider that two components are closer to each other if they have more common information and less discriminating information.

13.1 Refinement Difference

To measure the distance between two specifications, we may want to focus on the functional features that discriminate between them. As a first step, we will, in this section, consider the case of two specifications A and B such that $A \sqsubseteq B$, and wish to capture the difference between A and B. In order to justify the definition of *refinement difference*, which we give below, we consider the analogy with numeric distance.

We consider two real numbers a and b such that $a \leq b$. Observe that the difference between a and b is the smallest number x that satisfies the following equation

$$a + x \geq b.$$

Because we are looking for the smallest x, it will achieve the equality $a + x = b$; whence we find $x = b - a$. Hence the following definition.

Definition 13.1 *Given specifications A and B such that $A \sqsubseteq B$, the* refinement difference *between A and B is the least defined relation X such that*

$$A \sqcup X \sqsupseteq B.$$

When such a relation exists, we denote it by $B \ominus A$.

Given A and B such that $A \sqsubseteq B$, we know that there exists a relation D such that $A \sqcup D = B$, since $D = B$ satisfies this condition. We investigate whether there exists a minimal relation for which the condition holds and eventually what is its formula.

Proposition 13.1 *Given relations A and B such that $A \sqsubseteq B$, the* refinement difference *of A and B is given by*

$$B \ominus A = B \cap \overline{AL} \cup (A \cap \overline{B})L \cap (B \cup \overline{A}).$$

In order to formally justify our claim, we would have to show that A and $B \ominus A$ satisfy the consistency condition, and that their join refines B; furthermore, we would have to show that $B \ominus A$ is minimal with respect to all the feasible solutions. We will not do so, in this thesis; we will, however, give informal arguments and illustrative examples to the effect that $B \ominus A$ is indeed the refinement difference between A and B. We have the following proposition.

Proposition 13.2 *For any relation A, $A \ominus A = \emptyset$.*

Proof. We write the expression $A \ominus A$ using the definition of refinement difference.

$$\begin{aligned}
& A \ominus A \\
= & \qquad \{ \text{ definition of } \ominus \} \\
& A \cap \overline{AL} \cup (A \cap \overline{A})L \cap (A \cup \overline{A}) \\
= & \qquad \{ \text{ since } A \subseteq AL,\ A \cap \overline{AL} = \emptyset \} \\
& (A \cap \overline{A})L \cap (A \cup \overline{A}) \\
= & \qquad \{ \text{ set theory } \} \\
& \emptyset \cap L \\
= & \qquad \{ \text{ set theory } \} \\
& \emptyset.
\end{aligned}$$

qed

The empty relation ($\emptyset$) is the universal lower bound of the lattice of specification, hence it is the smallest value that a functional difference can take. In the following proposition, we check that this distance is the empty relation only whenever $A = B$.

Proposition 13.3 *Given A and B such that $A \sqsubseteq B$. If $B \ominus A = \emptyset$ then $A = B$.*

Proof. If $B \ominus A$ equals the empty relation, so do the two terms of the union (ref: the definition of refinement difference). We consider them in turn.

$$
\begin{array}{ll}
 & B \cap \overline{AL} = \emptyset \\
\Rightarrow & \quad \{ \text{ set theory } \} \\
 & B \subseteq AL \\
\Rightarrow & \quad \{ \text{ relational identity } \} \\
 & BL \subseteq AL \\
\Rightarrow & \quad \{ \text{ because } A \sqsubseteq B,\ AL \subseteq BL \ \} \\
 & AL = BL.
\end{array}
$$

On the other hand,

$$
\begin{array}{ll}
 & (A \cap \overline{B})L \cap (B \cup \overline{A}) = \emptyset \\
\Rightarrow & \quad \{ \text{ set theory } \} \\
 & (A \cap \overline{B})L \cap \overline{A \cap \overline{B}} = \emptyset \\
\Rightarrow & \quad \{ \text{ set theory: } X \cap \overline{Y} = \emptyset \Rightarrow X \subseteq Y. \ \} \\
 & (A \cap \overline{B})L \subseteq (A \cap \overline{B}) \\
\Rightarrow & \quad \{ \text{ because the inverse inclusion is a tautology } \} \\
 & (A \cap \overline{B})L = (A \cap \overline{B}).
\end{array}
$$

This equation has the form $QL \subseteq Q$, for $Q = (A \cap \overline{B})$. There are only two relations Q that satisfy the equation $QL \subseteq Q$; these are $Q = L$ and $Q = \emptyset$. Taking $Q = L$, we find $A = L$ and $B = \emptyset$, which contradicts the earlier premise $AL = BL$. Taking $Q = \emptyset$, we find:

$$
\begin{array}{ll}
 & (A \cap \overline{B}) = \emptyset \\
\Rightarrow & \quad \{ \text{ set theory } \} \\
 & A \subseteq B \\
\Rightarrow & \quad \{ \text{ because } AL = BL \text{ and } A \sqsubseteq B, \text{ we find } B \subseteq A \ \} \\
 & A = B.
\end{array}
$$

qed

Proposition 13.4 *Given an arbitrary relation A, the refinement difference $A \ominus \emptyset$ is defined and equals A.*

Proof. Because $\emptyset$ is the universal lower bound, it is refined by A; hence the refinement difference $A \ominus \emptyset$ is defined. We compute it from the definition:

$$
\begin{aligned}
& A \ominus \emptyset \\
= & \qquad \{ \text{ substitution } \} \\
& A \cap L \cup (\emptyset \cap \overline{A}) L \cap (A \cup L) \\
= & \qquad \{ \text{ simplification } \} \\
& A.
\end{aligned}
$$

qed

Let A and B be two relations such that $A \sqsubseteq B$, and let D be the refinement difference between A and B, i.e., $D = B \ominus A$. Rather than prove that A and D satisfy the consistency condition, and that B is the join of A and D, and that D is the smallest relation whose join with A yields B, we give informal arguments on how D is defined to satisfy these properties. We consider figure 13.1, where A is drawn on the first column, B is drawn on the second column, and D is drawn on the third column. Because A is refined by B, the domain of B is larger than the domain of A and the restriction of B to the domain of A is a subset of A. Whenever B is defined and A is not (ref: input x), relation D must behave like B; this justifies the component $(B \cap \overline{AL})$, which represents all the pairs (s, s') of B such that s is not in the domain of A. On the domain of A (ref: set $\{y, z\}$ in figure 13.1), two cases may occur: either A and B behave alike (ref: state y), in which case D has no information to add to A to obtain B, and is not defined for those states (because D must be minimal, it is not defined on any state unless it has to); or the images assigned by B are a proper subset of those assigned by A (ref: state z), in which case relation D must include the pairs of B (ref: $(z, 6)$) as well as (for the sake of minimality) all the pairs that do not belong to A (we include no pairs of A so that the join of A and D yields only the pair of B, and we include all the other pairs for the sake of minimality). The relation that takes the prescribed values is $(B \cup \overline{A})$. We can tell whether D must be defined (ref: case of state z) or must not (ref: case of state y) by taking the restriction of $B \cup \overline{A}$ to $(A \cap \overline{B})L$: this restriction excludes from consideration domain elements for which A and B assign the same images.

In order to help the reader develop an intuition for our definitions in terms of the lattice of refinement, we draw analogies between the refinement lattice and the lattice of the *divides* relation on positive natural numbers. In terms of the divides lattice, the proposition above provides in effect that whenever a divides b, the distance between a and b is the ratio $\frac{b}{a}$; hence e.g., the distance between 5 and 15 is 3, the distance between 12 and 60 is 5. Figure 13.2 gives a graphic illustration of refinement difference and justifies the name given to this measure of distance, especially in contrast with subsequent measures.

13.1.1 Illustrative Examples

In this subsection we give examples of relations A and B such that $A \sqsubseteq B$, and we compute the refinement difference $B \ominus A$. We start with simple examples.

Example. We consider the space S=**real** and we define relations A and B as follows:

$$A = \{(s, s') | s = s'^2\}$$

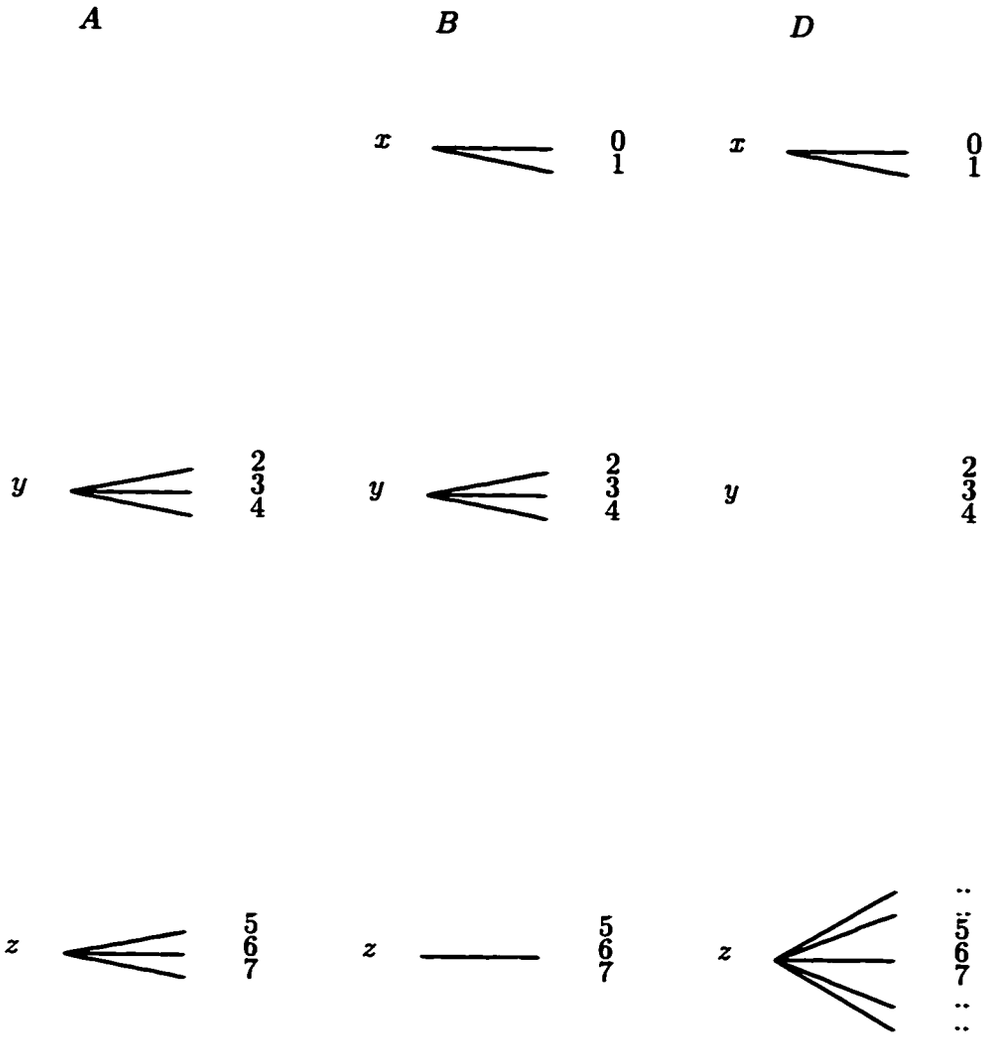

Figure 13.1: Formula of Refinement Difference

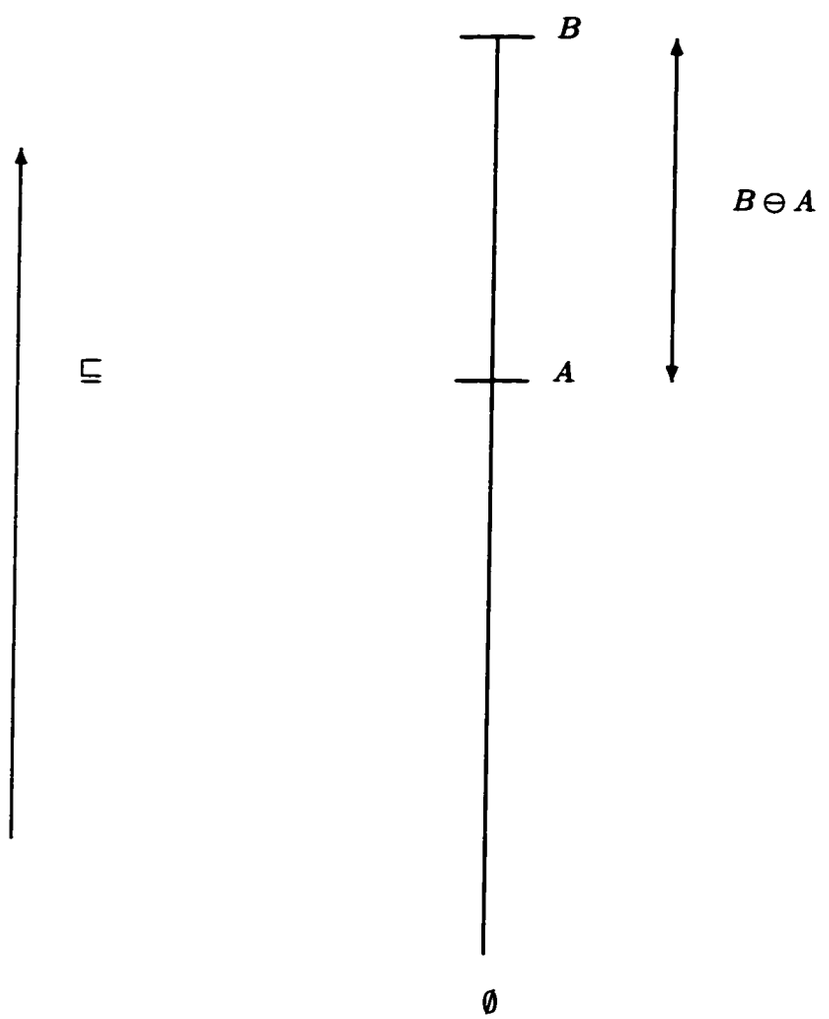

Figure 13.2: Refinement Difference

$$B = \{(s, s') | s < 0\} \cup \{(s, s') | s = s'^2\}.$$

Clearly, A is refined by B, since B has a larger domain and behaves as A does on the domain of A. We compute the refinement difference between A and B.

$$\begin{array}{ll} & B \ominus A \\ = & \quad \{ \text{ definition of refinement difference } \} \\ & B \cap \overline{AL} \cup (A \cap \overline{B})L \cap (B \cup \overline{A}) \\ = & \quad \{ \text{ because } A \subseteq B,\ A \cap \overline{B} = \emptyset \ \} \\ & B \cap \overline{AL} \\ = & \quad \{ \text{ expansion of } B \ \} \\ & \{(s, s') | s < 0\} \cap \overline{AL} \cup \{(s, s') | s = s'^2\} \cap \overline{AL} \\ = & \quad \{ \text{ we find } AL = \{(s, s') | s \geq 0\}, \text{ hence } \overline{AL} = \{(s, s') | s < 0\} \ \} \\ & \{(s, s') | s < 0\} \cup \{(s, s') | s = s'^2\} \cap \{(s, s') | s < 0\} \\ = & \quad \{ \text{ expanding the intersection } \} \\ & \{(s, s') | s < 0\} \cup \{(s, s') | s < 0 \wedge s = s'^2\} \\ = & \quad \{ \text{ the second term is empty } \} \\ & \{(s, s') | s < 0\}. \end{array}$$

This result is quite intuitive: the refinement difference between A and B is the term $\{(s, s') | s < 0\}$. □

Example. We consider the space S=**real** and we define relations A and B as follows:

$$A = \{(s, s') | s = s'^2\}$$
$$B = \{(s, s') | s = s'^2 \wedge s \geq 0\}.$$

Clearly, A is refined by B, since A and B have the same domain and B is a subset of A. We compute the refinement difference between A and B.

$$\begin{array}{ll} & B \ominus A \\ = & \quad \{ \text{ definition of refinement difference } \} \\ & B \cap \overline{AL} \cup (A \cap \overline{B})L \cap (B \cup \overline{A}) \\ = & \quad \{ \text{ because } AL = BL \ \} \\ & B \cap \overline{BL} \cup (A \cap \overline{B})L \cap (B \cup \overline{A}) \\ = & \quad \{ \text{ because } B \subseteq BL \ \} \\ & (A \cap \overline{B})L \cap (B \cup \overline{A}). \end{array}$$

We now compute the expression $(B \cup \overline{A})$.

$$
\begin{array}{ll}
& B \cup \overline{A} \\
= & \{ \text{ substitution } \} \\
& \{(s, s') | s = s'^2 \wedge s \geq 0 \vee s \neq s'^2\} \\
= & \{ \text{ logic simplification } \} \\
& \{(s, s') | s \neq s'^2 \vee s' \geq 0\}.
\end{array}
$$

On the other hand,

$$
\begin{array}{ll}
& (A \cap \overline{B})L \\
= & \{ \text{ set theory } \} \\
& \overline{(B \cup \overline{A})L} \\
= & \{ \text{ substitution } \} \\
& \{(s, s') | s = s'^2 \wedge s' < 0\} \circ L \\
= & \{ \text{ definition of product } \} \\
& \{(s, s') | \exists s' : s = s'^2 \wedge s' < 0\} \\
= & \{ \text{ simplification } \} \\
& \{(s, s') | s \geq 0\}.
\end{array}
$$

Now we resume the derivation of the refinement difference between A and B.

$$
\begin{array}{ll}
& B \ominus A \\
= & \{ \text{ derivation above } \} \\
& (A \cap \overline{B})L \cap (B \cup \overline{A}) \\
= & \{ \text{ expansions above } \} \\
& \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\}.
\end{array}
$$

Note that $D = \{(s, s') | s' \geq 0\}$ does satisfy the equation $A \sqcup D = B$, but is not minimal, since it refines the relation given above. □

13.2 Refinement Distance

In the previous section we had defined the refinement difference between two specifications in the case where the specifications are ordered by refinement. In this section we consider two arbitrary specifications that are not necessarily ordered.

13.2.1 Prime Specifications

In number theory, we say that two natural numbers are relatively prime if and only if their greatest common divisor (i.e., their meet with respect to the *divides* lattice) is 1 (i.e., the universal lower bound of the *divides* lattice). Likewise, we say that two specifications are relatively prime if and only if they have no functional details in common, i.e., their meet

with respect to the refinement lattice is the empty relation, which is the universal lower bound.

Definition 13.2 *Specifications A and B are said to be* relatively prime *if and only if their meet is the empty relation.*

This definition is inspired by the lattice of the *divides* relation over the set of natural numbers: two numbers are said to be *relatively prime* whenever their only common divisor (hence their meet) is 1 (the universal lower bound of the lattice).

Proposition 13.5 *Two specifications A and B are relatively prime if and only if their domains are disjoint.*

Proof. The condition is sufficient:

$$
\begin{array}{ll}
 & AL \cap BL = \emptyset \\
\Rightarrow & \quad \{ \text{ applying intersection on both sides } \} \\
 & AL \cap BL \cap (A \cup B) = \emptyset \\
\Rightarrow & \quad \{ \text{ definition of meet } \} \\
 & A \sqcap B = \emptyset.
\end{array}
$$

The condition is necessary:

$$
\begin{array}{ll}
 & A \sqcap B = \emptyset \\
\Rightarrow & \quad \{ \text{ definition of meet } \} \\
 & AL \cap BL \cap (A \cup B) = \emptyset \\
\Rightarrow & \quad \{ \text{ monotonicity } \} \\
 & AL \cap BL \cap A = \emptyset \wedge AL \cap BL \cap B = \emptyset \\
\Rightarrow & \quad \{ \text{ because } AL \cap A = A \text{ and } BL \cap B = B \} \\
 & A \cap BL = \emptyset \wedge AL \cap B = \emptyset \\
\Rightarrow & \quad \{ \text{ both express that } A \text{ and } B \text{ have disjoint domains } \} \\
 & AL \cap BL = \emptyset.
\end{array}
$$

qed

Intuitively, if two specifications are totally orthogonal, then it is reasonable to consider that the distance between them is their sum (i.e., their join).

Definition 13.3 *Given two specifications A and B that are relatively prime, their* refinement distance *is the join of A and B.*

In terms of the divides lattice, this definition provides that the distance between two natural numbers a and b that are relatively prime is the product ab: in order to go from a to b, we divide a by a (to find 1) then we multiply the result by b (to find b), for a total distance of $a \times b$.

Figure 13.3 gives further illustration to our definition: the expression $A \sqcup B$ is monotonic with respect to its arguments A and B (as long as they grow in an orthogonal fashion, i.e., while preserving $A \sqcap B = \emptyset$).

13.2.2 Arbitrary Specifications

We refer again to the analogy with the divides lattice: given two natural numbers a and b, we consider their respective decompositions as products of prime numbers; then we can measure the distance between a and b (vis á vis the divides ordering) as the product of all the prime factors that appear in a and not in b with the prime factors that appear in b and not in a.

Example. We consider the natural numbers $a = 210$ and $b = 330$. We write their prime number decompositions as follows:

$$a = 2 \times 3 \times 5 \times 7,$$

$$b = 2 \times 3 \times 5 \times 11.$$

Intuitively, we want to consider the distance between a and b as being identical to the distance between those prime factors that differ between a and b, namely 7 and 11. Because 7 and 11 are relatively prime, their distance (which is their join) equals their product. In this case, we find that the distance between a and b is 77. □

This example inspires the following definition.

Definition 13.4 *Given two specifications arbitrary A and B, the* refinement distance *between A and B is the join of $A \ominus (A \sqcap B)$ and $B \ominus (A \sqcap B)$.*

The expression $A \ominus (A \sqcap B) \sqcup B \ominus (A \sqcap B)$ will be abbreviated by $A \otimes B$ and will be referred to as the *refinement distance* between A and B. This expression is defined only if $A \ominus (A \sqcap B)$ and $B \ominus (A \sqcap B)$ have a join; note that the refinement difference expressions are, in turn, defined since in each case the first term refines the second (A refines $A \sqcap B$ and B refines $A \sqcap B$). We admit the following proposition without proof.

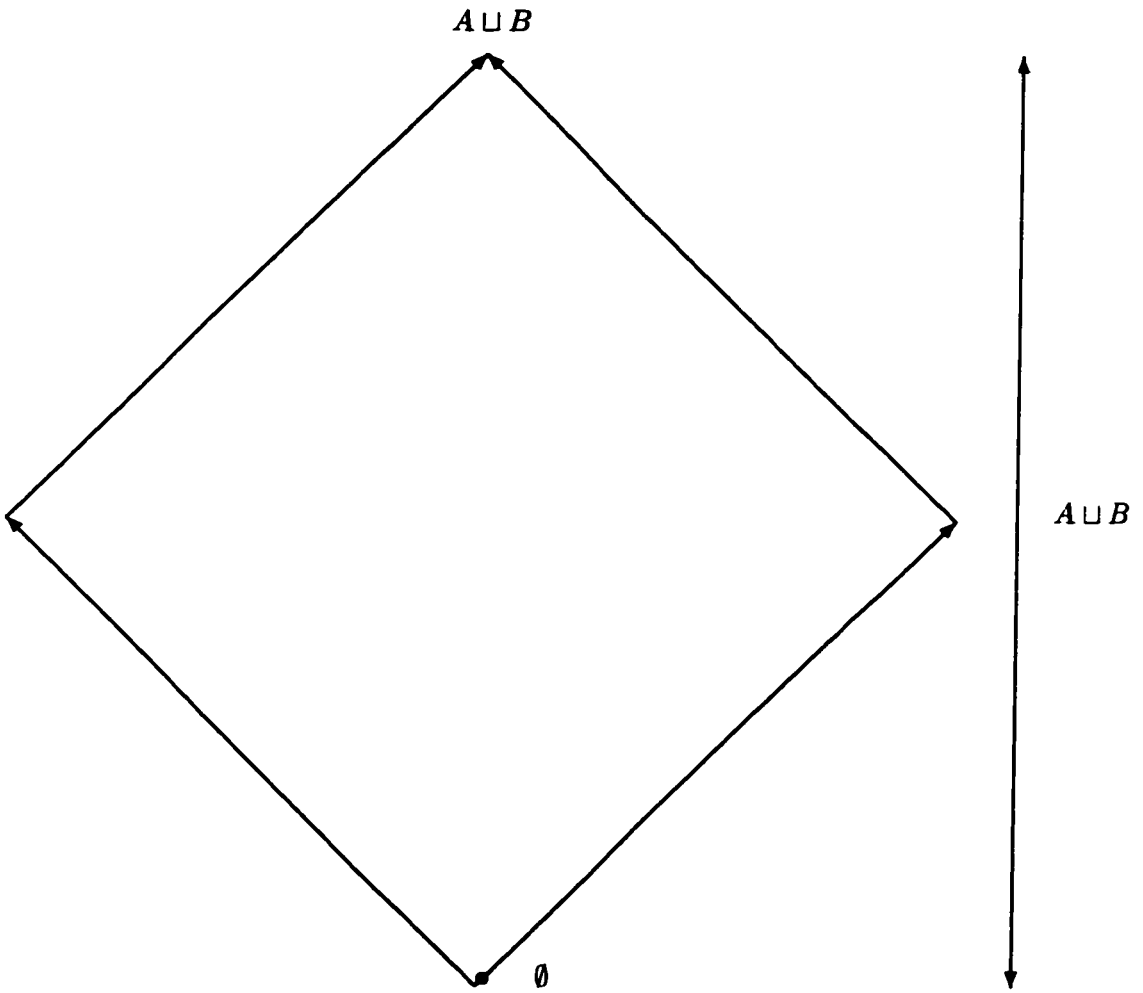

Figure 13.3: Refinement Distance of Prime Specifications

Proposition 13.6 *Given specifications A and B that satisfy the consistency condition, the refinement distance between A and B is defined, and is given by the following formula:*

$$A \otimes B = (A \sqcup B) \ominus (A \sqcap B).$$

Note that the refinement difference is defined since $A \sqcup B \sqsupseteq A \sqcap B$. Note also the analogy of this measure of distance with the distance between two real numbers: if we consider the set of real numbers with the $\leq$ ordering, then this set is a lattice, where the join is the max of its arguments and the meet is the min of its arguments. The distance between two numbers a and b (which is the absolute value of their difference) is given by the following formula (irrespective of their order: $a \leq b$ or $b \leq a$), where $\sqcup$ represents the max and $\sqcap$ represents the min:

$$|a - b| = (a \sqcup b) - (a \sqcap b).$$

Figure 13.4 gives a graphic illustration of the refinement distance, and shows why this is a natural definition of functional distance. This figure shows how $A \otimes B$ increases as A and B grow apart.

13.2.3 Illustrative Examples

Example. We consider the space $S =$ **real** and we let A and B be the following relations:

$$A = \{(s, s') | s < 0\} \cup \{(s, s') | s = s'^2\}$$
$$B = \{(s, s') | s = s'^2 \wedge s' \geq 0\}.$$

The first order of business is to compute their meet, which we denote by M. We find,

$$\begin{aligned}
& M \\
= & \quad \{ \text{ notation } \} \\
& A \sqcap B \\
= & \quad \{ \text{ definition } \} \\
& AL \sqcap BL \sqcap (A \cup B) \\
= & \quad \{ \ AL = L \ \} \\
& BL \sqcap (A \cup B) \\
= & \quad \{ \text{ distributivity, } BL \sqcap B = B \ \} \\
& BL \sqcap A \cup B \\
= & \quad \{ \ BL = \{(s, s') | s \geq 0\} \ \} \\
& \{(s, s') | s = s'^2\} \cup \{(s, s') | s = s'^2 \wedge s' \geq 0\} \\
= & \quad \{ \text{ second term is a subset of the first } \} \\
& \{(s, s') | s = s'^2\}.
\end{aligned}$$

Now we compute $A \ominus M$ and $B \ominus M$. We find

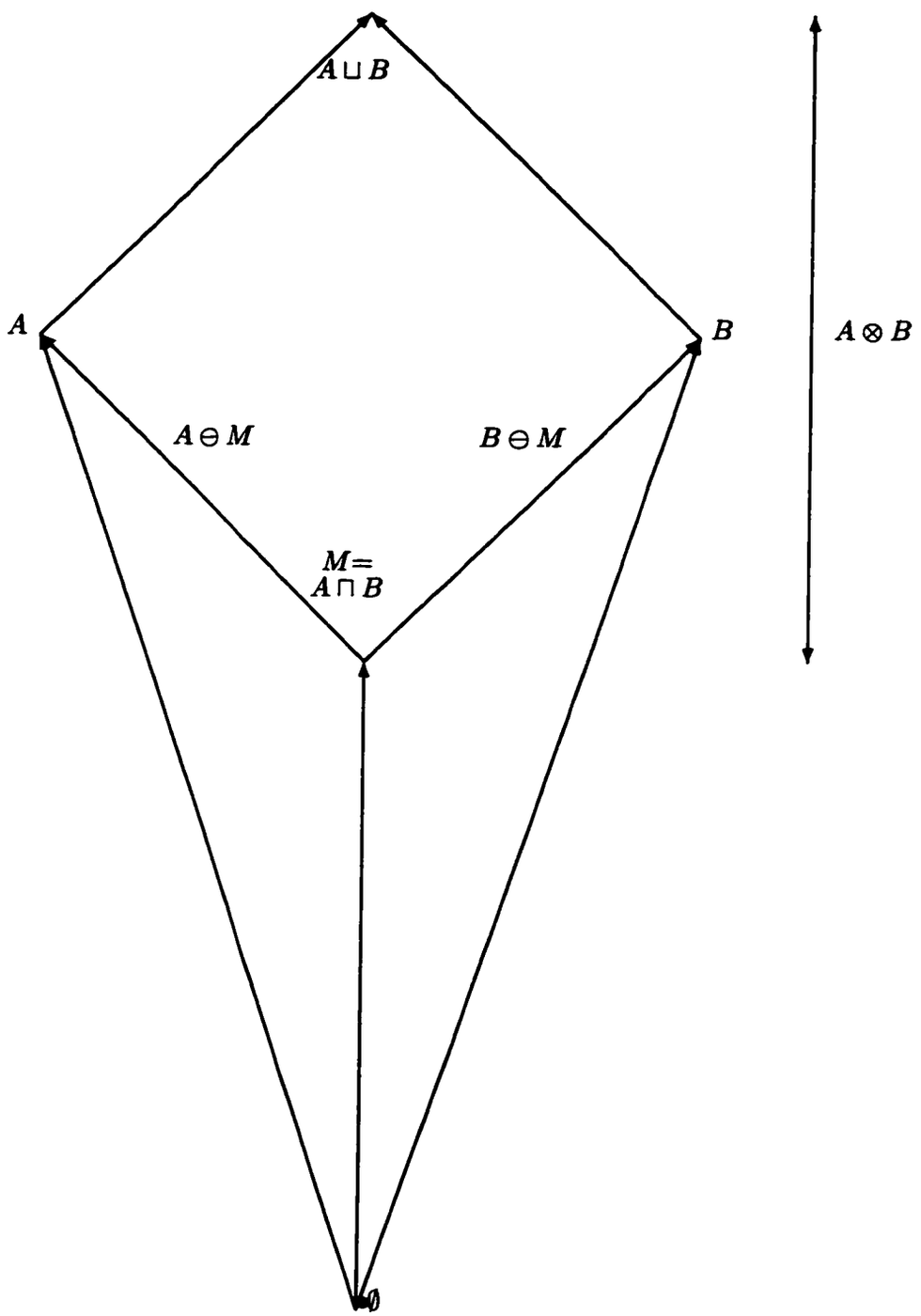

Figure 13.4: Refinement Distance of Arbitrary Specifications

$$
\begin{array}{ll}
& A \ominus M \\
= & \quad \{ \text{ first example, subsection 13.2.3 } \} \\
& \{(s, s') | s < 0\}.
\end{array}
$$

On the other hand,

$$
\begin{array}{ll}
& B \ominus M \\
= & \quad \{ \text{ second example, subsection 13.2.3 } \} \\
& \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\}.
\end{array}
$$

The symmetric refinement difference between A and B is computed as follows:

$$
\begin{array}{ll}
& A \otimes B \\
= & \quad \{ \text{ definition } \} \\
& (A \ominus M) \sqcup (B \ominus M) \\
= & \quad \{ \text{ substitution } \} \\
& \{(s, s') | s < 0\} \sqcup \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\} \\
= & \quad \{ \text{ disjoint domains } \} \\
& \{(s, s') | s < 0\} \cup \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\}.
\end{array}
$$

This relation captures the information that A has but B does not have, as well as the information that B has but A does not have. As such, it reflects the refinement distance between A and B. □

13.3 Refinement Ratio

13.3.1 Definition

The foregoing discussions enable us now to define a measure of functional distance between two specifications A and B (that are not necessarily ordered by the refinement ordering, but do satisfy the consistency condition). In order to understand the definition of our measure of functional distance, one must place it in the context of software reuse: We consider, e.g., that A is a user query and that B is the specification of a component that is available in a reuse library. We are interested in assessing the functional distance between A and B in order to determine whether it is worthwhile to modify B to satisfy A; the alternative to modifying B to satisfy A is merely to write a program for A from scratch. The option of modifying B is all the more attractive that A and B have more common functional information (more code of B can be reused), and less refinement distance (less code of B must be modified).

Given that common functional information between A and B is reflected by the *meet* of A and B, and given that the amount of functional features that set A and B apart is reflected by the refinement distance between A and B, we naturally define the refinement

ratio between A and B as follows.

Definition 13.5 *Given two arbitrary relations A and B, the* refinement ratio *between A and B is the relational vector denoted by $\phi(A, B)$ and defined by*

$$\phi(A, B) = \begin{pmatrix} A \otimes B \\ A \sqcap B \end{pmatrix}.$$

When a distance is measured by a positive number (as is typical for distances in general) then it is implicit that the smaller the distance, the closer the objects whose distance we are measuring. Because our difference is a vector, and because its entries are partially ordered, we must provide an additional definition to clarify the property of proximity.

Definition 13.6 *Given specifications A, B and A', B', we say that the distance between A and B is* shorter *than the distance between A' and B' if and only if*

$$(A \otimes B \sqsubseteq A' \otimes B') \wedge (A' \sqcap B' \sqsubseteq A \sqcap B).$$

To reflect this property, we may refer to the first entry of the distance $\phi(A, B)$ (which is $A \otimes B$) as the *numerator* of the refinement ratio, and we refer to the second entry (which is $A \sqcap B$) as the *denominator* of the refinement ratio.

We may denote this property by the notation $\phi(A, B) \leq \phi(A', B')$, even though ϕ is not a numeric-valued function. As a special case of this definition, we consider the situation where $A' = A$.

Definition 13.7 *Given specifications A, B and B', we say that A is* functionally closer *to B than to B' if and only if the refinement ratio between A and B is shorter than the refinement ratio between A and B'.*

To give some intuition for these definitions, we consider again the analogy of our refinement lattice with the *divides* lattice over the set of natural numbers.

Example. We consider natural numbers $a = 210$ and $b = 330$ and we show their respective decompositions as products of prime numbers. We find,

$$a = 2 \times 3 \times 5 \times 7,$$
$$b = 2 \times 3 \times 5 \times 11.$$

Our definition of distance provides that the distance between a and b is given by the vector

$$\phi(a,b) = \begin{pmatrix} 77 \\ 30 \end{pmatrix}.$$

Also, we find that the distance between c and d (defined below) is shorter than the distance between a and b.

$$c = 2 \times 3 \times 5 \times 17 \times 7 = 3570,$$
$$d = 2 \times 3 \times 5 \times 17 \times 11 = 5610,$$

because the distance between c and d is given by

$$\phi(c,d) = \begin{pmatrix} 77 \\ 510 \end{pmatrix},$$

and 30 does divide 510. Finally, we find that b is closer to a than b' that is defined as follows:

$$b' = 2^2 \times 3 \times 5 \times 11.$$

because the distance between a and b' is given by

$$\phi(a,b') = \begin{pmatrix} 154 \\ 30 \end{pmatrix},$$

and 77 divides 154. □

We conclude this subsection with a few simple but instructive propositions, which we give without proof (their proofs are straightforward applications of definitions).

Proposition 13.7 *The distance between A and $\emptyset$, for an arbitrary relation A, is*

$$\phi(A,\emptyset) = \begin{pmatrix} A \\ \emptyset \end{pmatrix}.$$

Proposition 13.8 *The distance between A and A, for an arbitrary relation A, is*

$$\phi(A,A) = \begin{pmatrix} \emptyset \\ A \end{pmatrix}.$$

13.3.2 Illustrative Examples

Example. We consider the space $S =$ **real** and we let A and B be the following relations:

$$A = \{(s,s') | s < 0\} \cup \{(s,s') | s = s'^2\}$$
$$B = \{(s,s') | s = s'^2 \wedge s' \geq 0\}.$$

In the example of section 13.2.3, we had found that the meet of A and B is

$$A \sqcap B = \{(s, s') | s = s'^2\}.$$

Further, we had found that the refinement distance between A and B is:

$$A \otimes B = \{(s, s') | s < 0\} \cup \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\}.$$

We can rewrite this as:

$$A \otimes B = \{(s, s') | s \geq 0 \Rightarrow (s \neq s'^2 \vee s' \geq 0)\}.$$

Whence we find:

$$\phi(A, B) = \begin{pmatrix} \{(s, s') | s \geq 0 \Rightarrow (s \neq s'^2 \vee s' \geq 0)\} \\ \{(s, s') | s = s'^2\} \end{pmatrix}.$$

Also, if we take relation A' defined by:

$$A' = \{(s, s') | s < 0 \wedge s' = -1\} \cup \{(s, s') | s = s'^2\}$$

we find that A is closer to B than A', by virtue of the following development.

$$\begin{array}{ll} & A' \sqcap B \\ = & \quad \{ \text{ similar development to } A \sqcap B \ \} \\ & \{(s, s') | s = s'^2\} \\ = & \quad \{ \text{ inspection } \} \\ & A \sqcap B. \end{array}$$

On the other hand,

$$\begin{array}{ll} & A \otimes B \\ = & \quad \{ \text{ developments above } \} \\ & \{(s, s') | s < 0\} \cup \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\} \\ \sqsubseteq & \quad \{ \text{ definition, properties of refinement } \} \\ & \{(s, s') | s < 0 \wedge s' = -1\} \cup \{(s, s') | s \geq 0 \wedge (s \neq s'^2 \vee s' \geq 0)\} \\ = & \quad \{ \text{ similar developments to } A \otimes B \ \} \\ & A' \otimes B. \end{array}$$

□

13.4 Concluding Remarks

In this chapter, we have introduced a number of functional (semantic) measures of distance between specifications; because components can be represented by their specifications, these measures of distance can be used to measure the distance between two specifications or the distance between a specification and a component. The measures are semantic, in the sense that they reflect the semantic properties of the specifications at hand and are independent of the way in which specifications are represented. We discussed three different measures, which we review briefly below:

- *Refinement Difference.* This measure is applicable whenever the specifications are ordered, e.g., A is refined by B, and reflects the amount of functionality that must be added (by the join) to A to obtain B. This measure of distance decreases as the components grow closer, but (unlike traditional distances) it is not symmetric.

- *Refinement Distance.* This measure is applicable even when the specifications are not ordered, but only when they satisfy the consistency condition; this measure reflects the amount of functional information that A has and B does not have, as well as the amount of functional information that B has and A does not have. This measure of distance satisfies all the axioms of distance (is symmetric, equals the minimal element if and only if the arguments are identical, satisfies the triangle inequality), but (unlike traditional distances) it is not defined for any pair of specifications, and takes its values in a partially ordered set.

- *Refinement Ratio.* This measure is applicable to any pair of specifications that satisfy the consistency condition, and reflects the functional information that the components have in common as well as the functional information that sets them apart. It is structured as a fraction, which increases whenever its numerator increases or its denominator decreases.

In the next chapter we complement this study by an investigation of structural measures of distance; while functional measures of distance deal with the semantics of the specifications at hand, structural measures deal with their syntax and their representation.

Chapter 14

A Measure of Structural Distance

14.1 Functional vs Structural Distance

In the previous chapter we had defined and discussed a measure of *functional distance* (also called *semantic distance*) which reflects the amount of information that two specifications have in common, and matches it against the amount of information that discriminates between these specifications. It practice, it may be useful to assess the functional distance that separates two specifications in the context of software adaptation: Given a user query K and an available component C, we consider it worthwhile to modify C to satisfy K only if C has much information in common with K (as reflected by $C \sqcap K$) and little discriminating information with K (as reflected by $C \otimes K$); in other words, it is worthwhile to modify C to satisfy K only if the functional distance $\phi(C, K)$ is small (small numerator, large denominator).

While it is intuitively appealing, the notion of functional distance cannot reliably be used to predict the modification effort, because the modification effort is not necessarily a linear function of the functional distance —as we illustrate in the simple example below.

Example. We consider the following program on variables x, y, z of type **integer**, which we call A (by abuse of notation we use the same name for the program and for the relation that it defines).

```
A =
begin
z:=0;
while y<>0 do
   begin
   y:= y-1;
   z:=z+x
   end
end;
```

The relation computed by this program is given as follows:

$$A = \{(s, s') | y \geq 0 \wedge x(s') = x(s) \wedge y(s') = 0 \wedge z(s') = z(s) + x(s) \times y(s)\}.$$

In order to show that functional distance is not proportionate with modification effort, we derive two versions of A: a version B that is derived from A by a minor modification; and a version C that is derived from A by means of a major modification. Then we show that C is functionally closer to A than B even though, structurally, B is closer to A than C.

```
B =
begin
z:=0;
while y>0 do
   begin
   y:=y-1;
   z:=z+x
   end
end;
```

```
C =
begin
z:=0;
if x=0 then y:=0;
while y<>0 do
   begin
   if even(y) then
      begin
      y:=y div 2;
      x:=2*x
      end
   else
      begin
      y:=y-1;
      z:=z+x
      end
   end
end;
```

Without going into the details (which we leave to the interested reader), we find the following formulae for B and C:

$$B = \{(s, s') | s < 0 \wedge s' = s\} \cup A,$$

$$C = A.$$

Note that because the two terms in the formula of B have disjoint domains, their union is the join; hence we rewrite B as follows:

$$B = \{(s, s') | s < 0 \wedge s' = s\} \sqcup A,$$

Again, without going into the details (which we leave to the interested reader), we find the following formula for the distance between A and B:

$$\phi(A,B) = \begin{pmatrix} \{(s,s') | s < 0 \wedge s' = s\} \\ A \end{pmatrix}.$$

As for the distance between A and C we find, trivially,

$$\phi(A,C) = \begin{pmatrix} \emptyset \\ A \end{pmatrix}.$$

Clearly, we do have

$$\phi(A,C) \leq \phi(A,B),$$

hence C is functionally closer to A than B —even though it took a trivial modification of A to obtain B and an extensive modification of A to produce C. □

This example illustrates briefly the orthogonality between functional distance and structural distance (which reflects the modification effort): we can produce a large functional distance with a minor modification (example: B) and a small functional distance with a profound modification (example: C). In light of this observation, one may argue that functional distance is immaterial since it cannot reliably predict the modification effort. Our position can be defined by the following premises.

- While structural distance can be defined to reflect the modification effort between a user query and an available software component, it cannot be used to predict modification effort. In order to determine the modification effort one has to perform the modification, but we are precisely interested in estimating the modification effort prior to the modification process, to determine whether the process is worthwhile/cost-effective.

- By contrast, functional distance can be estimated by inspection of the specifications at hand (rather than the modification process), hence can be measured upstream of the modification process (thereby helping to decide whether the modification is cost-effective/worthwhile). While modification effort is not proportionate to functional distance, we do expect it nevertheless to be somewhat correlated.

- Both structural distance and functional distance are required to complete our understanding of the issues involved in software modification, and both are needed to define such notions as modification efficiency, and modifiability (see chapter 15).

For a long term perspective, we expect to conduct further experimental research to establish correlations between functional distance and structural distance, so that when we derive the functional distance by inspection of the specifications at hand, this gives us some information about the structural distance that we expect to cover to perform the modification at hand (i.e., the effort required to carry out the modification).

14.2 Definition of Structural Distance

14.2.1 Modification Effort

Given a user query K and an available software component C, we are interested in defining a measure of distance that reflects the effort it takes to modify component C to satisfy specification K. For the sake of argument, we assume that the modification of C to satisfy K proceeds according to the prescriptions of chapter 12. The following rules provide the necessary equations to compute the structure distance.

Refinement Effort Rule 1 Refinement Step*: A specification of the form $K \sqcup K$ is refined by K.*
Effort Estimation Equation*:*

$$\eta(K, K) = 1 + \rho(K).$$

This rule is applied whenever we must refine the specification $K \sqcup C$ as part of the modification of C to satisfy K: If we find that $K = C$, then this join is refined into C. We propose to count the effort involved in this step as one (1), to which we add whatever costs are associated to the use of C to satisfy K. This cost, which we denote by $\rho(K)$ includes such factors as the cost of instantiation, parameter tuning, compiling, linking, etc.

Refinement Effort Rule 2 Refinement Step*: Given a space S with variables x_i, $1 \leq i \leq N$, and given an expression $E(s)$ on S, the specification*

$$\{(s, s') | x_1(s') = E(s) \wedge \forall i > 1 : x_i(s') = x_i(s)\}$$

refines and is refined by

$$x := E.$$

Effort Estimation Equation*:*

$$\eta(\{(s, s') | x_1(s') = E(s) \wedge \forall i > 1 : x_i(s') = x_i(s)\}, x := E) = 1.$$

This rule is applied whenever we must refine the specification

$$\{(s, s') | x_1(s') = E(s) \wedge \forall i > 1 : x_i(s') = x_i(s)\} \sqcup x := E$$

as part of the refinement of $C = (x := E)$ to satisfy

$$K = \{(s, s') | x_1(s') = E(s) \wedge \forall i > 1 : x_i(s') = x_i(s)\}.$$

Presumably, this rule is followed by application of rule 1, which reduces the specification to the assignment statement

$$x := E.$$

We consider that application of rule 2 takes a unit effort.

Refinement Effort Rule 3 Refinement Step*: If it is found that C does not help in deriving a solution to K, then a solution to K must be developed from scratch.*
Effort Estimation Equation*:*

$$\eta(K, C) = 1 + \gamma(K),$$

where $\gamma(K)$ is the cost of developing a solution to K from scratch.

Clearly, if C does not help in deriving a solution to K then the effort of adapting C to K is the effort of developing a solution to K, to which we add one (1) to account for the effort it takes to establish that C does not help to solve K. Unlike the other effort estimation rules, this rule is informal, because it depends on an informal judgment of whether C can or cannot be used to derive a solution to K.

Refinement Effort Rule 4 Refinement Step*: If specifications K and K' satisfy the condition $K \cap K'L \subseteq K'$ then their join is defined; further this join refines and is refined by*

$$\textbf{if } t \textbf{ then } K \textbf{ else } K',$$

where $t(s) \equiv s \in dom(K)$.
Effort Estimation Equation*:*

$$\begin{aligned}&\eta((K \sqcup K'), (\textbf{ if } t \textbf{ then } C \textbf{ else } C'))\\ &= 1 + \eta((\textbf{ if } t \textbf{ then } K \textbf{ else } K'), (\textbf{ if } t \textbf{ then } C \textbf{ else } C')).\end{aligned}$$

This rule is applied to give specification $K \sqcup K'$ the same structure as the component to be modified; we consider that this step takes a unit effort, followed by the effort it takes to match the two specifications component by component.

Refinement Effort Rule 5 Refinement Step*: When a specification of the form ((*$\textbf{if } t \textbf{ then } K)^* \sqcup \textbf{est}(\neg t)$*) satisfies the following conditions*

- *$t \rightarrow K$ is progressively finite.*
- *$t \subseteq KL$,*

then the join is defined, and the specification refines and is refined by $\textbf{while } t \textbf{ do } K$.
Effort Estimation Equation*:*

$$\begin{aligned}&\eta((\textbf{ if } t \textbf{ then } K)^* \sqcup \textbf{est}(\neg t),\ \textbf{while } t \textbf{ do } R')\\ &= 3 + \eta(\textbf{while } t \textbf{ do } K, \textbf{while } t \textbf{ do } R').\end{aligned}$$

This rule is applied when a join of two specifications with the appropriate forms can be equated to a while statement; because of the verification conditions involved in this rule, we take its required effort to be 3, plus the subsequent effort it takes to match the two while loops.

Refinement Effort Rule 6 Refinement Step: *Specification* $(K \circ K') \sqcup (C \circ C')$ *is refined by* $(K \sqcup C) \circ (K' \sqcup C')$.
Effort Estimation Equation:

$$\eta(K \circ K', C \circ C') = 1 + \eta(K, C) + \eta(K', C').$$

This rule is applied when the specification is given the same structure as the component (in this case, a sequence structure), and the join is propagated inside this structure.

Refinement Effort Rule 7 Refinement Step: *Specification* $(K||K') \sqcup (C||C')$ *is refined by* $(K \sqcup C)||(K' \sqcup C')$.
Effort Estimation Equation:

$$\eta(K||K', C||C') = 1 + \eta(K, C) + \eta(K', C').$$

This rule is applied when the specification is given the same structure as the component (in this case, a parallel structure), and the join is propagated inside this structure.

Refinement Effort Rule 8 Refinement Step: *Specification* $(K \sqcap K') \sqcup (C \sqcap C')$ *is refined by* $(K \sqcup C) \sqcap (K' \sqcup C')$.
Effort Estimation Equation:

$$\eta(K \sqcap K', C \sqcap C') = 1 + \eta(K, C) + \eta(K', C').$$

This rule is applied when the specification is given the same structure as the component (in this case, a parallel structure), and the join is propagated inside this structure.

Refinement Effort Rule 9 Refinement Step: *Specification* $K^* \sqcup C^*$ *is refined by* $(K \sqcup C)^*$.
Effort Estimation Equation:

$$\eta(K^*, C^*) = 1 + \eta(K, C).$$

This rule is applied when the specification is written as a closure (to match the structure of the program) and the join is propagated inside the structure.

Refinement Effort Rule 10 Refinement Step: *Specification* $t \rightarrow K \sqcup t \rightarrow C$ *is refined by* $t \rightarrow (K \sqcup C)$.
Effort Estimation Equation:

$$\eta(t \rightarrow K, t \rightarrow C) = 1 + \eta(K, C).$$

This rule is used when a join is propagated inside a prerestriction.

Refinement Effort Rule 11 Refinement Step: *Specification* **var** $x : T$ $K \sqcup$ **var** $x : T$ C *is refined by* **var** $x : T$ $K \sqcup C$.
Effort Estimation Equation:

$$\eta(\textbf{var } x : TK, \textbf{var } x : TC) = 1 + \eta(K, C).$$

This rule is applied when a join is propagated inside a new variable scope.

Refinement Effort Rule 12 Refinement Step: *Specification* **if** t **then** K **else** $K' \sqcup$ **if** t **then** C **else** C' *is refined by* **if** t **then** $K \sqcup C$ **else** $K' \sqcup C'$.
Effort Estimation Equation:

$$\eta(\textbf{ if } t \textbf{ then } K \textbf{ else } K', \textbf{ if } t \textbf{ then } C \textbf{ else } C') = 1 + \eta(K, C) + \eta(K', C').$$

This rule is applied when the join is propagated inside an alternation structure.

Refinement Effort Rule 13 Refinement Step: *Specification* **while** t **do** $K \sqcup$ **while** t **do** C *is refined by* **while** t **do** $K \sqcup C$.
Effort Estimation Equation:

$$\eta(\textbf{ while } t \textbf{ do } K, \textbf{while } \ t \textbf{ do } C) = 1 + \eta(K, C).$$

This rule is applied when the join is propagated inside a while statement structure.

Refinement Effort Rule 14 Refinement Step: *Specification* $\textbf{est}(t)$ *is refined by*

$$\textbf{est}(t) \,\square\, \textbf{prs}(t).$$

Effort Estimation Equation:

$$\eta(\textbf{est}(t), C) = 1 + \eta(\textbf{est}(t) \,\square\, \textbf{prs}(t), C).$$

This rule is applied whenever, in order to match a sequence structure of C, we decompose K as a sequence structure.

Refinement Effort Rule 15 Refinement Step: *Specification* $\textbf{est}(t)$ *is refined by*

$$\textbf{est}(\textbf{true}) \,\square\, \textbf{est}(t).$$

Effort Estimation Equation:

$$\eta(\textbf{est}(t), C) = 1 + \eta(\textbf{est}(\textbf{true}) \,\square\, \textbf{est}(t), C).$$

This rule is applied whenever, in order to match a sequence structure of C, we decompose K as a sequence structure.

Refinement Effort Rule 16 Refinement Step: *Specification* K *is refined by* $\kappa(K) \,\square\, K$.
Effort Estimation Equation:

$$\eta(K, C) = 1 + \eta(\kappa(K) \,\square\, K, C).$$

This rule is applied whenever, in order to match a sequence structure of C, we decompose K as a sequence structure.

Refinement Effort Rule 17 Refinement Step: *Specification K is refined by $K \square \Gamma(K)$.*
Effort Estimation Equation:

$$\eta(K, C) = 1 + \eta(K \square \Gamma(K), C).$$

This rule is applied whenever, in order to match a sequence structure of C, we decompose K as a sequence structure.

Refinement Effort Rule 18 Refinement Step: *Specification* $\mathbf{prs}(t)$ *is refined by* $\mathbf{prs}(t)^*$.
Effort Estimation Equation:

$$\eta(\mathbf{prs}(t), C) = 1 + \eta(\mathbf{prs}(t)^*, C).$$

This rule is applied whenever, in order to match the closure structure of C, we structure K as a closure.

Refinement Effort Rule 19 Refinement Step: *Specification $\kappa(K)$ is refined by $\kappa(K)^*$.*
Effort Estimation Equation:

$$\eta(\kappa(K), C) = 1 + \eta(\kappa(K)^*, C).$$

This rule is applied whenever, in order to match the closure structure of C, we structure K as a closure.

Refinement Effort Rule 20 Refinement Step: *Specification K^* is refined by*

$$(\ \mathbf{if}\ t\ \mathbf{then}\ K)^*$$

for all predicate t.
Effort Estimation Equation:

$$\eta(K^*, C) = 1 + \eta((\ \mathbf{if}\ t\ \mathbf{then}\ K)^*, C).$$

This rule is applied whenever we wish to highlight a while statement structure in K to match a while statement in C.

Because these rules are not exhaustive, we may sometimes apply arbitrary refinements and count their associated effort as one.

14.2.2 Structural Distance

Using the rules of *modification effort*, we can define the notion of structural distance, as follows.

Definition 14.1 *Given specifications K and C, the* structural distance *from C to K is denoted by $\sigma(K, C)$ and defined by:*

$$\sigma(K, C) = \mathbf{Min}\{\eta(K, C)\},$$

where the minimum is taken over the set of all the possible refinements of $K \sqcup C$.

In other words, the structural distance from C to K is the smallest total modification effort that is required to adapt C to satisfy K. Unlike traditional distances, ours is not symmetric: we define the distance from C to K, where C is a component and K is a specification. It is possible that this measure can be generalized to reflect the distance between two arbitrary specifications; we do not consider this possibility for now.

According to this definition we cannot compute the structural distance from C to K unless we have inspected all the possible refinements of $K \sqcup C$. Because this is impractical, we will most often follow a refinement which we consider to be shortest, then either take the associated effort to be the structural distance, or simply consider that this effort is an upper bound to the structural distance.

14.3 Illustrative Examples

We consider the simple example where we are looking to satisfy the user query $K = \mathit{OptCompiler}$ for an optimizing compiler, and all we find in the reuse library is a plain compiler $C = \mathit{Compiler}$. According to chapter 12, the adaptation of C to K is a refinement of the specification $K \sqcup C$. Because the refinement steps are trivial, we will use the comments space to keep track of the refinement effort $\eta(K, C)$. We proceed as follows:

$$
\begin{array}{ll}
 & K \sqcup C \\
= & \quad \{\ \eta(K,C), \text{ by substitution } \} \\
 & \mathit{OptCompiler} \sqcup \mathit{Compiler} \\
\sqsubseteq & \quad \{\ =1+\eta(\mathit{OptCompiler}, \mathit{Compiler}), \text{ by refinement } \}
\end{array}
$$

We assume that *GenCompiler* is a generic specification for a compiler, whereas *Compiler* is the specific compiler that was retrieved. Consequently, we do assume that *Compiler* refines *GenCompiler*. We resume the refinement,

$$
\begin{array}{ll}
 & (\mathit{GenCompiler} \circ \mathit{Optimizer}) \sqcup \mathit{Compiler} \\
\sqsubseteq & \quad \{\ =2+\eta(\mathit{GenCompiler} \circ \mathit{Optimizer}, \mathit{Compiler} \circ \Gamma(\mathit{Compiler}))\ \} \\
 & (\mathit{GenCompiler} \circ \mathit{Optimizer}) \sqcup (\mathit{Compiler} \circ \Gamma(\mathit{Compiler})) \\
\sqsubseteq & \quad \{\ =3+\eta(\mathit{GenCompiler}, \mathit{Compiler}) + \eta(\mathit{Optimizer}, \Gamma(\mathit{Compiler}))\ \} \\
 & (\mathit{GenCompiler} \sqcup \mathit{Compiler}) \circ (\mathit{Optimizer} \sqcup \Gamma(\mathit{Compiler})) \\
\sqsubseteq & \quad \{\ =4+\eta(\mathit{Compiler}, \mathit{Compiler}) + \eta(\mathit{Optimizer}, \Gamma(\mathit{Compiler})), \text{ by refinement } \} \\
 & (\mathit{Compiler} \sqcup \mathit{Compiler}) \circ (\mathit{Optimizer} \sqcup \Gamma(\mathit{Compiler})) \\
\sqsubseteq & \quad \{\ =5+\eta(\mathit{Compiler}, \mathit{Compiler}) + \eta(\mathit{Optimizer}, \Gamma(\mathit{Compiler})), \text{ by rule 1 } \} \\
 & (\mathit{Compiler}) \circ (\mathit{Optimizer} \sqcup \Gamma(\mathit{Compiler}))
\end{array}
$$

We use $\rho(C)$ to represent the reuse effort associated with the available component C; this effort may include such costs as instantiation, assessment, integration, etc. We resume the refinement.

$$\sqsubseteq \quad \{ =6+\rho(Compiler)+\gamma(Optimizer), \text{ by rule 3 } \}$$
$$Compiler \,\square\, Optimizer.$$

Hence in order to satisfy specification *OptCompiler* it suffices to combine the existing component *Compiler* to an optimizer to be developed from scratch, for a total effort of

$$\eta(OptCompiler, Compiler) = 6 + \rho(Compiler) + \gamma(Optimizer).$$

Given that we have prospected a possible refinement of $Compiler \sqcup OptCompiler$, we can give the following lower bound for the structural distance between *Compiler* and *OptCompiler*, namely:

$$\sigma(OptCompiler, Compiler) \leq 6 + \rho(Compiler) + \gamma(Optimizer).$$

14.4 Concluding Remarks

In this chapter we have focused on defining a measure of structural (syntactic) distance between specifications. We have resolved to define structural distance in the narrow context of program modification, and have decided to represent the structural distance between a specification K and a component C as the effort it takes to modify C to satisfy K. For the sake of uniformity, we have chosen to represent this modification effort, not by person-months (as one would represent effort), but rather by means of the weighted number of refinement rules that are used in the program modification calculus that is presented in chapter 12. Our measure of structural distance can be characterized by the following premises.

- Unlike measures of functional distance, which take their values in the partially ordered set of specifications, structural distance takes numeric values, hence ranges over a totally ordered set.
- Unlike traditional measures of distance, which take their values in the set of real numbers, structural distance takes its values in the set of natural numbers.
- Whereas functional distance can be computed between two specifications, structural distance can only be defined between a specification and a component, and can only be interpreted in the context of program modification.
- Whereas functional distance is intrinsic to the specifications at hand, structural distance pertains to the representation of the specifications.
- Most reuse-related applications of distance rely on structural distance, because it reflects a concrete quantity, the modification effort; unlike functional distance measures, however, the measure of structural distance cannot be estimated prior to carrying out a program modification.

- Because functional measures of distance are intrinsic to the specifications at hand (rather than their representation), they can be computed by inspection of the specifications; because they are arbitrarily independent of structural distance, however, functional measures of distance cannot, in principle, be used to predict structural distance.

- In the absence of other alternatives, it is reasonable to use measures of functional distance as predictors of structural distance. The experiments we conducted in chapter 11 using the *meet* substantiates this claim.

In the next chapter we use this measure of structural distance as well as the measures of functional distance we introduced in chapter 13 to defines analytical measures of component modifiability.

Chapter 15

Component Modifiability

In chapters 13 and 14 we have defined two measures of proximity between specifications, namely *functional distance* and *structural distance*. In this chapter we discuss how these measures can be used to define and discuss *modifiability*. First, we recognize that there are several interpretations of component modifiability, which we review in turn, associating a distinct definition to each, by means of our measures of distance. These are: *modification worthiness* ; *modification efficiency* ; and *component modifiability* . Among all the measures of functional distance that we have introduced in chapter 13, we use a single measure, which synthesizes many of the properties of the other measures: the *refinement ratio*, which is the vector made up of the refinement distance and the *meet*. Because the refinement ratio is the only measure of functional distance that we use in this chapter, we will, for the sake of simplicity, refer to it merely as *functional distance* (to contrast it with structural distance).

15.1 Modification Worthiness

Given a software component C and a user query K we consider the question of whether it is worthwhile to modify C to satisfy K —the alternative being to write a solution to K from scratch and dispose of C. We consider two approaches to this question, one based on structural distance, the other based on functional distance.

15.1.1 Structural Measure of Worthiness

Whether or not the modification of C to satisfy K is worthwhile depends on two factors:

- The cost of developing a solution to K from scratch; this can be estimated by existing cost models, which use predictions of the size of K's programming solution [14] or observations on the functional complexity of specification K [1].
- The cost of modifying C to satisfy K; this cost cannot be estimated ahead of time, before the modification occurs. Rather, it can only be estimated as the modification takes place —which means that it cannot be used to decide whether to carry out the modification.

In principle, a modification is considered worthwhile only if the cost of the modification is less then the cost of development from scratch, i.e,

$$\eta(K, C) < \gamma(K).$$

In practice, to provide for risks involved in the development process and in the modification process, one could consider that a modification is worthwhile only if the modification effort $\eta(K, C)$ is less than or equal to a small ratio of the development effort $\gamma(K)$.

15.1.2 Functional Measure of Worthiness

The structural measure of modification worthiness suffers from two weaknesses: first, the development cost can only be estimated on the basis of an estimate of the size of the projected programming solution to K; it is not clear how such a size can be estimated with accuracy. Second, the (minimal) modification cost cannot be estimated prior to the modification; in addition, even if a modification does take place, it does not provide an estimate of the minimal modification effort —rather it only provides an upper bound of the minimal effort. Hence we turn to an alternative measure of worthiness, based on functional distance.

Given a user query K and an available component C, we consider that it is worthwhile to modify C to satisfy K when the following conditions hold:

- *Component C possesses many of the functional features of K.* The more features of K component C has, the less we have to modify C to satisfy K. Every feature of K that C has represents a savings: we will not have to write code into C to provide this feature. Note that the *meet* of K and C reflects common functional features between K and C.

- *Few functional features set specifications K and C apart.* Whenever K has a feature that C does not need, we have to write code into C to provide it. Also, whenever C has a feature that K does not, we have to abstract it away as we modify C; each such feature represents information that does not pertain with K, but which we must understand and work around. Note that the refinement distance between K and C reflects discriminating functional features between K and C.

In light of this discussion, it is clear that *functional distance* between K and C, which is the vector made up of the refinement distance between K and C and their meet, is an adequate representation of the modification worthiness of C to satisfy K.

Unlike the structural measure of worthiness discussed above, the functional measure of worthiness can be assessed before the modification takes place, and can in fact be used to decide whether the modification is worthwhile. This measure does, however, have a major drawback: it infers the modification effort from the functional distance, assuming in effect that modification effort is a *linear* function of functional distance; we have shown in chapter 14 that this is not always the case. Hence in the absence of other sources we can always use the functional distance as a predictor of modification effort; but we must remember that the correlation between these two quantities can be weak.

Example. In order to illustrate the notion of functional distance as reflecting modification worthiness, we consider the following example:

$$K = OptCompiler$$
$$C = Compiler.$$

The functional distance between $K = OptCompiler$ and $C = Compiler$ is given by:

$$\phi(OptCompiler, Compiler) = \begin{pmatrix} OptCompiler \otimes Compiler \\ OptCompiler \sqcap Compiler \end{pmatrix}.$$

Because *OptCompiler* is a refinement of *Compiler* (they have the same domain, former is a subset of latter), the meet of *OptCompiler* and *Compiler* is the latter. We focus our attention on the first entry of the distance vector:

$$\begin{aligned}
& OptCompiler \otimes Compiler \\
= & \quad \{ \text{ because } OptCompiler \text{ refines } Compiler \} \\
& OptCompiler \ominus Compiler \\
= & \quad \{ \text{ functional difference (chap. 13), } OC = OptCompiler,\ C = Compiler \} \\
& OC \cap \overline{CL} \cup (C \cap \overline{OC})L \cap (OC \cup \overline{C}) \\
= & \quad \{ \ dom(OC) = dom(C) = \{\text{Pascal Programs}\} \ \} \\
& (C \cap \overline{OC})L \cap (OC \cup \overline{C}) \\
= & \quad \{ \ (C \cap \overline{OC}) \text{ maps all programs into unoptimized code } \} \\
& (OC \cup \overline{C}).
\end{aligned}$$

Hence the functional distance between *OptCompiler* and *Compiler* is given by:

$$\phi(OptCompiler, Compiler) = \begin{pmatrix} OptCompiler \cup \overline{Compiler} \\ Compiler \end{pmatrix}.$$

Note that the denominator of this functional distance, *Compiler*, is very high in the refinement ordering (represents a great deal of functional detail). On the other hand, the numerator is very low: it maps source programs into correct optimized code or into arbitrary incorrect code. Because of the wide range of outputs for any source program, this numerator is fairly low in the refinement ordering. Because the numerator of $\phi(OptCompiler, Compiler)$ is low and its denominator is high, this functional distance is small, suggesting high modification worthiness. The illustrative example discussed in chapter 14 does confirm that it is worthwhile to modify *Compiler* to satisfy *OptCompiler*, the main additional cost being that of developing an optimizer. Note however that the derivation of functional distance does not recognize this fact, because it deals exclusively with functional details of *OptCompiler* and *Compiler*, not with their structural details.

□

15.2 Modification Efficiency

Given a user query K and a component C, we are interested in the efficiency with which component C can be adapted to satisfy specification K. To this effect, we need to refer to the functional distance between K and C ($\phi(K,C)$) as well as the structural distance between K and C ($\sigma(K,C)$).

Had $\phi(K,C)$ and $\sigma(K,C)$ both been defined in numeric terms, *modification efficiency* would have been defined merely as the ratio of $\phi(K,C)$ over $\sigma(K,C)$. This ratio would reflect the functional distance that is covered by unit of modification effort.

Because functional distance is not defined in numeric terms, we define modification efficiency by a partial order, using the following definition.

Definition 15.1 *Given a user query K and a software component C, we are considering to modify C to satisfy K. The associated* modification efficiency *is the vector denoted by $\epsilon(K,C)$ and defined by*

$$\epsilon(K,C) = \begin{pmatrix} \phi(K,C) \\ \sigma(K,C) \end{pmatrix}.$$

To complement this definition, we must stipulate under what condition we consider that a modification has a greater modification efficiency.

Definition 15.2 *We consider that modification efficiency $\epsilon(K,C)$ is greater than modification efficiency $\epsilon(K',C')$ (denoted, by abuse of notation, by $\epsilon(K,C) \geq \epsilon(K',C')$) if and only if $\phi(K,C) \geq \phi(K',C')$ and $\sigma(K,C) \leq \sigma(K',C')$.*

To reflect this definition, we refer to the first entry of $\epsilon(K,C)$ (i.e, $\phi(K,C)$) as the *numerator* of the modification efficiency, and we refer to the second entry of $\epsilon(K,C)$ (i.e, $\sigma(K,C)$) as the *denominator* of the modification efficiency.

Example. If we consider again the example of modifying a plain compiler *Compiler* to satisfy the specification of an optimized compiler *OptCompiler*, and use the results of this and previous chapters, we find the following results:

$$\phi(OptCompiler, Compiler) = \begin{pmatrix} OptCompiler \cup \overline{Compiler} \\ Compiler \end{pmatrix}.$$

$$\sigma(OptCompiler, Compiler) \leq 6 + \rho(Compiler) + \gamma(Optimizer).$$

Note that we did not give an exact estimate of the structural distance, but rather an upper bound of it: the figure we have on the right hand side is the modification effort

of the modification we have carried out in the example of chapter 14; it may or may not be the smallest modification effort (i.e, the structural distance). Consequently, what we give below is not an estimate of the modification efficiency; rather it is a lower bound of the modification efficiency (if the structural distance is actually shorter than we have observed, then the modification efficiency is higher). We find,

$$\epsilon(OptCompiler, Compiler) \geq \left(\begin{array}{c} \left(\begin{array}{c} OptCompiler \cup \overline{Compiler} \\ Compiler \end{array} \right) \\ 6 + \rho(Compiler) + \gamma(Optimizer) \end{array} \right).$$

□

15.3 Definition of Component Modifiability

The measure of modification efficiency, which we have discussed in the previous section, reflects properties of the two parameters involved, namely K (the user query) and C (the software component at hand). In this section we wish to define a measure of modification efficiency that depends solely on component C.

Had we defined functional distance in numeric terms, we would have defined the modifiability of component C as the derivative of functional distance over structural distance:

$$\frac{d\phi}{d\sigma}.$$

Such a quantity would reflect how much functional distance can be covered per unit of structural distance. Because functional distance is not a numeric value, we will not be able to use this definition; we will, however, approximate it.

15.3.1 Pointwise Modifiability

Given a software component C, we are interested in the modification efficiency of C to satisfy specification $C \sqcup \delta$, where δ is a *small* functional increment.

Definition 15.3 *The* pointwise modifiability *of component C with respect to increment δ is the modification efficiency $\epsilon(C \sqcup \delta, C)$. We denote it with $\pi_\delta(C)$.*

If we consider the formula for $\epsilon(K, C)$ for the special case when $K = C \sqcup \delta$, we find:

$$\pi_\delta(C) = \left(\begin{array}{c} \left(\begin{array}{c} (C \sqcup \delta) \ominus C \\ C \end{array} \right) \\ \sigma(C \sqcup \delta, C) \end{array} \right).$$

The formula proposed here stems from two observations: first, $C \sqcup \delta$ is a refinement of C, hence their refinement distance is the refinement difference of $C \sqcup \delta$ and C; second, for the same reason, the meet of $C \sqcup \delta$ and C is C.

In the special case when C and δ are prime, the pointwise modifiability of C with respect to δ takes the even simpler form:

$$\pi_\delta(C) = \begin{pmatrix} \begin{pmatrix} \delta \\ C \end{pmatrix} \\ \sigma(C \sqcup \delta, C) \end{pmatrix}.$$

The pointwise modifiability of component C increases as δ increases (for the same modification effort) or as $\sigma(C \sqcup \delta, C)$ decreases (for the same δ). In other words, the pointwise modification of C increases as it becomes less costly to achieve the increment of functionality δ.

15.3.2 Overall Modifiability

Pointwise modifiability deals with the modifiability of a component with respect to a specific elementary (infinitesimal) modification, represented by δ. Overall modifiability attempts to overcome the dependence on a specific functional increment. To this effect, we consider a set Δ that contains a number of elementary functional increments δ_i, and we define the modifiability of C as the vector of pointwise modifiabilities that we find with each δ_i. We can imagine, for example, that set Δ depends on the application domain of C and represents the main increments that we may wish to add to C.

Example. As a simple example of set Δ, imagine that C is a compiler. Then we wish to include the following increments in set Δ:

- The output is peephole optimized:

  $$\delta_0 = \{(s, c) | \mathit{peephole}(c)\},$$

  where s is a source program, c is an object code program, and predicate $\mathit{peephole}(c)$ means that c is peephole optimized.

- The output is globally optimized:

  $$\delta_1 = \{(s, c) | \mathit{global}(c)\}.$$

- The input may include object modules (ref: object Pascal):

  $$\delta_2 = \{(s, c) | \mathit{object}(s) \wedge \mathit{correct}(s, c)\}.$$

□

We now define overall modifiability, with respect to a vector of possible modifications.

Definition 15.4 *Given a component C and a domain specific set Δ of function increments,*

$$\Delta = \{\delta_0, \delta_1, .., \delta_k\}.$$

The overall modifiability *of component C is the vector denoted by $\mu(C)$ and defined by*

$$\mu(C) = \begin{pmatrix} \pi_{\delta_0}(C) \\ ... \\ \pi_{\delta_i}(C) \\ ... \\ \pi_{\delta_k}(C) \end{pmatrix}.$$

As usual, we do not content ourselves with defining a measure; we also discuss the ordering that this measure implies.

Definition 15.5 *Component C is said to be* more modifiable *than component C' if and only if:*

$$\forall \delta_i \in \Delta : \pi_{\delta_i}(C) \geq \pi_{\delta_i}(C').$$

This definition presents a weakness: very few pairs (C, C') may satisfy this strong requirement. Weaker comparison criteria may be defined, such as:

- There exists at least one element δ in Δ for which $\pi_\delta(C) \geq \pi_\delta(C')$ and there is no element δ in Δ for which $\pi_\delta(C) < \pi_\delta(C')$.
- The number of elements δ in Δ for which $\pi_\delta(C) > \pi_\delta(C')$ greater than the number of elements δ in Δ for which $\pi_\delta(C) < \pi_\delta(C')$.
- Define the sum of pointwise modifiabilities, and compare sums of pointwise modifiabilities to determine which component is most modifiable.

By weakening the criterion for inequality, we make the ordering more meaningful (if very few pairs are ordered, we do not have much of an ordering).

15.4 Concluding Remarks

In this chapter we have discussed means to use the measures of functional distance (introduced in chapter 13) and structural distance (introduced in chapter 14) as a basis for defining measures of modifiability. To this effect, we proceed in a stepwise fashion, as follows:

- *Modification Worthiness.* We consider a component C and a specification K and ponder the worthiness of modifying C to satisfy K by comparing the modification cost (reflected in the structural distance) and the cost of developing a solution to K from scratch. To this effect, we discuss two approaches to modification worthiness: one based on structural distance and one based on functional distance, and show the superiority of the latter.

- *Modification Efficiency.* This measure takes a component C and a specification K and produces a quantity that reflects the cost effectiveness of modifying C to satisfy K. To derive a measure of cost effectiveness, we introduce a measure of cost (by means of the structural distance, which reflects modification effort) and a measure of benefit (by means of the functional distance between C and K, which represents the gain in functionality that is achieved by the modification).

- *Component Modifiability.* Modification efficiency involves two factors: the component at hand, C, and the specification that the component must be modified to satisfy, K. In order to derive a measure of modifiability which is intrinsic to the component, we consider infinitesimal modifications originating in C and observe their modification efficiency. To this effect, we consider a set of small functional increments that one may want to add to C, and estimate their associated modification efficiency. From these quantities, we can derive a measure of modifiability of the component: a component C is more modifiable than a component C' if it exhibits higher modification efficiency on the set of functional increments. We discuss the significance of this set of functional increments, and find that in practice it makes sense to define such a set in a domain-specific manner: For a given application domain, one can think of a set of functional increments that one may want to add to an application.

Part VI

Conclusion

Summary, Assessment and Prospects

Summary

This thesis deals with measuring the reusability of a component; we take two distinct approaches to this problem. The first approach, which is elaborated in parts II and III, is empirical and relies on a return on investment model. The second approach, which is elaborated in parts IV and V, is analytical and relies on capturing reuse activities by means of formal specifications. In this section, we briefly review the two approaches and discuss our contributions.

Empirical Approach

Traditionally, empirical approaches to measuring reusability proceed by identifying software features that promote reuse, associating metrics to reflect these features, and using the metrics to quantify reusability and compare reusable components. There are two weaknesses in this approach: First, the link between the features and the measure of reusability is not formally established; second, some of the features may be unquantifiable or may be imperfectly reflected by the associated metrics. To address these weaknesses, we have taken an approach that can be characterized by the following premises: first, the measure of reusability is related to specific features of the software component by means of a mathematical cost-benefit formula; second, the features that appear in the formula can all be quantified.

Among our main findings in deriving the empirical measure of reusability, we mention:

- **Definition of a component-level ROI model**, that reflect the cost benefit analysis of storing a component for software reuse. Given a reuse library, and a software component (say C), we view the decision to store component C in the library as an investment-like decision, where the upfront investment costs include the cost of validating and packaging the component, and the long term periodic costs include the overhead of maintaining the component in the library. In return, the long term benefits of storing the component include gains in programmer productivity and program quality. On the basis of this view, we define the reusability of a software component as the return on investment associated with the option of storing the component in the reuse library. In deriving this measure, we equate all the costs

and benefits (including quality benefits) in terms of man months, and compute the reusability value using existing software cost models as well as empirical data on software reuse. We view this measure as a means to quantify the decision of whether a component should be or should not be stored in the reuse library.

- **Deriving auxiliary reusability measures**, such as long term reusability, intrinsic reusability, long term intrinsic reusability, break even frequency, and long term break even frequency. Long term reusability is the reusability of a component over an indefinite number of years. Intrinsic reusability is the derivative of the reusability measure with respect to frequency; hence, it reflects exclusively the features of the component and can be used to compare components in case we have no information on their frequency of reuse. Break-even frequency is the frequency for which the reusability takes a special value; the interest of break-even frequency is that we no longer ask the question "what is the expected reuse frequency of this component?"; rather we ask "do we expect the reuse frequency of this component to exceed this break-even value?". Long term intrinsic reusability and long term break even frequency are the values of intrinsic reusability and break even frequency over an indefinite period of time.

- **A detailed investigation of all the cost factors that intervene in the reusability measures**, and a discussion of means to compute them in practice. To estimate the cost factors, we use a number of existing cost models as well as existing statistical data from controlled experiments.

- **Development of a CASE tool** that assists in collecting appropriate information and computes the different reusability measures. This tool elicits information about the components at hand, the software library where the component is being introduced, as well as the expected use of the component and produces estimates of the various reusability measures of the component. Subsequent versions of this tool are designed to make provisions for incomplete information, to calibrate cost models to an organization, and to maintain organization-wide reusability information.

Analytical Approach

Our experience with the empirical approach to software reusability leads us to the observation that the reusability of a component (as defined by the return on investment model) is dependent on two families of factors: intrinsic factors, that deal with qualities of the component; and environmental factors, that deal with features of the environment in which the component is reused. Intrinsic factors include: generality, encapsulation, abstraction, which promote the likelihood that the component matches a given query (black box reuse); modularity, understandability, structure, which promote the ease of modification of the component (white box reuse). It is possible to define these features, represent them and reason about them provided we have a mathematical foundation that captures reuse related processes and products.

The purpose of parts IV and V is to provide such a foundation; among the main contributions of these parts, we mention the following:

- **A specification model** that can be used to specify reusable components and queries of reuse libraries.

- **The implementation of a reuse library** that features perfect retrieval precision (all retrieved components are relevant) and arbitrarily good retrieval recall (ratio of retrieved components over relevant components). The retrieval recall can be improved arbitrarily by inserting a larger number of specifications (hence discriminating more and more among components).

- **The definition of a calculus of program modifications**, whose purpose is to guide the modification of a component C to satisfy a query K in a stepwise fashion.

- **The definition of measures of functional distance between specifications.** We have defined several measures of semantic distance between specifications and have investigated their use for approximate retrieval. Generally speaking, these measures reflect the amount of functionality that two specifications have in common as well as the amount of functionality that discriminates between two specifications. These measures of distance can be used in practice to identify, among a set of components in a reuse library, those that are closest to a given query; also, given a query and a candidate component, some measures can help answer the question "is C close enough to K that it is worthwhile to modify C to satisfy K?".

- **The definition of measures of structural distance between specifications.** We have investigated applications of this measure of distance for the analytical definition of measures of modifiability.

Assessment

Among the distinguishing features of our empirical work on a measure of reusability, we mention:

- All the cost factors are quantified, and all are quantified in a uniform unit, namely the person month.

- The relationship between the individual cost factors and the measure of reusability is a straightforward mathematical relationship, which results from an economic analysis of the situation at hand.

- All the cost factors that are used in the reusability measure can be estimated with adequate precision at the time when reusability must be evaluated (and the reuse decision must be taken).

- Because it is based on an economic analysis, the reusability can be readily used for decision making, following the same criteria that are applied within the organization to other investment decisions. For example one may decide that components with a negative reusability are rejected outright, that components with a reusability greater

than 2 are accepted outright, and that component with a reusability between 0 and 1 are inspected further.

As for our work on a formal basis of software reuse, we feel that it enables us to capture a wide range of processes and activities using a single mathematical framework. With its emphasis on formal specifications, correctness preservation, and retrieval precision, our work attempts to achieve the same goal (for software reuse) as structured programming (for program construction).

Prospects

Among the prospects we envisage for our work on an empirical measure of reusability, we mention:

- Continue the development of the ARESsystem, by elaborating version 1.0 and producing versions 1.1 and 1.2.
- Define normal values of reusability for existing reuse libraries (for example, in the ASSET library, compute the minimum reusability, maximum reusability, average reusability, standard deviation, reusability variance).
- Define reusability standards based on the normal values: for example, a reuse library whose reusability values are found to be lower than some threshold can be diagnosed as having weak inclusion criteria (undeserving components are included); a reuse library whose reusability values are found to be higher than some threshold can be diagnosed as having strong inclusion criteria (some useful components may have been overlooked); a reuse library that has a reusability variance greater than some value can be diagnosed as having erratic (inconsistent) inclusion criteria.

As for the formal basis of software reuse, we are interested in extending it by consolidating its mathematical results, and on the practical side by correlating analytical measures to structural metrics of the components at hand.

Appendices

This part includes four appendices: in the first appendix we present the original verbatim text of our survey form, whose rationale we discuss in the second appendix. In the third appendix we present the source code for our reusability assessment package Finally, in appendix D we present data pertaining to the experimentation of our reusability measures on sample Grady Booch components [15]. These are: a list, a queue, a dequeue, a ring and a string. For each, we present the source code in Ada, the logfile of PC-Metric, as well as the logfile of ARES; these results are commented in chapter 8.

Appendix A

Reuse Survey Form

In this survey we seek the assistance of reuse practitioners in deriving an estimation model for a reusability measure of a given software component. In the preamble we explain the context of our work; then we give a detailed questionnaire that we ask interested parties to fill out and return to us. We are using this form to derive a number of separate cost models so that even if you cannot answer all the questions, we can still make use of your data. This survey encourages intelligent guessing when hard numbers are not available. Your best guess or expert opinion is better than no data at all.

The information you provide will be treated confidentially. Only summary statistics which do not reveal data specifically about individuals or organizations will be published.

A.1 Preamble

We are working on a measure of software reusability. Our measure aims to quantify the decision of the domain engineer, who must determine whether a given software component (asset) is worthy of being stored in a library of components for possible future reuse —in light of its reuse potential. Our measure reflects the expected long term person-month savings achieved by storing the component, and takes into account the upfront investment costs as well as the long term returns (in terms of product quality and programmer productivity).

Our measure depends on such factors as: the estimated cost to prepare the component for storage (for future retrieval); the cost of maintaining the component in store; the cost of retrieving it for verbatim reuse, as well as white box reuse; the gain in quality, achieved by reusing this component (versus developing it from scratch); the estimated frequency with which this component will be retrieved, and the frequency with which it will actually be (re-)used; the discount rate that the management of your organization applies on its investments, and the terms within which your organization expects investments to pay off; the cost of instantiating or modifying components once they are retrieved and selected for reuse.

The purpose of this form is to collect data about all these cost factors and build models that allow us to estimate them at the time when the decision to store the component must be made.

A.2 Interviewee's Affiliation

Company: _____

_____

Address: _____

_____

Name of the participants: _____

Phone numbers: _____

---

e-mails: _____

Date: _____

- Do you wish to receive a copy of the results of this survey?

  ___ Yes ___ No

A.3 Library Identity

In this part, we wish to enquire whether you maintain a library of software components; if so, we are interested in collecting some data about the application domain of this library as well as some indication of its level of usage.

Does your organization maintain a library of software components?

___ Yes ___ No

If you are maintaining a library, please answer the following questions. If not, please go directly to section 5.

1. **Name of the library:**

   ---

2. **Size (number of components):**

   ---

3. **Library class:**

   a. For private use _____
   b. Public domain
   c. Developed in an academic environment
   d. For use at a single site
   e. For use at multiple sites

4. **Library type:**

a. Scientific or mathematical components _____
b. Communications or telecommunications components
c. Process control components
d. Graphics, animation or image processing components
e. Robotics or mechanical automation components
f. Artificial intelligence components
g. Systems or support components
h. Other (please specify) _____

If possible, characterize the domain at two or three levels of generality. For example

Telecommunications software
Communications protocols
ISO compatible packages.

Given that the first level is given in the question above, give below the second and third levels.

Second level:_____
Third level:_____

5. **Level of Activity:**
Give an estimate of the level of software development activity that is making use of the software reuse library at hand. We measure this level of activity by means of two factors:

Volume of code produced annually:

_____Lines of Code

In the measure of volume of code, we include the code that is reused. For the sake of uniformity, we define a line of code as any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line.

Total annual manpower:_____PersonMonths

Volume of Function Points produced annually:_____FP

A.4 Library Operation

A.4.1 Storage Costs

1. **Entry structure:** We are interested in collecting data about what information is recorded in the entry of a software component; also, we wish to enquire about the structure of this information.

What is the structure of a component entry in the reuse library?
_____
_____
_____
_____

2. **Entry costs:** It is conceivable that the cost of entering a component in the database depends on some features of the component (e.g. the amount of data that you have to collect about it). If so, we wish to identify those features.

Does the cost of including a component depend on the component?
___ Yes ___ No

If so, what features does it depend on? _____
_____
_____
_____
_____

What was the cost of including the component at hand? _____
(person-days)

A.4.2 Retrieval Algorithms

We consider two kinds of retrieval in a library of software components: *exact retrieval*, where a requirements specification is submitted for the purpose of identifying library components that satisfy those requirements; *approximate retrieval*, where a requirements specification is submitted for the purpose of identifying library components that approximate those requirements as closely as possible.

1. **Design of the Algorithm for Exact Retrieval:** Exact retrieval could be based on matching keywords, comparing functional features, applying a formal procedure to check whether a library component subsumes the submitted requirements specification, etc.. We are interested in characterizing the algorithm as precisely as possible.

   What algorithm is used in this library for exact retrieval?
   _____
   _____
   _____
   _____

2. **Design of the Algorithm for approximate retrieval:**
   We assume that whenever we cannot find in the library a component that satisfies our query, we invoke some procedure of approximate retrieval, whereby we do not insist that candidate components subsume the requirements of the query; rather we content ourselves with components that approximate them (e.g. satisfy most of the requirements, partly satisfy all of them, ..).

   Please give details about the retrieval algorithm; in particular, we are interested in determining under what condition do we consider that a candidate component (approximate)-matches the query that is submitted.

   _____
   _____
   _____
   _____
   _____
   _____

3. **Performance of the Algorithm of approximate retrieval:** Retrieval algorithms are evaluated on the basis of their *precision*, which measures the ratio of the number of retrieved and relevant components over the number of retrieved components; and their *recall*, which measures the ratio of the number of retrieved and relevant components over the number of relevant components. As usual, expert estimates are an acceptable substitute for hard data.

   We are interested in an estimation of the average performance of the approximate algorithm that is being used, in terms of its precision and recall. Be as specific as possible (e.g. the precision of this algorithm, computed over a total of 200 approximate retrievals, is 45 percent).

A.5 Component Identity

We are assuming that you are monitoring a situation where software components are maintained in store for the purpose of software reuse; this store may be a formal software library, or some informal approximation thereof. We seek information about a component of your choice, about which you have accurate data. If you have, and wish to provide, data about more than one component, please duplicate parts A.5 to A.8.3 (pages 225 to 229).

1. **Name of the Component:**

   ---

2. **Function:** Describe the exact function of the component, with sufficient detail to discriminate it from other components in the library. If a formal specification is available for this component (and is not classified) so much the better.

   Component's functional specification:

   ---

3. **Competition:** If you maintain a library of software components, do you know of any components in the library that are more specific than the current component? For example, if the current component is a sorting routine that acts on a generic array type, we want to know whether there exists a sorting routine on integer arrays in the library.

Routine name	How it specializes component at hand
---	---
---	---
---	---

4. **Generality:** List all the parameters that can be set by a user of this component as part of an instanciation; for each parameter, give the type and the size (in case the size cannot be inferred from the type).

Parameter name	Type	Size	Comment
---	---	---	---
---	---	---	---
---	---	---	---
---	---	---	---
---	---	---	---

5. **Source Language:** Programming Language(s) in which the component is written:

A.6 Frequency

Among the key factors that intervene in the estimation of reusability is the frequency with which the component is expected to be reused for the particular application domain of the library. We recognize two separate measures of frequency, namely: the frequency with which the component is used verbatim, after instanciation; and the frequency with which it is used after modification.

1. **Frequency of Exact Retrieval:**
   Give the frequency of use of this component in a typical/average year. This is NOT the number of times the component is retrieved (it may be retrieved but not used); it is the number of times the component is ACTUALLY used VERBATIM, by instanciation (rather than by modification).

   Frequency:_____times / a year

2. **Frequency of Approximate Retrieval:** Give the frequency with which the component at hand is approximate -retrieved in a typical year and reused after modification. DO NOT include the number of times when the component was retrieved then discarded; ONLY include those cases when the component was reused, after modification.

   Frequency:_____times / a year

A.7 Ivestment Factors

Investment factors include: the cost of reengineering the component; the cost of verifying the component (up to the standards of the reuse library); the cost of baselining the component. These will be discussed in turn below.

A.7.1 Reengineering costs

We use the term 'Reengineering' to refer to the process whereby a component that was derived for a single use is reengineered so as to be more general, more parameterized, and to have more standard interfaces. The cost of this task depends primarily on structural properties of the component. Give the following data about this component (to the extent that it is available):

Reengineering cost, in PM: _____

Development cost, in PM: _____
Function Point Count, FP: _____

Structural features (obtained, eventually, by a metrics package)

Size, in LOC: _____

Halstead's Total number of tokens, N: _____
number of operators, N1: _____
number of operands, N2: _____
volume, V: _____
effort, E: _____
development time, T: _____
number of subprograms, NS: _____

McCabe's Cyclomatic Complexity, VG: _____
average cyclomatic complexity, AVG: _____

A.7.2 Verification costs

In addition to the verification the component may have undergone within its original host system, it must be further verified before inclusion in a reuse library.

Give a summary of the reliability standards that are enforced in the reuse library at hand (required reliability, structural properties, reliability-related programming standards):

Give the cost of verification: _____
(person-months)

A.7.3 Baselining Costs

Once the component has been duly reengineered and verified, it must be included in the reuse library. We are interested in the costs associated with this procedure.

Does your organization have a formal procedure for component review prior to including a component in a reuse library? _____

If so, give main steps/phases of this procedure:

How many people are involved in this process? _____
How long did it take to baseline this component? _____

A.8 Customization Factors

We are interested in evaluation costs, instanciation costs for components that are used verbatim, and modification costs for components that are used after modification.

A.8.1 Evaluation Costs

Evaluation refers to the steps whereby a component that has been retrieved is evaluated with respect to such questions as: can this component be used verbatim? does it have to be modified? does the cost of modification approach or exceed the cost of development from scratch? was this component retrieved by mistake (lack of precision of the retrieval algorithm)?

Give measures of person-month effort of evaluating this component over all reuse occurences that you have recorded in the past.

Approximate Date	Evaluation Effort
----------	----------
----------	----------
----------	----------
----------	----------
----------	----------

If you have no precise data, could you give an expert opinion about the average instanciation cost?

---------- (person months)

A.8.2 Instantiation Costs

Give measures of person-month effort required to instantiate this component over all verbatim reuse occurences that you have recorded in the past.

Approximate Date	Evaluation Effort
----------	----------
----------	----------
----------	----------
----------	----------
----------	----------

If you have no precise data, could you give an expert opinion about the average instanciation cost?

---------- (person months)

A.8.3 Adaptation Costs

Give measures of person-month effort required to adapt this component over all non verbatim reuse occurences that you have recorded in the past.

Approximate Date	Evaluation Effort
----------	----------
----------	----------

---------- ----------

---------- ----------

---------- ----------

If you have no precise data, could you give an expert opinion
about the average instanciation cost?

(person months)

A.9 Strategic Factors

Our measure of reusability depends on two strategic factors, which are the *discount rate* and the *investment period*.

1. **Discount Rate**: We define the discount rate as follow: if you have the opportunity to exchange 1 person month today against $(1+d)$ person months in a year, what is the smallest value of d under which you will accept the opportunity.

   Does your organization have a standard discount rate pertaining
   to the time value of its resources (dollars, person-months, etc..)?

   ___ Yes ___ No

   If it does, give the value of the standard discount rate.

   ---

2. **Investment Period**: We define the investment period as the length of time within which your organization expects to recover its investment costs.

   Does your organization have a standard investment period pertaining
   to investment-like decisions?

   ___ Yes ___ No

   If it does, give the value of the standard investment period.

A.10 Concluding Remarks

Upon replying to these questions, please mail, e-mail or fax this completed form to

Rym Mili
Eric Jonsson School of Engineering and Computer Science

University of Texas at Dallas
Box 830688
Richardson, TX 75083-0688 USA

e-mail: rmili@utdallas.edu
fax: (214) 883 2349

if you know of any other practitioners/developers who may be willing to give answers, please feel free to pass it on to them.

Thank you very much for your contribution.

Appendix B

A Reuse Survey

B.1 Survey Form

In this section we review the survey form and discuss the rationale behind the questions that it asks; then we report on the survey that we have conducted with this form. The original text of this survey will be presented here, i small font size; our comments are presented in normal size Roman font. The original version of the survey form is given in appendix A.

B.1.1 Interviewee's Affiliation

This part is optional; you may decline to answer part or all of the questions of this part. In any case, no affiliation information will be published; however, information about your organization will allow us to identify organization-related trends or correlations.

Company: _____

_____

Address: _____

_____

_____

Name of the participants: _____

_____

_____

Phone numbers: _____

_____

_____

e-mails: _____

_____

_____

Date: _____

- Do you wish to receive a copy of the results of this survey?

___ Yes ___ No

This identifies the organization and presents the interviewee's function and responsibilities within the organization. We do not publish the organization's name, but we do need it for our purposes, to identify organization-specific or industry-specific trends and correlations.

B.1.2 Library Identity

In this part, we wish to enquire whether you maintain a library of software components; if so, we are interested in collecting some data about the application domain of this library as well as some indication of its level of usage.

Does your organization maintain a library of software components?

___ Yes ___ No

If you are maintaining a library, please answer the following questions. If not, please go directly to section 5.

1. **Name of the library:**

_____

2. **Size (number of components):**

_____

3. **Library class:**

a. For private use _____
b. Public domain
c. Developed in an academic environment
d. For use at a single site
e. For use at multiple sites

4. **Library type:**

a. Scientific or mathematical components _____
b. Communications or telecommunications components
c. Process control components
d. Graphics, animation or image processing components
e. Robotics or mechanical automation components
f. Artificial intelligence components
g. Systems or support components
h. Other (please specify) _____

If possible, characterize the domain at two or three levels of generality. For example

Telecommunications software
Communications protocols
ISO compatible packages.

Given that the first level is given in the question above, give below the second and third levels.

Second level:_____
Third level:_____

The library class gives an indication of the size of the potential clientele of the library. This, in turn, allows us to normalize many measures to the size of the user community. For example, if component A is used twice as often as component B, is it because A is twice as useful or because the user community that has access to A is twice as large?

On the other hand, the library type gives an indication of the general application domains of components in the library. It is quite conceivable that the reusability characteristics of components vary a great deal by application domain: for example, numeric routines have been routinely reused in numerical analysis applications, with little or no reuse technology. By contrast, data processing components have not, historically been available for reuse. The rationale for this contrast is the following:

- Numeric analysis components have a low coupling, i.e. communicate with the outside world over a narrow bandwidth. This enhances their chance of being matched with a particular application.
- While in numeric applications the source of difficulty is the complexity of the function to be computed, in data processing applications the source of difficulty is the organization of data. The likelihood that the same function be used often is greater than the likelihood that the same data organization be used often.
- Numeric analysis applications represent, on average, a larger development effort per line of code than data processing applications. Hence there is more incentive to reuse numeric applications than data processing applications.

5. **Level of Activity:**
Give an estimate of the level of software development activity that is making use of the software reuse library at hand. We measure this level of activity by means of two factors:

Volume of code produced annually:

_____Lines of Code

```
In the measure of volume of code, we include the code that
is reused. For the sake of uniformity, we define a line of
code as any line of program text that is not a comment or
blank line, regardless of the number of statements or
fragments of statements on the line.

Total annual manpower:_____Person Months

Volume of Function Points produced annually:_____FP
```

This question allows us to normalize our results for the level of development activity within the organization. We propose three distinct (but closely related) measures of activity level: the volume of code produced annually; the total manpower spent on development; and the total volume of function points that are produced annually. It is reasonable to expect that the reuse frequency of a component is linearly dependent on the level of development activity within that organization.

B.1.3 Library Operation

B.1.3.1 Storage Costs

1. **Entry structure:** We are interested in collecting data about what information is recorded in the entry of a software component; also, we wish to enquire about the structure of this information.

```
What is the structure of a component entry in the reuse library?
   ____________________________________________________________
   ____________________________________________________________
   ____________________________________________________________
   ____________________________________________________________
```

The effort to store a component in the library depends, of course, on the structure of this entry. Note that this applies whether we are dealing with manual storage and retrieval, or with automated storage and retrieval.

2. **Entry costs:** It is conceivable that the cost of entering a component in the database depends on some features of the component (e.g. the amount of data that you have to collect about it). If so, we wish to identify those features.

```
Does the cost of including a component depend on the component?
          ___ Yes        ___ No

   If so, what features does it depend on? _____________________
                                           _____________________
                                           _____________________
                                           _____________________
                                           _____________________

What was the cost of including the component at hand? __________
                                                      (person-days)
```

In principle, we expect that the cost of storing a component in the library depend on the procedures that are in force in the organization at hand, but does not depend necessarily on the component at hand (or depends so little that it is not perceptible). Nevertheless, in case interviewees feel that some aspects of the component influence storage costs, we are interested to learn about them.

B.1.4 Retrieval Algorithms

We consider two kinds of retrieval in a library of software components: *exact retrieval*, where a requirements specification is submitted for the purpose of identifying library components that satisfy those requirements; *approximate retrieval*, where a requirements specification is submitted for the purpose of identifying library components that approximate those requirements as closely as possible.

1. **Design of the Algorithm for Exact Retrieval:** Exact retrieval could be based on matching keywords, comparing functional features, applying a formal procedure to check whether a library component subsumes the submitted requirements specification, etc.. We are interested in characterizing the algorithm as precisely as possible.

   What algorithm is used in this library for exact retrieval?

   In principle, this question is meaningful only if we are dealing with an automated retrieval algorithm. In practice, even informal algorithms may be correlated with performance measurement. Note that we have already asked for the size of the reuse library; our intention is that the combination of the size and the algorithm should give us means to build an estimation model for retrieval costs.

2. **Design of the Algorithm for approximate retrieval:** We assume that whenever we cannot find in the library a component that satisfies our query, we invoke some procedure of approximate retrieval, whereby we do not insist that candidate components subsume the requirements of the query; rather we content ourselves with components that approximate them (e.g. satisfy most of the requirements, partly satisfy all of them, ..).

   Please give details about the retrieval algorithm; in particular, we are interested in determining under what condition do we consider that a candidate component (approximate)-matches the query that is submitted.

   We realize that for informal retrieval procedures (using e.g. hypertext or library-inspired techniques) there is no difference, in fact, between exact retrieval and approximate retrieval: The same process is used to find components that match or

approximate the user query. If formal procedures are used, the algorithm for approximate retrieval may be significantly different, and may have a substantially different performance, than the exact algorithm: consider, for illustration, the algorithm discussed in chapter 11. In this algorithm, a library component is considered to approximate-match the user query if it minimizes (some aspect of) functional distance to the query.

3. **Performance of the Algorithm of approximate retrieval:** Retrieval algorithms are evaluated on the basis of their *precision*, which measures the ratio of the number of retrieved and relevant components over the number of retrieved components; and their *recall*, which measures the ratio of the number of retrieved and relevant components over the number of relevant components. As usual, expert estimates are an acceptable substitute for hard data.

   We are interested in an estimation of the average performance of the approximate algorithm that is being used, in terms of its precision and recall. Be as specific as possible (e.g. the precision of this algorithm, computed over a total of 200 approximate retrievals, is 45 percent).

We expect that the performance of the approximate retrieval, in those instances where it is distinct from the exact retrieval, is fairly independent from the precise algorithm (or from the definition of approximate match) and is dependent instead on the size of the reuse library. Nevertheless, we choose to submit this question, in case a trend appears.

B.2 Component Identity

Our measure of reusability depends on a wide range of cost factors that are specific to the component at hand. These are the subject of parts B.2 to B.5.3.

We are assuming that you are monitoring a situation where software components are maintained in store for the purpose of software reuse; this store may be a formal software library, or some informal approximation thereof. We seek information about a component of your choice, about which you have accurate data. If you have, and wish to provide, data about more than one component, please duplicate parts B.2 to B.5.3 (pages 236 to 242).

1. **Name of the Component:**

   ---

2. **Function:** Describe the exact function of the component, with sufficient detail to discriminate it from other components in the library. If a formal specification is available for this component (and is not classified) so much the better.

Component's functional specification:

A precise definition of the function of the component is essential if we are to understand what determines the reuse frequency of a component, or what makes a component reuse worthy. In particular, the more generic the component's function, the more likely the component is to be reused.

3. **Competition:** If you maintain a library of software components, do you know of any components in the library that are more specific than the current component? For example, if the current component is a sorting routine that acts on a generic array type, we want to know whether there exists a sorting routine on integer arrays in the library.

Routine name	How it specializes component at hand
---	---
---	---
---	---

This question complements the previous question: a component may well be very generic and offer a considerable reuse potential. But if the reuse library contains a large number of components that specialize this component (in different ways). If the retrieval algorithm has a good recall (i.e. retrieves all the components that are relevant) then this component runs a risk of being overshadowed by those that specialize it; hence the rationale for this question.

4. **Generality:** List all the parameters that can be set by a user of this component as part of an instantiation; for each parameter, give the type and the size (in case the size cannot be inferred from the type).

Parameter name	Type	Size	Comment
---	---	---	---
---	---	---	---
---	---	---	---
---	---	---	---
---	---	---	---

The genericity of a component is a very subjective notion, which is quite difficult to define formally. A trivial way to measure it is by means of the bandwidth of communication that occurs between the component and its calling environment. This is the rationale for this question, where we are interested not only in the number of parameters, but also their type and (to the extent that it is not always implied by the type) their size.

5. **Source Language:** Programming Language(s) in which the component is written:

Programming languages play a vital role in promoting or hindering the reusability of a given component. The feature that has the greatest impact in this regard is that of scoping rules: Modular languages in general, and object oriented languages in particular, are best equipped to promote reuse; whereas block structured languages, despite their widespread use, are ill-equipped.

B.3 Frequency

In this section we enquire about the frequency of reuse of the component at hand, by considering in turn the frequency of exact retrieval and the frequency of approximate retrieval. Our intention is to relate these frequencies to three sets of features: first, features of the component's functional specification (such as, how generic the specification is, how low is its coupling); second, features of the component's implementation (such as the simplicity of its design, the ease of instantiating it or adapting it); finally, features of the application domain of the reuse library (such as the pervasiveness of the component within the application domain).

Among the key factors that intervene in the estimation of reusability is the frequency with which the component is expected to be reused for the particular application domain of the library. We recognize two separate measures of frequency, namely: the frequency with which the component is used verbatim, after instantiation; and the frequency with which it is used after modification.

1. **Frequency of Exact Retrieval:** Give the frequency of use of this component in a typical/average year. This is NOT the number of times the component is retrieved (it may be retrieved but not used); it is the number of times the component is ACTUALLY used VERBATIM, by instantiation (rather than by modification).

   Frequency:_____**times / a year**

2. **Frequency of Approximate Retrieval:** Give the frequency with which the component at hand is approximate -retrieved in a typical year and reused after modification. DO NOT include the number of times when the component was retrieved then discarded; ONLY include those cases when the component was reused, after modification.

   Frequency:_____**times / a year**

B.4 Investment Factors

Investment factors include: the cost of reengineering the component; the cost of verifying the component (up to the standards of the reuse library); the cost of baselining the component. These will be discussed in turn below.

B.4.1 Reengineering costs

We use the term 'Reengineering' to refer to the process whereby a component that was derived for a single use is reengineered so as to be more general, more parameterized, and to have more standard interfaces. The cost of this task depends primarily on structural properties of the component. Give the following data about this component (to the extent that it is available):

```
Reengineering cost, in PM: _____________________________________

Development cost, in PM: _____________________________________
Function Point Count, FP: _____________________________________

Structural features (obtained, eventually, by a metrics package)

   Size, in LOC: _____________________________________________

   Halstead's Total number of tokens, N: _____________________
              number of operators, N1: _____________________
              number of operands,  N2: _____________________
              volume, V: ___________________________________
              effort, E: ___________________________________
              development time, T: _________________________
              number of subprograms, NS: ___________________

   McCabe's Cyclomatic Complexity, VG: _____________________
              average cyclomatic complexity, AVG: ___________
```

Existing software cost estimation models have built in equations that deal with the cost of modifying a component, which is what reengineering really is. However, we felt that reengineering for the purpose of software reuse is sufficiently unique in its goals (genericity, design integrity, documentation) and its processes that a specific cost estimation model may be justified. In this question, we enquire about traditional cost drivers (such as the line of code count and the function point count) as well as structural features of the component at hand (such as Halstead's metrics and McCabe metrics).

B.4.2 Verification costs

In addition to the verification the component may have undergone within its original host system, it must be further verified before inclusion in a reuse library.

```
Give a summary of the reliability standards that are enforced in the
reuse library at hand (required reliability, structural properties,
reliability-related programming standards):

_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________
_____________________________________________________________________


Give the cost of verification: _____________________________________
                                                  (person-months)
```

The comment we made earlier about reengineering costs applies equally well to verification costs: even though existing cost models make provisions for estimating the cost of verifying a component, we felt that it is worthwhile to investigate the possibility of a reuse specific model.

B.4.3 Baselining Costs

Once the component has been duly reengineered and verified, it must be included in the reuse library. We are interested in the costs associated with this procedure.

Does your organization have a formal procedure for component review prior to including a component in a reuse library? _____

If so, give main steps/phases of this procedure:

_____
_____
_____
_____
_____
_____

How many people are involved in this process? _____
How long did it take to baseline this component? _____

We expect that baselining vary little from component to component, and that they depend primarily on the procedures that are practiced in the organization at hand. By asking this question, we seek to get an idea about the range of values of these costs.

B.5 Customization Factors

We are interested in evaluation costs, instantiation costs for components that are used verbatim, and modification costs for components that are used after modification.

B.5.1 Evaluation Costs

Evaluation refers to the steps whereby a component that has been retrieved is evaluated with respect to such questions as: can this component be used verbatim? does it have to be modified? does the cost of modification approach or exceed the cost of development from scratch? was this component retrieved by mistake (lack of precision of the retrieval algorithm)?

Give measures of person-month effort of evaluating this component over all reuse occurrences that you have recorded in the past.

Approximate Date	Evaluation Effort
_____	_____
_____	_____
_____	_____

---------- ----------

---------- ----------

If you have no precise data, could you give an expert opinion about the average instantiation cost?

(person months)

The step of *evaluation* is one step that has no equivalent in the traditional development lifecycle. Hence it is unlikely that existing cost models, that are geared towards software development, can be of much help in assessing the cost of this step. On the other hand, we expect that the cost of this step is fairly small, fairly independent on the component, rather dependent on the operational procedures that are in use in the organization; and in general we expect that practicing analysts generally have a fairly accurate idea about this cost.

B.5.2 Instantiation Costs

Give measures of person-month effort required to instantiate this component over all verbatim reuse occurrences that you have recorded in the past.

Approximate Date	Instantiation Effort
----------	----------
----------	----------
----------	----------
----------	----------
----------	----------

If you have no precise data, could you give an expert opinion about the average instantiation cost?

(person months)

The comment we made earlier about evaluation costs applies equally well to instantiation costs: because these steps are not part of the traditional development cycle, it is unlikely that traditional cost estimation models can be of use in estimating their cost. Like evaluation costs, we expect instantiation costs to be fairly small and rather dependent on the organization's operational procedures. Unlike evaluation costs, we expect that instantiation costs be dependent on the component at hand (although perhaps marginally).

B.5.3 Adaptation Costs

Give measures of person-month effort required to adapt this component

over all non verbatim reuse occurrences that you have recorded in the past.

Approximate Date	Evaluation Effort
_____	_____
_____	_____
_____	_____
_____	_____
_____	_____

If you have no precise data, could you give an expert opinion about the average instantiation cost?

_____
(person months)

Unlike evaluation costs and instantiation costs, adaptation costs are provided for by existing development cost models. But because they are provided for in the context of software maintenance, we feel that perhaps the conditions are sufficiently different from software reuse to warrant a separate enquiry.

B.6 Strategic Factors

Our measure of reusability depends on two strategic factors, which are the *discount rate* and the *investment period.*

1. **Discount Rate:** We define the discount rate as follow: if you have the opportunity to exchange 1 person month today against $(1 + d)$ person months in a year, what is the smallest value of d under which you will accept the opportunity.

   Does your organization have a standard discount rate pertaining to the time value of its resources (dollars, person-months, etc..)?

   ___ Yes ___ No

   If it does, give the value of the standard discount rate.

   _____
   _____

As a default option, a respondent may give the corporate discount rate, as decided by the corporate strategic planners. The respondent may revise the corporate figure upward or downwards, dependent on whether the software development department

is experiencing a staffing shortage (making it less worthwhile to invest person months now, even if it produces subsequent gains) or a staffing surplus (making it worthwhile to divert staffing resources to software reuse, if only for the sake of keeping them busy and motivated). In the first case we revise d upwards; the second case we revise it downwards from the corporate value.

2. **Investment Period:**

We define the investment period as the length of time within which your organization expects to recover its investment costs.

Does your organization have a standard investment period pertaining to investment-like decisions?

___ Yes ___ No

If it does, give the value of the standard investment period.

As a default option, a respondent may give the corporate investment cycle, as provided by the corporate strategic planners. The respondent may revise the figure downwards if, for example, he does not have management support for his venture into software reuse, and wishes to make his case on a short term basis. Alternatively, if corporate management is committed to software reuse, and is interested in the long term, this figure may be revised upwards.

B.7 Concluding Remarks

Upon replying to these questions, please mail, e-mail or fax this completed form to

Rym Mili
Eric Jonsson School of Engineering and Computer Science
University of Texas at Dallas
Box 830688
Richardson, TX 75083-0688 USA

e-mail: rmili@utdallas.edu
fax: (214) 883 2349

if you know of any other practitioners/developers who may be willing to give answers, please feel free to pass it on to them.

Thank you very much for your contribution.

Appendix C

Reusability Estimation Program

```
wm title . "ARES Version 1.0"
wm geometry . 750x250
. configure -background SeaGreen2
message .msg -font -Adobe-times-bold-r-normal--*-180* \
-relief raised \
-width 1000 \
-justify center \
-borderwidth 1 -text " Automated Reuse Estimation System \nversion 1.0  " -padx 0

button .click -text "click here" \
  -command {
  destroy .
  exec wish -f mainmenu
  }

pack .msg -side top -expand yes -fill both
pack .click -side right
-----------------------------------------------------

#!/usr/bin/wish -f
#define the frame for placing the menu
wm title . "ARES Version 1.0"
frame .mbar -relief raised -bd 2
.mbar configure -background blue

frame .dummy -width 20c -height 10c
.dummy configure -background lightblue3
pack .mbar .dummy -side top -fill x

# realize the frame on the screen
#message .msg -font -Adobe-times-medium-r-normal--*-180* -relief raised \
#-width 1000 \
#-borderwidth 1 -text " Automated Reuse Estimation System  "
#pack .msg -side  bottom -expand yes -fill both

#define the file menu
menubutton .mbar.file -text "    File    " -underline 0 \
-menu .mbar.file.menu
```

```
menubutton .mbar.lifecycle -text "    Lifecycle    " -underline 0 \
-menu .mbar.lifecycle.menu
#realize the menus on the frame
pack .mbar.file .mbar.lifecycle -side left
#define the options for the file menu
menu .mbar.file.menu
.mbar.file.menu add radiobutton -label Open -variable simvar \
-command "exec wish -f error &"
.mbar.file.menu add radiobutton -label Copy -variable simvar \
-command "exec wish -f error &"
.mbar.file.menu add radiobutton -label Delete -variable simvar \
-command "exec wish -f error &"
.mbar.file.menu add radiobutton -label Search -variable simvar \
-command "exec wish -f error &"
.mbar.file.menu add command -label "Quit" \
-command "destroy ."

#define the options for the lifecycle menu
menu .mbar.lifecycle.menu
.mbar.lifecycle.menu add cascade -label "Synchronous" \
-menu .mbar.lifecycle.menu.synchronous
.mbar.lifecycle.menu add cascade -label "Asynchronous" \
-menu .mbar.lifecycle.menu.asynchronous

#define the menu for the Synchronous menu
menu .mbar.lifecycle.menu.synchronous
.mbar.lifecycle.menu.synchronous add radiobutton -label "Waterfall Predefined" \
-command "exec wish -f syncwaterfall & "
.mbar.lifecycle.menu.synchronous add radiobutton -label "Waterfall Customized" \
-command "exec wish -f error & "
.mbar.lifecycle.menu.synchronous add radiobutton -label "Spiral" \
-command "exec wish -f error & "

#define the menu for the Asynchronous menu
menu .mbar.lifecycle.menu.asynchronous
.mbar.lifecycle.menu.asynchronous add radiobutton -label "Waterfall Predefined" \
-command "exec wish -f error & "
.mbar.lifecycle.menu.asynchronous add radiobutton -label "Waterfall Customized" \
-command "exec wish -f error & "
.mbar.lifecycle.menu.asynchronous add radiobutton -label "Spiral" \
-command "exec wish -f error & "

#realize the final menu setup on the window
tk_menuBar .mbar .mbar.file .mbar.lifecycle
focus .mbar

-----------------------------------------------------------------

#!/usr/bin/wish -f
#define the canvas with the required parameters and of the required type
wm title . "ARES Version 1.0 "
canvas .c -width 20c -height 21c -bg plum1 -scrollregion {0 0 1000 700} \
-confine false  -relief raised
#realize the canvas on the screen
```

```
.c configure -background SeaGreen2
pack .c
bind .c <Any-Motion> { puts "%x,%y"}
frame .f2
pack .f2 -side top
button .f2.b1 -relief groove -text "Synchronous Waterfall Predefined Lifecycle"
pack .f2.b1 -side left
frame .f1
pack .f1 -side bottom
button .f1.b1 -relief flat -text "Do you want to adopt this lifecycle?"
update
pack .f1.b1 -side left

button .f1.b2 -text "OK" -command {exec wish -f syncmenu &
   }
update
button .f1.b5 -text "Cancel" -command "quit quit"
update
pack .f1.b2 .f1.b5  -fill x  -side left

proc quit button {
destroy .
}
.c create rectangle 120 60 270 80 -outline black
.c create text 122 78 -text "Component Specification" -anchor sw

.c create rectangle 160 100 310 120 -outline black
.c create text 172 118 -text "Component Design" -anchor sw

.c create rectangle 200 140 350 160 -outline black
.c create text 222 158 -text "Coding and Test" -anchor sw

.c create line 270 70 295 70 -fill black
.c create line 295 70 295 100 -fill black -arrow last

.c create line 310 110 335 110 -fill black
.c create line 335 110 335 140 -arrow last

.c create line 210 214 235 214
.c create text 240 220  -text "if reusable" -anchor sw -font 7x13
.c create line 320 214 345 214

.c create rectangle 240 260 380 280 -outline black
.c create text 262  278  -text "Reengineering" -anchor sw

.c create rectangle 280 300 430 320 -outline black
.c create text 312  318  -text "Verification" -anchor sw

.c create rectangle 320 340 470 360 -outline black
.c create text 362  358  -text "Baselining" -anchor sw

.c create line 380 270 405 270
.c create line 405 270 405 300 -arrow last
```

```
.c create line 430 310 455 310
.c create line 455 310 455 340 -arrow last

.c create rectangle 120 450 270 470 -outline black
.c create text 145 468 -text "Exact Retrieval" -anchor sw

.c create rectangle 350 450 500 470 -outline black
.c create text 358 468 -text "Approximate Retrieval" -anchor sw

.c create rectangle 120 510 270 530 -outline black
.c create text 155 528 -text "Instantiation" -anchor sw

.c create rectangle 350 510 500 530 -outline black
.c create text 390 528 -text "Adaptation" -anchor sw

.c create line 195 470 195 510 -arrow last
.c create line 425 470 425 510 -arrow last

.c create rectangle 230 570 380 590 -outline black
.c create text 244 588  -text "Integration and Test" -anchor sw

.c create line 270 520 285 520
.c create line 285 520 285 570 -arrow last

.c create line 335 520 350 520
.c create line 335 520 335 570 -arrow last

.c create rectangle 270 630 420 650 -outline black
.c create text 298 648 -text "Implementation" -anchor sw

.c create rectangle 310 690 500 710 -outline black
.c create text 315 708 -text "Operations and Maintenance" -anchor sw

.c  create line 420 640 435 640
.c create line 435 640 435 690 -arrow last

.c create line 380 580 395 580
.c create line 395 580 395 630 -arrow last

-------------------------------------------------------------------------

#!/usr/bin/wish -f
####define the frame for placing the menu
wm title . "ARES Version 1.0"
frame .mbar -relief raised -bd 2
.mbar configure -background SeaGreen2
frame .dummy -width 18c -height 8c
pack .mbar .dummy -side top -fill x

### Initialization ###
                set pdperpm   0.05
  #20 person days per person month

  set defaultlambda 0.62
```

```
#ratio freqx/freq

                set invcycle 0
#investment cycle

                set discount 0
#discount rate

set rls      0
#number of full time persons operating library

set sr       0
#annual effort searching library

set n        0
#number of components in library

               set stor     0
#cost of storing component

set exact    0
#cost of exact retrieval

set aprox    0
#cost of approximate retrieval

set freqx    0
#freqency of exact retrieval

set freqa    0
#frequency of approximate retrieval

set freq     0
#freqx + freqa

set lambda   0
#freqx/freq

set de       0
#development effort

set ruse     0
#reuse rate

set su       0
#software understanding

set dm       0
#percent design modification

set cm       0
#percent code modification

set reeng    0
```

```
   #reengineering cost

   set verif    0
   #verif cost

   set base     0
   #baselining cost

   set instance 0
   #instanciation cost

   set adapt    0
   #adaptation cost

   set rest     0
   #integration and test cost

   set first    0
   #first part of lifecycle

   set operprime 0
   #annual operating cost without reuse

   set oper      0
   #annual operating cost with reuse

   set absolutereu 0
   #absolute reusability

   set intrinsicreu 0
   #intrinsic reusability

   set breakevenfreq 0
   #break even frequency

   set longtermabsreu 0
   #long term absolute reusability

   set longtermintreu 0
   #long term intrinsic reusability

   set longtermbef 0
   #long term break even frequency

####define the reuse menu
menubutton .mbar.reuse -text "    Reuse Costs     " -underline 0 \
-menu .mbar.reuse.menu
.mbar.reuse configure -background SeaGreen2

menubutton .mbar.reusecosts -text "   Display Reuse Costs   " -underline 0 \
-menu .mbar.reusecosts.menu
.mbar.reusecosts configure -background SeaGreen2

menubutton .mbar.lifecycle -text "  Reusability Measures  " -underline 0 \
```

```
-menu .mbar.lifecycle.menu
.mbar.lifecycle configure -background SeaGreen2

menubutton .mbar.reusemeasures -text "  Display Reusability Measures  " -underline 0 \
-menu .mbar.reusemeasures.menu
.mbar.reusemeasures configure -background SeaGreen2

pack .mbar.reuse .mbar.reusecosts .mbar.lifecycle .mbar.reusemeasures -side left

###### REUSE COSTS ###############################################################
menu .mbar.reuse.menu
.mbar.reuse.menu configure -fg SeaGreen

.mbar.reuse.menu add cascade -label " Strategic Factors " \
-menu .mbar.reuse.menu.strategic
.mbar.reuse.menu add cascade -label " Domain Factors " \
-menu .mbar.reuse.menu.domain
.mbar.reuse.menu add cascade -label "Component Factors "\
-menu .mbar.reuse.menu.component

.mbar.reuse.menu add command -label "Quit" \
-command {destroy .
  destroy .c}


##########################STRATEGIC FACTORS#####################################
menu .mbar.reuse.menu.strategic
.mbar.reuse.menu.strategic configure -fg SeaGreen

.mbar.reuse.menu.strategic add cascade -label " Corporate Factors "\
       -menu .mbar.reuse.menu.strategic.corporate
.mbar.reuse.menu.strategic add cascade -label " Library Factors "\
       -menu .mbar.reuse.menu.strategic.library


############CORPORATE FACTORS###################
#### TOP Investment Cycle  ###
menu .mbar.reuse.menu.strategic.corporate
.mbar.reuse.menu.strategic.corporate configure -fg SeaGreen

.mbar.reuse.menu.strategic.corporate add checkbutton -label "Investment Cycle" \
-command  {
                        toplevel .invest
                        wm title .invest "Investment Cycle"
                        wm geometry .invest 550x150

                       message .invest.invest1 -width 500 -justify left \
-text "Do you know what is the \
         investment cycle practiced at your organization:" -padx 0
                         button .invest.invest2 -text Yes \
-command {
toplevel .icycle
wm title .icycle "Investment Cost: Enter Values"
wm geometry .icycle 550x150
```

```
message .icycle.icycle1 -width 500 -justify left \
-text "Give the Investment Cycle:"\
-padx 0
entry .icycle.cmd -width 20 -relief sunken -textvariable \
                        invcycle
#########

button .icycle.icycle2 -text ok \
 -command {destroy .icycle
           destroy .invest
          }
button .icycle.icycle3 -text cancel \
-command {
destroy .icycle
set invcycle 0 }
pack .icycle.icycle1 .icycle.cmd -side top -expand true
pack .icycle.icycle2  -side left
pack .icycle.icycle3 -side right
}
  button .invest.invest3 -text No -command {

 toplevel .icyclez
                        wm title .icyclez "Investment Cost: Default Value"
                        wm geometry .icyclez 550x150

                        message .icyclez.icyclez1 -width 500 -justify left \
-text "ARES will take the default \
value of 3 years."

button .icyclez.icyclez2 -text ok \
 -command {
                                  set invcycle 3
                                  #########

                                  destroy .icyclez
        destroy .invest}

button .icyclez.icyclez3 -text Cancel \
-command {
  set invcycle 0
  ##########
  destroy .icyclez }

                        pack .icyclez.icyclez1 -side top
pack .icyclez.icyclez2 -side left
pack .icyclez.icyclez3 -side right
}

button .invest.invest4 -text Cancel \
 -command {
  set invcycle 0
  destroy .invest }
```

```
 pack .invest.invest1 -side top
                        pack .invest.invest2 .invest.invest3 -side left
 pack .invest.invest4 -side right

 }

######### Discount Rate ###############
.mbar.reuse.menu.strategic.corporate add checkbutton -label "Discount Rate" \
-command  {
 set discount 0
 ##########

                        toplevel .disc
                        wm title .disc "Discount Rate"
                        wm geometry .disc 550x150

                       message .disc.disc1 -width 500 -justify left \
-text "Do you know your organization's \
       discount rate?" -padx 0
                        button .disc.disc2 -text Yes \
-command {
toplevel .dcycle
wm title .dcycle "Discount Rate: Enter Values"
wm geometry .dcycle 550x150

message .dcycle.dcycle1 -width 500 -justify left \
-text "Give the Discount Rate:"\
-padx 0
entry .dcycle.cmd -width 20 -relief sunken -textvariable \
                      discount
##########

button .dcycle.dcycle2 -text ok \
 -command {destroy .dcycle
           destroy .disc
                 }

button .dcycle.dcycle3 -text cancel \
-command {
set discount 0
destroy .dcycle
}
pack .dcycle.dcycle1 .dcycle.cmd -side top -expand true
pack .dcycle.dcycle2  -side left
pack .dcycle.dcycle3 -side right
}

 button .disc.disc3 -text No -command {

toplevel .dcyclez
                      wm title .dcyclez "Discount Rate: Default Value"
                      wm geometry .dcyclez 550x150

                      message .dcyclez.dcyclez1 -width 500 -justify left \
```

```
 -text "ARES will take the default \
 value of 0.15."

 button .dcyclez.dcyclez2 -text ok \
  -command {
                                  set discount 0.15
                                  ##########

  destroy .dcyclez
          destroy .disc}

 button .dcyclez.dcyclez3 -text Cancel \
 -command {
                  set discount 0
  ##########

  destroy .dcyclez }

                          pack .dcyclez.dcyclez1 -side top
 pack .dcyclez.dcyclez2 -side left
 pack .dcyclez.dcyclez3 -side right
 }

 button .disc.disc4 -text Cancel \
  -command {
    set discount 0
    ##########
    destroy .disc }

                           pack .disc.disc2 .disc.disc3 -side left
  pack .disc.disc4 -side right
  pack .disc.disc1 -side top
  }


 ####### LIBRARY COST FACTORS ###############
 #### Cost of storing/searching + library size ##################
 menu .mbar.reuse.menu.strategic.library
 .mbar.reuse.menu.strategic.library configure -fg SeaGreen

 .mbar.reuse.menu.strategic.library add checkbutton -label "Storing/Search Costs" \
 -command  {
  set rls 0
  set sr 0
  set n 0
  set stor 0
                          ##########

                          toplevel .lib
                          wm title .lib "Storing/ Search Cost"
                          wm geometry .lib 550x250

                         message .lib.lib1 -width 500 -justify left \
 -text "What is the number of full \
```

```
        time persons assigned to the task of operating the \
reuse library?" -padx 0
entry .lib.cmd -width 20 -relief sunken -textvariable \
                        rls
##########

message .lib.lib2 -width 500 -justify left \
-text "What is the total annual effort \
(in Person Month) spent searching the reuse library?" -padx 0
entry .lib.cmd1 -width 20 -relief sunken -textvariable \
sr
##########

message .lib.lib3 -width 500 -justify left \
-text "Wat is the current size of the  \
reuse library (number of components)?" -padx 0
entry .lib.cmd2 -width 20 -relief sunken -textvariable \
n
##########


button .lib.lib4 -text ok \
 -command {
                  set stor [expr (12*$rls/$n) + ($sr/$n)]
          ##########
  destroy .lib}

button .lib.lib5 -text cancel \
-command {
destroy .lib
set rls 0
set sr 0
set n 0
set stor 0}

                        pack .lib.lib1 .lib.cmd .lib.lib2 .lib.cmd1 .lib.lib3 \
     .lib.cmd2 -side top -expand true
pack .lib.lib4 -side left
pack .lib.lib5 -side right
}

### Exact Approx. Retieval ########
.mbar.reuse.menu.strategic.library add checkbutton -label "Exact/Approx Retrieval" \
-command  {
                        toplevel .retv
                        wm title .retv "Exact/Approx.Retrieval"
                        wm geometry .retv 550x150

                        ##########
          set pdperpm 0.05
                                set exact 0
set aprox 0

                                set extab(1,1) [expr 1.0 * $pdperpm]
```

```
                                    set extab(1,2) [expr 0.8 * $pdperpm]
                                    set extab(1,3) [expr 0.5 * $pdperpm]
                                    set extab(2,1) [expr 1.5 * $pdperpm]
                                    set extab(2,2) [expr 1.0 * $pdperpm]
                                    set extab(2,3) [expr 0.6 * $pdperpm]
                                    set extab(3,1) [expr 2.0 * $pdperpm]
                                    set extab(3,2) [expr 1.2 * $pdperpm]
                                    set extab(3,3) [expr 0.7 * $pdperpm]

                                    set aptab(1,1) [expr 2.0 * $pdperpm]
                                    set aptab(1,2) [expr 1.6 * $pdperpm]
                                    set aptab(1,3) [expr 1.0 * $pdperpm]
                                    set aptab(2,1) [expr 3.0 * $pdperpm]
                                    set aptab(2,2) [expr 2.0 * $pdperpm]
                                    set aptab(2,3) [expr 1.2 * $pdperpm]
                                    set aptab(3,1) [expr 4.0 * $pdperpm]
                                    set aptab(3,2) [expr 2.4 * $pdperpm]
                                    set aptab(3,3) [expr 1.4 * $pdperpm]

                                    if {$n <= 100} \
   { set sizerange 1} \
   elseif {$n <= 500} \
  { set sizerange 2} \
  else {set sizerange 3}

##########

message .retv.retv1 -width 500 -justify left \
-text "Do you know the exact values of \
exact retrieval and approximate retrieval?" -padx 0
button .retv.retv2 -text Yes \
-command { toplevel .exap
          wm title .exap "Exact/Approx. Retrieval Values"
          wm geometry .exap 550x150

          message .exap.exap1 -width 500 -justify left \
  -text "Give the cost of exact \
     retrieval (person days)"
                                entry .exap.cmd -width 20 -relief sunken -textvariable\
  exact
  ##########

         message .exap.exap2 -width 500 -justify left \
  -text "Give the cost of approximate \
     retrieval (person days)"
                                entry .exap.cmd1 -width 20 -relief sunken -textvariable\
  aprox
                                ##########


  button .exap.ok -text ok \
    -command {
                                                set exact [expr $exact * $pdperpm]
                                                set aprox [expr $aprox * $pdperpm]
```

```
            ##########
      destroy .exap
      destroy .retv
      }

                                  button .exap.cancel -text cancel \
    -command { destroy .exap
       set exact 0
       set aprox 0}
                                  pack .exap.exap1 .exap.cmd .exap.exap2\
      .exap.cmd1 -side top -expand true
                                  pack .exap.ok -side left
  pack .exap.cancel -side right
                              }

                       button .retv.retv3 -text No \
-command { toplevel .deflt
 wm title .deflt "Exact/Approx. Retrieval: Default Values"
                                 wm geometry .deflt 550x150
 message .deflt.deflt1 -width 500 -justify left \
 -text "Tell what method is used for \
   searching your reuse:"

                                button .deflt.opt1 -text "By Hand Inspection" \
 -command {
  set searchmethod 1
                                        set exact $extab($sizerange,$searchmethod)
         set aprox $aptab($sizerange,$searchmethod)
  ##########

  toplevel .byhand
  wm title .byhand "By Hand Inspection"
  wm geometry .byhand 550x150
  message .byhand.byhand1 -width 500 -justify left \
  -text "We have taken the \
     following values for exact and approximate \
     retrieval: \n          exact retrieval: $exact, \
     \n          approximate retrieval: $aprox. "

  button .byhand.ok -text Ok \
  -command {destroy .byhand
    destroy .deflt
    destroy .retv
                }
                                        button .byhand.cancel -text Cancel \
  -command { set exact 0
     set aprox 0
     destroy .byhand
                                                }
                                        pack .byhand.byhand1 -side top
  pack .byhand.ok -side left
  pack .byhand.cancel -side right

  }
```

```
                                button .deflt.opt2 -text "Hypertext Methods" \
-command {
 set searchmethod 2
                                          set exact $extab($sizerange,$searchmethod)
        set aprox $aptab($sizerange,$searchmethod)
 ##########

 toplevel .hypert
 wm title .hypert "Hypertext Methods"
 wm geometry .hypert 550x150
 message .hypert.hypert1 -width 500 -justify left \
 -text "We have taken the \
    following values for exact and approximate \
    retrieval: \n          exact retrieval: $exact, \
    \n          approximate retrieval: $aprox. "

 button .hypert.ok -text Ok \
 -command {destroy .hypert
   destroy .deflt
   destroy .retv
                 }
                                          button .hypert.cancel -text Cancel \
 -command { set exact 0
    set aprox 0
    destroy .hypert
                                                  }
                                          pack .hypert.hypert1 -side top
 pack .hypert.ok -side left
 pack .hypert.cancel -side right

       }
                               button .deflt.opt3 -text "Automatic" \
-command {
 set searchmethod 3
                                          set exact $extab($sizerange,$searchmethod)
        set aprox $aptab($sizerange,$searchmethod)
 ##########

 toplevel .auto
 wm title .auto "Automatic"
 wm geometry .auto 550x150
 message .auto.auto1 -width 500 -justify left \
 -text "We have taken the \
    following values for exact and approximate \
    retrieval: \n          exact retrieval: $exact, \
    \n          approximate retrieval: $aprox. "

 button .auto.ok -text Ok \
 -command {destroy .auto
   destroy .deflt
   destroy .retv
                 }
                                          button .auto.cancel -text Cancel \
```

```
  -command { set exact 0
     set aprox 0
     destroy .auto
                                                  }
                                              pack .auto.auto1 -side top
  pack .auto.ok -side left
  pack .auto.cancel -side right

          }
 button .deflt.cancel -text "Cancel" \
                  -command {
  set searchmethod 0
  destroy .deflt
           }
                                  pack .deflt.deflt1 .deflt.opt1 .deflt.opt1 \
      .deflt.opt2 .deflt.opt3 -side top
                                  pack .deflt.cancel -side right

                                 }
                        button .retv.retv4 -text Cancel \
       -command { destroy .retv
}
                        pack .retv.retv1 -side top
       pack .retv.retv2 -side left
       pack .retv.retv3 -side left
       pack .retv.retv4 -side right

                     }

############################DOMAIN FACTORS#####################################
#### Frequencies  ###
menu .mbar.reuse.menu.domain
.mbar.reuse.menu.domain configure -fg SeaGreen

.mbar.reuse.menu.domain add checkbutton -label "  Frequencies  " \
-command  {
       set deflambda 0.62
       ##########

                        toplevel .freqs
                        wm title .freqs "Frequencies of Reuse"
                        wm geometry .freqs 550x150

                       message .freqs.freqs1 -width 500 -justify left \
-text "Do you know what is the \
         frequency of exact retrieval and the frequency of approximate\
         retrieval?" -padx 0
                        button .freqs.yes -text Yes \
-command {
toplevel .freqvals
wm title .freqvals "Frequencies: Enter Values"
wm geometry .freqvals 550x150

message .freqvals.freqvals1 -width 500 -justify left \
```

```
-text \
"Give the annual frequency of exact retrieval:" -padx 0
entry .freqvals.cmd -width 20 -relief sunken -textvariable\
    freqx
                        ##########

message .freqvals.freqvals2 -width 500 -justify left \
-text \
"Give the annual frequency of approximate retrieval:" \
  -padx 0
entry .freqvals.cmd2 -width 20 -relief sunken -textvariable\
    freqa
                        ##########


button .freqvals.ok -text ok \
 -command {
                                    set freq [expr $freqx + $freqa]
                                set lambda [expr $freqx/$freqa]
   ##########

   destroy .freqvals
           destroy .freqs
          }

button .freqvals.cancel -text cancel \
-command {
destroy .freqvals
set freqx 0
set freqa 0
set freq  0
set lambda 0}

pack .freqvals.freqvals1 .freqvals.cmd \
            .freqvals.freqvals2 .freqvals.cmd2 -side top -expand true
                        pack .freqvals.ok -side left
pack .freqvals.cancel -side right
}

button .freqs.no -text No \
-command {
              toplevel .freqdeflt
                              wm title .freqdeflt "Frequencies: Default Values"
                              wm geometry .freqdeflt 550x150

                              message .freqdeflt.freqdeflt1 -width 500 -justify left \
     -text "We have a default \
             value for the ratio between exact frequency and \
     total reuse frequency: $deflambda. \
     Thanks to this default value, it suffices that we\
     get an estimate of the total reuse frequency.\
     Do you have an estimate of the total frequency?" \
     -padx 0
```

```
                              button .freqdeflt.yes -text Yes \
         -command {
        toplevel .ratio
        wm title .ratio "Annual Frequency: Enter Values"
        wm geometry .ratio 550x150

               message .ratio.ratio1 -width 500 -justify left \
-text \
"Give an estimate of the total annual frequency \
 of reuse:" -padx 0
entry .ratio.cmd -width 20 -relief sunken \
    -textvariable \
freq
                                        ##########

button .ratio.ok -text ok \
 -command {
                                                    set lambda $deflambda
                       set freqx [expr $lambda * $freq]
                  set freqa [expr (1-$lambda) * $freq]
                                                    ##########

   destroy .ratio
           destroy .freqdeflt
   destroy .freqs
                     }

button .ratio.cancel -text cancel \
-command {
destroy .ratio
set freqx 0
set freqa 0
set freq  0
set lambda 0}

pack .ratio.ratio1 .ratio.cmd \
                               -side top -expand true
                                pack .ratio.ok -side left
pack .ratio.cancel -side right
}


     button .freqdeflt.no -text No \
  -command {
       toplevel .anfreq
       wm title .anfreq "Annual Frequency: Default Value"
       wm geometry .anfreq 550x150

              message .anfreq.anfreq1 -width 500 -justify left \
-text \
"For the sake of this calculation, we will take \
 the total annual frequency of reuse to be 3,\
 which amounts to about two exact retrievals \
 and one approximate retrieval every year. \
```

```
  You may want to compare this against the\
  breakeven frequency, which is given by the system. \
  You may also consider that you will be given an \
  estimate of the intrinsic reusability of this \
  component, which is independent of the reuse  \
  frequency." -padx 0

 button .anfreq.ok -text ok \
  -command {
                                                    set freq 3
                    set lambda deflambda
                    set freqx [expr $lambda * $freq]
                    set freqa [(1- $lambda) * $freq]
    ##########

    destroy .anfreq
            destroy .freqdeflt
    destroy .freqs
                      }

 button .anfreq.cancel -text cancel \
 -command {
 destroy .anfreq
 set freqx 0
 set freqa 0
 set freq  0
 set lambda 0}

  pack .anfreq.anfreq1 -side top
                                          pack .anfreq.ok -side left
  pack .anfreq.cancel -side right
            }

       button .freqdeflt.cancel -text cancel \
           -command {
 destroy .freqdeflt
 set freqx 0
 set freqa 0
 set freq  0
 set lambda 0}

                         pack .freqdeflt.freqdeflt1 -side top
                   pack .freqdeflt.yes -side left
   pack .freqdeflt.no -side left
   pack .freqdeflt.cancel -side right
          }

           button .freqs.cancel -text Cancel \
 -command {
 destroy .freqs
 set freqx 0
 set freqa 0
 set freq  0
 set lambda 0}
```

```
                    pack .freqs.freqs1 -side top
    pack .freqs.yes -side left
    pack .freqs.no -side left
    pack .freqs.cancel -side right
                   }

 ########################COMPONENT  FACTORS#####################################
 #### DEVELOPMENT EFFORT  ###
 menu .mbar.reuse.menu.component
 .mbar.reuse.menu.component configure -fg SeaGreen

 .mbar.reuse.menu.component add checkbutton -label " Development Effort " \
 -command  {
                          toplevel .devef
                          wm title .devef "Development Effort"
                          wm geometry .devef 550x150

                         message .devef.devef1 -width 500 -justify left \
 -text "Give an estimate of the development effort\
 for this component (in person months)" -padx 0
 entry .devef.com -width 20 -relief sunken \
   -textvariable de

 button .devef.ok -text ok \
  -command {destroy .devef.devef1
            destroy .devef
           }

 button .devef.cancel -text cancel \
 -command {
 set de 0
 destroy .devef.devef1
 destroy .devef
 }

 pack .devef.devef1 .devef.com   \
                            -side top -expand true
                        pack .devef.ok -side left
 pack .devef.cancel -side right
 }

 #### REUSE RATE ####
 .mbar.reuse.menu.component add checkbutton -label " Reuse Rate " \
 -command  {
                          toplevel .rate
                          wm title .rate "Reuse Rate"
                          wm geometry .rate 550x300

                        message .rate.rate1 -width 500 -justify left \
 -text "Rate the extent of software reuse in \
   your organization:" -padx 0

 button .rate.cancel -text cancel \
```

```
-command {
set ruse 0
destroy .rate.rate1
destroy .rate
}

button .rate.ok -text ok \
-command {
set dr [expr $de*$ruse]
##########
destroy .rate.rate1
destroy .rate
}

                        button .rate.opt1 -text "No Reuse" \
 -command {
                                  set ruse 1
  ##########
  }

                        button .rate.opt2 -text "Reuse across a project" \
 -command {
  set ruse 1.15
  ##########
        }

                        button .rate.opt3 -text "Reuse across a program" \
 -command {
  set ruse 1.49
  ##########
         }

                        button .rate.opt4 -text "Reuse across a product line" \
 -command {
  set ruse 2.10
  ##########
         }

                        button .rate.opt5 -text "Reuse across a many product lines" \
 -command {
         set ruse 2.45
  ##########
           }

              pack .rate.rate1 -side top
      pack .rate.opt1 .rate.opt2 .rate.opt3 .rate.opt4 .rate.opt5 \
     -side top
              pack .rate.cancel -side right
      pack .rate.ok -side left
              }

#### Reengineering Cost####
.mbar.reuse.menu.component add cascade -label " Reengineering Cost " \
       -menu .mbar.reuse.menu.component.reeng
```

```
menu .mbar.reuse.menu.component.reeng
.mbar.reuse.menu.component.reeng configure -fg SeaGreen
##########S. Understanding #####
.mbar.reuse.menu.component.reeng add checkbutton -label " Software Understanding " \
-command  "exec wish -f table"

######### D. Modification ######
.mbar.reuse.menu.component.reeng add checkbutton -label " Design Modification " \
       -command {
                        toplevel .dm
                        wm title .dm "Design Modification"
                        wm geometry .dm 550x150

                       message .dm.dm1 -width 500  -justify left -text  \
  "Provide below the portion of design that you expect\
   to modify as part of reengineering effort (percentage)"\
   -padx 0
entry .dm.com -width 20 -relief sunken \
  -textvariable dmx
##########

button .dm.cancel -text cancel \
-command {
set dm 0
destroy .dm.dm1
destroy .dm
}

button .dm.ok -text ok \
-command {
                                       set dm [expr $dmx/100]
###########
destroy .dm.dm1
destroy .dm
}

                    pack .dm.dm1 .dm.com -side top
      pack .dm.ok -side left
      pack .dm.cancel -side right

}

########## C. Modification #####
.mbar.reuse.menu.component.reeng add checkbutton -label " Code Modification " \
       -command {
                        toplevel .cm
                        wm title .cm "Design Modification"
                        wm geometry .cm 550x150

                       message .cm.cm1 -width 500  -justify left -text  \
  "Provide below the portion of coding that you expect\
   to modify as part of reengineering effort (percentage)"\
   -padx 0
```

```
entry .cm.com -width 20 -relief sunken \
  -textvariable cmx
##########

button .cm.cancel -text cancel \
-command {
set cm 0
destroy .cm.cm1
destroy .cm
}

button .cm.ok -text ok \
-command {
                        set cm [expr $cmx/100]
##########
destroy .cm.cm1
destroy .cm
}

                       pack .cm.cm1 .cm.com -side top
      pack .cm.ok -side left
      pack .cm.cancel -side right

}

###Reengineering Costs###
.mbar.reuse.menu.component.reeng add checkbutton -label " Reengineering Costs" \
       -command {
                         toplevel .reengcost
                         wm title .reengcost "Reengineering Costs"
                         wm geometry .reengcost 550x150

                         set reeng [expr ($su + (0.4 * $dm) + (0.3 * $cm)) * $de]
##########

                        message .reengcost.reengcost1 -width 500  -justify left -text  \
   "For reengineering costs, we have found $reeng. \
   Do you wish to override this estimate ? " \
   -padx 0

button .reengcost.cancel -text cancel \
-command {
set reeng 0
##########
destroy .reengcost.reengcost1
destroy .reengcost
}

button .reengcost.yes -text Yes \
-command {

                                toplevel .noreeng
                                wm title .noreeng ""
                                wm geometry .noreeng 550x150
```

```
                                message .noreeng.noreeng1 -width 500  -justify left -text  \
                "Give the adjusted estimate of the reengineering costs \
  in person months: " \
                 -padx 0
                entry .noreeng.com -width 20 -relief sunken \
         -textvariable reeng
       ##########

     button .noreeng.cancel -text cancel \
 -command {
 set reeng 0
 ##########
 destroy .noreeng.noreeng1
 destroy .noreeng
 }

 button .noreeng.ok -text ok \
 -command {
 destroy .noreeng.noreeng1
 destroy .noreeng
 destroy .reengcost
 }

                              pack .noreeng.noreeng1 .noreeng.com -side top
             pack .noreeng.ok -side left
           pack .noreeng.cancel -side right
                               }


 button .reengcost.no -text No \
 -command {
 destroy .reengcost.reengcost1
 destroy .reengcost
 }
       pack .reengcost.reengcost1 -side top
       pack .reengcost.cancel -side right
       pack .reengcost.yes -side left
       pack .reengcost.no -side left
 }
 ###VERIFICATION COST###
 .mbar.reuse.menu.component add checkbutton -label " Verification Cost " \
       -command {
                         toplevel .verifcost
                         wm title .verifcost "Verification Cost"
                         wm geometry .verifcost 550x150

                         set verif [expr 0.21 * $ruse * $de]
 ##########

                        message .verifcost.verifcost1 -width 500  -justify left -text  \
  "We are interested in estimating the verification costs \
   associated with reengineering this component. Our default \
   formula estimates these costs at: $verif. \
```

```
   Do you wish to override this estimate ? " \
   -padx 0

button .verifcost.cancel -text cancel \
-command {
set verif 0
##########
destroy .verifcost.verifcost1
destroy .verifcost
}

button .verifcost.yes -text Yes \
-command {

                              toplevel .noverf
                              wm title .noverf ""
                              wm geometry .noverf 550x150

                              message .noverf.noverif1 -width 500  -justify left -text  \
               "Give the adjusted estimate of the verification costs \
 in person months: " \
                -padx 0
               entry .noverf.com -width 20 -relief sunken \
       -textvariable verif
     ##########

    button .noverf.cancel -text cancel \
-command {
set verif 0
##########
destroy .noverf.noverif1
destroy .noverf
}

button .noverf.ok -text ok \
-command {
destroy .noverf.noverif1
destroy .noverf
destroy .verifcost
}

                           pack .noverf.noverif1 .noverf.com -side top
            pack .noverf.ok -side left
          pack .noverf.cancel -side right
                            }


button .verifcost.no -text No \
-command {
destroy .verifcost.verifcost1
destroy .verifcost
}
      pack .verifcost.verifcost1 -side top
      pack .verifcost.cancel -side right
```

```
        pack .verifcost.yes -side left
        pack .verifcost.no -side left
}

###BASELINING COST###
.mbar.reuse.menu.component add checkbutton -label " Baselining Cost " \
        -command {
                          toplevel .base
                          wm title .base "Baselining Cost"
                          wm geometry .base 550x150

                          set defbase 3.5
                          set base [expr  $defbase * $pdperpm ]
 ##########

                         message .base.base1 -width 500  -justify left -text  \
  "The default value for baselining costs is $defbase\
   person days, which is $base person months. Do you \
   wish to override this estimate ? " \
   -padx 0

button .base.cancel -text cancel \
-command {
set base 0
##########
destroy .base.base1
destroy .base
}

button .base.yes -text Yes \
-command {

                                toplevel .nobase
                                wm title .nobase ""
                                wm geometry .nobase 550x150

                                message .nobase.nobase1 -width 500  -justify left -text  \
                "Give an alternative estimate in\
 in person days: " \
                 -padx 0
                entry .nobase.com -width 20 -relief sunken \
        -textvariable base
      ##########

    button .nobase.cancel -text cancel \
-command {
set base 0
##########
destroy .nobase.nobase1
destroy .nobase
}

button .nobase.ok -text ok \
-command {
```

```
destroy .nobase.nobase1
destroy .nobase
destroy .base
}

                              pack .nobase.base1 .nbase.com -side top
            pack .nobase.ok -side left
          pack .nobase.cancel -side right
                               }


button .base.no -text No \
-command {
destroy .base.base1
destroy .base
}
       pack .base.base1 -side top
       pack .base.cancel -side right
       pack .base.yes -side left
       pack .base.no -side left
}


#### CUSTOMIZATION COSTS ####
.mbar.reuse.menu.component add cascade -label " Customization Costs " \
       -menu .mbar.reuse.menu.component.cust

menu .mbar.reuse.menu.component.cust
.mbar.reuse.menu.component.cust configure -fg SeaGreen

########### Instantiation costs ############
.mbar.reuse.menu.component.cust add checkbutton -label " Instanciation Costs " \
       -command {
                         toplevel .inst
                         wm title .inst "Instanciation Costs"
                         wm geometry .inst 550x150

                        set instance [expr 0.046 * $de]
##########

                        message .inst.inst1 -width 500  -justify left -text  \
  "Our estimate of the average/typical instanciation cost \
   for this component is: $instance person months. \nDo you wish \
   to override it? " \
   -padx 0

button .inst.cancel -text cancel \
-command {
set instance 0
destroy .inst.inst1
destroy .inst
}

button .inst.yes -text Yes \
```

```
 -command {

                                toplevel .noinst
                                wm title .noinst "Override Instanciation Costs"
                                wm geometry .noinst 550x150

                                message .noinst.noinst1 -width 500  -justify left -text  \
                  "Provide an alternative value of typical/average \
  instanciation costs in person months: " \
                    -padx 0
                  entry .noinst.com -width 20 -relief sunken \
          -textvariable instance
        ##########

     button .noinst.cancel -text cancel \
 -command {
 set instance 0
 ##########
 destroy .noinst.noinst1
 destroy .noinst
 }

 button .noinst.ok -text ok \
 -command {
 destroy .noinst.noinst1
 destroy .noinst
 destroy .inst
 }

                              pack .noinst.noinst1 .noinst.com -side top
              pack .noinst.ok -side left
            pack .noinst.cancel -side right
                               }


button .inst.no -text No \
-command {
destroy .inst.inst1
destroy .inst
}
       pack .inst.inst1 -side top
       pack .inst.cancel -side right
       pack .inst.yes -side left
       pack .inst.no -side left
}

########### Adaptation costs ############
.mbar.reuse.menu.component.cust add checkbutton -label " Adaptation Costs " \
       -command {
                        toplevel .adapt
                        wm title .adapt "Adaptation Costs"
                        wm geometry .adapt 550x150

                        set adapt [expr 0.67  * $de ]
```

```
 **********

                        message .adapt.adapt1 -width 500  -justify left -text  \
  "Our estimate of the average/typical adaptation cost \
   for this component is: $adapt  person months. \nDo you wish \
   to override it? " \
   -padx 0

button .adapt.cancel -text cancel \
-command {
set adapt 0
destroy .adapt.adapt1
destroy .adapt
}

button .adapt.yes -text Yes \
-command {

                               toplevel .noadapt
                               wm title .noadapt "Override Adaptation Costs"
                               wm geometry .noadapt 550x150

                               message .noadapt.noadapt1 -width 500  -justify left -text  \
                "Provide an alternative value of typical/average \
 adaptation costs in person months: " \
                 -padx 0
                entry .noadapt.com -width 20 -relief sunken \
       -textvariable adapt
     **********

    button .noadapt.cancel -text cancel \
-command {
set adapt 0
**********
destroy .noadapt.noadapt1
destroy .noadapt
}

button .noadapt.ok -text ok \
-command {
destroy .noadapt.noadapt1
destroy .noadapt
destroy .adapt
}

                             pack .noadapt.noadapt1 .noadapt.com -side top
           pack .noadapt.ok -side left
          pack .noadapt.cancel -side right
                              }


button .adapt.no -text No \
-command {
destroy .adapt.adapt1
```

```
destroy .adapt
}
        pack .adapt.adapt1 -side top
        pack .adapt.cancel -side right
        pack .adapt.yes -side left
        pack .adapt.no -side left
}
########### Integration and Test ############
.mbar.reuse.menu.component.cust add checkbutton -label " Integration and Test Costs " \
        -command {
                          toplevel .intg
                           wm title .intg "Integration and Test Costs"
                           wm geometry .intg 550x150

                          set rest [expr 0.19 * $de]
##########

                          message .intg.intg1 -width 500  -justify left -text  \
  "Our estimate of the average/typical integration and test cost \
   for this component is: $rest  person months. \nDo you wish \
   to override it? " \
   -padx 0

button .intg.cancel -text cancel \
-command {
set rest 0
destroy .intg.intg1
destroy .intg
}

button .intg.yes -text Yes \
-command {

                                  toplevel .nointg
                                  wm title .nointg "Override Instanciation Costs"
                                  wm geometry .nointg 550x150

                                  message .nointg.nointg1 -width 500  -justify left -text  \
                "Provide an alternative value of typical/average \
 integration and test costs in person months: " \
                 -padx 0
                entry .nointg.com -width 20 -relief sunken \
        -textvariable rest
  set first [expr $de -$rest]
          ##########

    button .nointg.cancel -text cancel \
-command {
set rest 0
destroy .nointg.nointg1
destroy .nointg
}

button .nointg.ok -text ok \
```

```
-command {
destroy .nointg.nointg1
destroy .nointg
destroy .intg
}

                              pack .nointg.nointg1 .nointg.com -side top
            pack .nointg.ok -side left
          pack .nointg.cancel -side right
                               }


button .intg.no -text No \
-command {
destroy .intg.intg1
destroy .intg
}
       pack .intg.intg1 -side top
       pack .intg.cancel -side right
       pack .intg.yes -side left
       pack .intg.no -side left
}


#### QUALITY FACTOR COSTS ####
.mbar.reuse.menu.component add cascade -label " Operational Costs " \
       -menu .mbar.reuse.menu.component.qual

menu .mbar.reuse.menu.component.qual
.mbar.reuse.menu.component.qual configure -fg SeaGreen

########### Reengineered Component ############
.mbar.reuse.menu.component.qual add checkbutton -label " Reengineered Component " \
       -command {
                          toplevel .recomp
                          wm title .recomp "Operational Costs: Reengineered Component"
                          wm geometry .recomp 550x150

                         set oper [expr 0.067 * $de]
##########

                         message .recomp.recomp1 -width 500  -justify left -text  \
  "Our estimate of the annual operating cost of this component \
   if it were used after reengineering is :$oper   person months. \nDo \
   you wish to override this estimate ?" \
   -padx 0

button .recomp.cancel -text cancel \
-command {
set oper 0
##########
destroy .recomp.recomp1
destroy .recomp
}
```

```
button .recomp.yes -text Yes \
-command {

                                toplevel .norecomp
                                wm title .norecomp "Override Operational Costs For\
   Reengineered Component"
                                wm geometry .norecomp 550x150

                                message .norecomp.norecomp1 -width 500  -justify left -text  \
                "Provide an alternative value of the annual operating \
 cost of this component if it were used after reengineering \
 (in person months): " \
                 -padx 0
                entry .norecomp.com -width 20 -relief sunken \
       -textvariable oper
      ##########

    button .norecomp.cancel -text cancel \
-command {
set oper 0
destroy .norecomp.norecomp1
destroy .norecomp
}

button .norecomp.ok -text ok \
-command {
destroy .norecomp.norecomp1
destroy .norecomp
destroy .recomp
}

                              pack .norecomp.norecomp1 .norecomp.com -side top
            pack .norecomp.ok -side left
          pack .norecomp.cancel -side right
                               }


button .recomp.no -text No \
-command {
destroy .recomp.recomp1
destroy .recomp
}
       pack .recomp.recomp1 -side top
       pack .recomp.cancel -side right
       pack .recomp.yes -side left
       pack .recomp.no -side left
}
########### Component as is ############
.mbar.reuse.menu.component.qual add checkbutton -label " Component As Is " \
       -command {
                          toplevel .asis
                          wm title .asis "Operational Costs: Component As Is"
                          wm geometry .asis 550x150
```

```
                        set operprime [expr 0.1 * $de]
##########

                        message .asis.asis1 -width 500  -justify left -text  \
  "Our estimate of the annual operating cost of this component \
   if it were used as is : $operprime person months. \nDo \
   you wish to override this estimate ?" \
   -padx 0

button .asis.cancel -text cancel \
-command {
set operprime 0
destroy .asis.asis1
destroy .asis
}

button .asis.yes -text Yes \
-command {

                                toplevel .noasis
                                wm title .noasis "Override Operational Costs For \
   For Component Used As Is"
                                wm geometry .noasis 550x150

                                message .noasis.noasis1 -width 500  -justify left -text  \
               "Provide an alternative value of the annual operating \
 cost of this component if it were used as is\
 (in person months): " \
                -padx 0
               entry .noasis.com -width 20 -relief sunken \
        -textvariable operprime

    button .noasis.cancel -text cancel \
-command {
set operprime 0
destroy .noasis.noasis1
destroy .noasis
}

button .noasis.ok -text ok \
-command {
destroy .noasis.noasis1
destroy .noasis
destroy .asis
}

                            pack .noasis.noasis1 .noasis.com -side top
            pack .noasis.ok -side left
          pack .noasis.cancel -side right
                             }


button .asis.no -text No \
```

```
-command {
destroy .asis.asis1
destroy .asis
}
       pack .asis.asis1 -side top
       pack .asis.cancel -side right
       pack .asis.yes -side left
       pack .asis.no -side left
}

###### DISPLAY REUSE COSTS ##########
menu .mbar.reusecosts.menu
.mbar.reusecosts.menu configure -fg SeaGreen

.mbar.reusecosts.menu add radiobutton -label "   Values   " \
    -command {
                        toplevel .dispvars
                        wm title .dispvars " Display Reuse Costs "
                        wm geometry .dispvars 400x550

                       message .dispvars.dispvars1 -width 500  -justify left -text  \
  " STRATEGIC FACTORS \
 \n    Investment Cycle              :    $invcycle. \
 \n    Discount Rate                 :    $discount. \
 \n \
 \n LIBRARY FACTORS \
                          \n    Size of Library               : $n components. \
 \n    Searching Cost                : $sr person months. \
 \n    Cost of Exact Retrieval       : $exact person months. \
 \n    Cost of Approx. Retrieval     : $aprox person months. \
                          \n \
 \n DOMAIN FACTORS \
 \n    Freq. of Exact Retrieval      : $freqx. \
 \n    Freq. of Approx. Retrieval    : $freqa. \
 \n \
 \n COMPONENT FACTORS \
 \n    Reuse Rate                    : $ruse. \
 \n    Reengineering Costs           : $reeng person months. \
 \n          Software Understanding  : $su person months. \
 \n          Modified Design         : $dm %. \
 \n          Modified Code           : $cm %. \
 \n    Verification Costs            : $verif person months. \
 \n    Baselining Costs              : $base person months. \
 \n    Customization Costs \
         \n       Instantiation Costs     : $instance person months. \
 \n          Adaptation Costs        : $adapt person months. \
 \n          Integration and Test    : $rest person months. \
 \n    Quality Factors \
 \n          Annual Operating Cost Without Reuse: $operprime person months \
 \n          Annual Operating Costs With Reuse  : $oper person months. \
 "

button .dispvars.click -text "click here"\
-command {
```

```
destroy .dispvars.dispvars1
destroy .dispvars
}
                        pack .dispvars.dispvars1 -side top
       pack .dispvars.click -side right
     }

.mbar.reusecosts.menu add radiobutton -label " Graph  Representation " \
    -command {
     exec  wish -f error &
     }

######## REUSE FORMULAE ####################
#define the options for the lifecycle menu
menu .mbar.lifecycle.menu
.mbar.lifecycle.menu configure -fg SeaGreen

.mbar.lifecycle.menu add cascade -label "Limited Term Reusability" \
   -menu .mbar.lifecycle.menu.short

.mbar.lifecycle.menu add cascade -label " Long Term Reusability " \
   -menu .mbar.lifecycle.menu.long

#### define menu for limited term reusability ####
menu .mbar.lifecycle.menu.short
  .mbar.lifecycle.menu.short configure -fg SeaGreen

.mbar.lifecycle.menu.short add radiobutton -label " Absolute Reusability " \
   -command {
    set ic [expr $reeng +$verif + $base]

    set absolutereu \
[expr ((exp([expr $invcycle*log([expr 1+$discount])])-1) \
      / \
      ($ic*$discount*exp($invcycle*log(1+$discount)))) \
      * \
      ($freqx*($first-$exact-$instance) \
       +$freqa*($first-$aprox-$adapt) \
       -$freq*($operprime-$oper) \
       -$stor) \
                    + \
      ($freq*($operprime-$oper)) \
      * \
      ((exp((1+$invcycle)*log(1+$discount)) \
-$discount*$invcycle \
-(1+$discount)) \
      / \
      ($ic*$discount*$discount*exp($invcycle*log(1+$discount)))) \
      - \
      1]
          ##################

          toplevel .abs
          wm title .abs " Absolute Reusability "
```

```
            wm geometry .abs 450x150

            message .abs.abs1 -width 500  -justify left -text  \
    "The Absolute Reusability for this component is : $absolutereu. "


            button .abs.click -text "click here"\
 -command {
 destroy .abs.abs1
 destroy .abs
 }
            pack .abs.abs1 -side top
            pack .abs.click -side right
     }

 .mbar.lifecycle.menu.short add radiobutton -label " Intrinsic Reusability " \
    -command {
     set ic [expr $reeng +$verif + $base]
     set oneplusd [expr 1+$discount]
     set oneplusdpowery [expr exp($invcycle*log($oneplusd))]
     set dsquare [expr $discount * $discount]
     ###

     set intrinsicreu \
 [expr (($oneplusdpowery -1) / ($ic*$discount*$oneplusdpowery)) \
 * \
 ($lambda*($aprox+$adapt-$exact-$instance) + \
       ($first -$aprox-$adapt-$operprime+$oper)) \
                + \
 (($oneplusd*$oneplusdpowery-$discount*$invcycle-$oneplusd) \
      /($ic*$dsquare*$oneplusdpowery)) \
                * \
 ($operprime -$oper)]
            ##########

            toplevel .int
            wm title .int " Intrinsic Reusability "
            wm geometry .int 450x150

            message .int.int1 -width 500  -justify left -text  \
    "The Intrinsic Reusability for this component is : $intrinsicreu. "


            button .int.click -text "click here"\
 -command {
 destroy .int.int1
 destroy .int
 }
            pack .int.int1 -side top
            pack .int.click -side right
     }

 .mbar.lifecycle.menu.short add radiobutton -label " Break Even Frequency " \
    -command {
```

```
    set ic [expr $reeng +$verif + $base]
    set oneplusd [expr 1+$discount]
    set oneplusdpowery [expr exp($invcycle*log($oneplusd))]
    set dsquare [expr $discount * $discount]
    ###

    set intrinsicreu \
[expr (($oneplusdpowery -1) / ($ic*$discount*$oneplusdpowery)) \
* \
($lambda*($aprox+$adapt-$exact-$instance) + \
      ($first -$aprox-$adapt-$operprime+$oper)) \
               + \
(($oneplusd*$oneplusdpowery-$discount*$invcycle-$oneplusd) \
     /($ic*$dsquare*$oneplusdpowery)) \
               * \
($operprime -$oper)]
          ##########

   set breakevenfreq \
[expr (1/$intrinsicreu) \
* \
(1+($stor/$ic)  \
  * \
  (($oneplusdpowery -1) \
           / \
    ($discount*$oneplusdpowery)))]
          ##########

          toplevel .break
          wm title .break " Break Even Frequency"
          wm geometry .break 450x150

          message .break.break1 -width 500  -justify left -text  \
   "The Break Even Frequency for this component is : $breakevenfreq. "


          button .break.click -text "click here"\
-command {
destroy .break.break1
destroy .break
}
          pack .break.break1 -side top
          pack .break.click -side right
    }

#### define menu for long term reusability ####
menu .mbar.lifecycle.menu.long
.mbar.lifecycle.menu.long configure -fg SeaGreen

.mbar.lifecycle.menu.long add radiobutton -label "Long Term Absolute \
  Reusability " \
   -command {
    set ic [expr $reeng +$verif + $base]
    ####
```

```
    set longtermabsreu \
[expr (1/$ic*$discount)* \
 ($freqx*($first-$exact-$instance)+ \
  $freqa*($first-$aprox-$adapt)- \
  $freq*($operprime-$oper)- \
  $stor) \
  + \
  ((1+$discount)/($ic*$discount*$discount)) \
  *$freq*($operprime-$oper) \
  - \
  1]
          toplevel .abslong
          wm title .abslong " Absolute Reusability "
          wm geometry .abslong 450x150

          message .abslong.abslong1 -width 500  -justify left -text  \
   "The Absolute Reusability for this component is : $longtermabsreu. "


          button .abslong.click -text "click here"\
-command {
destroy .abslong.abslong1
destroy .abslong
}
          pack .abslong.abslong1 -side top
          pack .abslong.click -side right
    }

.mbar.lifecycle.menu.long add radiobutton -label "Long Term Intrinsic \
   Reusability " \
   -command {
    set longtermintreu \
[expr (1/($ic*$discount))* \
 ($lambda*($aprox+$adapt-$exact-$instance) \
 + \
 ($first-$aprox-$adapt-$operprime+$oper)) \
 + \
 ((1+$discount)/($ic*$discount*$discount)) \
  * \
  ($operprime - $oper) \
              ]
          ##########

          toplevel .longint
          wm title .longint "Long Term Intrinsic Reusability "
          wm geometry .longint 450x150

          message .longint.longint1 -width 500  -justify left -text  \
  "The Long Term Intrinsic Reusability for this component is : $longtermintreu. "


          button .longint.click -text "click here"\
-command {
```

```
destroy .longint.longint1
destroy .longint
}
           pack .longint.longint1 -side top
           pack .longint.click -side right

    }

.mbar.lifecycle.menu.long add radiobutton -label "Long Term Break Even  \
     Frequency " \
   -command {
   set longtermbef 0
    }

###### DISPLAY REUSABILITY MEASURES ##########
menu .mbar.reusemeasures.menu
.mbar.reusemeasures.menu config -fg SeaGreen

.mbar.reusemeasures.menu add radiobutton -label "   Values    " \
    -command {
           toplevel .valmes
           wm title .valmes "Display Reusability Measures"
           wm geometry .valmes 550x250

           message .valmes.valmes1 -width 500  -justify left -text  \
   "Short Term Measures for Investment Cycle $invcycle:
   \n            Absolute Reusability:  $absolutereu. \
   \n            Intrinsic Reusability:  $intrinsicreu. \
   \n            Break Even Frequency:   $breakevenfreq. \
   \n \
   \nLong Term Reusability Measures \
   \n            Long Term Absolute Reusability: $longtermabsreu.\
   \n            Long Term Intrinsic Reusability: $longtermintreu. \
   \n            Long Term Break Even Frequency: $longtermbef. \
    "


           button .valmes.click -text "click here"\
-command {
destroy .valmes.valmes1
destroy .valmes
}
           pack .valmes.valmes1 -side top
           pack .valmes.click -side right
    }

.mbar.reusemeasures.menu add radiobutton -label " Graph  Representation " \
    -command {
     exec  wish -f error &
     }

#### realize the final menu setup on the window ####
tk_menuBar .mbar .mbar.reuse .mbar.lifecycle
focus .mbar
```

```
---------------------------------------------------------------

#!/usr/bin/wish -f

### Top frame ###################################
frame .frame2
pack .frame2 -side top

label .frame2.text1 -text "Reengineering costs depend on software understanding.\
       In view of the following table, give a rating for required software \
       understanding:" -padx 0
pack .frame2.text1 -side top
###Initialize variable ##########################
set su 0

### define canvas ###############################
wm title . "Reengineering Cost"
canvas .reeng -width 27c -height 10c -bg plum1 -scrollregion {0 0 1000 350} \
-confine false  -relief raised

#realize the canvas on the screen
pack .reeng
bind .reeng <Any-Motion> { puts "%x,%y"}

### Bottom Frame ################################
frame .frame1
pack .frame1 -side top

button .frame1.opt1 -text "Very Low" \
  -command { set su 0.5 }
     ##########
pack .frame1.opt1 -fill x -side left

button .frame1.opt2 -text "Low" \
  -command { set su 0.4 }
     ##########
pack .frame1.opt2 -fill x -side left

button .frame1.opt3 -text "Nominal" \
  -command { set su 0.3 }
     ##########
pack .frame1.opt3 -fill x -side left

button .frame1.opt4 -text "High" \
  -command { set su 0.2 }
     ##########
pack .frame1.opt4 -fill x -side left

button .frame1.opt5 -text "Extra High" \
  -command { set su 0.1 }
            ###########
pack .frame1.opt5 -fill x -side left
```

```
frame .frame3
pack .frame3 -side bottom

button .frame3.ok -text ok \
    -command {
     destroy .
     }
pack .frame3.ok -side left

button .frame3.cancel -text cancel \
    -command {
     set su 0
     destroy .
     }
pack .frame3.cancel -side right

### Canvas ########################################

.reeng create line 50 34 925 34  -fill black
.reeng create line 50 36 925 36  -fill black

.reeng create text 55 54 -text "  Very Low  " -anchor sw
.reeng create text 230 54 -text "  Low  " -anchor sw
.reeng create text 405 54 -text "  Nominal  " -anchor sw
.reeng create text 580 54 -text "  High  " -anchor sw
.reeng create text 755 54 -text "  Extra High  " -anchor sw

.reeng create text 55 96 -text  "  Low cohesion," -anchor sw
.reeng create text 230 96 -text "  Moderately low" -anchor sw
.reeng create text 405 96 -text "  Reasonably well" -anchor sw
.reeng create text 580 96 -text "  High cohesion," -anchor sw
.reeng create text 755 96 -text "  Strong modularity," -anchor sw

.reeng create text 55 120 -text  "  High coupling," -anchor sw
.reeng create text 230 120 -text "  cohesion,     " -anchor sw
.reeng create text 405 120 -text "  structured,  " -anchor sw
.reeng create text 580 120 -text "  Low coupling." -anchor sw
.reeng create text 755 120 -text "  Information hiding. " -anchor sw

.reeng create text 55 144 -text  "  Spaghetti code. " -anchor sw
.reeng create text 230 144 -text "  High coupling.  " -anchor sw
.reeng create text 405 144 -text "  Weak areas.     " -anchor sw
.reeng create text 580 144 -text "                  " -anchor sw
.reeng create text 755 144 -text "                  " -anchor sw

.reeng create text 55 192 -text  "  No match." -anchor sw
.reeng create text 230 192 -text "  Some correlation.   " -anchor sw
.reeng create text 405 192 -text "  Moderate correlation " -anchor sw
.reeng create text 580 192 -text "  Good correlation." -anchor sw
.reeng create text 755 192 -text "  Clean Match. " -anchor sw

.reeng create text 55 240 -text  "  Obscure code. Missing" -anchor sw
.reeng create text 230 240 -text "  Some code commentary" -anchor sw
.reeng create text 405 240 -text "  Moderate level of headers" -anchor sw
```

```
.reeng create text 580 240 -text "  Good code commentary" -anchor sw
.reeng create text 755 240 -text "  Self descriptive code;" -anchor sw

.reeng create text 55 264 -text  "  obscure or incomple-" -anchor sw
.reeng create text 230 264 -text "  and headers, some   " -anchor sw
.reeng create text 405 264 -text "  code commentary and" -anchor sw
.reeng create text 580 264 -text "  and headers; useful" -anchor sw
.reeng create text 755 264 -text "  Documentaion up to date," -anchor sw

.reeng create text 55 288 -text  "  te documentation." -anchor sw
.reeng create text 230 288 -text "  useful documentation." -anchor sw
.reeng create text 405 288 -text "  documentation. " -anchor sw
.reeng create text 580 288 -text "  documentation; weak areas." -anchor sw
.reeng create text 755 288 -text "  well organized." -anchor sw

.reeng create line 50 72 925 72  -fill black

.reeng create line 50 168 925 168  -fill black

.reeng create line 50 216 925 216  -fill black

.reeng create line 50 336 925 336  -fill black
.reeng create line 50 338 925 338  -fill black

.reeng create line 50 36 50 336  -fill blue

.reeng create line 225 36 225 336  -fill blue

.reeng create line 400 36 400 336  -fill blue

.reeng create line 575 36 575 336  -fill blue

.reeng create line 750 36 750 336  -fill blue

.reeng create line 925 36 925 336  -fill blue
```

Appendix D

Experimental Data: Assessing the Reusability of Booch's Components

We have investigated five components from Booch's library of software components [15]: a list, a queue, a dequeue, a ring and a string. For each we present below the Ada source code, the result file from PC-Metric[1]), the result file from ARES (Pascal version) with constant retrieval costs, then the result file from ARES with variable retrieval costs.

D.1 A List

D.1.1 Ada Source Code

```
generic

   type Item is private;

   package List_Single_Unbounded_Unmanaged is

      type List is private;

      Null_List: constant List;

      procedure Copy (From_The_List: in List;
      To_The_List: in out List);

      procedure Clear (The_List: in out List);

      procedure Construct (The_Item: in Item;
   And_The_List: in out List);

      procedure Set_Head  (Of_The_List: in out List;
   To_The_Item: in     Item);
```

[1] ©, Set Laboratories Inc, Mulino, Or, 97042.

```
      procedure Swap_Tail (Of_The_List: in out List;
   And_The_List:  in out List);

      function Is_Equal (Left: in List;
 Right: in List) return Boolean;

      function Length-Of (The_List: in List) return Natural;

      function Is_Null   (The_List: in List) return Boolean;

      function Head_Of   (The_List: in List) return Item;

      function Tail_Of   (The_List: in List) return List;

      Overflow: exception;

      List_Is_Null:  exception;

  private

       type Node;

       type List is access Node;

       Null_List: constant List := null;

  end List_Single_Unbounded_Unmanaged;

  package body List_Single_Unbounded_Unmanaged is

     procedure Copy (From_The_List: in List;
     To_The_List:  in out List) is

         From_Index:  List := From_The_List;
 To_Index:    List;

     begin
     if From_The_List = null then
To_The_List := null;

     else
To_The_List := new Node'(The_Item => From_Index.The_Item,
 Next     => null);
        To_Index:= To_The_List;
From_Index := From_Index.Next;

while From_Index /= null loop;
   To_Index.Next := new Node'(The_Item => From_Index.The_Item,
      Next     => null);

          To_Index := To_Index.Next;
   From_Index := From_Index.Next;
       end loop;
```

```
      end if;

      exception
 when Storage_Error => raise Overflow;

      end Copy;

      procedure Clear (The_List: in out List) is
      begin
      The_List:= null;
      end Clear;

      procedure Construct (The_Item:  in Item;
   And_The_Item:  in out List) is

      begin
      And_The_List := new Node'(The_Item => The_Item,
Next     => And_The_List);

      exception
 when Storage_Error => raise Overflow;

      end Construct;

      procedure Set_Head (Of_The_List:  in out List;
  To_The_Item:  in     Item) is

      begin
Of_The_List.The_Item:= To_The_Item;

      exception
 when Constraint_Error => raise List_Is_Null;

      end Set_Head;

     procedure Swap_Tail (Of_The_List:  in out List;
  And_The_List: in out List) is

     Temporary_Node: List;

     begin

     Temporary_Node:= Of_The_List.Next;
     Of_The_List.Next := And_The_List;
     And_The_List := Temporary_Node;

     exception
when Constraint_Error => raise List_Is_Null;

     end Swap_Tail;

     function Is_Equal (Left:  in List;
Right: in List) return Boolean is
```

```
      Left_Index:  List := Left;
      Right_Index: List := Right;

      begin

      while Left_Index /= null loop
 if Left_Index.The_Item /= Right_Index.The_Item then
    return False;
        end if;

 Left_Index := Left_Index.Next;
 Right_Index := Right_Index.Next;

      end loop;

      return (Right_Index = null);

      exception
 when Constraint_Error => return False

      end Is_Equal;

      function Length_Of (The_List: in List) return Natural is

 Count : Natural := 0;
 Index : List    := The_List;

      begin

      while Index /= null loop
 Count := Count + 1;
 Index := Index.Next;
      end loop;

      return Count;

      end Length_Of;

      function Is_Null (The List: in List) return Boolean is

      begin
return (The_List = null);
      end Is_Null;

      function Head_Of (The_List: in List) return Item is

      begin
      return Heap(The_List.The_Head).The_Item;
      exception
when Cosntraint_Error => raise List_Is_Null;
      end Head_Of;

      function Tail_Of (The_List: in List) return List is
```

```
     begin
     return Heap(The_List.The_Head).Next;
     exception
when Constraint_Error  => raise List_Is_Null;
     end Tail_Of;

   end package body List_Single_Unbounded_Unmanaged;
```

D.1.2 Report 1, PC-Metric

```
6/12/1996
PC-METRIC (ADA) Version 4.0
Summary Complexity Report for: LIST.RP1
---------------------------------------

Total Operators (N1):     161
Total Operands (N2):       86

Software Science Length (N):                247
Estimated Software Science Length (N^):     202
Purity Ratio (P/R):                        0.82

Software Science Volume (V):      1356
Software Science Effort (E):      51038

Estimated Errors using Software Science (B^):      0
Estimated Time to Develop, in hours (T^):          1


Cyclomatic Complexity (VG1):               13
Extended Cyclomatic Complexity (VG2):      13
Average Cyclomatic Complexity:              1
Average Extended Cyclomatic Complexity:     1

Lines of Code (LOC):                      191
Number of Comment Lines:                    0
Number of Blank Lines:                     68
Number of Executable Semi-colons (<;>):    46
Number of Tasks:                            0
Number of Packages:                         1
Number of Procedures/Functions:            10
```

D.1.3 ARES Result File, Constant Retrieval Costs

```
Analyst Name: Rym Mili
Date:  06/14/96
-----------------------------------------


ARES, version 0.0
```

```
Component Name: list


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.


Domain Factors:
   FreqX: 10.000. FreqA:  1.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.046. Baselining:  0.175
   Instanciation:  0.001. Adaptation:  0.012
   Upstream Costs:  0.015. Downstream Costs:  0.004
   Operating Cost with reuse: 0.001. Without reuse: 0.002


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -0.33
Intrinsic Reusability:   0.14
Break Even Frequency:  13.39

Long Term Reusability Measures

Long Term Absolute Reusability:   0.59
Long Term Intrinsic Reusability:   0.36
Long Term Break Even Frequency:   7.30
```

D.1.4 ARES Result File, Variable Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/15/96
-------------------------------------------


ARES, version 0.0


Component Name: 1


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.
```

```
Domain Factors:
   FreqX:  6.000. FreqA:  5.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.046. Baselining:  0.175
   Instanciation:  0.001. Adaptation:  0.012
   Upstream Costs:  0.015. Downstream Costs:  0.004
   Operating Cost with reuse: 0.001. Without reuse: 0.002


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -1.186
Intrinsic Reusability:   0.059
Break Even Frequency:  31.104

Long Term Reusability Measures

Long Term Absolute Reusability:   0.547
Long Term Intrinsic Reusability:   0.209
Long Term Break Even Frequency:  12.717
```

D.2 A Queue

D.2.1 Ada Source Code

```
generic

    type Item is private;

package Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
is
   type Queue is limited private;

   procedure Copy (From_The_Queue:  in Queue;
   To_The_Queue:    in out Queue);

   procedure Clear (The_Queue:  in out Queue);

   procedure Add (The_Item:  in Item;
  To_The_Queue: in out Queue);

   procedure Pop (The_Queue:  in out Queue);

   function Is_Equal (Left: in Queue;
      Right: in Queue) return Boolean;

   function Length_Of (The_Queue:  in Queue) return Natural;
```

```
   function Is_Empty (The_Queue:  in Queue) return Boolean;

   function Front_Of (The_Queue: in Queue) return Item;

   Overflow: exception;

   Underflow:  exception;

   private

      type Node;

      type Structure is access Node;

      type Queue is
 record
    The_Front: Structure;
    The_Back: Structure;
         end record;

end Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;


package body
Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator is

   procedure Copy (From_The_Queue:  in Queue;
   To_The_Queue:    in out Queue) is

      From_Index : Structure := From_The_Queue.The_Front;
      To_Index   : Structure;

   begin

   if From_The_Queue.The_Front = null
   then
      To_The_Queue.The_Front := null;
      To_The_Queue.The_Back := null;

   else
      To_The_Queue.The_Front :=
new Node'(The_Item => From_Index.The_Item,
  Next     => null);

      To_The_Queue.The_Back := To_The_Queue.The_Front;
      To_Index := To_The_Queue.The_Front;
      From_Index := From_Index.Next;

     while From_Index /= null loop
To_Index.Next := new Node'(The_Item => From_Index.The_Item,
   Next     => null);

        To_Index := To_Index.Next;
        From_Index := From_Index.Next;
```

```
         To_The_Queue.The_Back := To_Index;

      end loop;

   end if;

   exception
      when Storage_Error => raise Overflow;


   end Copy;


   procedure Clear (The_Queue:  in out Queue) is

   begin
   The_Queue := Queue'(The_Front => null;
       The_Back  => null);

   end Clear;

   procedure Add (The_Item:  in Item;
  To_The_Queue: in out Queue) is

   begin
   if To_The_Queue.The_Front = null
   then
      To_The_Queue.The_Front := new
Node'(The_Item => The_Item,
       Next     => null);

      To_The_Queue.The_Back := To_The_Queue.The_Front;

   else
      To_The_Queue.The_Back.Next := new Node' (The_Item => The_Item,
       Next     => null);

      To_The_Queue.The_Back := To_The_Queue.The_Back.Next;

   end if;

   exception

      when Storage_Error => raise Overflow;

   end Add;

   procedure Pop (The_Queue : in out Queue) is

   begin
   The_Queue.The_Front := The_Queue.The_Front.Next;
   if The_Queue.The_Front = null
   then
      The_Queue.The_Back := null;
```

```
   end if;

   exception
      when Constraint_Error  =>  raise Underflow;

   end Pop;

   function Is_Equal (Left: in Queue;
      Right: in Queue) return Boolean is

      Left_Index :  Structure  := Left.The_Front;
      Right_Index:  Structure  := Right.The_Front;

   begin
   while Left_Index /= null loop
      if Left_Index.The_Item /= Right_Index.The_Item
      then
return False;
      else
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
      end if;
   end loop;
   return (Right_Index = null);

   exception
      when Constraint_Error => return False;

   end Is_Equal;


   function Length_Of (The_Queue:  in Queue) return Natural is

      Count : Natural := 0;
      Index : Structure := The_Queue.The_Front;

   begin
   while Index /= null loop
      Count := Count + 1;
      Index := Index.Next;
   end loop;
   return Count;
   end Length_Of;

   function Is_Empty (The_Queue:  in Queue) return Boolean is

   begin
   return (The_Queue.The_Front = null);
   end Is_Empty;

   function Front_Of (The_Queue: in Queue) return Item is

   begin
      return The_Queue.The_Front.The_Item;
```

```
   exception
      when Constraint_Error => raise Underflow;
   end Front_Of;

end  Queue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

D.2.2 Report 1, PC-Metric

```
6/12/1996
PC-METRIC (ADA) Version 4.0
Summary Complexity Report for: QUEUE.RP1
----------------------------------------

Total Operators (N1):      196
Total Operands (N2):       109

Software Science Length (N):                  305
Estimated Software Science Length (N^):       185
Purity Ratio (P/R):                          0.61

Software Science Volume (V):       1645
Software Science Effort (E):       98597

Estimated Errors using Software Science (B^):       1
Estimated Time to Develop, in hours (T^):           2


Cyclomatic Complexity (VG1):                 13
Extended Cyclomatic Complexity (VG2):        13
Average Cyclomatic Complexity:                1
Average Extended Cyclomatic Complexity:       1

Lines of Code (LOC):                        192
Number of Comment Lines:                      0
Number of Blank Lines:                       64
Number of Executable Semi-colons (<;>):      48
Number of Tasks:                              0
Number of Packages:                           1
Number of Procedures/Functions:               8
```

D.2.3 ARES Result File, Constant Retrieval Costs

```
Analyst Name: Rym Mili
Date:  06/14/96
-------------------------------------------


ARES, version 0.0


Component Name: queue
```

```
Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.


Domain Factors:
   FreqX: 10.000. FreqA:  1.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.088. Baselining:  0.175
   Instanciation:  0.002. Adaptation:  0.024
   Upstream Costs:  0.029. Downstream Costs:  0.007
   Operating Cost with reuse: 0.002. Without reuse: 0.004


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:   1.42
Intrinsic Reusability:   0.28
Break Even Frequency:   5.99

Long Term Reusability Measures

Long Term Absolute Reusability:   1.63
Long Term Intrinsic Reusability:   0.71
Long Term Break Even Frequency:   3.35
```

D.2.4 ARES Result File, Variable Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/15/96
-----------------------------------------------


ARES, version 0.0


Component Name: q


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.010. Approximate:  0.015.
```

```
Domain Factors:
   FreqX:  6.000. FreqA:  5.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.088. Baselining:  0.175
   Instanciation:  0.002. Adaptation:  0.024
   Upstream Costs:  0.029. Downstream Costs:  0.007
   Operating Cost with reuse: 0.002. Without reuse: 0.004


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -0.778
Intrinsic Reusability:   0.084
Break Even Frequency:  20.274

Long Term Reusability Measures

Long Term Absolute Reusability:   1.529
Long Term Intrinsic Reusability:   0.316
Long Term Break Even Frequency:   7.580
```

D.3 A Dequeue

D.3.1 Ada Source Code

```
generic

    type Item is private;

package
Dequeue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator
is
   type Dequeue is limited private;

   procedure Copy (From_The_Dequeue:  in Dequeue;
   To_The_Dequeue:    in out Dequeue);

   procedure Clear (The_Dequeue:  in out Dequeue);

   procedure Add (The_Item:  in Item;
  To_The_Dequeue: in out Dequeue;
  At_The_Location: in Location);

   procedure Pop (The_Dequeue:  in out Dequeue
  At_The_Location: in Location);

   function Is_Equal (Left: in Dequeue;
      Right: in Dequeue) return Boolean;

   function Length_Of (The_Dequeue:  in Dequeue) return Natural;
```

```
   function Is_Empty (The_Dequeue:  in Dequeue) return Boolean;

   function Front_Of (The_Dequeue: in Dequeue) return Item;

   function Back_Of (The_Dequeue: in Dequeue) return Item;

   Overflow: exception;

   Underflow:  exception;

   private

      type Node is
 record
    The_Item: Item;
    Next: Structure
         end record;

      type Structure is access Node;

      type Dequeue is
 record
    The_Front: Structure;
    The_Back: Structure;
         end record;

end Dequeue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;


package body
Dequeue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator is

   procedure Copy (From_The_Dequeue:  in Dequeue;
   To_The_Dequeue:    in out Dequeue) is

      From_Index : Structure := From_The_Dequeue.The_Front;
      To_Index   : Structure;

   begin

   if From_The_Dequeue.The_Front = null
   then
      To_The_Dequeue.The_Front := null;
      To_The_Dequeue.The_Back := null;

   else
      To_The_Dequeue.The_Front :=
new Node'(The_Item => From_Index.The_Item,
  Next     => null);

      To_The_Dequeue.The_Back := To_The_Dequeue.The_Front;
      To_Index := To_The_Dequeue.The_Front;
      From_Index := From_Index.Next;
```

```
      while From_Index /= null loop
 To_Index.Next := new Node'(The_Item => From_Index.The_Item,
    Next     => null);

         To_Index := To_Index.Next;
         From_Index := From_Index.Next;
         To_The_Dequeue.The_Back := To_Index;

      end loop;

   end if;

   exception
      when Storage_Error => raise Overflow;


   end Copy;


   procedure Clear (The_Dequeue:  in out Dequeue) is

   begin
   The_Dequeue := Dequeue'(The_Front => null;
       The_Back  => null);

   end Clear;

   procedure Add (The_Item:  in Item;
  To_The_Dequeue: in out Dequeue;
  At_The_Location: in Location) is

   begin
   if To_The_Dequeue.The_Front = null
   then
      To_The_Dequeue.The_Front := new
Node'(The_Item => The_Item,
       Next     => null);

      To_The_Dequeue.The_Back := To_The_Dequeue.The_Front;

   else
      if At_The_Location = Front
      then
To_The_Dequeue.The_Front :=  new Node' (The_Item => The_Item,
       Next     => To_The_Dequeue.The_Front);

      else
To_The_Dequeue.The_Back.Next :=
    new Node'(The_Item => The_Item,
      Next => null);
         To_The_Dequeue.The_Back := To_The_Dequeue.The_Back.Next;
      end if;
```

```
   end if;

   exception

      when Storage_Error => raise Overflow;

   end Add;

   procedure Pop (The_Dequeue : in out Dequeue;
  At_The_Location: in Location) is

      Index : Structure := The_Dequeue.The_Front;

   begin
   if Index.Next = null then
      The_Dequeue.The_Front := null;
      The_Dequeue.The_Back  := null;
   elseif
      At_The_Location=Front
   then
      The_Dequeue.The_Front := The_Dequeue.The_Front.Next;

   else
      while Index.Next /= The_Dequeue.The_Back loop
         Index := Index.Next;
      end loop;
      The_Dequeue.The_Back := Index;
      The_Dequeue.The_Back.Next := null;
   end if;

   exception
      when Constraint_Error  =>  raise Underflow;

   end Pop;

   function Is_Equal (Left: in Dequeue;
      Right: in Dequeue) return Boolean is

      Left_Index :  Structure  := Left.The_Front;
      Right_Index:  Structure  := Right.The_Front;

   begin
   while Left_Index /= null loop
      if Left_Index.The_Item /= Right_Index.The_Item
      then
return False;
      else
Left_Index := Left_Index.Next;
Right_Index := Right_Index.Next;
      end if;
   end loop;
   return (Right_Index = null);

   exception
```

```
      when Constraint_Error => return False;

   end Is_Equal;


   function Length_Of (The_Dequeue:  in Dequeue) return Natural is

      Count : Natural := 0;
      Index : Structure := The_Dequeue.The_Front;

   begin
   while Index /= null loop
      Count := Count + 1;
      Index := Index.Next;
   end loop;
   return Count;
   end Length_Of;

   function Is_Empty (The_Dequeue:  in Dequeue) return Boolean is

   begin
   return (The_Dequeue.The_Front = null);
   end Is_Empty;

   function Front_Of (The_Dequeue: in Dequeue) return Item is

   begin
      return The_Dequeue.The_Front.The_Item;

   exception
      when Constraint_Error => raise Underflow;
   end Front_Of;

   function Back_Of (The_Dequeue: in Dequeue) return Item is

   begin
      return The_Dequeue.The_Back.The_Item;

   exception
      when Constraint_Error => raise Underflow;
   end Front_Of;

end  Dequeue_Nonpriority_Nonbalking_Sequential_Unbounded_Unmanaged_Noniterator;
```

D.3.2 Report 1, PC-Metric

```
6/12/1996
PC-METRIC (ADA) Version 4.0
Summary Complexity Report for: DEQUEUE.RP1
----------------------------------------

Total Operators (N1):      241
Total Operands (N2):       141
```

```
Software Science Length (N):                382
Estimated Software Science Length (N^):     202
Purity Ratio (P/R):                        0.53

Software Science Volume (V):      2098
Software Science Effort (E):      141471

Estimated Errors using Software Science (B^):      1
Estimated Time to Develop, in hours (T^):          2


Cyclomatic Complexity (VG1):              16
Extended Cyclomatic Complexity (VG2):     16
Average Cyclomatic Complexity:             1
Average Extended Cyclomatic Complexity:    1

Lines of Code (LOC):                      231
Number of Comment Lines:                    0
Number of Blank Lines:                     70
Number of Executable Semi-colons (<;>):    58
Number of Tasks:                            0
Number of Packages:                         1
Number of Procedures/Functions:             9
```

D.3.3 ARES Result File, Constant Retrieval Costs

```
Analyst Name: Rym Mili
Date:  06/14/96
-------------------------------------------


ARES, version 0.0


Component Name: dequeue


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.


Domain Factors:
   FreqX: 10.000. FreqA:  1.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.127. Baselining:  0.175
   Instanciation:  0.002. Adaptation:  0.035
   Upstream Costs:  0.042. Downstream Costs:  0.010
```

```
   Operating Cost with reuse: 0.003. Without reuse: 0.005


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:   2.58
Intrinsic Reusability:   0.38
Break Even Frequency:   4.23

Long Term Reusability Measures

Long Term Absolute Reusability:   2.31
Long Term Intrinsic Reusability:   0.94
Long Term Break Even Frequency:   2.35
```

D.3.4 ARES Result File, Variable Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/15/96
-------------------------------------------


ARES, version 0.0


Component Name: d


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.015. Approximate:  0.023.


Domain Factors:
   FreqX:  6.000. FreqA:  5.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.127. Baselining:  0.175
   Instanciation:  0.002. Adaptation:  0.035
   Upstream Costs:  0.042. Downstream Costs:  0.010
   Operating Cost with reuse: 0.003. Without reuse: 0.005


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -0.527
Intrinsic Reusability:   0.099
Break Even Frequency:  16.349
```

```
Long Term Reusability Measures

Long Term Absolute Reusability:   2.176
Long Term Intrinsic Reusability:   0.382
Long Term Break Even Frequency:   5.793
```

D.4 A Ring

D.4.1 Ada Source Code

```
generic

   type Item is private;

package Ring_Sequential_Unbounded_Unmanaged_Noniterator is

   type Ring is limited private;

   type Direction is (Forward, Backward);

   procedure Copy (From_The_Ring : in Ring;
   To_The_Ring : in out Ring);

   procedure Clear (The_Ring : in out Ring);

   procedure Insert (The_Item : in Item;
     In_The_Ring : in out Ring);

   procedure Pop (The_Ring : in out Ring);

   procedure Rotate (The_Ring:  in out Ring;
     In_The_Direction: in Direction);

   procedure Mark (The_Ring: in out Ring);

   procedure Rotate_To_Mark (The_Ring: in out Ring);

   function Is_Equal (Left: in Ring;
      Right: in Ring) return Boolean;

   function Extent_Of (The_Ring: in Ring) return Natural;

   function Is_Empty (The_Ring: in Ring) return Boolean;

   function Top_Of (The_Ring: in Ring) return Item;

   function At_Mark (The_Ring: in Ring) return Boolean;

   Overflow: exception;

   Underflow:  exception;
```

```
   Rotate_Error:  exception;

   private

      type Node;

      type Structure is access Node;

      type Ring is
 record
    The_Top: Structure;
    The_Mark: Structure;
         end record;

end Ring_Sequential_Unbounded_Unmanaged_Noniterator;


package body Ring_Sequential_Unbounded_Unmanaged_Noniterator is


   procedure Copy (From_The_Ring : in Ring;
   To_The_Ring : in out Ring) is

      From_Index:  Structure := From_The_Ring.The_Top;
      To_Index: Structure;

   begin
   if From_The_Ring.The_Top = null
   then
      To_The_Ring.The_Top := null;
      To_The_Ring.The_Mark := null;

   else
      To_The_Ring.The_Top := new Node'(Previous => null,
       The_Item => From_Index.The_Item,
       Next => null);

      To_Index := To_The_Ring.The_Top;
      if From_The_Ring.The_Mark := From_Index
      then
To_The_Ring.The_Mark := To_Index;
      end if;

      From_Index := From_Index.Next;

      while From_Index /= From_The_Ring.The_Top loop
To_Index.Next := new Node'(Previous => To_Index,
   The_Index => From_Index.The_Item,
   Next => null);
         To_Index := To_Index.Next;
if From_The_Ring.The_Mark = From_Index
then
   To_The_Ring.The_Mark := To_Index;
         end if;
```

```
 From_Index := From_Index.Next;

      end loop;

      To_The_Ring.The_Top.Previous := To_Index;
      To_Index.Next := To_The_Ring.The_Top;

   end if;

   exception

      when Storage_Error => raise Overflow;

   end Copy;


   procedure Clear (The_Ring : in out Ring) is
   begin
   The_Ring := Ring'(The_Top => null, The_Mark => null);
   end Clear;

   procedure Insert (The_Item : in Item;
     In_The_Ring : in out Ring) is

   begin
   if In_The_Ring.The_Top = null
   then
      In_The_Ring.The_Top  = new Node'(Previous => null,
       The_Item => The_Item,
       Next => null);

      In_The_Ring.The_Top.Previous := In_The_Ring.The_Top;
      In_The_Ring.The_Top.Next     := In_The_Ring.The_Top;
      In_The_Ring.The_Top.The_Mark := In_The_Ring.The_Top;

   else
      In_The_Ring.The_Top :=
new Node'(Previous => In_The_Ring.The_Top.Previous,
   The_Item => The_Item,
   Next     => In_The_Ring.The_Top);
      In_The_Ring.The_Top.Next.Previous := In_The_Ring.The_Top;
      In_The_Ring.The_Top.Previous.Next := In_The_Ring.The_Top;
   end if;

   exception
      when Storage_Error => raise Overflow

   end Insert;

   procedure Pop (The_Ring : in out Ring) is

   begin
   if The_Ring.The_Top = The_Ring.The_Top.Next
```

```
   then
      The_Ring.The_Top := null;
      The_Ring.The_Mark:= null;

   else
      The_Ring.The_Top.Previous.Next := The_Ring.The_Top.Next;
      The_Ring.The_Top.Next.Previous := The_Ring.The_Top.Previous;

      if The_Ring.The_Mark = The_Ring.The_Top
      then
The_Ring.The_Mark := The_Ring.The_Top.Next;
      end if;

      The_Ring.The_top := The_Ring.The_Top.Next;

   end if;

   exception

      when Constraint_Error => raise Underflow;

   end Pop;


   procedure Rotate (The_Ring:  in out Ring;
     In_The_Direction: in Direction) is

   begin

   if In_The_Direction = Forward
   then
      The_Ring.The_Top := The_Ring.The_Top.Next;
   else
      The_Ring.The_Top := The_Ring.The_Top.Previous;
   end if;

   exception

      when Constraint_Error => raise Rotate_Error

   end Rotate;


   procedure Mark (The_Ring: in out Ring) is
   begin
   The_Ring.The_Mark := The_Ring.The_Top;
   end Mark;

   procedure Rotate_To_Mark (The_Ring: in out Ring) is
   begin
   The_Ring.The_Top := The_Ring.The_Mark;
   end Rotate_To_Mark;

   function Is_Equal (Left: in Ring;
```

```
      Right: in Ring) return Boolean is

      Left_Index : Structure := Left.The_Top;
      Right_Index: Structure := Ringht.The_Top;

   begin
   if Left_Index.The_Item /= Right_Index.The_Item
   then
      return False;

   elsif
      (Left.The_Mark = Left_Index) and then
      (Right.The_Mark /= Right_Index) then
      return False;

   else
      Left_Index := Left_Index.Next;
      Right_Index := Right_Index.Next;

      while Left_Index /= Left.The_Top loop
if Left_Index.The_Item /= Right_Index.The_Item
then return False;
elsif (Left.The_Mark = Left_Index) and then
       (Right.The_Mark /= Right_Index)
         then return False;
else
    Left_Index := Left_Index.Next;
    Right_Index := Right_Index.Next;
         end if;
      end loop;

      return (Right_Index = Right_Index.The_Top);

  end if;

  exception

     when Constraint_Error => return (Left.The_Top = Right.The_Top);

  end Is_Equal;


  function Extent_Of (The_Ring: in Ring) return Natural is

     Count : Natural := 0;
     Index : Structure := The_Ring.The_Top;

  begin

  Index := Index.Next;
  Count := Count +1;

  while Index /= The_Ring.The_Top loop
     Count := Count + 1;
```

```
      Index := Index.Next;
   end loop;

   return Count;

   exception

      when Constraint_Error => return 0;

   end Extent_Of;

   function Is_Empty (The_Ring: in Ring) return Boolean is
   begin
   return(The_Ring.The_Top = null);
   end Is_Empty;

   function Top_Of (The_Ring: in Ring) return Item is

   begin
   return The_Ring.The_Top.The_Item;

   exception

      when Constraint_Error => raise Underflow;

   end Top_Of;

   function At_Mark (The_Ring: in Ring) return Boolean is

   begin
   return (The_Ring.The_Top = The_Ring.The_Mark);
   end At_Mark;

end Ring_Sequential_Unbounded_Unmanaged_Noniterator;
```

D.4.2 Report 1, PC-Metric

```
6/13/1996
PC-METRIC (ADA) Version 4.0
Summary Complexity Report for: RING.RP1
---------------------------------------

Total Operators (N1):      356
Total Operands (N2):       247

Software Science Length (N):                  603
Estimated Software Science Length (N^):       257
Purity Ratio (P/R):                          0.43

Software Science Volume (V):       3470
Software Science Effort (E):     369456

Estimated Errors using Software Science (B^):       1
Estimated Time to Develop, in hours (T^):           6
```

```
Cyclomatic Complexity (VG1):                22
Extended Cyclomatic Complexity (VG2):       24
Average Cyclomatic Complexity:               1
Average Extended Cyclomatic Complexity:      1

Lines of Code (LOC):                       293
Number of Comment Lines:                     0
Number of Blank Lines:                      98
Number of Executable Semi-colons (<;>):     77
Number of Tasks:                             0
Number of Packages:                          1
Number of Procedures/Functions:             12
```

D.4.3 ARES Result File, Constant Retrieval Costs

```
Analyst Name: Rym Mili
Date:  06/09/96
-------------------------------------------


ARES, version 0.0


Component Name: ring


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.


Domain Factors:
   FreqX: 10.000. FreqA:  1.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.331. Baselining:  0.175
   Instanciation:  0.006. Adaptation:  0.090
   Upstream Costs:  0.109. Downstream Costs:  0.026
   Operating Cost with reuse: 0.009. Without reuse: 0.014


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:   5.78
Intrinsic Reusability:   0.65
Break Even Frequency:   2.10
```

```
Long Term Reusability Measures

Long Term Absolute Reusability:   4.22
Long Term Intrinsic Reusability:   1.58
Long Term Break Even Frequency:   1.09
```

D.4.4 ARES Result File, Variable Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/15/96
-------------------------------------------


ARES, version 0.0


Component Name: r


Displaying Cost Factors:


Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.046. Approximate:  0.070.


Domain Factors:
   FreqX:  6.000. FreqA:  5.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.331. Baselining:  0.175
   Instanciation:  0.006. Adaptation:  0.090
   Upstream Costs:  0.109. Downstream Costs:  0.026
   Operating Cost with reuse: 0.009. Without reuse: 0.014


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -0.225
Intrinsic Reusability:   0.104
Break Even Frequency:  13.170

Long Term Reusability Measures

Long Term Absolute Reusability:   3.948
Long Term Intrinsic Reusability:   0.496
Long Term Break Even Frequency:   3.475
```

D.5 A String

D.5.1 Ada Source Code

```
generic

   type Item is private;

package String_Sequential_Bounded_Unmanaged_Noniterator is

   type String (The_Size: Positive) is limited private;

   procedure Copy (From_The_String : in String;
   To_The_String   : in out String);

   procedure Copy (From_The_Substring: in Substring;
   To_The_String:      in out String);

   procedure Clear (The_String:    in out String);

   procedure Prepend (The_String: in String;
      To_The_String: in out String);

   procedure Prepend (The_Substring: in Substring;
      To_The_String: in out String);

   procedure Append (The_String: in String;
     To_The_String: in out String);

   procedure Append (The_Substring: in Substring;
     To_The_String: in out String);

   procedure Insert (The_String:    in String;
     In_The_String: in out String;
     At_The_Position: in Positive);

   procedure Insert (The_Substring:    in Substring;
     In_The_String: in out String;
     At_The_Position: in Positive);

   procedure Delete (In_The_String:  in out String;
     From_The_Position:  in Positive;
     To_The_Position: in Positive);

   procedure Replace (In_The_String: in out String;
      At_The_Position: in Positive;
      With_The_String: in String);

   procedure Replace (In_The_String: in out String;
      At_The_Position: in Positive;
      With_The_Substring: in Substring);

   procedure Set_Item (In_The_String: in out String;
       At_The_Position: in Positive;
```

```
        With_The_Item: in Item);

    function Is_Equal (Left: in String;
       Right: in String) return Boolean;

    function Is_Equal (Left: in Substring;
       Right: in String) return Boolean;

    function Is_Equal (Left: in String;
       Right: in Substring) return Boolean;

    function Is_Less_Than (Left: in String;
                Right: in String) return Boolean;

    function Is_Less_Than (Left: in Substring;
           Right: in String) return Boolean;

    function Is_Less_Than (Left: in String;
           Right: in Substring) return Boolean;

    function Is_Greater_Than (Left: in String;
              Right: in String) return Boolean;

    function Is_Greater_Than (Left: in Substring;
              Right: in String) return Boolean;

    function Is_Greater_Than (Left: in String;
              Right: in Substring) return Boolean;

    function Length_Of (The_String: in String) return Natural;

    function Is_Null (The_String: in String) return Boolean;

    function Item_Of (The_String: in String;
      At_The_Position:  in Positive) return Item;

    function Substring_Of (The_String: in String) return Substring;

    function Substring_Of (The_String: in String;
  From_The_Position: in Positive;
  To_The_Position:  in Positive) return Substring;

    Overflow:  exception;

    Position_Error: exception;

private

    type Structure is access Substring;

    type String (The_Size: Positive) is
       record
 The_Length: Natural := 0;
 The_Items: Substring (1.. The_Size);
```

```
      end record;

end String_Sequential_Bounded_Unmanaged_Noniterator;

package body String_Sequential_Bounded_Unmanaged_Noniterator is

   procedure Copy (From_The_String: in String;
   To_The_String: in out String) is

   begin
   if From_The_String.The_Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items (1.. From_The_String.The_Length)
      := From_The_String.The_Items (1.. From_The_String.The_Length);
      To_The_String.The_Length
      := From_The_String.The_Length;
   end if;
   end Copy;

   procedure Copy (From_The_Substring: in Substring;
   To_The_String: in out String) is

   begin
   if From_The_Substring'Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items (1.. From_The_Substring'Length)
      := From_The_Substring;
      To_The_String.The_Length
      := From_The_Substring'Length;
   end if;
   end Copy;

   procedure Clear (The_String: in out String) is
   begin
   The_String.The_Length := 0;
   end Clear;

   procedure Prepend (The_String: in String;
      To_The_String: in out String) is

      New_Length : Natural :=
To_The_String.The_Length + The_String.The_Length;

   begin
   if New_Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items ((The_String.The_Length +1) ..
       New_Length)
```

```
      :=
      To_The_String.The_Items(1 .. To_The_String.The_Length);

      To_The_String.The_Items (1.. The_String.The_Length)
      :=
      The_String.The_Items(1.. The_String.The_Length);

      To_The_String.The_Length := New_Length;

   end if;

   end Prepend;

   procedure Prepend (The_Substring: in Substring;
      To_The_String: in out String) is

      New_Length : Natural :=
To_The_String.The_Length + The_Substring'Length;

   begin
   if New_Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items ((The_Substring'Length +1) ..
       New_Length)
      :=
      To_The_String.The_Items(1 .. To_The_String.The_Length);

      To_The_String.The_Items (1.. The_Substring'Length)
      :=
      The_Substring;

      To_The_String.The_Length := New_Length;

   end if;

   end Prepend;

   procedure Append (The_String: in String;
      To_The_String: in out String) is

      New_Length : Natural :=
To_The_String.The_Length + The_String.The_Length;

   begin
   if New_Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items ((The_String.The_Length +1) ..
       New_Length)
      :=
      To_The_String.The_Items(1 .. To_The_String.The_Length);
```

```
      To_The_String.The_Length := New_Length;

   end if;

   end Append;

   procedure Append (The_Substring: in Substring;
      To_The_String: in out String) is

      New_Length : Natural :=
To_The_String.The_Length + The_Substring'Length;

   begin
   if New_Length > To_The_String.The_Size then
      raise Overflow;

   else
      To_The_String.The_Items ((The_Substring'Length +1) ..
       New_Length)
      :=
      To_The_String.The_Items(1 .. To_The_String.The_Length);

      To_The_String.The_Length := New_Length;

   end if;

   end Append;

   procedure Insert (The_String:  in String;
     In_The_String : in out String;
     At_The_Position : in Positive) is

   New_Length : Natural
      := In_The_String.The_Length + The_String.The_Length;
   End_Position : Natural
      := At_The_Position + The_String.The_Length;

   begin
   if At_The_Position > In_The_String.The_Length then
      raise Position_Error;

   elseif
      New_Length > In_The_String.The_Size then
raise Overflow;

   else
      In_The_String.The_Items(End_Position .. New_Length)
      :=
      In_The_String.The_Items (At_The_Position .. In_The_String.The_Length);

      In_The_String.The_Items (At_The_Position .. (End_Position -1))
      :=
      The_String.The_Items (1..The_String.The_Length);
```

```
      In_The_String.The_Length := New_Length;

   end if;

   end Insert;

   procedure Insert (The_Substring:  in Substring;
     In_The_String : in out String;
     At_The_Position : in Positive) is

   New_Length : Natural
      := In_The_String.The_Length + The_Substring'Length;
   End_Position : Natural
      := At_The_Position + The_Substring'Length;

   begin
   if At_The_Position > In_The_String.The_Length then
      raise Position_Error;

   elseif
      New_Length > In_The_String.The_Size then
raise Overflow;

   else
      In_The_String.The_Items(End_Position .. New_Length)
      :=
      In_The_String.The_Items (At_The_Position .. In_The_String.The_Length);

      In_The_String.The_Items (At_The_Position .. (End_Position -1))
      :=
      The_Substring;

      In_The_String.The_Length := New_Length;

   end if;

   end Insert;

   procedure Delete (In_The_String : in out String;
     From_The_Position : in Positive;
     To_The_Position : in Positive) is

   New_Length : Natural;

   begin

   if (From_The_Position > In_The_String.The_Length) or else
      (To_The_Position > In_The_String.The_Length) or else
      (From_The_Position > To_The_Position)
   then
      raise Position_Error;

   else
```

```
      New_Length :=
      In_The_String.The_Length
      - (To_The_Position - From_The_Position + 1);

      In_The_String.The_Items (From_The_Position .. New_Length)
      :=
      In_The_String.The_Items ((To_The_Position +1) ..
       In_The_String.The_Length);

      In_The_String.The_Length := New_Length;

   end if;

   end Delete;

   procedure Replace   (In_The_String : in out String;
At_The_Position : in Positive;
With_The_String : in String) is

   End_Position : Natural
:= At_The_Position + With_The_String.The_Length -1;

   begin

   if (At_The_Position > In_The_String.The_Length) or else
      (End_Position > In_The_String.The_Length)
   then
      raise Position_Error;

   else
      In_The_String.The_Items (At_The_Position .. End_Position)
      :=
      With_The_String.The_Items (1 .. With_The_String.The_length);
   end if;

   end Replace;

   procedure Replace   (In_The_String : in out String;
At_The_Position : in Positive;
With_The_Substring : in Substring) is

   End_Position : Natural
:= At_The_Position + With_The_Substring'Length -1;

   begin

   if (At_The_Position > In_The_String.The_Length) or else
      (End_Position > In_The_String.The_Length)
   then
      raise Position_Error;

   else
      In_The_String.The_Items (At_The_Position .. End_Position)
      :=
```

```
      With_The_Substring;
   end if;

   end Replace;

   procedure Set_Item (In_The_String : in out String;
       At_The_Position : in Positive;
                        With_The_Item : in Item) is
   begin

   if At_The_Position > In_The_String.The_length
   then
      raise Position_Error;

   else
      In_The_String.The_Items (At_The_Position)
      :=  With_The_Item;

   end if;

   end Set_Item;

   function Is_Equal (Left:  in String;
      Right: in String) return Boolean is

   begin

   if Left.The_Length /= Right.The_Length
   then
      return False;

   else
      for Index in 1 .. Left.The_Length loop
if Left.The_Items(Index) /= Right.The_Items(Index)
then
   return False;
        end if;
     end loop;

     return True;
  end if;

  end Is_Equal;

  function Is_Equal (Left:  in Substring;
     Right: in String) return Boolean is

  begin

  if Left'Length /= Right.The_Length
  then
     return False;

  else
```

```
     for Index in 1 .. Left'Length loop
 if Left(Left'First+Index-1) /= Right.The_Items(Index)
 then
    return False;
        end if;
      end loop;

      return True;
   end if;

   end Is_Equal;

   function Is_Less_Than (Left: in String;
  Right: in String) return Boolean is

   begin

   for Index in 1 .. Left.The_Length loop

      if Index > Right.The_Length
      then
 return False;

      elseif Left.The_Items(Index) < Right.The_Items(Index)
      then
 return True;

      elseif Right.The_Items(Index) < Left.The_Items(Index)
      then
 return False;

      end if;

   end loop;

   return (Left.The_Length < Right.The_Length);

   end Is_Less_Than;

   function Is_Less_Than (Left: in Substring;
  Right: in String) return Boolean is

   begin

   for Index in 1 .. Left'Length loop

      if Index > Right.The_Length
      then
return False;

      elseif Left(Left'First + Index -1) < Right.The_Items(Index)
      then
return True;
```

```
      elseif Right.The_Items(Index) < Left (Left'First + Index -1 )
      then
 return False;

      end if;

   end loop;

   return (Left'Length < Right.The_Length);

   end Is_Less_Than;

   function Is_Greater_Than (Left: in String;
  Right: in String) return Boolean is

   begin

   for Index in 1 .. Left.The_Length loop

      if Index > Right.The_Length
      then
 return True;

      elseif Left.The_Items(Index) < Right.The_Items(Index)
      then
 return False;

      elseif Right.The_Items(Index) < Left.The_Items(Index)
      then
 return True;

      end if;

  end loop;

  return False;

  end Is_Greater_Than;

  function Is_Greater_Than (Left: in Substring;
 Right: in String) return Boolean is

  begin

  for Index in 1 .. Left'Length loop

     if Index > Right.The_Length
     then
return True;

     elseif Left(Left'First + Index -1) < Right.The_Items(Index)
     then
return False;
```

```
      elseif Right.The_Items(Index) < Left (Left'First + Index -1 )
      then
 return True;

      end if;

   end loop;

   return False;

   end Is_Greater_Than;

   function Length_Of (The_String:  in String) return Natural is

   begin
      return The_String.The_Length;
   end Length_Of;

   function Is_Null (The_String : in String) return Boolean is

   begin
      return (The_String.The_Length = 0);
   end Is_Null;

   function Item_Of (The_String : in String;
     At_The_Position : in Positive) return Item is

   begin
   if At_The_Position > The_String.The_Length
   then
      raise Position_Error;

   else
      return The_String.The_Items(At_The_Position);
   end if;

   end Item_Of;

   function Substring_Of (The_String: in String) return Substring is

   begin
      return The_String.The_Items (1 .. The_String.The_length);
   end Substring_Of;

   function Substring_Of (The_String:  in String;
From_The_Position: in Positive;
To_The_Position: in Positive) return Substring is

   begin

   if (From_The_Position > The_String.The_Length)
      or else
      (To_The_Position > The_String.The_Length)
      or else
```

```
      (From_The_Position > To_The_Position -1)
   then
      raiuse Error_Position;

   else
      return The_String.The_Items
 (From_The_Position ..  To_The_Position);

   end if;

   end Substring_Of;

end package body String_Sequential_Bounded_Unmanaged_Noniterator;
```

D.5.2 Report 1, PC-Metric

```
6/12/1996
PC-METRIC (ADA) Version 4.0
Summary Complexity Report for: STRING.RP1
----------------------------------------

Total Operators (N1):      565
Total Operands (N2):       412

Software Science Length (N):                  977
Estimated Software Science Length (N^):       239
Purity Ratio (P/R):                          0.24

Software Science Volume (V):       5542
Software Science Effort (E):     937779

Estimated Errors using Software Science (B^):       2
Estimated Time to Develop, in hours (T^):          14


Cyclomatic Complexity (VG1):                 29
Extended Cyclomatic Complexity (VG2):        35
Average Cyclomatic Complexity:                1
Average Extended Cyclomatic Complexity:       1

Lines of Code (LOC):                        604
Number of Comment Lines:                      0
Number of Blank Lines:                      187
Number of Executable Semi-colons (<;>):     123
Number of Tasks:                              0
Number of Packages:                           1
Number of Procedures/Functions:              24
```

D.5.3 ARES Result File, Constant Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/14/96
-----------------------------------------------
```

```
ARES, version 0.0

Component Name: string

Displaying Cost Factors:

Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.005. Approximate:  0.007.

Domain Factors:
   FreqX: 10.000. FreqA:  1.000. Freq: 11.000

Component Specific Factors
   Development Effort:  0.840. Baselining:  0.175
   Instanciation:  0.016. Adaptation:  0.230
   Upstream Costs:  0.278. Downstream Costs:  0.065
   Operating Cost with reuse: 0.023. Without reuse: 0.034

Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:   8.15
Intrinsic Reusability:   0.85
Break Even Frequency:   1.39

Long Term Reusability Measures

Long Term Absolute Reusability:   5.63
Long Term Intrinsic Reusability:   2.05
Long Term Break Even Frequency:   0.66
```

D.5.4 ARES Result File, Variable Retrieval Costs

```
Analyst Name: Rym Mili
Date: 06/15/96
-----------------------------------------------

ARES, version 0.0

Component Name: s

Displaying Cost Factors:
```

```
Strategic Factors:
   Investment Cycle:   5.00. Discount Rate:   0.15.
   Storage:  0.055. Exact:  0.130. Approximate:  0.195.


Domain Factors:
   FreqX:  6.000. FreqA:  5.000. Freq: 11.000


Component Specific Factors
   Development Effort:  0.840. Baselining:  0.175
   Instanciation:  0.016. Adaptation:  0.230
   Upstream Costs:  0.278. Downstream Costs:  0.065
   Operating Cost with reuse: 0.023. Without reuse: 0.034


Short Term Measures, for Investment Cycle =   5.00

Absolute Reusability:  -0.288
Intrinsic Reusability:   0.081
Break Even Frequency:  14.548

Long Term Reusability Measures

Long Term Absolute Reusability:   5.249
Long Term Intrinsic Reusability:   0.529
Long Term Break Even Frequency:   2.574
```

Bibliography

[1] A. J. Albrecht. Measuring application development productivity. In *Proceedings IBM Applic. Dev. Symposium*, Monterey, California, October 1979.

[2] G. Arango. Domain analysis: From art to engineering discipline. *Fifth International Workshop on Software Specification and Design*, pages 152–159, April 1989.

[3] G. Arango and R. Prieto-Diaz. Domain analysis concepts and research directions. *IEEE Proceedings*, pages 9–32, 1991.

[4] J. Bailey and V. Basili. The software-cycle model for re-engineering and reuse. In *Proccedings TRI-Ada Conference*, San Jose, CA, October 1991.

[5] V. Basili, G. Caldiera, and G. Cantone. A reference architecture for the component factory. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):53–80, January 1992.

[6] V. Basili and H. D. Rombach. Support for comprehensive reuse. *Softawre Engineering Journal, IEEE British Computer Society*, 6(5):303–316, September 1991.

[7] D. Bell, I. Morrey, and J. Pugh. *Software Engineering: A Programming Approach.* Prentice Hall, NY, 1992.

[8] T. Biggerstaff and A. Perlis Editors. *Software Reusability.* ACM Press, Reading, MA, 1989.

[9] B. Boehm. *Recent Developments in Ada COCOMO.* Private correpondence, 1989.

[10] B. Boehm. Megaprogramming. In *Key note address. ACM Computer Science Conference*, Phoenix, AZ, March 1994.

[11] B. Boehm. *COCOMO 2.0 Model User's Manual.* Private correpondence, 1995.

[12] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future software life cycle processes. *Annals of Software Engineering Special Volume on Software Process and product Measurement*, 1995.

[13] B. Boehm and W. Scherlis. Megaprogramming. In *Proceedings, DARPA Software Technology Conference*, April 1992.

[14] B.W. Boehm. *Software Engineering Economics.* Prentice Hall, Englewood Clifss, NJ, 1981.

[15] Grady Booch. *Software Components with Ada.* Benjamin Cummings, Menlo Park, CA, 1987.

[16] N. Boudriga, A. Mili, F. Mili, and R. Zalila (Mili). A relational approach to the specification of data types: The generalized model. *Computer Languages*, 17(2):101–131, september 1992.

[17] N. Boudriga, A. Mili, and R.T. Mittermeir. Semantic-based software retrieval to support rapid prototyping. *Structured Programming*, 13:109–127, 1992.

[18] N. Boudriga, A. Mili, and R. Zalila. An automated tool for specification validation: Design and preliminary implementation. In *Proceedings, Hawaii International Conference on System Sciences*, pages 74–82, 1992.

[19] C. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science.* Springer Verlag, 1997.

[20] G. Caldiera and V. Basili. Identifying and qualifying reusable software components. *IEEE Computer*, pages 61–70, february 1991.

[21] P.S. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In *Proc. 2nd Intl. Workshop on Software Reusability*, pages 99–108, Lucca, Italy, march 1993.

[22] Dynamic Research Corporation. Software reuse library standards and metrics. Technical report, Dynamic Research Corporation, Andover, MA, May 1994.

[23] M. A. Cusumano. *Japan's Software Factories.* Oxford University Press, New York, 1991.

[24] C.J. Date. *Introduction to Database Systems, 4th ed.* Addison Wesley, Reading, Ma, 1986.

[25] A. Davis. *Software Requirements.* Prentice-Hall, New Jersey, 1993.

[26] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. Lassie: A knowledge-based software information system. *Communications of the ACM*, 34(5), May 1991.

[27] M. Sitaraman et al. Components based software using resolve. *SIGSOFT Software Engineering Notes*, 1994.

[28] J. Favaro. A comparison of approaches to reuse investment analysis. In *Proceedings, Fourth International Conference on Software Reuse*, Orlando, FL, April 1996.

[29] N. Fenton. Software measurement: A necessary scientific basis. *IEEE TSE*, 20(3):199–206, march 1994.

[30] N. E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, London, 1993.

[31] J. Fitz-Enz. Who is the dp professional? *Datamation*, pages 124–129, September 1978.

[32] W.B. Frakes and T.P. Pole. Proteus: A reuse library system that supports multiple representation methods. *ACM SIGIR Forum*, 24:43–55, 1990.

[33] W.B. Frakes and T.P. Pole. Empirical study of software reuse. *IEEE Transactions on Software Engineering*, 20(8):617–630, August 1994.

[34] M. Frappier. Program construction by parts. Technical report, PhD, University of Ottawa, Ottawa, Ontario, Canada, 1995.

[35] M. Frappier, J. Desharnais, and A. Mili. Mathematics for program construction by parts. In *Proceedings Conf on Mathematics of Program Construction*, Kloster, Germany, July 1995.

[36] M. Frappier, J. Desharnais, and A. Mili. Mathematics for program construction by parts. *Science of Computer Programming*, to appear, to appear.

[37] J. E. Gaffney and R. D. Cruickshank. A general economics model of software reuse. In *Software Reusability*, Software Productivity Consortium, SPC-92119-CMC, september 1992.

[38] R.J. Hall. Generalized behaviour-based retrieval. In *Proceedings, 16th Int. Conf. on Soft. Eng.*, Sorrento, Italy, May 1994.

[39] Ivar Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, Workingham, England, 1994.

[40] S. Katz, C.A. Richter, and T-S. The. Paris: A system for reusing partially interpreted schemas. In *Proc. 9th Intl. Conf. on Software Engineering*, pages 377–385, Monterey, CA, april 1987.

[41] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, MA, 1980.

[42] W. Lim. A cost justification model for software reuse. In *Proceedings, Fifth Annual Workshop on Software Reuse*, 1992.

[43] W. Lim. Software reuse metrics. *IEEE Software*, September 1994.

[44] B. Liskov and V. Berzins. An appraisal of program specifications. In P. Wegner, editor, *Research Directions in Software Technology.* MIT Press, 1979.

[45] R. Malan and K. Wentzel. Economics of software reuse revisited. Technical Report HPL-93-31, Hewlett Packard, April 1993.

[46] A. Mili, R. Mili, and R.T. Mittermeir. Storing and retrieving software components: a refinement based approach. In *Proc. 16th International Conference on Software Engineering*, Sorento, Italy, May 1994.

[47] R. Mili and J. Raymond. A measure of reusability. In *Proceedings, Reuse 94: Pragmatic Approaches to Reuse Integration*, Morgantown, WV, August 1994.

[48] R. Mili and J. Raymond. Assessing reusability: An economics based measure. In *Reuse 95*, Morgantown, WV, August 1995.

[49] R. Mili and J. Raymond. Measuring the reusability of a component: A return on investment approach. In *Proceedings, Sixth International Conference on Software Quality*, Ottawa, Ont., October 1996.

[50] R. Zalila (Mili). *A Relational Method for Specification Validation and Its Automated Support.* Thèse de Doctorat de Spécialité (PhD), University of Tunis II, Tunisia, May 1991.

[51] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet. *A Mathematical Foundation for Programming.* Allyn and Bacon, Boston, Ma, 1986.

[52] R. Mittermeir and R. Mili. Ex-ante reusability assessment. In *Proceedings, Fourth International Conference on Re-Technologies for Information Systems*, Bled, Slovenia, June 1995.

[53] R. T. Mittermeir and E. Kofler. Layered specifications to support reusability and integrability. *Journal of Systems Integration*, 3(3):273–302, sept. 1993.

[54] Th. Moineau and M.C. Gaudel. Software reusability through formal specifications. In *Proc. 1st Intl. Workshop on Software Reusability*, number Memo Nr 57 in technical reports, Dortmund, 1991.

[55] J. Morrison. *Software Reuse in Japan.* TTI, CO, 1992.

[56] E. Ostertag, J. Hendler, R. Prieto-Diaz, and C. Braun. Computing similarity in a reuse library system: An ai-based approach. *ACM TOSEM*, 1(3):205–228, July 1992.

[57] R. A. Paul. Metric-guided software reuse. In *Workhop ICSI'94*, Sao Paulo, Brazil, August 1994.

[58] D.E. Perry and S.S. Popovich. Inquire: Predicate-based use and reuse. In *Proceedings, Knowledge Based Software Engineering Conference*, Chicago, IL, September 1993.

[59] D.E. Perry and S.S. Popovich. Inquire: Predicate-based use and reuse. Technical report, AT&T Bell Laboratories, 1994.

[60] A. Podgurski and L. Pierce. Behaviour sampling: a technique for automated retrieval of reusable components. In *In Proceedings, 14th International Conference on Software Engineering*, pages 300–304, New York, NY: ACM Press, 1992.

[61] J. Poulin and J. M. Caruso. A reuse metrics and return on investment model. In *Proceedings: Advances in Software Reuse*, Lucca, Italy, March 1993.

[62] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34:88–97, 1991.

[63] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, 1987.

[64] A. P. Sage. *Economic Systems Analysis.* North Holland, New York, NY, 1983.

[65] S. Schach. *Software Engineering.* Aksen Associates, NY, 1993.

[66] R. W. Selby. Empirically analysing software reuse in a production environment. In W. Tracz, editor, *Software Reuse: Emerging Technology.* IEEE Computer Society Press, Los Alamitos, California, 1988.

[67] STARS. Repository guidelines for the software technology for adaptable, reliable systems (stars) program. Technical Report CDRL Sequence Number 0460, STARS, DoD, Washington, DC, March 1989.

[68] P. Suppes. *Axiomatic Set Theory.* New York, NY: Dover Publications, 1972.

[69] W. Tracz. *Software Reuse Myths.* Washington, DC: IEEE Computer Society Press, 1988.

[70] B. Weide, W. Ogden, and M. Sitaraman. Recasting algorithms to encourage reuse. *IEEE Software*, 11(5):80–88, September 1994.

[71] B.W. Weide, W.F. Ogden, and S.H. Zweben. Reusable software components. In M. Yovits, editor, *Advances in Computers*, pages 1–65. Academic Press, 1991.

[72] G. M. Weinberg. *The Psychology of Computer Programming.* Van Nostrand Reinhold, New York, NY, 1971.

[73] G. M. Weinberg. The psychology of improved programming performance. *Datamation*, pages 82–85, November 1972.

[74] G. M. Weinberg. The psychology of change in development methodology. *Shared-Guide*, pages 93–100, 1979.

[75] G. M. Weinberg and E. L. Schulman. Goals and performance in computer programming. *Human Factors*, 16(1):70–77, 1974.

[76] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications.* McGraw Hill, New York, NY, 1992.

[77] A. Moorman Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM TOSEM*, 4(2):146–170, April 1995.

[78] A. Moormann Zaremski and J. M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Mathodology*, 4(1):146–170, april 1995.

[79] A. Moorman Zremski and J. M. Wing. Specification matching of software components. In *Proceedings, SIGSOFT '95: Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. New York, NY: ACM Press, 1995.

Index