...TORY SERIES...

...ED SO... ...RE... ...APERS... ...LUME...

...NUMBER 1...

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-90-005

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME VIII

NOVEMBER 1990

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development Branch)

The University of Maryland (Computer Sciences Department)

Computer Sciences Corporation (Systems Development Operation)

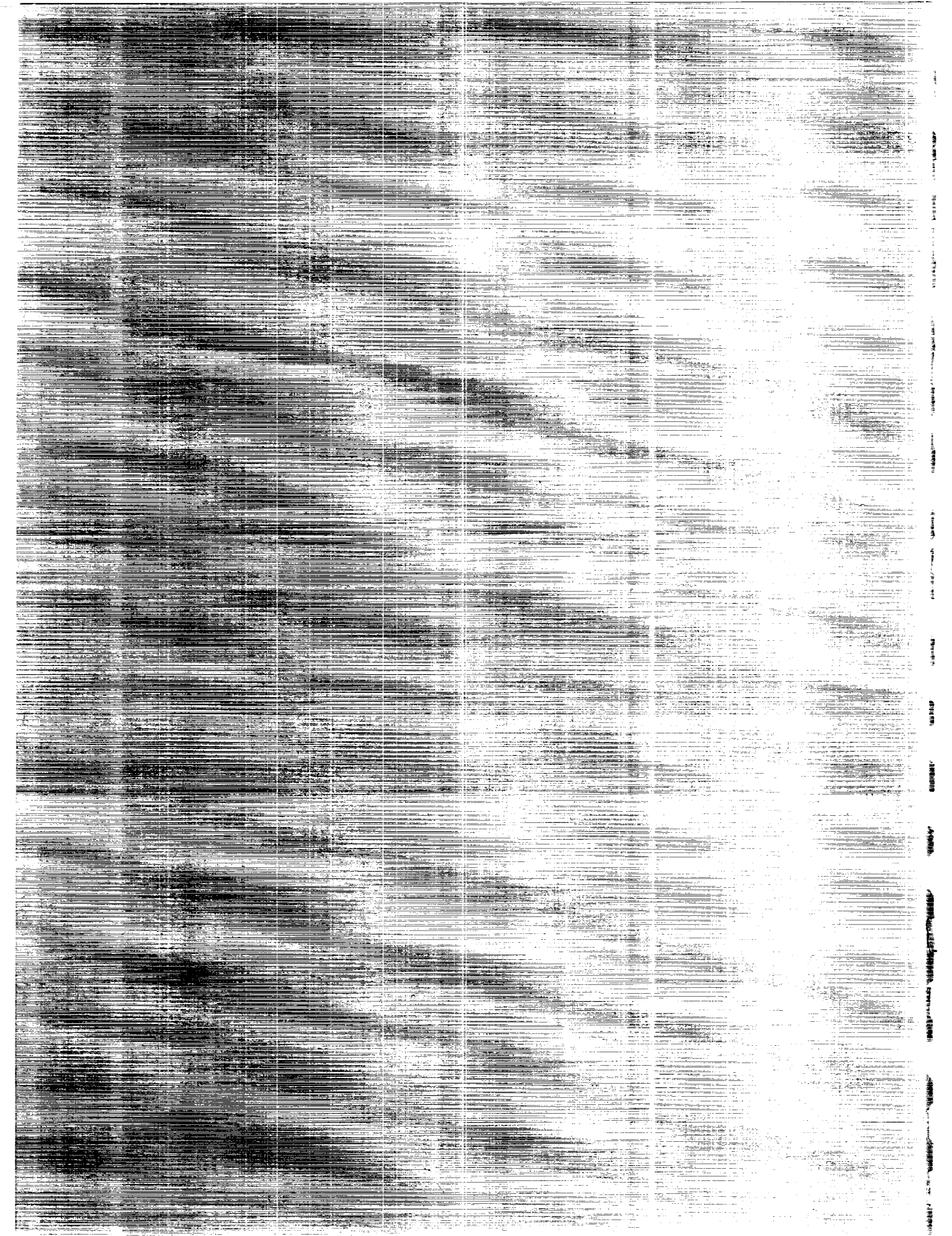
The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. The papers contained in this document appeared previously as indicated in each section.

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

TABLE OF CONTENTS

SECTION 1–INTRODUCTION



SECTION 1 - INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) during the period November 1989, through October 1990. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the eighth such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document.

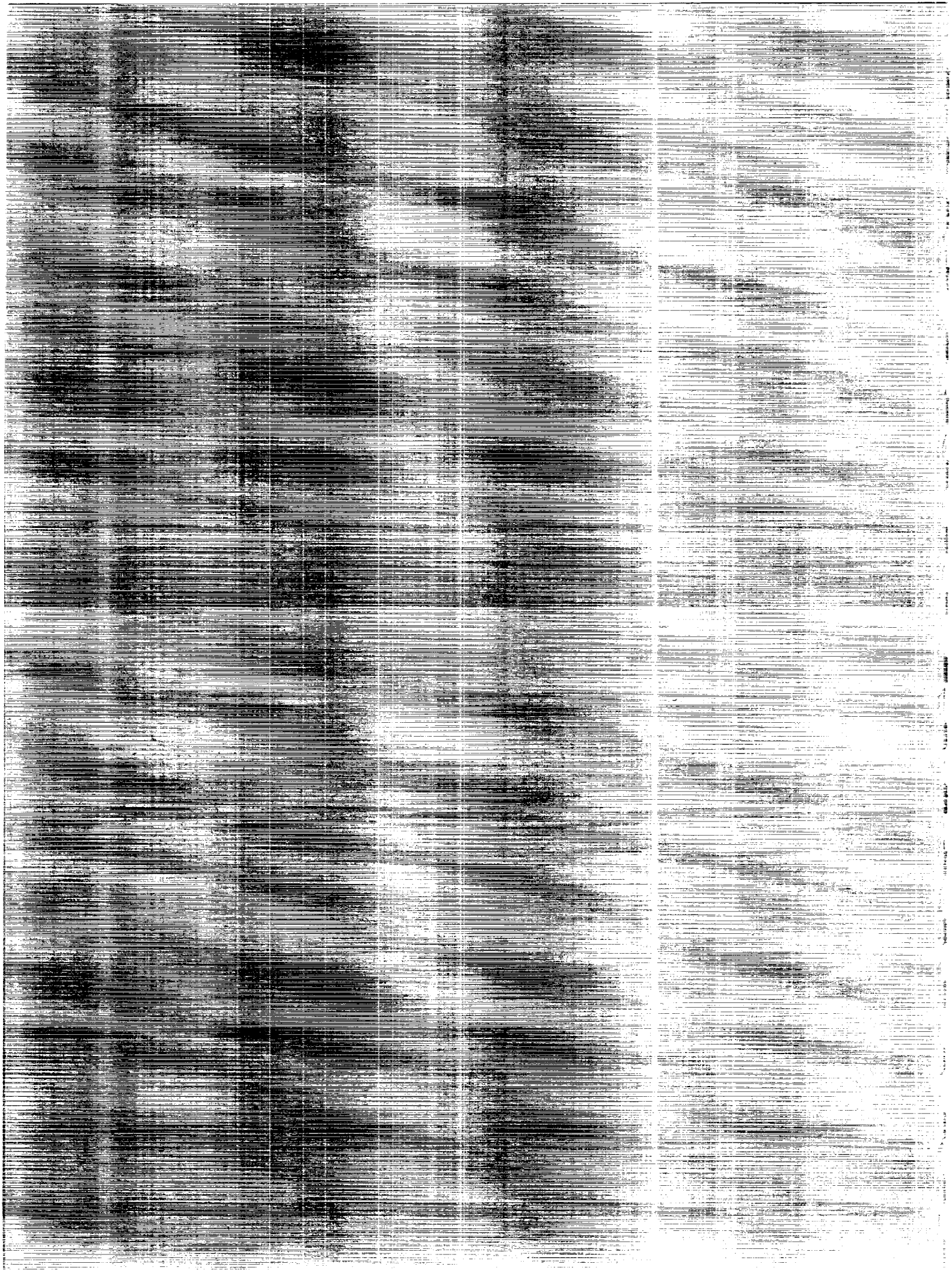
For the convenience of this presentation, the seven papers contained here are grouped into four major categories:

- Software Measurement Studies
- Software Models Studies
- Software Tools Studies
- Ada Technology Studies

The first category presents experimental research and evaluation of software measurement; the second category presents studies on models for software reuse; the third category presents a software tool evaluation; the last category represents Ada technology and includes studies in the areas of reuse and specifications.

The SEL is actively working to understand and improve the software development process at Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the Collected Software Engineering Papers and other SEL publications.

SECTION 2–SOFTWARE MEASUREMENT STUDIES



SECTION 2 - SOFTWARE MEASUREMENT STUDIES

The technical paper included in this section was originally prepared as indicated below.

- "Design Measurement: Some Lessons Learned," H. Rombach, IEEE Software, March 1990

Design Measurement: Some Lessons Learned

H. Dieter Rombach, *University of Maryland at College Park*

Most software measurements are derived from source code. A promising addition to the field is design measurement, which applies measurement principles to front-end products and processes.

Measurement is becoming recognized as a useful way to soundly plan and control the execution of software projects. However, current measurement practices are deficient in four ways:

- They emphasize the back end of development, mainly the coding and testing phases;
- they are biased toward software products, as opposed to processes;
- they are based on unsound measurement methodologies; and
- they are not integrated with development activities.

In short, most software measurements are derived solely from source code. *Design* measurement — as Figure 1 illustrates — is the application of measurement to design *processes* (the word I use to refer to all kinds of activities) and/or the resulting design *products* (the word I use to refer to all kinds of documents).

Design measurement is a desirable addition to traditional code-based measures because it lets you capture important aspects of the product and the process earlier in the life cycle so you can take corrective actions earlier. In turn, this benefit leads to a potentially high payoff, since we know that errors are more costly if committed early in the life cycle and not caught until much later.

In this article, I extract from several measurement projects some of the important lessons I have learned about measurement in general and design measurement in particular. I have synthesized these lessons into a design-measurement framework in an attempt to communicate my personal measurement experience to other software engineers.

My general measurement experience was gained on the Distos/Incas[1] project at the University of Kaiserslautern, West Germany; several projects at the National Aeronautics and Space Administration's Software Engineering Laboratory at the

0740-7459/90/0300/0017/$01.00 © 1990 IEEE

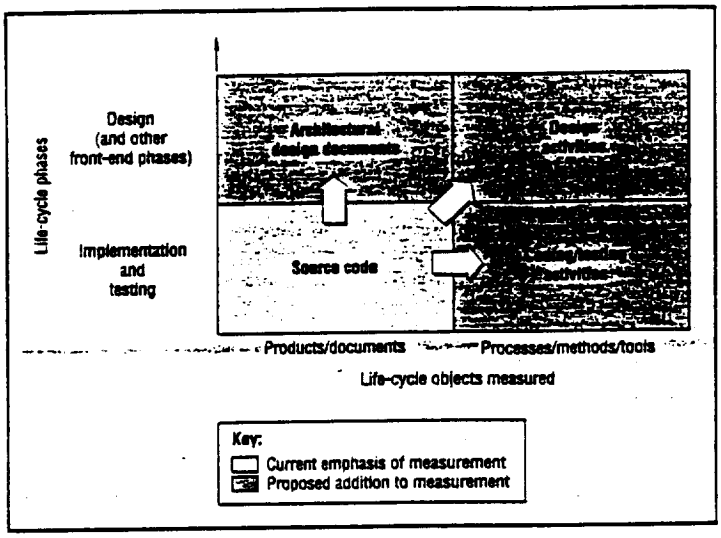


Figure 1. The scope of measurement.

Goddard Space Flight Center in Greenbelt, Md.; and the Tailoring a Measurement Environment project at the University of Maryland.

My design-measurement experience was gained on the Distos/Incas project, which measured intercomponent dependencies to predict future maintenance behavior; a study at the University of Maryland that compared the effect of various design methods; and various studies at the Software Engineering Laboratory to develop quantitative design baselines.

Measurement

From each of these projects, I learned important lessons about "effective" software measurement. These lessons tended to fall in three areas:

- how measurement must be applied in individual experiments or case studies,
- how measurement can help continuously improve an organization's state of the practice, and
- why measurement requires automated support.

Experiments and case studies. As part of the Distos/Incas project on distributed systems, my colleagues and I developed the object-oriented language Lady.[1] One objective of this language was to improve the maintainability of the distributed software written in it. The funding agency, the German Ministry for Research and Technology, requested empirical evidence of whether (and to what degree) this objective had been met. To find out, we conducted a large, controlled experiment in which we developed 12 systems, six in Lady and six in a traditional procedural language. We then studied their consequent maintenance.[2]

This experiment has taught us seven lessons:

- There are many types of measurement goals. Measurement goals can differ in the type of object the measurement focuses on, in their intended effect on the object, and in the people interested in them. A measurement goal may focus on object types such as processes, products, languages, methods, and tools. Its intended effect is either passive (when you want to understand the object) or active (when you want to predict, control, and improve the object). The people interested range from language and tool developers to customers and users.

In the Distos/Incas experiment, we had two main goals. First, we wanted to determine and explain differences in the maintenance behavior of systems implemented in Lady and those implemented in the traditional language. Second, we wanted to predict the maintenance behavior of Lady systems based on structural complexity.

The first goal focused on the languages used; its intended effect was passive because it was meant to help us understand Lady's effect on maintainability; and it reflected the interest of the language designers. The second goal focused on the product and maintenance process; its intended effect was active because it was meant to help us guide and control the appropriate use of Lady to build maintainable systems; and it reflected the interest of the managers and developers planning to use Lady.

- Models and measures are inseparable. Measures are intended to characterize some aspect of a software object in quantitative terms, but different models of the same aspect are possible. Without an explicit specification of the chosen model, it is impossible to judge the appropriateness of the quantitative measures selected.

In the Distos/Incas experiment, the maintainability model was based on the cost required to perform a change during maintenance and the effect of the change on the maintained product. With this model, measures like "effort in staff hours to perform a change" and "number of modules affected by a change" were justified. To predict maintenance behavior based on structural complexity, we chose Sallie Henry and Dennis Kafura's model for information flow between components.[3] In this model, measures such as "number of incoming information flows per module" and "number of outgoing information flows per module" were justified.

- You need different types of measures. We learned that you need both *abstract* and *specific* measures, *process* and *product* measures, *direct* and *indirect* measures, and *objective* and *subjective* measures.

Most measures reported in the literature are based on some abstract model (for example, control-flow measures based on abstract program graphs). Such abstract measures must be tailored to the specific characteristics of the object to be measured (for example, Ada control-flow measures must be based on Ada's specific control-flow features).

Product measures (such as design complexity) are not sufficient to support active measurement goals. Planned improvement of quality and productivity is only possible through measurably improved (such as fewer design errors) development processes.

Direct measures are intended to quantify some quality aspect (the number of staff hours spent on design is a direct measure of design cost, for example); indirect measures of some quality aspect are in-

tended to predict this quality based on other information that can be derived earlier (for example, the number of product requirements may be an indirect measure to predict the number of staff hours you expect to be spent on design).

Objective measures (such as lines of code) are defined well enough so that two people should compute the identical value from the same object independently. Subjective measures (such as staff experience) are computed based on a subjective estimation or a compromise among a group of people. Objective measures are easier to automate than subjective measures.

In the Distos/Incas experiment, the measure "number of incoming information flows per module" is an abstract measure. To collect this measure, we had to determine how "incoming information flow" could be measured from programs implemented in Lady. The maintenance-effort measures are process measures; the structural-complexity measures are product measures. The maintenance-effort measures are direct measures of cost; the structural-complexity measures are direct measures of product complexity and indirect measures of maintenance cost and effect — they do not directly characterize maintenance cost and effect but are expected to help predict them.

- Measurement-based analysis results are only as good as the data they are based on. It is important to recognize the limits of interpreting measures depending on their scale (that is, nominal, ordinal, interval, or ratio) and the validity of the underlying data. Validating data is a very time-consuming and often underestimated task. However, the sensitive task of interpreting data becomes guesswork if you try to use inappropriate interpretations or fail to consider the validity of the underlying data.

In the Distos/Incas experiment, we used the complexity measures only as ordinal measures because we felt that they could predict that a more complex Lady system would require more effort per maintenance change, but not how much more. About half my time on the Distos/Incas experiment was spent on data validation.

- You need a sound experimental approach. A measurement-based experiment requires extensive planning, tedious data collection and validation, and careful interpretation of the collected data. As in other experimental disciplines, you need a formal approach to experimentation.

In the Distos/Incas experiment, we formulated an approach for the experimental validation of structural-complexity measures. Our approach has six steps:

1. Model the quality of interest (maintainability, in this case) and quantify it into direct measures.
2. Model the product complexity in a way that lets you identify all the aspects that may affect the quality of interest.

A measurement-based experiment requires extensive planning, tedious data collection and validation, and careful interpretation.

3. Explicitly state your hypotheses about the effect of product complexity on the quality of interest.
4. Plan and perform an appropriate experiment or case study, including the collection and validation of the prescribed data.
5. Analyze the data and validate the hypotheses.
6. Assess the just-completed experimental validation and, if necessary, prepare for future experimental validations by refining the quality and product-complexity models, your hypotheses, the experiment itself, and the procedures used for data collection, validation, and analysis. In a way, this step is a built-in validation and improvement of the experimental validation itself.

- You must report specific measurement results in context. It is not useful to report measurement results from an experiment or case study without carefully characterizing the study's context. The way you present your results should put the reader in a position to repeat the experiment or case study. Only then can the reader agree or disagree with your conclusions. It is not only useless to present results out of context, it is also dangerous, because it may lead to inappropriate perceptions.

I have published some of the results of the Distos/Incas experiment together with the necessary context information.[2] The results suggest that Lady programs are more maintainable than traditional-language programs and that structural complexity is a useful predictor for a component's maintainability.

The advantage of providing the experimental context is that readers can agree or disagree with the experimental approach chosen. For readers who disagree with the approach, the results have no value; for readers who agree with the approach, the results may confirm or add to their current understanding. In the Distos/Incas experiment, we found that structural complexity cannot be compared across language boundaries based on the suggested language-specific complexity measures. However, the proposed abstract complexity measures seem appropriate.

- You must assess each experimental validation itself. It is important to transfer knowledge gained from one experiment to the next. This lets you state better goals, use better measures, and interpret the results in a broader context.

In the Distos/Incas experiment, this assessment was explicitly integrated, as step 6, into our experimental approach. As a consequence of this postmortem assessment, we posed many new questions, some of which led to the follow-up experiments I outline later.

Continuous improvement. At the University of Maryland, we have developed a general measurement approach called the goal/question/metric paradigm.[4] The GQM paradigm is broader in scope and formulated in more operational terms than the specific experimental approach applied in the Distos/Incas experiment. However, both agree on two major measurement principles: First, measurement must be top-down — measurement goals define what measures should be collected. Second, the data interpretation must take place in the context of some

goal and hypothesis.

The GQM paradigm has four steps:

1. State measurement goals in operational terms. You do this step using templates, which help you formulate goals and refine them into questions and measures.
2. Plan measurement procedures to support the collection and validation of data needed to compute the measures prescribed in step 1.
3. Collect and validate data.
4. Analyze and interpret the collected data and measures in the context of the questions and goals stated in step 1.

We have expanded the GQM paradigm into the quality-improvement paradigm, which aims to facilitate continuous improvement of an organization's software-engineering practices.[5] The quality-improvement paradigm embodies three basic measurement principles: First, measurement must be applied continuously to all projects in an organization. Second, measurement must be an integral part of each project — "development" must include *both* software construction and measurement. Third, the experience gained from each project must be recorded in a measurement database and be made available to future projects.

The quality-improvement paradigm has six steps:

1. Characterize the project environment.
2. State improvement goals in operational terms. Again, this is done through templates that help you formulate goals and refine them into questions and measures.
3. Plan the project (by selecting appropriate methods and tools) and the measurement procedures to support the collection and validation of data prescribed in step 2.
4. Perform the project and the data collection and validation.
5. Analyze and interpret the collected data in the context of the questions and improvement goals stated in step 2.
6. Return to step 1 armed with the experience gained from this project.

Applying the quality-improvement paradigm at NASA's Software Engineering Laboratory has led to a broad body of measurement experience.[5,6] At the SEL, the quality-improvement paradigm is now an integral part of development (and just recently maintenance) activities to identify the quality goals of interest, use standard measurement procedures to collect the necessary data from ongoing production projects, validate and interpret the data, and maintain a corporate measurement database.

The Goddard Space Flight Center has benefited from this measurement-based improvement approach in many ways, ranging from a better understanding of the weaknesses and strengths of its environment, to better planning, to the development of a new standard set of development methods and tools, to higher productivity and the production of higher quality software.

Introducing measurement to improve an organization's development practices requires fundamental changes of the organization.

My own active involvement in the SEL has helped me better understand several issues related to the introduction of measurement into a production environment:

• Introducing measurement has far-reaching consequences. Introducing measurement to improve an organization's development practices requires fundamental changes of the organization. It does not just add data collection to the existing development activities — it *really* changes the existing development activities by making them more transparent.

In addition, the effective incorporation of measurement into an organization requires changes in the reward structure so it is consistent with the goals motivated by measurement and so the additional efforts spent on data collection and validation are rewarded. All in all, measurement can reveal the advantages and disadvantages of current practices and spur changes. Inappropriate measures can be countereffective because they may cause the *wrong* changes.

At the SEL, each project member fills out a data-collection form every time he makes a change — to capture the nature, cost, and effect of that change — and weekly — to capture the effort spent on activities and products. Filling out these forms has become as routine as writing code. Special measurement employees validate the collected forms, maintain the measurement database, and produce periodical reports.

• You must justify the cost of measurement. Measurement costs! The cost is acceptable if it is justified by the expected quality and productivity improvements. Measurement itself can be used to quantify the improvement potential by capturing the amount of rework, for example. The GQM paradigm itself helps you build the case that investment in capturing certain measures may pay off by tying them to an organization's obvious improvement needs.

At the SEL, each project spends, on average, 3 percent of its budget on data collection and validation. The organization spends an additional 4 to 6 percent on off-line data processing and analysis. However, you should expect a higher investment up front to build a new program.

• We must address both technology-transfer and research issues. The technology to establish an improvement program exists, as the SEL and other organizations have shown. Using the available technology is mainly a technology-transfer problem.

However, there are important areas that need more research. These areas include the formalization of measurement planning, support for data interpretation, support for learning based on measurement results and reusing what has been learned across projects and environments, and the appropriate automated support for all these activities, especially the appropriate organization of corporate measurement databases.

One of the largest corporate measurement databases exists at the SEL. Built over the last 12 years, it includes measurement data on product characteristics (size and complexity), process characteristics (effort, changes, and defects), the effec-

tiveness of methodologies (what types of faults were easily detected using method X), and project characteristics (methods and tools used, and personnel experience).[6] At first, measurement covered only the development stages, but maintenance has recently been added.[5]

Automated support. Much research remains to be done to properly integrate measurement into software development and maintenance and to provide automated support in the form of software-engineering environments.

In the Tailoring a Measurement Environment project at the University of Maryland, we address all these measurement-related issues in the context of the framework provided by the quality-improvement paradigm.[4] We try to formalize models and we support characterizing corporate environments, planning construction and measurement activities, collecting, validating, and analyzing data, and learning from the measurement results to do a better job in the next project. We are developing a series of TAME prototypes based on an architecture that supports all these activities.

From the TAME project, we have learned that:

- You need automated support. The amount of information accumulated in an organization that applies a measurement-based improvement approach cannot be handled manually. Also, without automated support, results cannot be made available to interested people in real time so they can be used to support project decisions.

In establishing the SEL program, we initially collected data without database support. After about six months of collecting maintenance data from only two projects, we depended on database support to maintain control of the data-collection process.

It takes more than just tools to support the automated collection of product data. We also need automated support that spans the entire set of measurement activities suggested by the quality-improvement paradigm. In the TAME project, we are developing tool support for the formulation of goals, the derivation of measures, the interpretation of data, the reporting of measurement results, and the maintenance of an experience base.

- You must integrate construction and measurement support. Measurement processes must be tailored to the construction processes they are to measure. The construction processes, on the other hand, must be designed to be measurable to the degree necessary.

Often, measurement is expected to answer questions about the construction process that cannot be answered based on the way construction is performed. Very often, the reason for such inconsistencies is that there exists no explicit agreement on how construction is or should be performed.

I distinguish between two design steps: architectural, or high-level, design and algorithmic, or low-level, design.

It is very hard to tailor measurement to heuristic construction processes. To address this problem, we are developing a language that lets us model any development process explicitly and instrument that process for measurement.[7] The explicit specification of some construction process may help clarify what the limitations for measuring it are and whether the need for additional measurements is urgent enough to consider changing the construction process to make it more measurable.

Design measurement

I distinguish between two design steps: architectural, or high-level, design and algorithmic, or low-level, design. Architectural design involves identifying software components and their interconnection; algorithmic design involves identifying data structures and the control flow within the architectural components.

Most design measurement reported in the literature measures product complexity at the end of the algorithmic design phase from program-design-language documents. Many of these measures (such as Tom McCabe's cyclomatic-complexity measure) capture product aspects equally well from program-design-language documents and source code, so it is not surprising that the results derived through these design measures do not differ from results derived through corresponding source-code measures.

In this article, I use the term "design measure" to refer to architectural design measures. In this context, the measurement of designs is more complicated because typically less information is documented in a formal, measurable way at this early stage.

When you try to measure software designs, you realize that the potential for measurement is limited by the measurability of the design documents. There is very often a discrepancy between the need for measuring a design aspect (such as number of separate design decisions) and its measurability or lack thereof (many design decisions are documented very informally or not at all).

Therefore, design measurement, more so than code measurement, can not only capture design aspects quantitatively, but it can also drive the development and use of more formal, better measurable design approaches. The same argument can be made in the case of design processes, which are typically heuristic rather than formally specified.

Design characterization. We need a way to characterize software designs based on architectural design measures. In the Distos/Incas experiment, we developed design measures accidentally when we tried to compare the structural complexity of products implemented in languages based on different structural concepts. To do so, we had to resort to comparisons at some abstract level.

We defined an abstract model that was general enough to be instantiated into the precise models underlying each language. In that regard, the abstract structural model represented the greatest common denominator between the different language-specific structural concepts (I have described the abstract model's in-

stantiations elsewhere[5]).

We then realized that the abstract model could also be instantiated to measure intercomponent complexity during design — completely for collection at the end of algorithmic design, partially at the end of architectural design.

From this experience, we have learned:

• Specific measures derived from the same abstract model can easily be compared across life-cycle phases. Abstract models and measures let you instantiate compatible measures to trace some design aspect across several life-cycle phases. Compatible measures help identify the life-cycle phases in which the aspect of interest (in our case, structural complexity) is predominantly addressed.

In the Distos/Incas experiment, we measured and traced structural complexity through several consecutive life-cycle phases — from architectural design through coding. It became obvious that most of the important structural decisions had been made irreversibly by the end of architectural design.

• It is difficult to isolate and understand the effects of design methods. This is due in part to the creative nature of the design process itself and in part to the heuristic and therefore unpredictable (as to their effect) nature of most design methods.

In the Distos/Incas experiment, we were tempted to attribute the observed superiority of systems implemented in Lady to the language's advanced structural features. This seemed to be a valid conclusion because we had kept all the other potentially contributing factors as constant as possible (we had trained students similarly, used the same design-tool support, and so on).

However, follow-up interviews led us to believe that the major contributor was the object-oriented design approach that we had tailored to support Lady's structural concepts. This means that, in this study, the synergy of language concepts and design support contributed the real benefits. However, we were convinced that appropriate design support in isolation promises more payoff than language support in isolation. Our conclusion agrees with other experience (in the Ada community, for example) that the best language concepts are useless without guidelines and support for their effective use.

Later studies at the SEL evaluated the potential effect of different design approaches on the resulting design documents. These results made us question our previous conclusion because they revealed that the designer's experience and background is much more important than the design approach used.[6]

• Architectural design information has more influence on maintainability than algorithmic design information. Several publications have described the relative importance of different algorithmically oriented design-complexity measures. Our experience suggests that it may not be worth distinguishing among them because they all seem to be relatively unimportant compared to intercomponent complexity.

Architectural design information has more influence on maintainability than algorithmic design information.

In the Distos/Incas experiment, we compared some algorithmic design measures (such as lines of code and McCabe's cyclomatic-complexity measure), some architectural design measures (such as Henry and Kafura's information-flow measure), and some hybrid design measures (combinations of architectural and algorithmic measures) regarding their ability to predict maintenance behavior.

As Figure 2 shows, in isolation, the algorithmic measures showed no significant correlation with maintainability. However, the architectural measures did (correlations in the range of 0.7 to 0.8, with a significance level of less than 0.01). The hybrid measures had only a slightly higher correlation with maintainability than the architectural measures, but there was no difference among them based on the algorithmic measure used. Overall, the correlations of hybrid design measures with maintainability were only about 0.1 lower than the correlations of the same measures computed from source code.

• The dependency between construction and measurement is even more obvious during design than it was during coding. If we believe it is important to measure certain architectural design product or process aspects, we must ensure their measurability.

Design product documentation methods vary in formality — ranging from informal English to (semi) formal graph notations. Most design product measures are tailored to capture the aspects formally specified according to a specific method. Thus, they cannot be applied across environments that use different design methods.

The creative nature of the design process means that many aspects cannot be formalized, and consequently measured, at all. While formalization (and consequent automation) is a solution for more mechanical processes (such as compilation), it is not feasible for design processes. The only feasible way to make complex creative processes more manageable and measurable is to divide them into smaller processes with well-defined interfaces that can be checked — the divide-and-conquer principle.

In the Distos/Incas project, we applied the divide-and-conquer principle in the form of a stepwise, refinement-oriented design process. (The Cleanroom method uses a similar but much more formal approach.[3]) In our approach, formal specifications were iteratively refined into lower level specifications. After each refinement step, the result is proven correct with respect to the input specification. This approach let us control the design process and lent itself to measurement (such as the number of design decisions and how much complexity each design step adds to the design document).

Design predictability. We must develop ways to predict maintainability with architectural design measures. This means that we must understand the relationship between a component's design characteristics and its maintenance behavior.

In the Distos/Incas project, we used our stepwise design approach and measured

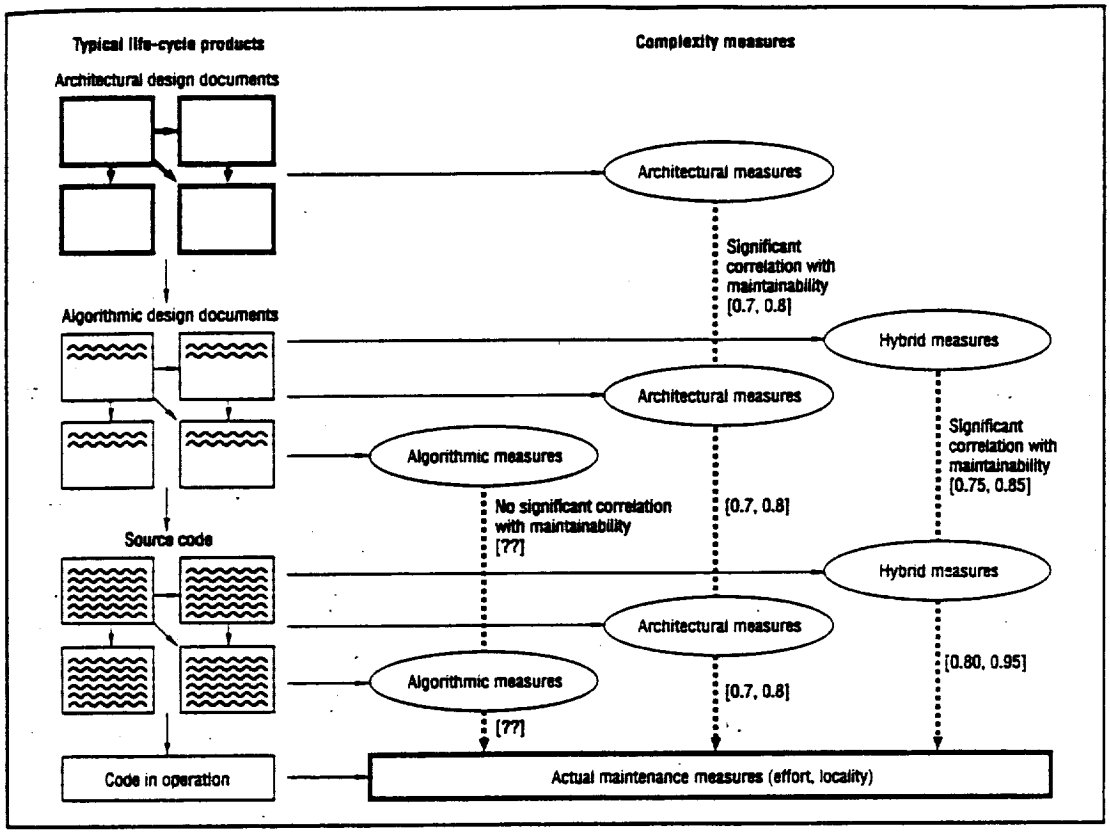


Figure 2. The capabilities of complexity measures to predict maintainability during design and coding phases.

the architecture of Lady components. We then measured their maintenance behavior with some effort- and error-based measures.

From this, we learned that:

- You *can* use design measures to predict maintainability. Generally, the design phase is considered to be where the signature of a system is created. If we can measure during that phase, we should be able to use this information to predict many process and product aspects as the life cycle progresses.

In the Distos/Incas experiment, we found that design measures could predict maintenance, locality, isolation effort, modification, and understandability almost as well as the corresponding code measures. Some of the measures were applicable as early as the end of architectural design.

- We should expand the definition of design measures. The Distos/Incas experiment supports the belief that true leverage is possible from measuring and understanding the architectural aspects of the design product and process, as opposed to the algorithmic aspects measured by traditional design measures. For example, design-process measures could capture design effort, errors committed and corrected during design, the effectiveness of design methods in supporting fundamental design principles, and the human aspect in assessing design alternatives and resolving conflicts.[10]

In the Distos/Incas experiment, we concentrated on measuring the structural aspects of design products. However, we also evaluated the stability of designs created according to design methods that supported different structural language concepts. It was very clear from comparisons of the evolving design versions and from the designers' comments that the design method tailored to the Lady language (which identifies three structural levels) resulted in fewer redesigns than the method tailored to traditional languages (which typically identify only two structural levels[2]).

In the SEL, we use a wide spectrum of design measures, ranging from subjective measures that capture the human experience with design methods, to measures that capture the effectiveness of design methods in preventing certain errors, to effort and error measures.[5]

- It is important to document all design decisions. It has long been recognized that missing design information makes it extremely difficult to maintain software efficiently. While the final design is important, the design rationale is at least as important if you are to understand design decisions and avoid recreating previously rejected design alternatives.

In the Distos/Incas experiment, we used more explicit design documents than are used in most production environments. However, the information-flow measures derived from the final design document had only average predictive capabilities. Further analysis revealed that whenever a component had implicit dependencies with other components its maintenance behavior was poorly predicted. Implicit dependencies between components included the use of the same constant, the use of the same algorithm, and architectural dependencies.[5]

These design decisions were not reflected explicitly in the final design document. Fortunately, we had stored all the versions created during development, so we could do a postmortem analysis to identify many implicit dependencies. This caused us to extend Henry and Kafura's information-flow model with implicit, global information flows.[8] The new design measure, which combines explicit and implicit information flows, was significantly more reliable in predicting maintenance behavior.

Research framework

Measurement is useful to understand, control, and improve products and processes based on objective data rather than subjective judgment. It also helps you build better models of processes and products. However, successful measurement requires more than a set of measures, just as successful design requires more than a set of design tools.

I suggest the following comprehensive design-measurement framework, which includes measurement approaches, mechanisms to model design aspects, the entire range of candidate design measures, and guidelines for reporting design-measurement results:

- Choose and tailor an effective measurement approach. I suggest the GQM and quality-improvement paradigms for both individual experiments and case studies as well as continuous organizational improvement. Both paradigms theoretically can exist without measurement, but you must measure if you want to evaluate and improve based on objective data rather than just subjective judgment.

Both paradigms incorporate measurement in a goal-oriented fashion: Measures serve goals! Both must be instantiated into an operational approach tailored to the specific environment characteristics.[4] In the TAME project, we developed templates and guidelines to help formally support the setting of goals and the derivation of measures.[4]

- Model the design aspects of interest. To use the paradigms properly, you must model the product and process aspects of interest. The product aspects of interest are those addressed and documented during the design phase (such as data and intercomponent structure, and control and information flow). The process aspects of interest are harder to model. In a separate project at the University of Maryland, we are developing a process-modeling language that acknowledges the need to specify mechanical and creative design aspects by combining imperative and constraint-oriented language principles.[7]
- Consider a variety of design measures. Candidate design measures address the design process and product, characterize design aspects directly and use design measures as indirect measures to help predict other qualities of interest (such as maintainability), and represent design information objectively, subjectively, and on different scales.

Design-process measures may capture effort distributions, defect profiles, or patterns of design-conflict resolutions. Design-product measures include measures of length, structural complexity, data-structure and dataflow complexity, and information-structure and information-flow complexity.

A direct design measure characterizes a design aspect. In comparing Lady systems to systems implemented in a traditional language, the measure "structural complexity in terms of incoming and outgoing information flows" was used as a direct measure of design-product complexity and the measure "effort in staff-hours spent on designing" as a direct measure of design cost.

An indirect design measure helps predict the expected value of a direct measure. To measure maintainability, meaningful direct measures might be "effort per maintenance change." The indirect design measure "structural complexity" has been identified in the Distos/Incas experiment to be a useful indirect measure for predicting maintainability.

Knowing the relationship between indirect and direct measures for a particular characteristic lets you predict whether requirements for this characteristic can be fulfilled and in turn, where necessary, to correct developments.

Objective design measures are preferred over subjective design measures. Examples of typical objective measures are "effort in staff hours spent on design" and "number of design components." Examples of typical subjective measures are "degree to which a design method was used" and "experience of staff with the design method." It is important to understand the scale of a given design measure and the corresponding implications on its interpretability.

- Define guidelines for reporting measurement results. The GQM and quality-improvement paradigms provide not only a good context for measurement but sound guidelines for reporting measurement results as well. You can use the steps of the quality-improvement paradigm as a structure to report results:

1. Characterize the environment to the degree necessary to understand the measurement goals, the experimental design, and the data interpretations.
2. Describe the measurement goals.
3. Describe the measures chosen.
4. Describe the experimental design, including procedures for data collection, validation, and analysis, as well as hypotheses.
5. Characterize the collected data.
6. Present the analysis results and validate the hypotheses.
7. Summarize the contribution of the results to the original goals and outline possible lessons for future measurement tasks.

Effective design measurement promises to contribute to quality and productivity. Design measurement has many dimensions and should be closely tied to the design methodology used. There are components of design-measurement technology available today, including general measurement approaches — the first TAME prototype is composed largely of available measurement technology.[4]

Design-measurement areas that require further research include the development of tractable (or measurable) design methods, the further formalization of measurement approaches, the identification of important design principles that need to be better understood through design measures, the integration of construction and measurement, and the quantification of intellectual design activities such as exploring and rejecting design alternatives.

Acknowledgments

My work with Juergen Nehmer and Victor Basili and my involvement in the Software Engineering Laboratory has contributed significantly to my understanding of design measurement. I thank Brad Ulery for reading and commenting on an early version of this article.

References

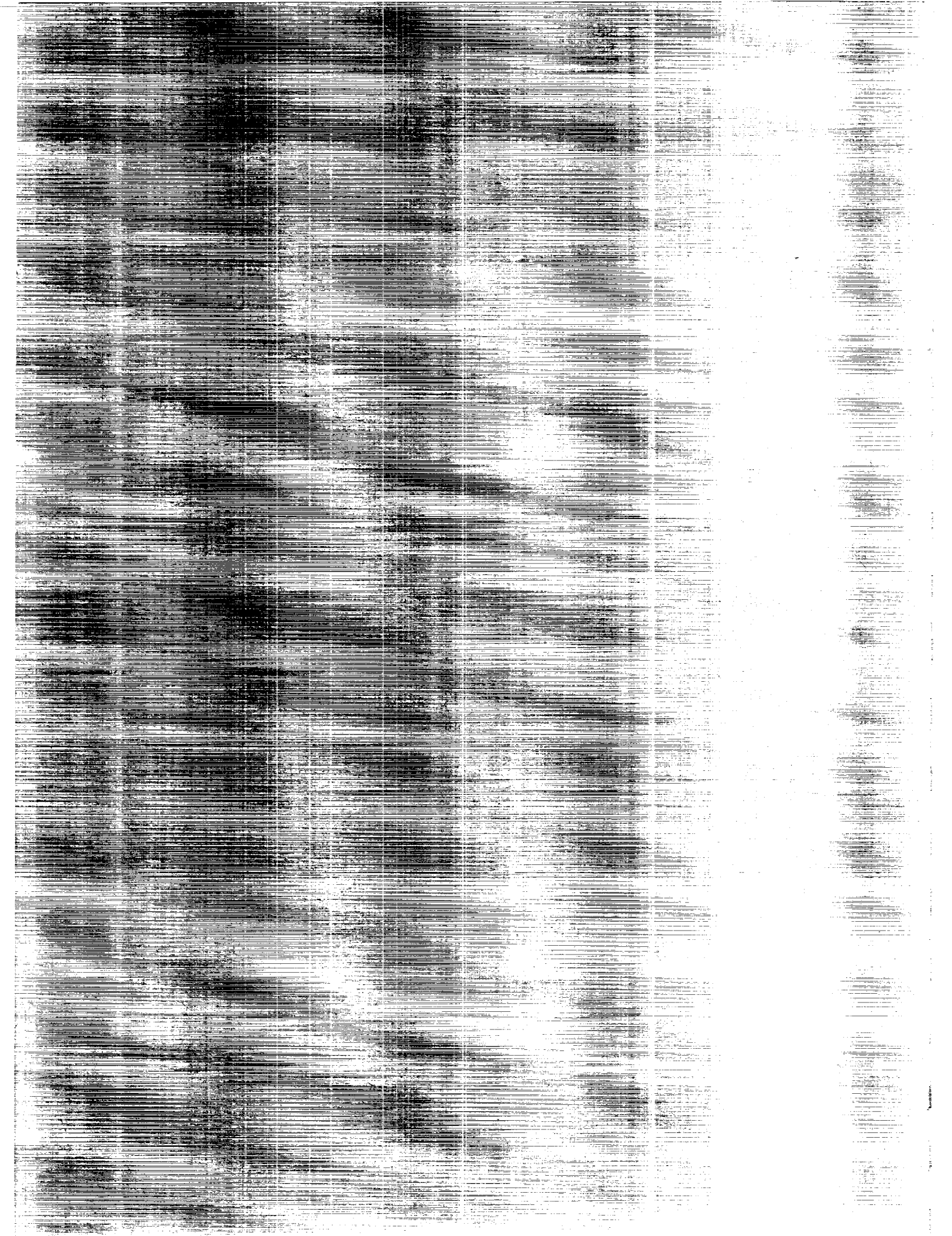
1. J. Nehmer et al., "Key Concepts of the Incas Multicomputer Project," *IEEE Trans. Software Eng.*, Aug. 1987, pp. 913-923.
2. H.D. Rombach, "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Trans. Software Eng.*, March 1987, pp. 344-354.
3. S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Trans. Software Eng.*, Sept. 1981, pp. 510-518.
4. V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, June 1988, pp. 758-773.
5. H.D. Rombach and B.T. Ulery, "Improving Software Maintenance through Measurement," *Proc. IEEE*, April 1989, pp. 581-595.
6. F.E. McGarry, S. Waligora, and T. McDermott, "Experiences in the SEL Applying Software Measurement," *Proc. 14th Ann. Software Eng. Workshop*, NASA/SEL Pub. SEL-89-007, NASA Goddard Space Flight Center, Greenbelt, Md., 1989.
7. H.D. Rombach and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proc. 22nd Hawaii Int'l Conf. System Sciences*, CS Press, Los Alamitos, Calif., 1989, pp. 165-174.
8. H.D. Rombach, "Software Design Metrics for Maintenance," *Proc. Ninth Ann. Software Eng. Workshop*, NASA/SEL Pub. SEL-84-004, NASA Goddard Space Flight Center, Greenbelt, Md., 1984.
9. R.W. Selby, V.R. Basili, and T. Baker, "Cleanroom Software Development: An Empirical Evaluation," *IEEE Trans. Software Eng.*, Sept. 1987, pp. 1,027-1,037.
10. B. Curtis et al., "On Building Software Process Models Under the Lamppost," *Proc. Ninth Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1987, pp. 96-103.

H. Dieter Rombach is an assistant professor of computer science at the University of Maryland at College Park. He is also affiliated with NASA's Software Engineering Laboratory and the University of Maryland's Institute for Advanced Computer Studies. His research interests include software methodologies, process and product measurement and modeling, integrated software-development environments, and distributed programming.

Rombach received a BS in mathematics and an MS in mathematics and computer science from the University of Karlsruhe, West Germany, and a PhD in computer science from the University of Kaiserslautern, West Germany. He is a member of the IEEE Computer Society, ACM, and the German Computer Society.

Address questions about this article to Rombach at Computer Science Dept., University of Maryland, College Park, MD 20742; CSnet dieter@cs.umd.edu.

SECTION 3–SOFTWARE MODELS STUDIES



SECTION 3 - SOFTWARE MODELS STUDIES

The technical papers included in this section were originally prepared as indicated below.

- Towards a Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes, V. Basili and H. Rombach, University of Maryland Technical Report TR-2446, April 1990
- "Viewing Maintenance as Reuse-Oriented Software Development," V. Basili, IEEE Software, January 1990

UMIACS–TR–90–47
CS–TR–2446

April 1990

Towards a Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*

Victor R. Basili and H. Dieter Rombach
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

Reuse of products, processes and related knowledge will be the key to enable the software industry to achieve the dramatic improvement in productivity and quality required to satisfy the anticipated growing demands. We need a comprehensive framework of models and model-based characterization schemes for better understanding, evaluating, and planning all aspects of reuse. In this paper we define requirements for comprehensive reuse models and related characterization schemes, assess state-of-the-art reuse characterization schemes relative to these requirements and motivate the need for more comprehensive reuse characterization schemes. We introduce a characterization scheme based upon a general reuse model, apply it and discuss its benefits, and suggest a model for integrating reuse into software development.

*Research for this study was supported in part by NASA grant NSG-5123, ONR grant NOOO14-87-K-0307 and Airmics grant DE-mail-84OR21400 to the University of Maryland.

TABLE OF CONTENTS:

1. INTRODUCTION

The existing gap between demand and our ability to produce high quality software cost-effectively calls for an improved software development technology. A reuse oriented development technology can significantly contribute to higher quality and productivity. Quality should improve by reusing proven experience in the form of products, processes and related knowledge such as plans, measurement data and lessons learned. Productivity should increase by using existing experience rather than creating everything from scratch. Many different approaches to reuse have appeared in the literature (e.g., [7, 9, 11, 13, 14, 15, 16, 21, 22, 23]).

Reusing existing experience is a key ingredient to progress in any area. Without reuse everything must be re-learned and re-created; progress in an economical fashion is unlikely. The goal of research in the area of reuse is the achievement of systematic approaches for effectively reusing existing experience to maximize quality and cost benefits.

This paper defines and demonstrates the usefulness of model-based reuse characterization schemes. From a number of important assumptions regarding the nature of software development and reuse we derive four essential requirements for any useful reuse models and related characterization schemes (Section 2). Existing models and characterization schemes are assessed with respect to these assumptions and the need for more comprehensive models and characterization schemes is established (Section 3). We introduce a reuse characterization scheme based on a general model of reuse (Section 4), and discuss its practical application and benefits (Section 5). Throughout the paper we use examples of reusing *generic Ada packages, design inspections, and cost models* to demonstrate our approach. Finally, we present a model for integrating and supporting reuse in software development (Section 6).

2. BASIC REQUIREMENTS FOR A REUSE CHARACTERIZATION SCHEME

The reuse approach presented in this paper is based on a number of assumptions regarding software development in general and reuse in particular. These assumptions are based on more than ten years of analyzing software processes and products [1, 3, 4, 5, 6, 19]. This section states our assumptions regarding development and reuse (Sections 2.1 and 2.2, respectively), and derives a set of characteristics required for any useful reuse characterization scheme (Section 2.3).

2.1. Software Development Assumptions

According to a common software development project model depicted in Figure 1, the goal of software development is to produce project deliverables (i.e., project output) that satisfy project needs (i.e., project input) [25]. This goal is achieved according to some development process model which coordinates personnel, practices, methods and tools.

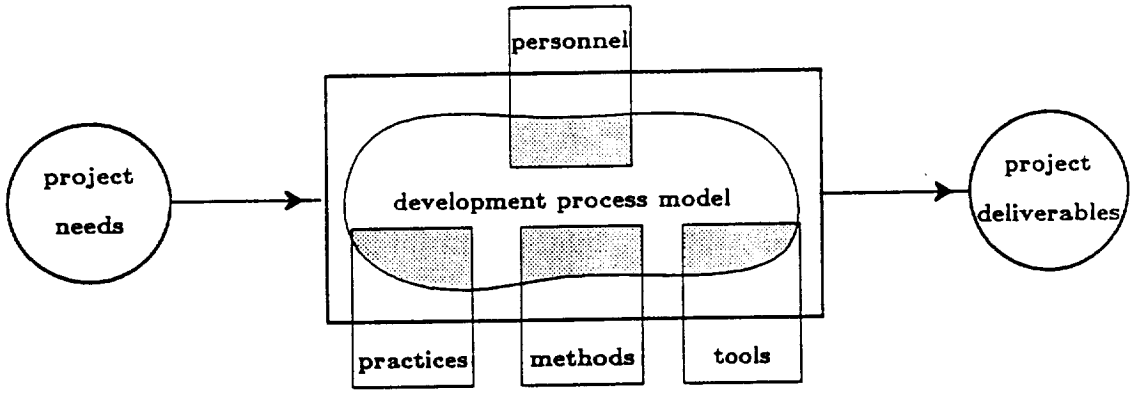


Figure 1: Software Development Project Model

With regard to software development we make the following assumptions:

(D1) **A single software development process model cannot be assumed for all software development projects:** Different project needs and other project characteristics may suggest and justify different development process models. The potential differences may range from different development process models themselves to different practices, methods and tools supporting these development process models to different personnel.

(D2) **Practices, methods and tools – including reuse–related ones – need to be tailored to the project needs and characteristics:** Under the assumption that practices, methods and tools support a particular development project, they need to be tailored to the needs and objectives, development process model, and other characteristics of that project.

2.2. Software Reuse Assumptions

Reuse–oriented software development (depicted in Figure 2) assumes that, given the project–specific need to develop an object 'x' that meets specification '$\bar{x}$', we take advantage of some already existing object 'x_k' e {'x_1', ..., 'x_n'} instead of developing 'x' from scratch. In this case, '$\bar{x}$' is not only the specification for 'x' but also the *reuse specification* for the set of reuse candidates 'x_1', ..., 'x_n'. Reuse includes the identification of a set of reuse candidates {'x_1', ..., 'x_k', ..., 'x_n'}, the evaluation of their potential to satisfy reuse specification '$\bar{x}$' effectively and the selection of the best–suited candidate 'x_k', the possible modification of the chosen candidate 'x_k' into 'x', and the integration of 'x' into the development process of the current project.

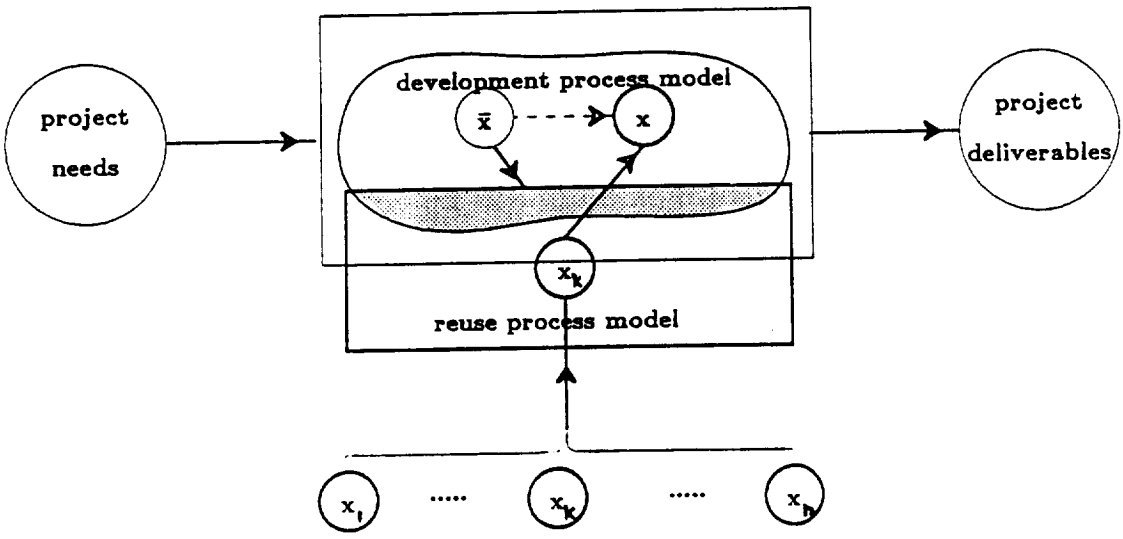


Figure 2: Reuse–Oriented Software Development Model

With regard to software reuse we make the following assumptions:

(R1) All experience can be reused: Typically, the emphasis is on reusing objects of type 'source code'. This limitation reflects the traditional view that software equals code. It ignores the importance of reusing software products across the entire life–cycle (which includes the planning as well as the production phases of a software development project), software processes and methods, and other kinds of knowledge such as models, measurement data or lessons learned.

The reuse of 'generic Ada packages' represents an example of product reuse. Generic Ada packages represent templates for instantiating specific package objects according to a parameter mechanisms. The reuse of 'design inspections' represents an example of process reuse. Design inspections are off-line fault detection and isolation methods applied during the module design phase. They can be based on different techniques for reading (e.g., ad hoc, sequential, control flow oriented, stepwise abstraction oriented). The reuse of 'cost models' represents an example of knowledge reuse. Cost models are used in the estimation, evaluation and control of project cost. They predict cost (e.g., in the form of staff–months) based on a number of characteristic project parameters (e.g., estimated product size in KLoC, product complexity, methodology level).

(R2) Reuse typically requires some modification of the object being reused: Under the assumption that software developments may be different in some way, modification of

experience from prior projects must be anticipated. The degree of modification depends on how many, and to what degree, existing object characteristics differ from their desired characteristics.

To reuse an Ada package 'list of integers' to organize a 'list of reals' we need to modify it. We can either modify the existing package by hand, or we can use a generic package 'list' which can be instantiated via a parameter mechanism for any base type.

To reuse a design inspection method across projects characterized by significantly different fault profiles, the underlying reading technique may need to be tailored to the respective fault profiles. If 'interface faults' replace 'control flow faults' as the most common fault type, we can either select a different reading technique all together (e.g., step-wise abstraction instead of control-flow oriented) or we can establish specific guidelines for identifying interface faults.

To reuse a cost model across projects characterized by different application domains, we may have to change the number and type of characteristic project parameters used for estimating cost as well as their impact on cost. If 'commercial software' is developed instead of 'real-time software', we may have to consider re-defining 'estimated product size' to be measured in terms of 'data structures' instead of 'lines of code' or re-computing the impact of the existing parameters on cost. Using a cost model effectively implies a constant updating of our understanding of the relationship between project parameters and cost.

(R3) Analysis is necessary to determine when and if reuse is appropriate: The decision to reuse existing experience as well as how and when to reuse it needs to be based on an analysis of the payoff. Reuse payoff is not always easy to evaluate. We need to understand (i) the objectives of reuse, (ii) how well the available reuse candidates are qualified to meet these objectives, and (iii) the mechanisms available to perform the necessary modification.

Assume the existence of a set of Ada generics which represent application-specific components of a satellite control system. The objective may be to reuse such components to build a new satellite control system of a similar type, but with higher precision. Whether the existing generics are suitable depends on a variety of characteristics: Their correctness and reliability, their performance in prior instances of reuse, their ease of integration into a new system, the potential for achieving the higher degree of precision through instantiation, the degree of change needed, and the existence of reuse mechanisms that support this change process. Candidate Ada generics may theoretically be well suited for reuse; however, without knowing the answers to these questions, they may not be reused due to lack of confidence that reuse will pay off.

Assume the existence of a design inspection method based on ad-hoc reading which has been used successfully on past satellite control software developments within a standard waterfall model. The objective may be to reuse the method in the context of the Cleanroom development method [18, 20]. In this case, the method needs to be applied in the context of a different life-cycle model, different design approach, and different design representations. Whether and how the existing method can be reused depends on our ability to tailor the reading technique to the stepwise refinement oriented design technique used in Cleanroom, and the required intensity of

reading due to the omission of developer testing. This results in the definition of the stepwise abstraction oriented reading technique [8].

Assume the existence of a cost model that has been validated for the development of satellite control software based on a waterfall life-cycle model, functional decomposition oriented design techniques, and functional and structural testing. The objective may be to reuse the model in the context of Cleanroom development. Whether the cost model can be reused at all, how it needs to be calibrated, or whether a completely different model may be more appropriate depends on whether the model contains the appropriate variables needed for the prediction of cost change or whether they simply need to be re-calibrated. This question can only be answered through thorough analysis of a number of Cleanroom projects.

(R4) Reuse must be integrated into the specific software development: Reuse is intended to make software development more effective. In order to achieve this objective we need to tailor reuse practices, methods and tools towards the respective development process.

We have to decide when and how to identify, modify and integrate existing Ada packages. If we assume identification of Ada generics by name, and modification by the generic parameter mechanism, we require a repository consisting of Ada generics together with a description of the instantiation parameters. If we assume identification by specification, and modification of the generic's code by hand, we require a suitable specification of each generic, a definition of semantic closeness of specifications so we can find suitable reuse candidates, and the appropriate source code documentation to allow for ease of modification. In the case of identification by specification we may consider identifying reuse candidates at high-level design (i.e., when the component specifications for the new product exist) or even when defining the requirements.

We have to decide on how often, when, and how design inspections should be integrated into the development process. If we assume a waterfall-based development life-cycle, we need to determine how many design inspections need to be performed and when (e.g., once for all modules at the end of module design, once for all modules of a subsystem, or once for each module). We need to state which documents are required as input to the design inspection, what results are to be produced, what actions are to be taken, and when, in case the results are insufficient, and who is supposed to participate.

We have to decide when to initially estimate cost and when to update the initial estimate. If we assume a waterfall-based development life-cycle, we may estimate cost initially based on estimated product and process parameters (e.g., estimated product size). After each milestone, the estimated cost can be compared with the actual cost. Possible deviations are used to correct the estimate for the remainder of the project.

2.3. Software Reuse Characteristics

The above software reuse assumptions suggest that 'reuse' is a complex concept. We need to build models and characterization schemes that allow us to define and understand, compare and evaluate, and plan the objectives of reuse, the candidate objects of reuse, the reuse process itself,

and the potential for effective reuse. Based upon the above assumptions, such models and characterization schemes need to exhibit the following characteristics:

(C1) **Applicable to all types of reuse objects**: We want to be able to characterize products, processes and all other types of related knowledge such as plans, measurement data or lessons learned.

(C2) **Capable of characterizing objects–before–reuse and objects–after–reuse**: We want to be able to characterize the reuse candidates (from here on called 'objects–before–reuse') as well as the object actually being reused in the current project (from here on called 'object–after–reuse'). This will enable us to (i) judge the suitability of a given reuse candidate based on the distance between its actual before–reuse and desired after–reuse characteristics, and (ii) establish criteria for useful reuse candidates (object–before–reuse characteristics) based on anticipated objectives for their (re)use (object–after–reuse characteristics).

(C3) **Capable of characterising the reuse process itself**: We want to be able to (i) judge the ease of bridging the gap between different object characteristics before– and after–reuse, and (ii) derive additional criteria for useful reuse candidates based on characteristics of the reuse process itself.

(C4) **Capable of being systematically tailored to specific project (i.e., development and reuse) needs and other characteristics**: We want to be able to adjust a given reuse characterization scheme to changing needs in a systematic way. This requires not only the ability to change the scheme, but also some kind of rationale that ties the given reuse characterization scheme back to its underlying model and assumptions. Such a rationale enables us to identify the impact of different environments and modify the scheme in a systematic way.

3. STATE–OF–THE–ART REUSE CHARACTERIZATION SCHEMES

A number of research groups have developed characterization schemes for reuse (e.g., [9, 11, 13, 21, 22]). The schemes can be distinguished as *special purpose schemes* and *meta schemes.*

The large majority of published characterization schemes have been developed for a special purpose. They consist of a fixed number of characterization dimensions. There intention is to characterize software products as they exist. Typical dimensions for characterizing source code objects in a repository are "function", "size", or "type of problem". Examples schemes include the schemes published in [11, 13], the ACM Computing Reviews Scheme, AFIPS's Taxonomy of Computer Science and Engineering, schemes for functional collections (e.g., GAMS, SHARE, SSP, SPSS, IMSL) and schemes for commercial software catalogs (e.g., ICP, IDS, IBM Software Catalog, Apple Book). It is obvious that special purpose schemes are not designed to satisfy the reuse modeling characteristics of section 2.3.

A few characterization schemes can be instantiated for different purposes. They explicitly acknowledge the need for different schemes (or the expansion of existing ones) due to different or changing needs of an organization. They, therefore, allow the instantiation of any imaginable scheme. An excellent example is Ruben Prieto–Diaz's facet–based meta–characterization scheme [14, 17]. Theoretically, meta schemes are flexible enough to allow the capturing of any reuse aspect. However, based on known examples of actual uses of meta schemes, such broadness seems not intended. Instead, most examples focus on product reuse, are limited to the objects–before–reuse, and ignore the reuse process entirely. Meta schemes were also not designed to satisfy the reuse modeling characteristics of section 2.3.

We have found that existing schemes – special purpose as well as meta schemes – do not satisfy our requirements. To illustrate the problems associated with their limitations, we use the following example scheme which can be viewed either as a special–purpose scheme or a specific

instantiation of a meta scheme*:

Each reuse candidate is characterized in terms of

- **name**: What is the object's name? (e.g., buffer.ada, sel_inspection, sel_cost_model)
- **function**: What is the functional specification or purpose of the object? (e.g., integer_queue, <element>_buffer, sensor control system, certify appropriateness of design documents, predict project cost)
- **use**: How can the object be used? (e.g., product, process, knowledge)
- **type**: What type of object is it? (e.g., requirements document, code document, inspection method, coding method, specification tool, graphic tool, process model, cost model)
- **granularity**: What is the object's scope? (e.g., system level, subsystem level, component level, module – package, procedure, function – level, entire life cycle, design stage, coding stage)
- **representation**: How is the object represented? (e.g., data, informal set of guidelines, schematized templates, formal mathematical model, languages such as Ada, automated tools)
- **input/output**: What are the external input/output dependencies of the object needed to completely define/extract it as a self–contained entity? (e.g., global data referenced by a code unit, formal and actual input/output parameters of a procedure, instantiation parameters of a generic Ada package, specification and design documents needed to perform a design inspection, defect data produced by a design inspection, variables of a cost model)
- **dependencies**: What are additional assumptions and dependencies needed to understand the object? (e.g., assumption on user's qualification such as knowledge of Ada or qualification to read, specification document to understand a code unit, readability of design document, homogeneity of problem classes and environments underlying a cost model)
- **application domain**: What application classes was the object developed for? (e.g. ground support software for satellites, business software for banking, payroll software)
- **solution domain**: What environment classes was the object developed in? (e.g., waterfall life–cycle model, spiral life–cycle model, iterative enhancement life–cycle model, functional decomposition design method, standard set of methods)
- **object quality**: What qualities does the object exhibit? (e.g., level of reliability, correctness, user–friendliness, defect detection rate, predictability)

Let's assess the above reuse characterization scheme relative to the four desired characteristics of section 2.3:

(C1) It is theoretically possible to characterize all types of experience according to the above scheme (in case of a meta scheme we could even create new ones). For example, a generic Ada package 'buffer.ada' may be characterized as having identifier 'buffer.ada', offering the function '<element>_buffer', being usable as a 'product' of type 'code document' at the 'package module level', and being represented in 'Ada'. The self–contained definition of the package requires knowledge regarding the instantiation parameters as well as its visibility of externally

* Characterization dimensions are marked with '–'; example categories for each dimension are listed in parenthesis.

defined objects (e.g., explicit access through WITH clauses, implicit access according to nesting structure). In addition, effective use of the object may require some basic knowledge of the language Ada and assume thorough documentation of the object itself. It may have been developed within the application domain 'ground support software', according to a 'waterfall life–cycle' and 'functional decomposition design', and exhibiting high quality in terms of 'reliability'.

(C2) The scheme is used to characterize reuse candidates (i.e., objects–before–reuse) only. However, in order to evaluate the reuse potential of an object–before–reuse in a given reuse scenario, one needs to understand the distance between its characteristics and the characteristics of the desired object (i.e., object–after–reuse). In the case of the Ada package example, the required function may be different, the quality requirements with respect to reliability may be higher, or the design method used in the current project may be different from the one according to which the package has been created originally. Without understanding the distance to be bridged between reuse requirements and reuse candidates it is hard to (a) predict the cost involved in reusing a particular object, and (b) establish criteria for populating a reuse repository that supports cost–effective reuse.

(C3) The scheme is not intended to characterize the reuse process at all. To really predict the cost of reuse we do not only have to understand the distance to be bridged between objects–before and objects–after–reuse (as pointed out above), but also the intended process to bridge it (i.e., the reuse process). For example, it can be expected that it is easier to bridge the distance with respect to function by using a parameterized instantiation mechanism rather than modifying the existing package by hand.

(C4) Their is no explicit rationale for the eleven dimensions of the example scheme. That makes it hard to reason about its appropriateness as well as modify it in any systematic way. There is no guidance in tailoring the example scheme to new needs neither with respect to what is to changed (e.g., only some categories, dimensions, or the entire implicitly underlying model) nor

how it is to be changed.

The result of this assessment suggests the urgent need for new, better reuse characterization schemes. In the next section, we suggest a model–based scheme which satisfies all four characteristics.

4. MODEL–BASED REUSE CHARACTERIZATION SCHEMES

In this section we define a model–based reuse characterization scheme satisfying the characteristics (C1–4) stated in section 2.3. We start this modeling approach with a very general reuse model satisfying satisfying the reuse assumptions, refine it step by step until it generates reuse characterization dimensions at the level of detail needed to understand, evaluate, motivate or improve reuse. This modeling approach allows us to deal with the complexity of the modeling task itself, and document an explicit rationale for the resulting model.

4.1. The Abstract Reuse Model

The general reuse model used in this section is consistent with the view of reuse represented in section 2.2. It assumes the existence of objects–before–reuse and objects–after–reuse, and a transformation between the two:

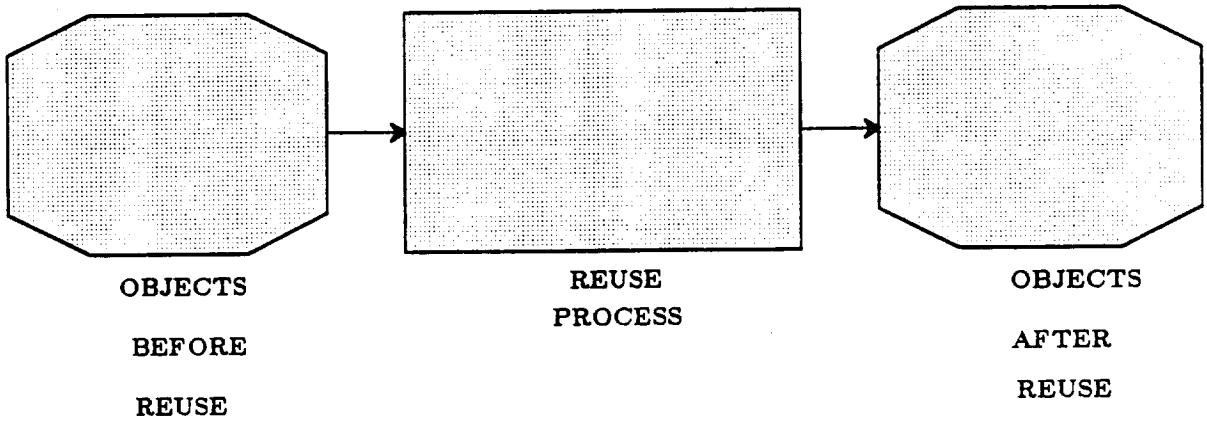


Figure 3: Abstract Reuse Model (Refinement level 0)

The objects–before–reuse represent experience from prior projects, have been evaluated as being of potential reuse value, and have been made available in some form of a repository. The objects–after–reuse are the (potentially modified) versions of objects–before–reuse integrated into some project other than the one they were initially created for. Object–after–reuse characteristics represent the 'reuse specification' for any candidate 'object–before–reuse'. Both the objects–before–reuse and the objects–after–reuse may represent any type of experience accumulated in the context of software projects ranging from products to processes to knowledge. The reuse process transforms objects–before–reuse into objects–after–reuse.

4.2. The First Model Refinement Level

Figure 4 depicts the result of the first refinement step of the general model of Figure 3.

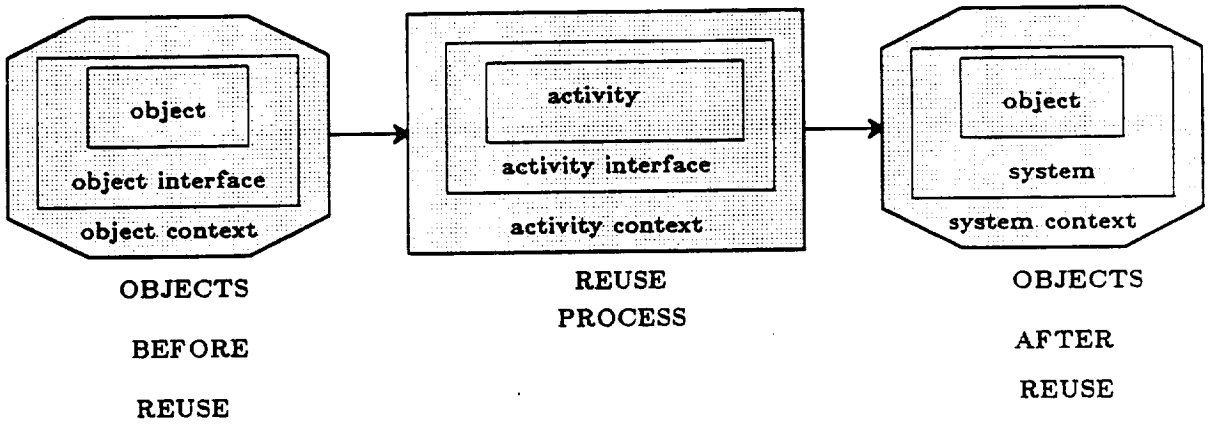


Figure 4: Our Reuse Model (Refinement level 1)

Each *object-before-reuse* is a specific candidate for reuse. It has various attributes that describe and bound the object. Most objects are physically part of a system, i.e. they interact with other objects to create some greater object. If we want to reuse an object we must understand its interaction with other objects in the system in order to extract it as a unit, i.e. *object interface*. Objects were created in some environment which leaves its characteristics on the object, even though those characteristics may not be visible. We call this the *object context*.

The *object-after-reuse* is a specification for a set of before-reuse candidates. Therefore, we may have to consider different attributes. The *system* in which the transformed object is integrated and the *system context* in which the system is developed must also be classified.

The *reuse process* is aimed at extracting the object-before-reuse from a repository based on the available object-after-reuse characteristics, and making it ready for reuse in the system and context in which it will be reused. We must describe the various *reuse activities* and classify them. The reuse activities need to be integrated into the reuse-enabling software development process. The means of integration constitute the *activity interface*. Reuse requires the transfer of experience across project boundaries. The organizational support provided for this experience transfer is referred to as *activity context*.

Based upon the goals for the specific project, as well as the organization, we must evaluate (i) the required qualities of the object–after–reuse, (ii) the quality of the reuse process, especially its integration into the enabling software evolution process, and (iii) the quality of the existing objects–before–reuse.

4.3. The Second Model Refinement Level

Each component of the First Model Refinement (Figure 4) is further refined as depicted in Figures 5(a–c) . It needs to be noted that these refinements are based on our current understanding of reuse and may, therefore, change in the future.

4.3.1. Objects–Before–Reuse

In order to characterize the object itself, we have chosen to provide the following six dimensions and supplementing categories: the object's name (e.g., buffer.ada), its function (e.g., integer_buffer), its possible use (e.g., product), its type (e.g., requirements document), its granularity (e.g., module), and its representation (e.g., Ada language). The object interface consists of such things as what are the explicit inputs/outputs needed to define and extract the object as a self–contained unit (e.g., instantiation parameters in the case of a generic Ada package), and what are additionally required assumptions and dependencies (e.g., user's knowledge of Ada). Whereas the object and object interface dimensions provide us with a snapshot of the object at hand, the object context dimension provides us with historical information such as the application classes the object was developed for (e.g., ground support software for satellites), the environment the object was developed in (e.g., waterfall life–cycle model), and its validated or anticipated quality (e.g., reliability).

The resulting model refinement is depicted in Figure 5a.

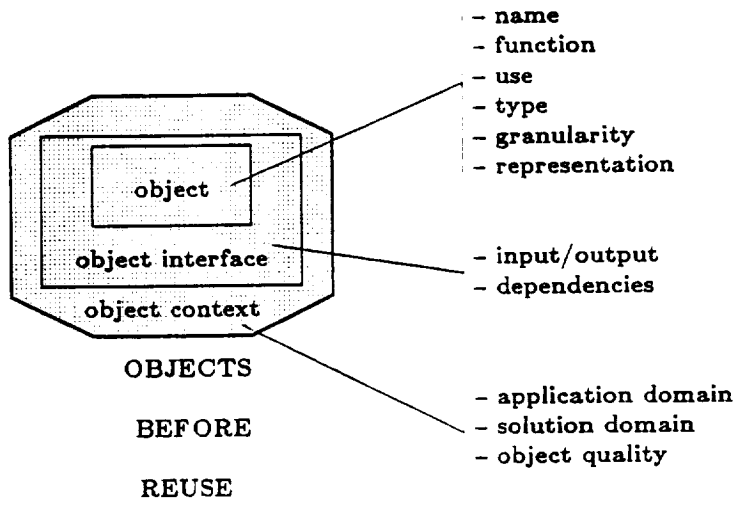


Figure 5a: Reuse Model (Objects–Before–Reuse / Refinement level 2)

A detailed definition of the above eleven dimensions – together with example categories – has already been presented in Section 3. In contrast to Section 3, we now have (i) a rationale for these dimensions (see Figure 5a) and (ii) understand that they cover only part (i.e., the objects–before–reuse) of the comprehensive reuse model depicted in Figure 4.

4.3.2. Objects–After–Reuse

In order to characterize objects–after–reuse, we have chosen the same eleven dimensions and supporting categories as for the objects–before–reuse. The resulting model refinement is depicted in Figure 5b:

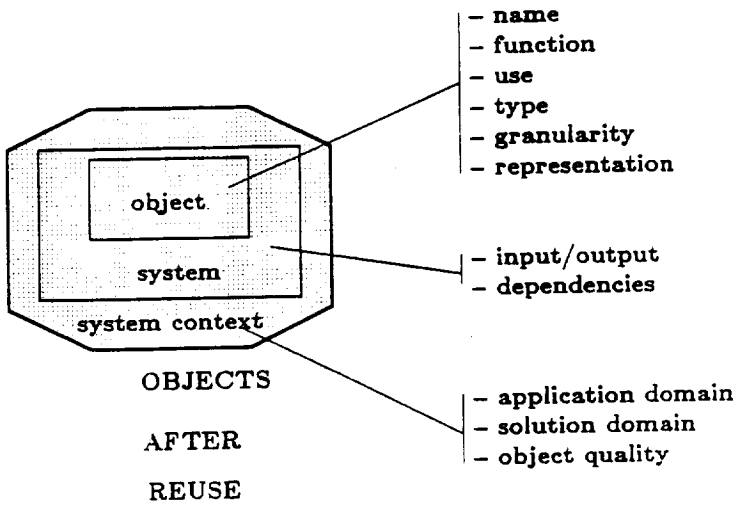


Figure 5b: Reuse Model (Objects–After–Reuse / Refinement level 2)

However, an object may change its characteristics during the actual process of reuse. Therefore, its characterizations before–reuse and after–reuse can be expected to be different. For example, an object–before–reuse may be a compiler (type) product (use), and may have been developed according to a waterfall life–cycle approach (solution domain). The object–after–reuse may be a compiler (type) process (use) integrated into a project based on iterative enhancement (solution domain).

This means that despite the similarity between the refined models of objects–before–reuse and objects–after–reuse, there exists a significant difference in emphasis: In the former case the emphasis is on the potentially reusable objects themselves; in the latter case, the emphasis is on the system in which these object(s) are (or are expected to be) reused. This explains the use of different dimension names: 'system' and 'system context' instead of 'object interface' and 'object context'.

The distance between the characteristics of an object–before–reuse and an object–after–reuse give an indication of the gap to be bridged in the event of reuse.

4.3.3. Reuse Process

The reuse process consists of several activities. In the remainder of this paper, we will use a model consisting of four basic activities: identification, evaluation, modification, and integration. In order to characterize each reuse activity we may be interested in its name (e.g., modify.p1), its function (e.g., modify an identified reuse candidate to entirely satisfy given object–after–reuse characteristics), its type (e.g., modification), and the mechanism used to perform its function (e.g., modification via parameterization). The interface of each activity may consist of such things as what the explicit input/output interfaces between the activity and the enabling software evolution environment are (e.g., in the case of modification: performed during the coding phase, assumes the existence of a specification), and what other assumptions regarding the evolution environment need to be satisfied (e.g., existence of certain configuration control policies). The activity context may include information about how experience is transferred from the object–before–reuse domain to the object–after–reuse domain (experience transfer), and the quality of each reuse activity (e.g., reliability, productivity).

This refinement of the reuse process is depicted in Figure 5c.

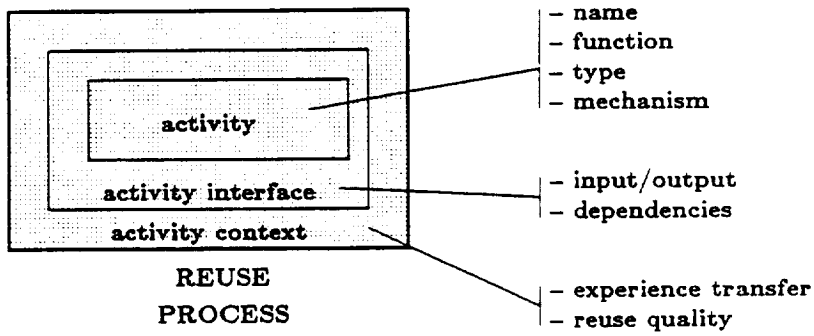


Figure 5c: Reuse Model (Reuse Process / Refinement level 2)

In more detail, the dimensions and example categories for characterizing the reuse process are:

- **REUSE PROCESS**: For each reuse activity characterize:

  + **Activity**:

    - **name**: What is the name of the activity? (e.g., identify.generics, evaluate.generics, modify.generics, integrate.generics)
    - **function**: What is the function performed by the activity? (e.g., select candidate objects $\{x_i\}$ which satisfy certain object categories of the object–after–reuse specification '$\bar{x}$'; evaluate the potential of the selected candidate objects of satisfying the given system and system context dimensions of the object–after–reuse specification '$\bar{x}$' and pick the most suited candidate 'x_k'; modify 'x_k' to entirely satisfy '$\bar{x}$'; integrate object 'x' into the current development project)
    - **type**: What is the type of the activity? (e.g., identification, evaluation, modification, integration)
    - **mechanism**: How is the activity performed? (in the case of identification: e.g., by name, by function, by type and function; in the case of evaluation: e.g., by subjective judgement, by evaluation of historical baseline measurement data; in the case of modification: e.g., verbatim, parameterized, template–based, unconstrained; in the case of integration: e.g., according to the system configuration plan, according to the project/process plan)

  + **Activity Interface**:

    - **input/output**: What are explicit input and output interfaces between the reuse activity and the enabling software evolution environment? (in the case of identification: e.g., specification for the needed object–after–reuse / set of candidate objects–before–reuse; in the case of modification: e.g., one selected object–before–reuse, specification for the needed object–after–reuse / object–after–reuse)
    - **dependencies**: What are other implicit assumptions and dependencies on data and information regarding the software evolution environment? (e.g., time at which reuse activity is performed – relative to the enabling development process: e.g., during design or coding stages; additional information needed to perform the reuse activity effectively: e.g., package specification to instantiate a generic package, knowledge of system configuration plan, configuration management procedures, or project plan)

  + **Activity Context**:

    - **experience transfer**: What are the support mechanisms for transferring experience across projects? (e.g., human, experience base, automated)
    - **reuse quality**: What is the quality of each reuse activity? (e.g., high reliability, high predictability of modification cost, correctness, average performance)

5. APPLYING MODEL–BASED REUSE CHARACTERIZATION SCHEMES

We demonstrate the applicability of our model–based reuse scheme by characterizing three hypothetical reuse scenarios related to product, process and knowledge reuse: Ada generics, design inspections, and cost models (Section 5.1). The characterization of the Ada generics scenario is furthermore used to demonstrate the benefits of model–based characterizations to describe/understand/motivate a given reuse scenario (Section 5.2), to evaluate the cost of reuse (Section 5.3), and to plan the population of a reuse repository (Section 5.4).

5.1. Example Reuse Characterizations

The characterization scheme of section 4 has been applied to the three examples of product, process and knowledge reuse introduced in section 2. The resulting characterizations are contained in tables 2, 3, and 4:

Dimensions	Reuse Examples		
	Ada generic	design inspection	cost model
name	buffer.ada	sel_inspection.waterfall	sel_cost_model.fortran
function	<element>_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada/ generic package	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Fortran) life–cycle model, functional de-composition design method	waterfall (Fortran) life–cycle model, standard set of methods	waterfall (Fortran) life–cycle model standard set of methods
object quality	high reliability (e.g., < 0.1 defects per KLoC for a given set of acceptance tests)	average defect detection rate (e.g., > 0.5 defects detected per staff_hour)	average predictability (e.g., < 5% pre-diction error)

Table 2: Characterization of Example Reuse Objects–Before–Reuse

Dimensions	Reuse Examples		
	Ada generics	design inspection	cost model
name	string_buffer.ada	sel_inspection.cleanroom	sel_cost_model.ada
function	string_buffer	certify appropriateness of design documents	predict project cost
use	product	process	knowledge
type	code document,	inspection method	cost model
granularity	package	design stage	entire life cycle
representation	Ada	informal set of guidelines	formal mathematical model
input/output	formal and actual instantiation params	specification and design document needed, defect data produced	estimated product size in KLOC, complexity rating, methodology level, cost in staff_hours
dependencies	assumes Ada knowledge	assumes a readable design, qualified reader	assumes a relatively homogeneous class of problems and environments
application domain	ground support sw for satellites	ground support sw for satellites	ground support sw for satellites
solution domain	waterfall (Ada) life–cycle model, object oriented design method	Cleanroom (Fortran) development model, stepwise refinement oriented design, statistical testing	waterfall (Ada) life–cycle model, revised set of methods
object quality	high reliability (e.g., < 0.1 defects per KLoC for a given set of acceptance tests), high performance (e.g., max. response times for a set of tests)	high defect detection rate (e.g., > 1.0 defects detected per staff_hour) wrt. interface faults	high predictability (e.g., $< 2\%$ prediction error)

Table 3: Characterization of Example Reuse Objects–After–Reuse

Dimensions	Reuse Examples		
	Ada generics	design inspection	cost model
name	modify.generics	modify.inspections	modify.cost_models
function	modify to satisfy target specification	modify to satisfy target specification	modify to satisfy target specification
type	modification	modification	modification
mechanism	parameterized (generic mechanism)	unconstrained	template-based
input/output	buffer.ada, reuse specification/ string_buffer.ada	sel_inspection.waterfall, reuse specification/ sel_inspection.cleanroom	sel_cost_model.fortran, reuse specification/ sel_cost_model.ada
dependencies	performed during coding stage, package specification needed, knowledge of system configuration plan	performed during planning stage, knowledge of project plan	performed during planning stage, knowledge of historical project profiles
experience transfer	experience base	human and experience base	human and experience base
reuse quality	correctness	correctness	correctness

Table 4: Characterization of Example Reuse Processes

5.2. Describing/Understanding/Motivating Reuse Scenarios

We will demonstrate the benefits of our reuse characterization scheme to describe, understand, and motivate the reuse of Ada generics as characterized in section 5.1.

We assume that in some project the need has arisen to have an Ada package implementing a 'string_buffer' with high 'reliability and performance' characteristics. This need may have been established during the project planning phase based on domain analysis, or during the design or coding stages. This package will be integrated into a software system designed according to

object–oriented principles. The complete reuse specification is contained in Table 3.

First, we identify candidate objects based on some subset of the object related characteristics stated in Table 3: string_buffer.ada, string_buffer, product, code document, package, Ada. The more characteristics we use for identification, the smaller the resulting set of candidate objects will be. For example, if we include the name itself, we will either find exactly one object or none. Identification may take place during any project stage. We will assume that the set of successfully identified reuse candidates contains 'buffer.ada', the object characterized in Table 2.

Now we need to evaluate whether and to what degree 'buffer.ada' (as well as any other identified candidate) needs to be modified and estimate the cost of such modification compared to the cost required for creating the desired object 'string_buffer' from scratch. Three characteristics of the chosen reuse candidate deviate from the expected ones: it is more general than needed (see function dimension), it has been developed according to a different design approach (see solution domain dimension), and it does not contain any information about its performance behavior (see object quality dimension). The functional discrepancy requires instantiating object 'buffer.ada' for data type 'string'. The cost of this modification is extremely low due to the fact that the generic instantiation mechanism in Ada can be used for modification (see Table 4). The remaining two discrepancies cannot be evaluated based on the information available through the characterizations in section 5.1. On the one hand, ignoring the solution domain discrepancy may result in problems during the integration phase. On the other hand, it may be hard to predict the cost of transforming 'buffer.ada' to adhere to object–oriented principles. Without additional information about either the integration of non–object–oriented packages or the cost of modification, we only have the choice between two risks. Predicting the cost of changes necessary to satisfy the stated object performance requirements is impossible because we have no information about the candidate's performance behavior. It is noteworthy that very often practical reuse seems to fail because of lack of appropriate information to evaluate the reuse implications a–priori, rather than because of technical infeasibility.

In case the object characterized in Table 2 has been modified successfully to satisfy the specification in Table 3, we need to integrate it into the ongoing development process. This task needs to be performed consistently with the system configuration plan and the process plan used in this project.

The characterization of both objects (before/after–reuse) and the reuse process allow us to understand some of the implications and risks associated with discrepancies between identified reuse candidates and target reuse specification. Problems arise when we have either insufficient information about the existence of a discrepancy (e.g., object performance quality in our example), or no understanding of the implications of an identified discrepancy (e.g., solution domain in our example). In order to avoid the first type of problem, one may either constrain the identification process further by including characteristics other than just the object related ones, or not have any objects without 'performance' data in the reuse repository. If we had included 'desired solution domain' and 'object performance' as additional criteria in our identification process, we may not have selected object 'buffer.ada' at all. If every object in our repository would have performance data attached to it, we at least would be able to establish the fact that there exists a discrepancy. In order to avoid the second type of problem, we need have some (semi–) automated modification mechanism, or at least historical data about the cost involved in similar past situations. It is clear that in our example any functional discrepancy within the scope of the instantiation parameters is easy to bridge due to the availability of a completely automated modification mechanism (i.e., generic instantiation in Ada). Any functional discrepancy that cannot be bridged through this mechanisms poses a larger and possibly unpredictable risk. Whether it is more costly to re–design 'buffer.ada' in order to adhere to object oriented design principles or to re–develop it from scratch is not obvious without past experience.

Based on the preceding discussion, the motivational benefits are that we have a sound rationale for suggesting the use of certain reuse mechanisms (e.g., automated in the case of Ada packages to reduce the modification cost), criteria for populating a reuse repository (e.g., do

exclude objects without performance data to avoid the unnecessary expansion of the search space), criteria for identifying reuse candidates effectively according to some reuse specification (e.g., do include solution domain to avoid the identification of candidates with unpredictable modification cost), or certain types of reuse specifications (e.g., require that each reuse request is specified in terms of all object dimensions, except probably name, and all system context dimensions).

5.3. Evaluating the Cost of Reuse

We will demonstrate the benefits of our reuse characterization scheme to evaluate the cost of reusing Ada generics as characterized in section 5.1.

The general evaluation goals are (i) characterize the degree of discrepancies between a given reuse specification (see Table 3) and a given reuse candidate (Table 2), and (ii) what is the cost of bridging the gap between before–reuse and after–reuse characteristics. The first type of evaluation goal can be achieved by capturing detailed information with respect to the object–before–reuse and object–after–reuse dimensions. The second goal requires the inclusion of data characterizing the reuse process itself and past experience about similar reuse activities.

We use the goal/question/metric paradigm to perform the above kind of goal–oriented evaluation [6, 8, 10]. It provides templates for guiding the selection of appropriate metrics based on a precise definition of the evaluation goal. Guidance exists at the level of identifying certain types of metrics (e.g., to quantify the object of interest, to quantify the perspective of interest, to quantify the quality aspect of interest). Using the goal/question/metric paradigm in conjunction with reuse characterizations like the ones depicted in Tables 2, 3, and 4, provides very detailed guidance as to what exact metrics need to be used. For example, evaluation of the Ada generic example suggests metrics to characterize discrepancies between the desired object–after–reuse and all before–reuse candidates in terms of (i) function, use, type, granularity, and representation on a nominal scale defined by the respective categories, (ii) input/output interface on an ordinal scale

'number of instantiation params', (iii) application and solution domains on nominal scales, and (iv) qualities such as performance based on benchmark tests.

5.4. Planning the Population of Reuse Repositories

We will demonstrate the benefits of our reuse characterization scheme to populate a reuse repository with generic Ada packages as characterized in section 5.1.

Reuse is economical from a project perspective if the effort required to bridge the gap between an object–before–reuse (available in some experience base) and the desired object–after–reuse is less than the effort required to create the object–after–reuse from scratch. Reuse is economical from an organization's perspective if the effort required for creating the reuse repository is less than the sum of all project–specific savings based on reuse.

Based on the above statement, populating a reuse repository constitutes an optimization problem for the organization. For example, high effort for populating a reuse repository may be justified if (i) small savings in many projects are expected, or (ii) large savings in a small number of projects are expected. For example, object 'buffer.ada' could have been transformed to adhere to object oriented principles prior to introducing it into the repository. This would have excluded the project specific risk and cost.

The cost of reusing an object–before–reuse from an experience base depends on its distance to the desired object–after–reuse and the mechanisms employed to bridge that distance. The cost of populating a reuse repository depends on how much effort is required to transform existing objects into objects–before–reuse. Both efforts together are aimed at bridging the gap between the project in which some objects were produced and the projects in which they are intended to be reused. The inclusion of a generic package 'buffer.ada' into the repository instead of specific instances 'integer_buffer.ada' and 'real–buffer.ada' requires some up–front transformation (i.e., abstraction). The advantage of creating an object 'buffer.ada' is that it reduces the project–specific cost of creating object 'string_buffer.ada' (or any other buffer for that matter) and

quantifies the cost of modification.

Finding the appropriate characteristics for objects–before–reuse to minimize project–specific reuse costs requires a good understanding of future reuse needs (objects–after–reuse) and the reuse processes to be employed (reuse process). The more one knows about future reuse needs within an organization, the better job one can do of populating a repository. For example, the object–before–reuse characteristics of Ada generics in Table 2 were derived from the corresponding object–after–reuse and reuse process characteristics in Tables 3 and 4. It would have made no sense to include Ada generics into the experience base that (i) are not based on the same instantiation parameters as all anticipated objects–after–reuse because modification is assumed via parameterized instantiation, (ii) do not exhibit high reliability and performance, and (iii) have not the same solution domain except we understand the implication of different solution domains. Without any knowledge of the object–after–reuse and reuse process characteristics, the task of populating a reuse repository is about as meaningful as investing in the mass–production of concrete components in the area of civil engineering without knowing whether we want to build bridges, town houses or high–rise buildings.

6. A REUSE–ORIENTED SOFTWARE ENVIRONMENT MODEL

Effective reuse according to the reuse–oriented software development model depicted in Figure 2 of Section 2 needs to take place in an environment that supports continuous improvement, i.e., recording of experience across all projects, appropriate packaging and storing of recorded experience, and reusing existing experience whenever feasible. Figure 6 depicts such an environment model.

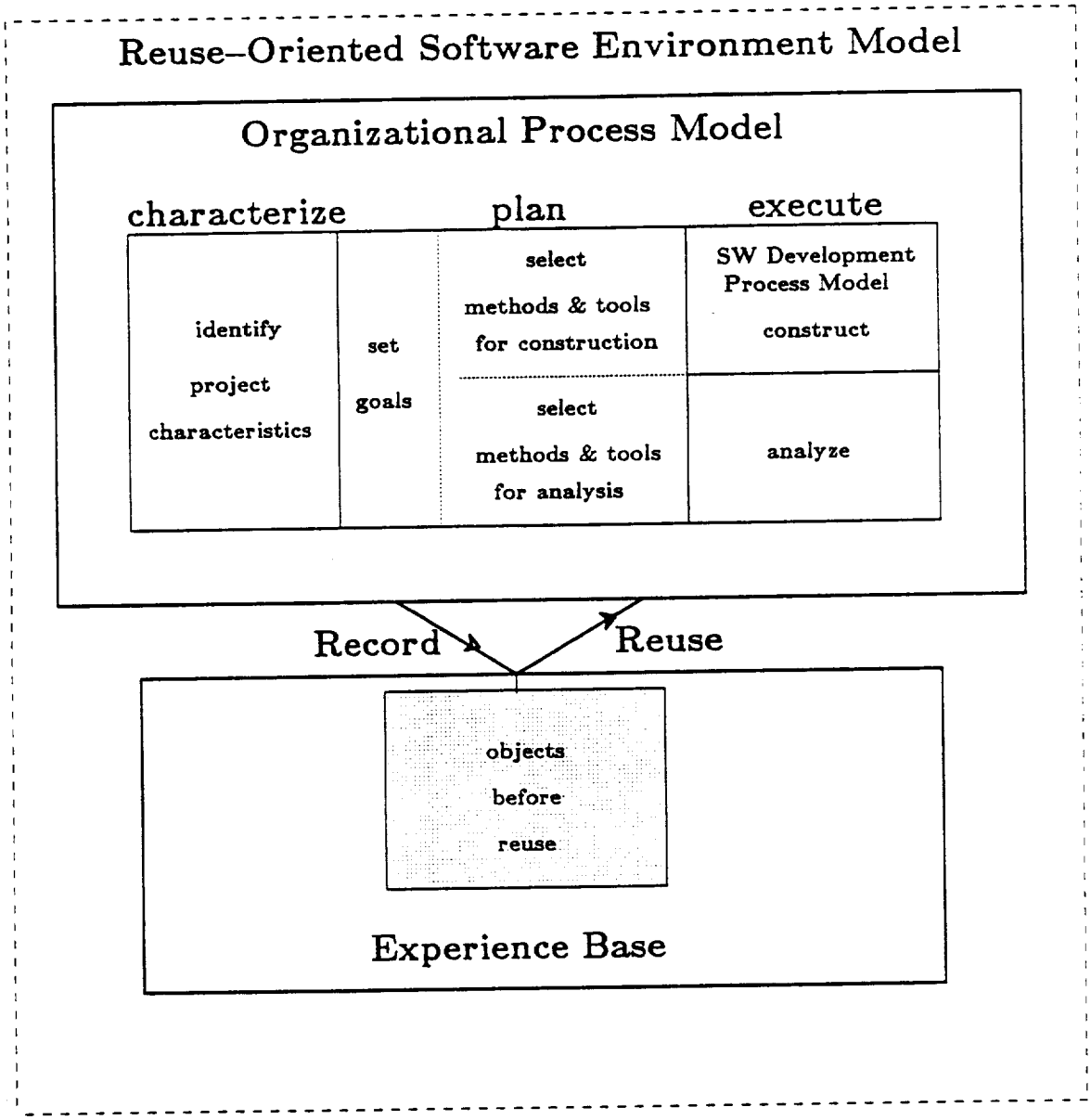


Figure 6: Reuse–Oriented Software Environment Model

Each project is performed according to an organization process model based on the improvement paradigm [2, 5]:

1. **Characterize**: Identify characteristics of the current project environment so that the

appropriate past experience can be made available to the current project.

2. **Plan**: (A) Set up the goals for the project and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances (e.g., based upon the goal/question/metric paradigm [6]).

   (B) Choose the appropriate software development process model for this project with the supporting methods and tools – both for construction and analysis.

3. **Execute**: (A) Construct the products according to the chosen development process model, methods and tools.

   (B) Collect the prescribed data, validate and analyze it to provide feedback in real–time for corrective action on the current project.

4. **Feedback**: (A) Analyze the data to evaluate the current practices, determine problems, record findings and make recommendations for improvement for future projects.

   (B) Package the experiences in the form of updated and refined models and other forms of structured knowledge gained from this and previous projects, and save it in an experience base so it can be available to future projects.

The experience base is not a passive entity that simply stores experience. It is an active organizational entity in the context of the reuse–oriented environment model which – in addition to storing experience in a variety of repositories – involves the constant modification of experience to increase its reuse potential. It plays the role of an organizational "server" aimed at satisfying project–specific requests effectively. The constant collection of measurement data regarding objects–after–reuse and the reuse processes themselves enables the judgements needed to populate the experience base effectively and to select the best suited objects–before–reuse to satisfy project–specific reuse needs based upon experiences. The organizational process model based on the improvement paradigm supports the integration of measurement–based analysis and construction.

For more detail about the reuse–oriented environment model, the reader is referred to [7].

7. CONCLUSIONS

The model–based reuse characterization scheme introduced in this paper has advantages over existing schemes in that it (a) allows us to capture the reuse of any type of experience, (b) distinguishes between objects–before–reuse, objects–after–reuse, and the reuse process itself, and (c) provides a rationale for the chosen characterizing dimensions. In the past most the scope of reuse schemes was limited to objects–before–reuse.

We have demonstrated the advantages of such a model–based scheme by applying it to the characterization of example reuse scenarios. Especially its usefulness for evaluating the cost of reuse and planning the population of reuse repositories were stressed.

Finally, we gave a model how we believe reuse should be integrated into an environment aimed at continuous improvement based on learning and reuse. A specific instantiation of such an environment, the 'code factory', is currently being developed at the University of Maryland [12]. In order to make reuse a reality, more research is required towards understanding and conceptualizing activities and aspects related to reuse, learning and the experience base.

8. ACKNOWLEDGEMENTS

We thank all our colleagues and graduate students who contributed to this paper, especially all members of the TAME and CARE project.

9. REFERENCES

[1] V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying", in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, December 1985.

[2] V. R. Basili, "Quantitative Evaluation of Software Methodology", Dept. of Computer Science, University of Maryland, College Park, TR-1519, July 1985 [also in Proc. of the First Pan Pacific Computer Conference, Australia, September 1986].

[3] V. R. Basili, "Viewing Maintenance as Reuse-Oriented Software Development", IEEE Software Magazine, January 1990, pp. 19-25.

[4] V. R. Basili and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments", Proc. of the Ninth International Conference on Software Engineering, Monterey, CA, March 30 - April 2, 1987, pp. 345-357.

[5] V. R. Basili and H. D. Rombach, "TAME: Integrating Measurement into Software Environments", Technical Report TR-1764 (or TAME-TR-1-1987), Dept. of Computer Science, University of Maryland, College Park, MD 20742, June 1987.

[6] V. R. Basili and H. D. Rombach "The TAME Project: Towards Improvement-Oriented Software Environments", IEEE Transactions on Software Engineering, vol. SE-14, no. 6, June 1988, pp. 758-773.

[7] V. R. Basili and H. D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment", Technical Report (UMIACS-TR-88-92, CS-TR-2158), Department of Computer Science, University of Maryland, College Park, MD 20742, December 1988.

[8] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, vol.SE-13, no.12, December 1987, pp.1278-1296.

[9] V. R. Basili and M. Shaw, "Scope of Software Reuse", White paper, working group on 'Scope of Software Reuse', Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July 1987 (in preparation).

[10] V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", IEEE Transactions on Software Engineering, vol.SE-10, no.3, November 1984, pp.728-738.

[11] Ted Biggerstaff, "Reusability Framework, Assessment, and Directions", IEEE Software Magazine, March 1987, pp.41-49.

[12] G. Caldiera and V. R. Basili, "Reengineering Existing Software for Reusability", Technical Report (UMIACS-TR-90-30, CS-TR-2419), Department of Computer Science, University of Maryland, College Park, MD 20742, February 1990.

[13] P. Freeman, "Reusable Software Engineering: Concepts and Research Directions", Proc. of the Workshop on Reusability, September 1983, pp. 63-76.

[14] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", IEEE Software, vol.4, no.1, January 1987, pp. 6-16.

[15] IEEE Software, special issue on 'Reusing Software', vol.4, no.1, January 1987.

[16] IEEE Software, special issue on 'Tools: Making Reuse a Reality', vol.4, no.7, July 1987.

[17] G. A. Jones and R. Prieto-Diaz, "Building and Managing Software Libraries", Proc. Compsac'88, Chicago, October 5-7, 1988, pp. 228-236.

[18] A. Kouchakdjian, V. R. Basili, and S. Green, "The Evolution of the Cleanroom Process in the Software Engineering Laboratory", IEEE Software Magazine (to appear 1990).

[19] F. E. McGarry, "Recent SEL Studies", in Proc. Tenth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt, MD, Dec. 1985.

[20] R. W. Selby, Jr., V. R. Basili, and T. Baker, "CLEANROOM Software Development: An Empirical Evaluation", IEEE Transactions on Software Engineering, vol. SE-13, no. 9, September 1987, pp.1027-1037.

[21] Mary Shaw, "Purposes and Varieties of Software Reuse", Proceedings of the Tenth Minnowbrook Workshop on Software Reuse, Blue Mountain Lake, New York, July, 1987.

[22] T. A. Standish, "An Essay on Software Reuse", IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984, pp.494-497.

[23] W. Tracz, "Tutorial on 'Software Reuse: Emerging Technology'", IEEE Catalog Number EHO278-2, 1988.

[24] J. Valett, B. Decker, J. Buell, "The Software Management Environment", in Proc. Thirteenth Annual Software Engineering Workshop, NASA/Goddard Space Flight Center, Greenbelt, MD, November 30, 1988.

[25] M. V. Zelkowitz (ed.), "Proceedings of the University of Maryland Workshop on 'Requirements for a Software Engineering Environment', Greenbelt, MD, May 1986", Technical Report TR-1733, Dept. of Computer Science, University of Maryland, College Park, MD 20742, December 1986 [to be published as a book, Ablex Publ., 1988].

IEEE Software

Viewing Maintenance as Reuse-Oriented Software Development

Victor R. Basili, *University of Maryland at College Park*

Treating maintenance as a reuse-oriented development process provides a choice of maintenance approaches and improves the overall evolution process.

If you believe that software should be developed with the goal of maximizing the reuse of experience in the form of knowledge, processes, products, and tools, the maintenance process is logically and ideally suited to a reuse-oriented development process. There are many reuse models, but the key issue is which process model is best suited to the maintenance problem at hand.

In this article, I present a high-level organizational paradigm for development and maintenance in which an organization can learn from development and maintenance tasks and then apply that paradigm to several maintenance process models. Associated with the paradigm is a mechanism for setting measurable goals so you can evaluate the process and the product and learn from experience.

An earlier version of this article was given as the keynote presentation at the Conference on Software Maintenance in October 1988.

Maintenance models

Most software systems are complex, and modification requires a deep understanding of the functional and nonfunctional requirements, the mapping of functions to system components, and the interaction of components. Without good documentation of the requirements, design, and code with respect to function, traceability, and structure, maintenance becomes a difficult, expensive, and error-prone task. As early as 1976, Les Belady and Manny Lehman reported on the problems with the evolution of IBM OS/360.[1] The literature is filled with similar examples.

Maintenance comprises several types of activities: correcting faults in the system, adapting the system to a changing operating environment (such as new terminals and operating-system modifications), and adapting the system to changes in the original requirements. The new system is

0740-7459/90/0100/0019/$01.00 © 1990 IEEE

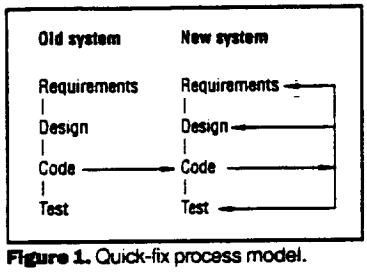


Figure 1. Quick-fix process model.

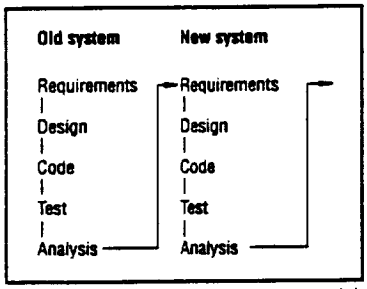


Figure 2. Interative-enhancement model.

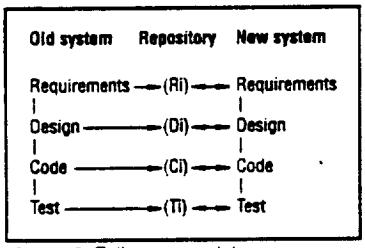


Figure 3. Full-reuse model.

like the old system, yet it is also different in a specific set of characteristics.

You can view the new version of the system as a modification of the old system or as a new system that reuses many of the old system's components. Although these two views have many aspects in common, they are very different in how you organize the maintenance process, the effects on future products, and the support environments required.

Consider the following three maintenance process models:

- the quick-fix model,
- the iterative-enhancement model, and
- the full-reuse model.

All three models reuse the old system and so are reuse-oriented. Which model you choose for a particular modification is determined by a combination of management and technical decisions that depend on the characteristics of the modification, the future evolution of the product line, and the support environment available.

Each model assumes that there is a complete and consistent set of documents describing the existing system, from requirements through code. Although this may be a naive assumption in practice, a side effect of this article's presentation should be to motivate organizations to gain the benefits of having such documentation.

Quick-fix model. The quick-fix model represents an abstraction of the typical approach to software maintenance. In the quick-fix model, you take the existing system, usually just the source code, and make the necessary changes to the code and the accompanying documentation and recompile the system as a new version. This may be as straightforward as a change to some internal component, like an error correction involving a single component or a structural change or even some functional enhancement.

Figure 1 demonstrates the flow of change from the old system's source code to the new version's source code. It is assumed — but not always true — that the accompanying documentation is also updated. You can view this model as reuse-oriented, since you can view the model as creating a new system by reusing the old system or as simply modifying the old system. However, viewing it in a reuse orientation gives you more freedom in the scope of change than viewing it in a modification or patch orientation.

Iterative-enhancement model. Iterative enhancement[2] is an evolutionary model proposed for development in environments where the complete set of requirements for a system was not fully understood or where the developer did not know how to build the full system. Although iterative enhancement was proposed as a development model, it is well suited to maintenance. It assumes a complete and consistent set of documents describing the system. The iterative-enhancement model

- starts with the existing system's requirements, design, code, test, and analysis documents;
- modifies the set of documents, starting with the highest-level document affected by the changes, propagating the changes down through the full set of documents; and
- at each step of the evolutionary process, lets you redesign the system, based on analysis of the existing system.

The process assumes that the maintenance organization can analyze the existing product, characterize the proposed set of modifications, and redesign the current version where necessary for the new capabilities.

Figure 2 demonstrates the flow of change from the highest-level document affected by the change through the lowest-level document. This model supports the reuse orientation more explicitly. An environment that supports the iterative-enhancement model clearly supports the quick-fix model.

Full-reuse model. While iterative enhancement starts with evaluating the existing system for redesign and modification, a full-reuse process model starts with the requirements analysis and design of the new system and reuses the appropriate requirements, design, and code from any earlier versions of the old system. It assumes a repository of documents and components defining earlier versions of the current system and similar systems. The full-reuse model

- starts with the requirements for the new system, reusing as much of the old system as feasible, and
- builds a new system using documents and components from the old system and from other systems available in your repository; you develop new documents and components where necessary.

Here, reuse is explicit, packaging of existing components is necessary, and analysis is required to select the appropriate components.

Figure 3 demonstrates the flow of various documents into the various document repositories (which are all part of the larger repository) and how those repositories are accessed for documents for the new development. There is an assumption that the items in the repository are classified according to a variety of characteristics, some of which I describe later in the article.

This repository may contain more than just the documents from the earlier system — it may contain documents from earlier versions, documents from other products in the product line, and some

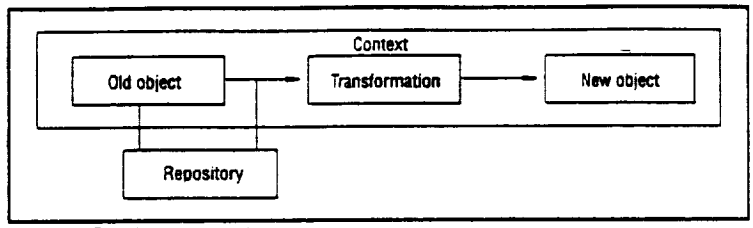


Figure 4. Simple reuse model.

generic reusable forms of documents. An environment that supports the full-reuse model clearly supports the other two models.

Model differences. The difference between the last two approaches is more one of perspective than style. The full-reuse model frees you to design the new system's solution from the set of solutions of similar systems. The iterative-enhancement model takes the last version of the current system and enhances it.

Both approaches encourage redesign, but the full-reuse model offers a broader set of items for reuse and can lead to the development of more reusable components for future systems. By contrast, the iterative-enhancement model encourages you to tailor existing systems to get the extensions for the new system.

Reuse framework

The existence of multiple maintenance models raises several questions. Which is the most appropriate model for a particular environment? a particular system? a particular set of changes? the task at hand? How do you improve each step in the process model you have chosen? How do you minimize overall cost and maximize overall quality?

To answer these questions, you need a model of the object of reuse, a model of the process that adapts that object to its target application, and a model of the reused object within its target application. Figure 4 shows a simple model for reuse. In this model, an object is any software process or product and a transformation is the set of activities performed when reusing that object.

The model steps are

- identifying the candidate reusable pieces of the old object,
- understanding them,
- modifying them to your needs, and
- integrating them into the process.

To flesh out the model, you need a framework for categorizing objects, transformations, and their context. The framework should cover various categories. For example, is the object of reuse a process or a product? In each category, there are various classification schemes for the product (such as requirements document, code module, and test plan) and for the process (such as cost estimation, risk analysis, and design).

Framework dimensions. There are a variety of approaches to classifying reusable objects, most notably the faceted scheme offered by Ruben Prieto-Díaz and Peter Freeman.[3] I offer here a scheme that categorizes three aspects of reuse: the reusable object, the reusable object's context, and the process of transforming that object. This scheme owes much to ideas presented at the 1987 Minnowbrook Workshop on Software Reuse.

I offer here a scheme that categorizes three aspects of reuse: the reusable object, the reusable object's context, and the process of transforming that object.

Object dimensions include:

- Reuse-object type. What is a characterization of the candidate reuse object? Sample process classifications include a design method and a test technique; product classifications include source code and requirements documents.
- Self-containedness. How independent and understandable is the candidate object? Sample classifications include syntactic independence (such as a data-coupling measure) and specification precision (such as functional notation and English).
- Reuse-object quality. How good is the candidate reuse object? Sample classifications include maturity (such as the number of systems using it), complexity (such as cyclomatic complexity), and reliability (such as the number of failures during previous use).

Context dimensions include:

- Requirements domain. How similar are the requirements domains of the candidate reuse object and the current project? Sample classifications are application (such as ground-support software for satellites) and distance (such as same application or similar algorithms but different problem focus).
- Solution domain. How similar are the evolution processes that resulted in the candidate reuse objects and the ones used in the current project? Sample classifications are process model (such as the waterfall model), design method (such as function decomposition), and language (such as Fortran).
- Knowledge-transfer mechanism. How is information about the candidate reuse objects and their context passed to current and future projects? People, such as a subset of the development team, provide a common knowledge-transfer mechanism.

Transformation dimensions include:

- Transformation type. How do you characterize transformation activities? Sample classifications include percent of change required, direction of change (such as general to domain-specific or project-specific to domain-specific), modification mechanism (such as verbatim, parameterized, template-based, or unconstrained), and identification mechanism (such as by name or by functional requirements).
- Activity integration. How do you integrate the transformation activities into the new system development? One sample classification is the phase where the activity is performed in the new development (for example, planning, requirements development, and design).
- Transformed quality. What is the contribution of the reuse object to the new system compared to the objectives set for it? Sample classifications are reliability (such as no failures associated with that component) and performance (such as satisfying a timing requirement).

Comparing the models. When applying the reuse framework to maintenance, the set of reuse objects is a set of product documents. You compare the models to see which is appropriate for the current set of changes according to the framework's three dimensions.

First consider the reuse-object dimension:

The objects of the quick-fix and iterative-enhancement models are the set of documents representing the old system. The object of the full-reuse model is any appropriate document in the repository.

For self-containedness, all the models depend on the unit of change. The quick-fix model depends on how much evolution has taken place, since the system may have lost structure over time as objects were added, modified, and deleted. In iterative enhancement, the evolved system's structure and understandability should improve with respect to the application and the classes of changes made so far. In the full-reuse model, the evolved system's structure, understandability, and generality should improve; the degree of improvement will depend on the quality and maturity of the repository.

For reuse-object quality, the quick-fix model offers little knowledge about the old object's quality. In iterative enhancement, the analysis phase provides a fair assessment of the system's quality. In full reuse, you have an assessment of the reuse object's quality across several systems.

Now consider the context dimensions:

For the requirements domain, the quick-fix and iterative-enhancement models assume that you are reusing the same application — in fact, the same project. The full-reuse model allows manageable variation in the application domain, depending on what is available in the repository.

For the solution domain, the quick-fix model assumes the same solution structure exists during maintenance as during development. There is no change in the basic design or structure of the new system. In iterative enhancement, some modification to the solution structure is allowed because redesign is a part of the model. The full-reuse model allows major differences in the solution structure: You can completely redesign the system from a structure based on functional decomposition to one based on object-oriented design, for example.

For the knowledge-transfer mechanism, the quick-fix and iterative-enhancement models work best when the same people are developing and maintaining the system. The full-reuse model can compensate for having a different team, assuming that you have application specialists and a well-documented reuse-object repository.

The quick-fix model's weaknesses are that the modification is usually a patch that is not well-documented, partly destroying the system structure and hindering future evolution.

Last, consider the transformation dimension:

For the transformation type, the quick-fix model typically uses activities like source-code lookup, reading for understanding, unconstrained modification, and recompilation. Iterative enhancement typically begins with a search through the highest-level (most abstract) document affected by the modification, changing it and evolving the subsequent documents to be consistent, using several modification mechanisms. The full-reuse model uses a library search and several modification mechanisms; those selected depend on the type of change. In full reuse, modification is done off-line.

For activity integration, all activities are performed at same time in the quick-fix model. Iterative enhancement associates the activities with all the normal development phases. In full reuse, you identify the candidate reusable pieces during project planning and perform the other activities during development.

For transformed quality, the quick-fix model usually works best on small, well-contained modifications because their effects on the system can be understood and verified in context. Iterative enhancement is more appropriate for larger changes where the analysis phase can provide better assessment of the full effects of changes. Full reuse is appropriate for large changes and major redesigns. Here, analysis and performance history of the reuse objects support quality.

Applying the models. Given these differences, you can analyze the maintenance process models and recommend where they might be most applicable.

But first, consider the relationship between the development and maintenance process models: You can consider development to be a subset of maintenance. Maintenance environments differ from development environments in the constraints on the solution, customer demand, timeliness of response, and organization.

Most maintenance organizations are set up for the quick-fix model but not for the iterative-enhancement or full-reuse models, since they are responding to timeliness — a system failure needs to be fixed immediately or a customer demands a modification of the system's functionality. This is best used when there is little chance the system will be modified again.

Clearly, these are the quick-fix model's strengths. But its weaknesses are that the modification is usually a patch that is not well-documented, the structure of the system has been partly destroyed, making future evolution of the system difficult and error-ridden, and the model is not compatible with development processes.

The iterative-enhancement model allows redesign that lets the system structure evolve making future modifications easier. It focuses on making the system as good as possible. It is compatible with development process models. It is a good approach to use when the product will have a long life and evolve over time. In this case, if timeliness is also a constraint, you can use the quick-fix model for patches and the iterative-enhancement model for long-term change, replacing the patches. The drawbacks are that it is a more costly and possibly less timely approach (in the

short run) than the quick-fix model and provides little support for generic components or future, similar systems.

The full-reuse model gives the maintainer the greatest degree of freedom for change, focusing on long-range development for a set of products, which has the side effect of creating reusable components of all kinds for future developments. It is compatible with development process models and, in fact, is the way development models should evolve. It is best used when you have multiproduct environments or generic development where the product line has a long life. Its drawback is that it is more costly in the short run and is not appropriate for small modifications (although you can combine it with other models for such changes).

My assessment of when to apply these models is informal and intuitive, since it is a qualitative analysis. To do a quantitative analysis, you would need quantitative models of the reuse objects, transformations, and context. You would need a measurement framework to characterize (via classification), evaluate, predict, and motivate management and technical decisions. To do this, you would need to apply to the models a mechanism for generating and interpreting quantitative measurement, like the goal/question/metric paradigm.[4-6] (See the box on p. 24 for a description of this paradigm and its application to choosing the appropriate maintenance process model.)

Reuse enablers

There are many support mechanisms necessary to achieve maximum reuse that have not been sufficiently emphasized in the literature. In this article, I have presented several: a set of maintenance models, a mechanism for choosing the appropriate models based on the goals and characteristics of the problem at hand, and a measurement and evaluation mechanism. To support these activities, there is a need for an improvement paradigm that helps organizations evaluate, learn, and enhance their software processes and products, a reuse-oriented evolution environment that encourages and supports reuse, and automated support for both the paradigm and environment as well as for measurement and evaluation.

Improvement paradigm. The improvement paradigm[4] is a high-level organizational process model in which the organization learns how to improve its products and process. In this model, the organization should learn how to make better decisions on which process model to use for the maintenance of its future products based on past performance. The paradigm has three parts: planning, analysis, and learning and feedback.

In planning, there are three integrated activities that are iteratively applied:

In the improvement paradigm, organizations should learn how to make better decisions on which process model to use for the maintenance of its future products based on past performance.

- Characterize the current project environment to provide a quantitative analysis of the environment and a model of the project in the context of that environment. For maintenance, the characterization provides product-dimension data, change and defect data, cost data and customer-context data for earlier versions of the system, information about the classes of candidate components available in the repository for the new system, and any feedback from previous projects with experience with different models for the types of modifications required.
- Set up goals and refine them into quantifiable questions and metrics using the goal/question/metric paradigm to get performance that has improved compared to previous projects. This consists of a top-down analysis of goals that iteratively decomposes high-level goals into detailed subgoals. The iteration terminates with subgoals that you can measure directly.
- Choose and tailor the appropriate construction model for this project and the supporting methods and tools to satisfy the project goals. Understanding the environment quantitatively lets you choose the appropriate process model and fine-tune the methods and tools needed to be most effective. For example, knowing the effect of earlier applications of the maintenance models and methods in creating new projects from old systems lets you choose and fine-tune the appropriate process model and methods that have been most effective in creating new systems of the type required from older versions and component parts in the repository.

In analysis, you evaluate the current practices, determine problems, record the findings, and make recommendations for improvement. You must conduct data analysis during and after the project. The goal/question/metric paradigm lets you trace from goals to metrics and back, which lets you interpret the measurement in context to ensure a focused, simpler analysis. The goal-driven operational measures provide a framework for the kind of analysis you need.

In learning and feedback, you organize and encode the quantitative and qualitative experience gained from the current project into a corporate information base to help improve planning, development, and assessment for future projects. You can feed the results of the analysis and interpretation phase back to the organization to change how it does business based on explicitly determined successes and failures.

In this way, you can learn how to improve quality and productivity and how to improve goal definition and assessment. You can start the next project with the experience gained from this and previous projects. For example, understanding the problems associated with each new version of a system provides insights into the need for redesign and redevelopment.

Reuse-oriented environment. Reuse can be more effectively achieved in an environment that supports reuse. (See the article by Ted Biggerstaff and Charles Richter[7] for a set of reusability technologies and the article by myself and Dieter Rombach[8] for a set of environment

Goal/question/metric paradigm

The goal/question/metric paradigm represents a systematic approach for setting project goals (tailored to the needs of an organization) and defining them in an operational, tractable way. Goals are associated with a set of quantifiable questions and models that specify metrics and data for collection. The tractability of this software-engineering process supports the analysis of the collected data and computed metrics in the appropriate context of the questions, models and goals, feedback (by integrating constructive and analytic activities), and learning (by defining the appropriate synthesis procedure for lower level into higher level pieces of experience.)

The goals are defined in terms of purpose (why the project is being analyzed), perspective (the models of interest and the point of view of the analysis), and the environment (the context of the project). When measuring a product or process, you ask questions in three general categories:

- product or process definition,
- definition of the quality perspectives of interest, and
- feedback.

Product definition includes physical attributes of the product, cost, changes and defects, and the context in which the product will be used. Process definition includes a model of the process, an evaluation of conformance to the model, and an assessment of the project-specific documents and experience with the application.

Definition of the quality perspectives of interest includes the quality models used (such as reliability and user friendliness) and the interpretation of the data collected relative to the models.

Feedback involves the return of information for improving the product and process based on the quality perspective of interest.

The following is an informal application of the goal/question/metric paradigm to a particular maintenance problem. The answers to some of the questions are obvious. The answers to others assume a database of experience that management must estimate if it is not available.

Goals. The goal-definition phase has three parts:

- Purpose: Analyze the new product requirements to determine the appropriate evolution model.
- Perspective: Examine the cost of the current enhancement and future evolution of the system from the organization's point of view.
- Environment: Along with the standard environmental factors, like resource and problem factors, you would like to pay special attention to the context dimensions in the reuse framework.

In the requirements domain, you typically use product objects from the same application domain, although you can choose objects from other domains in the repository, if they are generally applicable.

The solution domain defines the process models, methods, and tools used in the development of the old product. If you plan to use the same processes for the evolving product, there is no problem with reuse. If future evolution dictates changes to the solution domain, the full-reuse model lets you make these changes, but at the cost of reusing less of the old product.

For knowledge-transfer mechanism, you must determine what form of documentation is needed to transfer the required application, process, and product knowledge to the maintainers. If the maintenance group is the same as the development group, the major transfer mechanism is the people.

Product definition. With the goal defined, you then define your product. In this example, there are several products: the new product to be built (the new version of the system), the old versions, and any other relevant objects in the repository that may be reused.

For the category of physical attributes, sample questions are: How many requirements are there for the new system? What is the mapping of the requirements to system components in the old system? How independent are the components to be modified in the old system? What is the complexity of the old system and its individual components? What candidate objects are available in the repository and what are their object, context, and transformation classifications? How many new requirements, categorized by class (such as size, type, and whether it is a new, modified, or deleted requirement) are there that are not in the old system? How many components, categorized by class (such as size and type of change) in the old system must be changed, added, and deleted?

For the category of changes and defects, sample questions are: How many errors, faults, and failures (categorized by class) are there associated with the requirements and components that need to be changed? What is the profile of past and future changes to the system, categorized by class (such as cost and number of times a component has been and must be changed)?

For the category of cost, sample questions are: What was the cost of the original system? What was the cost of each prior version? What is the cost of each prior requirement change by class? What is the estimated cost of modifying the old system to meet the new requirements? What is the estimated cost of building a new system, reusing the experience and parts of the old system and the repository?

For the category of customer context, sample questions are: What are the various customer classes and how are they using the system? What are the estimated future enhancements based on your analysis of customer profiles, past modifications, and the state of technology in the application domain?

Quality perspective of interest. With the product defined, you now define the perspectives for the qualities that you are trying to achieve.

You should make a model of the system's evolution, along with its associated costs. Based on the data from the evolution of this system and other systems, as well as on the characteristics of the set of new requirements, the model should let you estimate the cost and benefits associated with each of the three process models and let you choose the appropriate one. Parameters for the model will include such items as the projected system lifetime, the number of future related systems, and the projected cost of changes for various classes of requirements.

Feedback. With the quality perspectives defined, you can now get the information needed to improve the product or process. The feedback should provide with deeper insights into the model and our environment.

Sample questions include: Is the model appropriate? How can the model be improved? How can the classifications be improved?

Other goals. There are many relevant goals. Consider the following examples:

- Evaluate the modification activities in the reuse model to improve them. Examine the cost and correctness of the resulting objects from the customer's point of view.
- Evaluate the components of the existing system to determine whether to reuse them. Examine their independence and functional appropriateness from the viewpoint of reuse in future systems.
- Predict the ability of a set of code components to be integrated into the current system from the developer's point of view.
- Encourage the reuse of a set of repository components built for reuse. Examine the reward structure from the manager's and developer's points of view.

characteristics.) Software-engineering environments provide such things as a project databases and support the interaction of people with methods, tools, and project data. However, experience is not controlled by the project database nor owned by the organization — so reuse exists only implicitly.

For effective reuse, you need to be able to incorporate the reuse process model in the context of development. You need to combine the development and maintenance models to maximize the context dimensions. You need to integrate characterization, evaluation, prediction, and motivation into the process. You need to support learning and feedback to make reuse viable. I propose that the reuse model can exist in the context of the improvement paradigm, making it possible to support all these requirements.

Automated support. The improvement paradigm and the reuse-oriented process model require automated support for the database, encoded experience, and the repository of previous projects and reusable components. A special issue of *IEEE Software*[9] offered a set of automated and automatable technologies for reuse. You need to automate as much of the measurement process as possible and to provide a tool environment for managers and engineers to develop project-specific goals and generate operational definitions based on these goals that specify the metrics needed for evaluation. This evaluation and feedback cannot be done in real time without automated support.

Furthermore, automated support will help in the postmortem analysis. For example, a system like Tailoring a Measurement Environment,[5] whose goal is to instantiate and integrate the improvement and goal/question/metric paradigms and help tailor the development process, can help support the reuse-oriented process model because it contains mechanisms to support systematic learning and reuse.

Applying the TAME concept to maintenance provides a mechanism for choosing the appropriate maintenance process model for a particular project and provides data to help you learn how to do a better job of maintenance.

The approach you take to maintenance depends on the nature of the problem and the size and complexity of the modification. Viewing maintenance as a reuse-oriented process in the context of the improvement paradigm gives you a choice of maintenance models and a measurement framework. You can evaluate the strengths and weaknesses of the different maintenance approaches, learn how to refine the various process models, and create an experience base from which to support further management and technical decisions.

If you do not adapt the maintenance approach, you will find it difficult to know which process model to use for a particular project, whether you are evolving the system appropriately, and whether you are maximizing quality and minimizing cost over the system lifetime. ❖

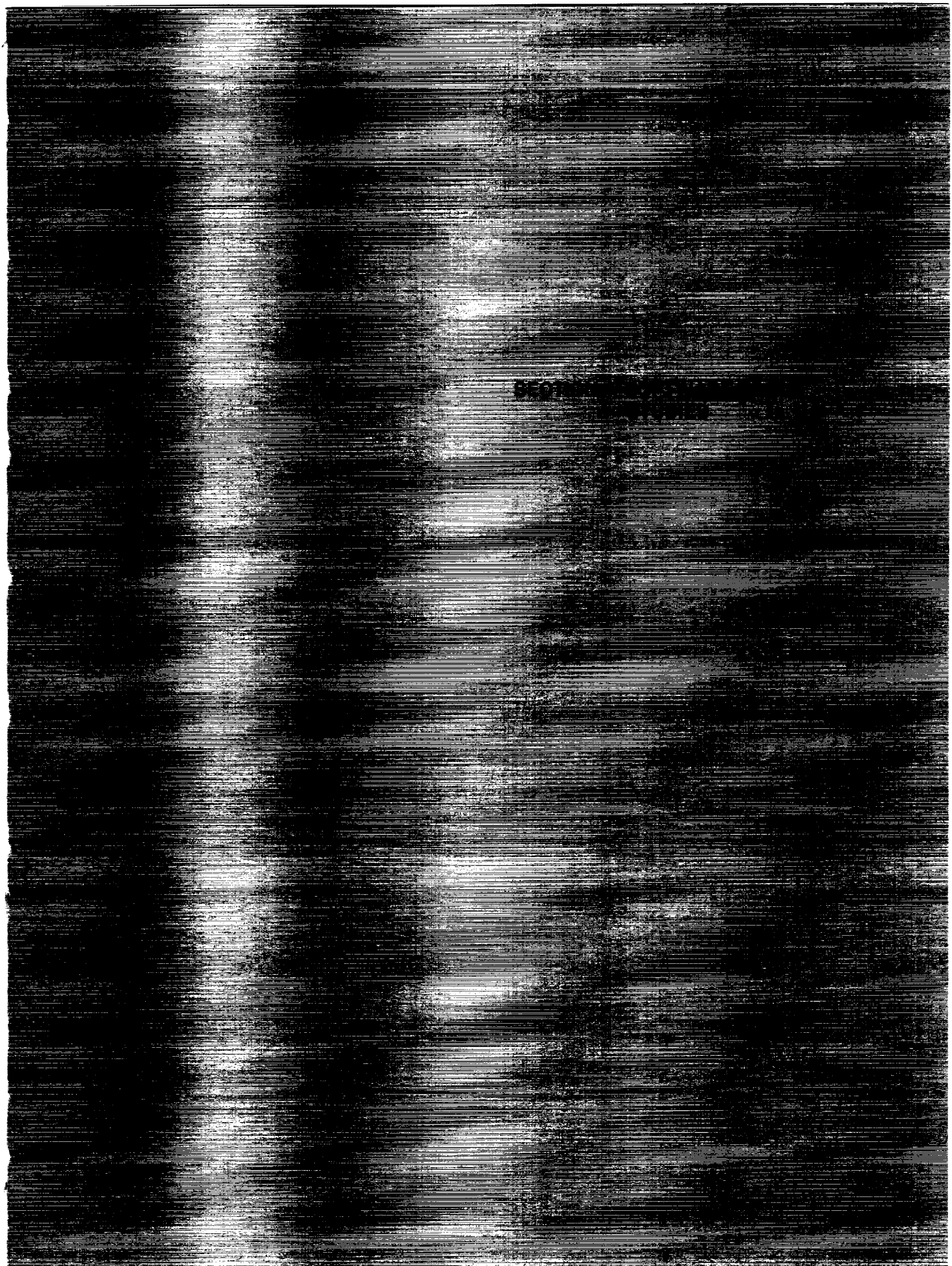
References

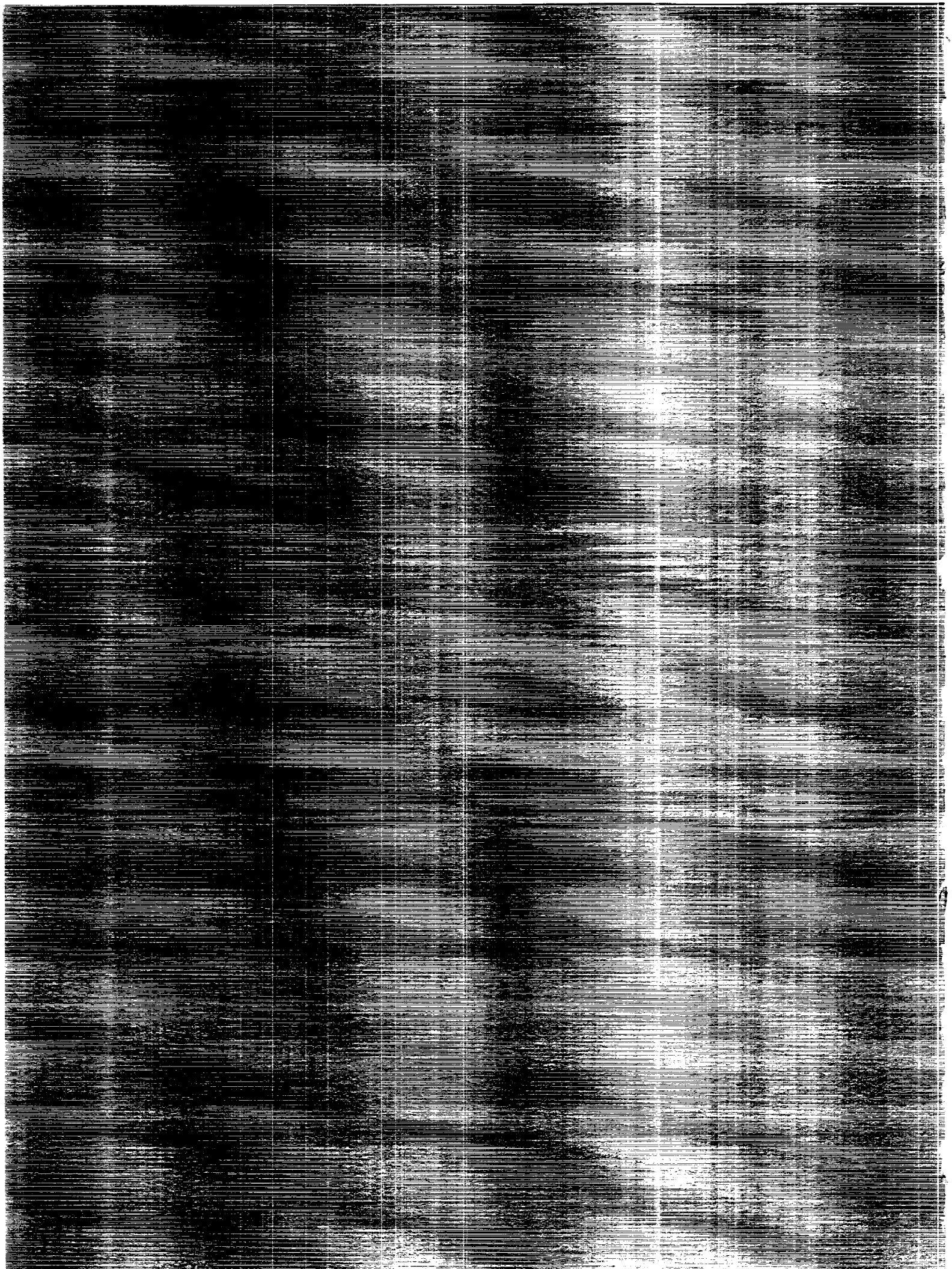
1. L. Belady and M. Lehman, "A Model of Large Program Development," *IBM Systems J.*, No.3, 1976, pp. 225-252.
2. V.R. Basili and A.J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. Software Eng.*, Dec. 1975, pp. 390-396.
3. R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, Jan. 1987, pp. 6-16.
4. V.R. Basili, "Quantitative Evaluation of Software Methodology," Tech. Report 1519, Computer Science Dept., Univ. of Maryland, College Park, Md., July 1985.
5. V.R. Basili and H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Trans. Software Eng.*, June 1988, pp. 758-773.
6. V.R. Basili and D.M. Weiss, "A Methodology for Collecting Valid Software-Engineering Data," *IEEE Trans. Software Eng.*, Nov. 1984, pp. 728-738.
7. V.R. Basili and H.D. Rombach, "Towards a Comprehensive Framework for Reuse: A Reuse-Enabling Software-Evolution Environment," Tech. Report UMIACS-TR-88-92, Computer Science Dept., Univ. of Maryland, College Park, Md., Dec. 1988.
8. T. Biggerstaff, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.
9. special issue on tools for reuse, *IEEE Software*, July 1987, pp. 6-72.

Victor R. Basili is a professor at the University of Maryland at College Park's Institute for Advanced Computer Studies and Computer Science Dept. His research interests include measuring and evaluating software development in industrial and government settings. He is a founder and principal of the Software Engineering Laboratory, a joint venture between the National Aeronautics and Space Administration, the University of Maryland, and Computer Science Corp.

Basili received a BS in mathematics from Fordham College, an MS in mathematics from Syracuse University, and a PhD in computer science from the University of Texas at Austin. He is a member of the IEEE Computer Society and is editor-in-chief of *IEEE Transactions on Software Engineering*.

Address questions about this article to Basili at Computer Science Dept., A.V. Williams Bldg., Rm. 4187, University of Maryland, College Park, MD 20742.





SECTION 4 - SOFTWARE TOOLS STUDIES

The technical paper included in this section was originally prepared as indicated below.

- "Evolution Towards Specifications Environment: Experiences With Syntax Editors," M. Zelkowitz, Information and Software Technology, April 1990

Evolution towards specifications environment: experiences with syntax editors

M V Zelkowitz

Language-based editors have been thoroughly studied over the last 10 years and have been found to be less effective than originally thought. The paper reviews some relevant aspects of such editors, describes experiences with one such editor (Support), and then describes two current projects that extend the syntax-editing paradigm to the specifications and design phases of the software life-cycle.

software design, environments, specification, syntax editors

SYNTAX EDITORS

Syntax-editing (or alternatively language-based editing) is a technique that had its beginning about 20 years ago (e.g., Emily[1]) and blossomed into a major research activity 10 years later (e.g., Mentor[2], CPS[3]). During the mid-1980s, major conferences were often dominated by syntax-editing techniques[4,5]. Many of these projects, however, have since been terminated or have taken a much lower profile. There are few widely used commercial products that use this technology. Why?

This paper briefly introduces the concept of syntax editing, describes one particular editor, and explains some experiences in using it. It is then shown how the syntax-editing paradigm is powerful but perhaps misapplied in the domain of source-program generation.

Just using a syntax editor for source-code production does not result in significantly higher productivity. By integrating specification generation with this source-code production, however, the author believes that increased productivity can be provided by making more of the life-cycle visible to the programmer. Two extensions to the current environment are described that apply syntax editing within a specifications environment to provide additional functionality over that of standard syntax editors.

With a conventional editor, the user may insert an arbitrary string of characters at any point in a file, and a later compilation phase will determine if there are any errors. With a syntax editor, however, only those choices permitted by the language grammar can be inserted, and the generation of source program and the processing of the program's syntax are intertwined operations. For example, for the statement nonterminal <stmt>, there are only a limited set of statement types that are permitted and only those legal strings can be entered by the user in response to that nonterminal on the screen.

The user interface is a major component of syntax editors. Depending on editor design, syntactic constructs can be specified via a mouse and pull-down menus, function keys on the keyboard, or special editing prompt commands. If the cursor is pointing to the <stmt> syntactic unit and the user specifies the **if** statement, then the text

```
if <expr> then
        <stmt>
else <stmt>
```

will replace <stmt> on the screen. Each nonterminal <...> is considered as a single editing character and syntactic constructs must be added or deleted in their entirety. In essence, the programmer is building the source-program parse tree in a top-down manner.

Pure syntax-editing is a simple macro-like substitution, and such macro substitutions exist in several conventional editors. For example, Emacs and Digital's LBE (Language Based Editor) both permit such substitutions anywhere in a program. Here, however, editors that go beyond simple substitution are being considered. Screen layout is often specified (e.g. unparsing the program tree to a 'pretty-printed' display), semantic information is usually checked (e.g., variable declarations, mixed types), and often the editor is part of an integrated package or environment of editor, interpreter, and debugging and testing tools.

Early on, many advantages of a syntax editor were stated:

- Source-program generation would be efficient as a single mouse or function key click would generate an entire construct.
- Productivity would increase as numerous errors such as missing **begin—end** pairs could not occur and mixed mode expressions would immediately be found by the editor at the point of insertion. Users could more easily use an unfamiliar language.
- Screen layout would be predefined, providing a uniform structure to all programs.

Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA.
Paper submitted: 27 August 1989.
Revised version received: 20 November 1989.

 0950-5849/90/030191-08

- The integrated package of tools enables testing and debugging to proceed more rapidly.

As shall be seen, the last of these reasons does indeed seem to be true; each of the others, however, seems to have a serious drawback as well as the supposed benefit.

As an example, the Support environment, designed by the author, is briefly described as an instance of the integrated syntax-editing genre[6]. It has many of the features implemented in such tools and is the basis for the extensions to specifications described later.

Support design

Support is an integrated environment built to process the CF-PASCAL subset of PASCAL and was used for three years (until the course contents changed) as the programming tool in the introductory programming course at the University of Maryland. It runs on both Berkeley Unix and IBM PC systems.

Design

Major features of Support include the following.

Text input Support uses both the command and function key mechanisms for input. If the cursor (represented by reverse video) covers the <stmt> unit, a menu at the bottom of the screen gives the available choices. For example, to insert an **if** statement, either a response of .2 or depressing function key 2 (on the PC keyboard) will insert the **if** construct.

Support also permits textual substitution for any syntactic unit. A user can type in an arbitrary line of characters, and an internal LALR parser builds the subtree for that construct. If the root of that subtree is permitted by the current cursor position, then it is attached to the program tree at that cursor position.

Using either input mechanisms, invalid syntax can never be entered. Using the menu for input permits only correct responses, and, for textual input, if the parser cannot resolve the typed-in text to a correct syntactic unit, an error is displayed and the program is not modified.

Windows Horizontal windows dividing the CRT screen are the major interface with the user. Each tool within Support controls its own window, and from two to four windows will typically be displayed at any one time.

Tools Various tools within Support aid in program design and development. The relationship among procedures in a program is handled by the Design window; an interpreter executes partially developed programs and includes features such as variable and statement tracing and breakdown monitoring. Statement trace and statement coverage windows are part of this structure. Data are displayed via the variable trace and the run-time display windows.

As an extension to the textual input mode, a small (i.e., size of screen) text editor called the Character Oriented EDitor (or COED) was implemented. Users insert or modify arbitrary sequences of characters in this window, have the text processed by the LALR parser mentioned above, and then have the text inserted into the program tree at the appropriate place in the program. The user can also pull an arbitrary section of program text into this window for modification. This also gave an easy cut-and-paste feature and the ability to move sections of code around in the program as a means to address some of the syntax-editing deficiencies that turned up.

Language and screen displays The grammar processed by Support (e.g., CF-PASCAL) is defined via an external data file that defines the syntax, some semantics, and screen layout. This feature turned out to be a major factor in allowing Support to be extended for other applications.

Table 1. Background of students

	Semester 1	Semester 2
First university computer course (%)	73	82
Took this course previously (%)	12	9
Took high-school course (%)	59	55
Never previously used computer (%)	26	24
Own microcomputer (%)	49	51

Experiences

Support was used from 1986 until 1989 in Computer Science I by approximately 200 to 300 students each semester. During the first two semesters data were collected from the 543 students that enrolled in the course. The background of the students is summarized in Table 1. As shown, about 75% had previous experience with programming and about half own their own computer.

Based on a 1 to 5 rating scale (1 = poor), students who owned their own computer (and presumably had more experience in programming) rated satisfaction with Support lower than those without their own computer (2.8 to 3.2). More revealing, students rated Support's text-editing capabilities much lower than those of an IBM mainframe also used during the semester (2.7 versus 3.7 for one semester, 3.3 versus 3.8 for the other). The author believes that users with experience with general text editors felt more restricted by the syntax-editing paradigm. On the other hand, novices with no previous experience felt aided by such restrictions.

Students using Support rates its debugging capabilities higher than those available on the IBM mainframe (3.8 versus 3.1 for one semester, 3.0 versus 2.9 for the other). The PC system was also rated as more available compared with the mainframe (3.9 versus 2.8 for one semester, 3.0 versus 2.9 for the other). Other results are presented elsewhere[7].

In summary, syntax editing seems to be viewed as a restriction on program development, but the integrated development and testing environment appears to be desired. A tool that simply develops source text does not seem to produce a large productivity increase. The results here are comparable to those found with other editing environments.

Retrospective

After several years of use and several redesigns and enhancements based on user needs and experiences, the four advantages claimed for such editors can be addressed more clearly. As shall be seen, for most of the advantages, there are some serious problems to overcome.

Efficient generation of source programs

For entering much of the text of a program, this is true, but unfortunately there are enough complications to slow down experienced programmers. For example, the PASCAL **if** statement has an optional **else** clause. Should the editor automatically insert the **else** and have the programmer delete it if not desired, or should it not be included with the corresponding need to add it if wanted? Support chose the latter model, but in either case the editor will be wrong about half of the time.

In Support's case, the screen displays no information about optional syntactic units, so the user needs to know where such units are located. There are two modes of moving forward through a program: the → key moves to the next syntactic unit displayed on the screen, while the enter key is similar but will insert any optional phrases between displayed syntactic units as it moves. In POE's case[4] the opposite occurs. All optional units are displayed initially, and the user must delete them if not specifically wanted.

A more serious consequence is that syntactic units are added top-down, but programmers usually think of algorithms as sequential actions. For adding new statements, there is not much difference between sequential insertion and top-down development of the BNF:

```
<stmt list> ::= <stmt>; <stmt list> |
                <stmt>
```

as both generate statements in a left-to-right manner. Insertion of expressions such as A+B*C, however, essentially means to build the tree in postfix order (e.g., "+", "A", "*", "B", "C"), which is not the natural sequence.

In some environments, such as CMU's Gandalf[9], this top-down linking to the program's parse tree is embedded in the user interface; in Support's case, however, the LALR parser mentioned earlier was added. Straight text will be parsed and entered in its true infix format. The COED editor within Support was a valuable extension that permitted programmers to add small sections of program text (up to 22 lines of input) without violating the basic top-down nature of program generation in a syntax editor.

Early detection of syntax and semantic errors

While true, this is not much of a benefit if its consequences are considered. Experienced programmers generally do not make many syntax errors as they enter text, although novices do. (This might explain Support's greater popularity among non-programmers than among programmers.)

There are cases where this supposed benefit is actually a hindrance. If an experienced programmer thinks of a sequence of code to enter and makes an error in input, a standard editor will ignore the error and continue entering data. After finishing entering code, the programmer can fix the earlier problem. With a syntax editor, however, only correct syntax can be entered. The system will usually halt and beep until corrective action is taken. Thus there is a disruption in a train of thought where some deep semantic issue needs to be put aside (and forgotten?) to fix some simple syntax.

Looking at both of these reasons, as languages get more complex (e.g., ADA) syntax editing might make more sense, but in relatively simple languages, like PASCAL and C, there seems to be few benefits. There is little experience with such editors for complex languages. Arcturus[10] is a prototype of an ADA editor, but it was not made commercially available.

Screen layout is predefined

This is also true, but again the predefined layout might not be what the programmer wants in all cases. It certainly helps the novice generate nicely indented listings, but as the programming task grows more complex, the number of special cases increases.

The placement of comments seems to pose a problem with all such editors. Comments are generally outside the language's defining BNF. Where do they appear in the listing? In Support they are tagged before the defining nonterminal. This works in some cases, but not all.

Uniform debugging and testing tools

This again is true, but a syntax editor is not needed for this feature. An integrated framework and data repository are needed for a source program. The current interest in CASE (computer-aided software engineering) tools exemplifies this, and Support is simply a CASE tool with a syntax editor for a base.

In summary, the experiences with Support are by no means unique and closely mimic experiences others have had with syntax editors. For example, Mentor, initially developed about eight years earlier at INRIA, has had a similar pattern of development and use[11]. Similar to experiences with Support:

- Novices used menus but experienced programmers rarely did.
- Experienced programmers wanted the full-screen Emacs editor for textual input and modification (providing functionality similar to the COED editor described here) using automatic parsing and unparsing of the Mentor input.
- Switching between Mentor and Emacs was difficult due to the inherent problems in placement of comments. On the other hand, Mentor was a powerful source-code maintenance system due to the integration of many program analysis tools for obtaining semantic information about a program. But just as in Support's case, such tools are mostly a function of

Mentor being an integrated environment and not simply an editor.

In conclusion, the drawbacks seem to be as serious as the advantage in syntax editing, which probably explains their lack of growth and popularity since the early '80s. As a final comment, source-code development is often stated as 15% of total life-cycle costs. Even if the editor reduced coding time to zero, that would still mean a productivity improvement of only 15%. Industry is looking for more than that.

SPECIFICATIONS

The previous discussion indicates that while syntax editing of source programs is a powerful technique, it probably has minimal effect on programmer productivity. As requirements, specification and coding take up to 75% of the costs to develop a system, however, improving those phases of the life-cycle might have a more dramatic impact on productivity. In addition, a mechanism to improve the flow between specifications to design to code would probably lead to fewer interface errors, hence decreasing the effort needed in testing and further increasing improved productivity.

For coding source programs, there are several programming techniques: procedural languages (e.g., PASCAL, C, ADA, COBOL), applicative languages (e.g., LISP, PROLOG), object-oriented programming (e.g., SMALLTALK, C++), etc. Their relative strengths and weaknesses for specific applications are fairly well established. For specification of a program, there are also several models (e.g., axiomatic, denotational, algebraic, functional); however, as yet there is no clear consensus as to which is most effective and how each applies to different application domains. This is still very much an open research question, with many ongoing projects studying various specification strategies.

Given the powerful syntax editing paradigm and its relative inability at improving source-code generation, the author decided to investigate it within a specification domain. After all, most specification languages have a syntax and semantics more complex than most programming languages, and some anecdotal data do seem to indicate that programmers would prefer syntax editors for sufficiently complex languages.

As stated previously, Support processes a language defined by an external grammar file, and it is constructed as a set of independent tools, each writing to virtual windows that are mapped to the actual computer screen. By modifying this grammar and by adding new support tools, Support becomes an interface 'shell' for a series of integrated environments. It can be used as a language processing meta-environment by providing the capabilities to read input, parse text, build parse trees, and manipulate multiple windows simultaneously. Using Support, two such extensions were developed that are described here: AS* (based on algebraic specifications) and FSQ (based on functional specifications).

```
(1)   sort sequence [sort something] is
(2)   constructor
(3)       epsilon;
(4)       cons : something, sequence;
(5)   operation head : sequence → something is axiom
(6)     head(epsilon) == ?;
(7)     head(cons(X,Y)) == X;
(8)   operation count : sequence → integer is axiom
(9)     count(epsilon) == 0;
(10)    count(cons(X,Y)) == 1+count(Y);
(11)    end;
```

Figure 1. Example of sequence specification

AS* for executable specifications

An algebraic specification is a series of axioms that link together the operations that can be applied to an abstract data type. As an extension to the Support environment, a specifications extension based on these algebraic axioms has been defined.

An AS* specification contains three features:

- a set of sort names that define new abstract objects and their constructors
- a signature, which defines a set of defined operations for manipulating the abstract objects
- a set of oriented equations (or axioms) that relate the defined operations and constructors to each other

Figure 1 gives a simple example of a specification for a sequence. Line (1) specifies that a class of objects of sort (i.e., type) 'sequence' is being defined and indicates that the new object will require as a parameter a sort 'something' that will be specified in a later binding. A generic class of sequences that will be instantiated by this later binding to 'something' is being defined. Lines (2)—(4) define the two constructors needed to create an object of this sort: 'epsilon' to return the empty object of sort 'sequence' and 'cons', which takes an element and a sequence and returns a new sequence with the element in it. The functionality of each constructor is given after its name with the sort name 'sequence' implied as last (e.g., 'epsilon' returns an empty 'sequence' and 'cons' requires a 'something' and a 'sequence' and returns a 'sequence'.) 'Epsilon' initializes objects of this sort and 'cons' creates new complex objects.

This object is manipulated by means of a set of defined operations. In this simple example, operations 'head' and 'count' are given with their signatures on lines (5) and (8). They are defined by the rewrite rules (axioms) on lines (6)—(10). 'Head' says to return the element last included into the sequence by the 'cons' function, while 'count' returns 0 for 'epsilon' (i.e., an empty list) or 1 plus the size of any non-null list with the first element removed. As can be seen, the formal definitions of each function includes recursive algorithms for computing its value by reducing any complex object to a finite set of applications of the constructor functions. The '?' on line (6) is equivalent to an error condition, and the implementation stops execution and issues an error message when

this occurs. (That is, it is illegal to take the 'head' of an empty list.)

For example, the list <X,Y,Z> is created by the construction:

```
cons(X,cons(Y,cons(Z,epsilon)))
```

and the operation 'count' uses this construction, as in:

```
count(<X,Y,Z>)=
1+count(<Y,Z>)=
1+1+count(<Z>)=
1+1+1+count(<epsilon>)=
1+1+1+0=
3
```

The use of the Knuth—Bendix algorithm[12] defines a proof of adequacy of the resulting algebraic equations by showing the equivalence of supposedly equal terms to the same ground (i.e., constant) terms. As the Knuth—Bendix algorithm is based on an ordering transformation from one term to a 'simpler' term, however, the algorithm defines an operation that can be 'executed' and proven to terminate. Therefore, any set of axioms that is 'Knuth—Bendix' can be transformed mechanically into a series of transformations that can be executed in some programming language, in this case PASCAL.

Similar to Larch and Larch/CLU[13], AS* specifications are independent of the underlying programming language and must be defined relative to any concrete language. Libraries of generic specifications can be used to form the basis of a reuse methodology where the generic specification is refined to an explicit specification in a specific programming language by binding the generic sorts to specific programming language types. In this case PASCAL is considered as the implementation vehicle, so to create ASPascal, the extension to PASCAL that contains AS* specifications, a link between a PASCAL object and an AS* sort must be indicated.

An explicit specification is created by a refinement of a generic specification via the **use** clause, as in:

```
sort intsequence is
    use sequence [integer]
    end;
```

which refines the generic sort 'sequence' given earlier and indicates that a new sort 'intsequence' is created by modifying 'sequence' with a binding of PASCAL integers to the free sort 'something' of Figure 1. The operations 'head' and 'count' in 'sequence' become 'intsequence_head' and intsequence_count' in the new sort, although the actual mapping to their new names is handled automatically and of no concern to the programmer.

The interface assumption is made that an explicit sort specification

```
sort newsort is ...
```

is equivalent to the PASCAL type declaration

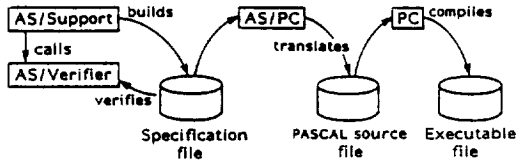


Figure 2. AS toolset*

```
type newsort = ...
```

The primitive PASCAL scalar types (char, Boolean, integer, real) may all be used in abstract sort definitions, and any explicit sort may also be used in a refinement. Thus

```
var A: intsequence;
```

simply creates a PASCAL variable A, which is of type 'intsequence'.

The power of this system is in alternative bindings. For example, real sequences could be created as

```
sort realsequence is use sequence [real] end;
```

Similarly, a sort such as a 'book' could be used to create a type 'library' as

```
sort library is use sequence [book] end;
```

As stated earlier, syntax editors might have greater use with more complex source languages, and the integrated tool set forms an effective basis for a CASE tool. Therefore, a prototype AS* system was built on top of the existing Support environment. Figure 2 represents this initial system that has been constructed. The four components are as follows.

AS/Support

AS/Support is a modification to the Support environment described earlier, which provides text-editing capabilities for creating specifications. It is also the control module that invokes the verification tool. AS/Support first checks axioms within operations for syntactic consistency. Because of the language-based design of the underlying environment, only syntactically correct axioms with the syntax

```
operation_name(<expression_list>)= =<expression>
```

can be entered by the user. After the user builds a sort, AS/Support formats the sort syntax into an appropriate format suitable for PROLOG and invokes AS/Verifier as a subprocess. AS/Verifier reads these axioms and checks executability. After passing all executability checks through AS/Verifier, the user may save the ASPascal program in a library for later translation by AS/PC or for later incorporation into another ASPascal program.

In case of failure, the causing axiom, if it can be determined, is highlighted to allow the user an interactive mechanism to change the specifications.

AS/Verifier

AS/Verifier, a PROLOG program, is called by AS/Support and verifies the set of axioms via the Knuth—Bendix algorithm. In general the axioms need to be a noetherian term rewriting system, and, if possible, AS/Verifier makes this determination. Of course, as the general problem is undecidable, in some cases the results are inconclusive. In any case, after one pass through the axioms, AS/Verifier will either succeed or indicate which axiom is currently failing so that the user may modify the definition and try again. As stated previously, if any error is found, an appropriate message is relayed back to AS/Support and displayed to the user.

For example, the 'sequence' definition of Figure 1 will be converted to the following clauses and passed to AS/Verifier:

```
as←sort (sequence, [epsilon, cons, head, count]).
function (1, epsilon, [], sequence).
function (2, cons, [something, sequence], sequence).
function (3, head, [sequence], something).
function (4, count, [sequence], integer).
axiom (5, head (epsilon), "?").
axiom (6, head (cons(x,y)),x).
axiom (7, count (epsilon),0).
axiom (8, count (cons(x,y)), 1 + count(y)).
```

(as←sort is the internal name for a new 'sort'.) The Knuth—Bendix algorithm either shows convergence of the axioms or indicates additional axioms that are needed; it may not indicate, however, when sufficient axioms have been added in the case of not converging rapidly enough (the usual problem with undecidability results). In this case, AS/Verifier does a single pass over the axioms and then terminates, indicating where the problem is with the axioms.

AS/PC

AS/PC is the translator, written in YACC, that converts specifications into standard PASCAL source programs. The code generally consists of a sequence of **if** statements, each checking the validity of the left-hand side of the axiom before executing the Knuth—Bendix reduction.

PC

PC is the standard system PASCAL compiler. At this point, the specifications have been converted to standard PASCAL, and any comparable compiler can be used for compilation and execution.

Specifications appear in programs as function calls in the host programming language. To translate such calls, it is necessary to determine, for each function reference, which explicit specification is being used. Thus a reference to 'head(thing)' where 'thing' is an 'intsequence' is translated to a call to 'intsequence_head(thing)', while 'head(realthing)' will result in 'realsequence_head(realthing)' for variable 'realthing' of sort 'realsequence'. (The details of the AS* implementation appear elsewhere[14].)

It should be clear that this translation does not result in a particularly efficient implementation; as a specifications or prototyping tool, however, efficiency is not its overriding purpose. The goal is to provide easily a correct extension to an existing system and to provide a verification tool, e.g., an oracle, that can be used as a test against an eventual efficient solution to the problem.

FSQ for software reuse

In the previous section, AS* was described as an environment based on an algebraic specification model for program specifications. Support is also being applied using the functional correctness model[15]. In this model, both a program and a specification are viewed as functions, and techniques have been developed to determine if both represent the same transformation of the data. This model of program development is briefly summarized and how Support is modified to aid in this process is then demonstrated.

Functional correctness

A specification f is a function. A box notation [...] is used to signify the function that a given string of text implements. If character string α represents a source program that implements exactly f, then $[\alpha] = f$, and it is stated that α is a solution to f.

Sequential program execution is modelled by function composition. If a sequence of statements $s = s_1;s_2; \ldots s_n$, then $[s] = [s_1] \circ \ldots \circ [s_n] = [s_n](\ldots)([s_1])) \ldots)$. Using techniques from denotational semantics, each statement s is a function from a program state to another state. Each program state is a function from variables to values and represents the abstract notion of data storage. Symbolic trace tables are used to derive the state functions for **if**, **while**, and assignment statements.

Program design is accomplished by converting a specification function f, written in a LISP-like notation, into a source program α, and then showing that $[\alpha] = f$. The specification f is called the abstract function and the program α the concrete design. Given this functional model, the basic theorem for functional correctness[16] can then be proved. Program p is correct with respect to specification function f if and only if $f \subseteq [p]$.

This model can be applied to three separate activities:

- Program verification. If f is a function and if p is a program, determine if they are the same function, i.e., $[p] = f$, or more generally $f \subseteq [p]$.
- Program design. If f is a function, then develop a program p such that $[p] = f$.
- Reverse engineering. If p is a program, then find a function f such that $[p] = f$

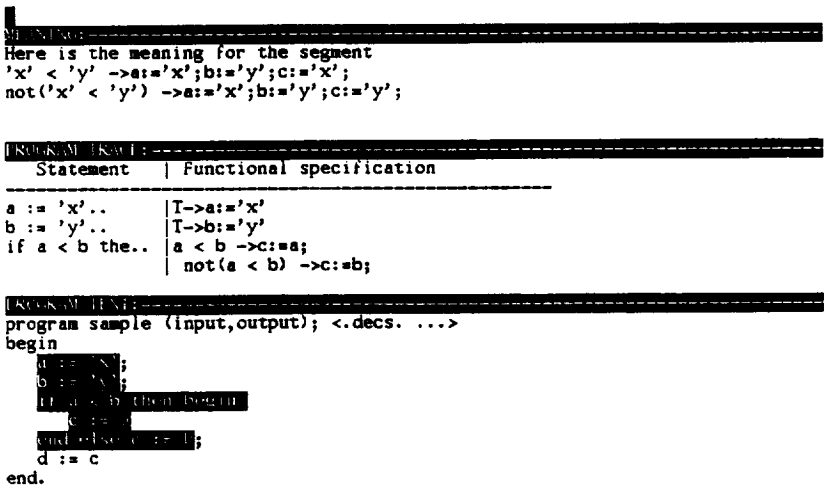


Figure 3. FSQ derived meaning for program fragment

FSQ extensions

The use of existing program fragments when developing a new program is one technique being studied for improving programmer productivity. Often, however, it is first necessary to determine exactly what these program fragments or procedures do. As formal specifications are rarely used, and documentation is generally quite inadequate, programmers are reluctant to use an existing procedure written by another from some previous project since the mental effort to truly understand that procedure is quite high.

To study this problem, the Support environment was extended with a new tool, Function Specification Qualifier (FSQ), to aid this process of determining the specifications for an existing component of a system. FSQ-1, a first prototype of this tool, is described.

FSQ is an additional tool to the basic CF-PASCAL programming environment in Support and works as follows:

- A programmer either builds a program using Support (and hence uses FSQ as a verification tool) or else reads one from the file system using the LALR parser internal to Support to build the parse tree (making FSQ a reverse engineering tool).
- The cursor is moved over the section of program that needs to be verified and FSQ is invoked via the command *.fsq*.
- FSQ symbolically executes each statement and determines its meaning. This is relayed back to the user, who either accepts this meaning (e.g., its specification) or manually simplifies it to another meaning.
- The derived meaning is stored in the Support syntax tree. If any part of a program is symbolically executed and already has a derived meaning, then that meaning will be used without further analysis. This meaning can then be carried along as part of the file system repository information on that object. Future users of that object will not have to derive the meaning again.

  Over time, more and more procedures in the system repository will have such derived meanings, making it more efficient to reuse such objects frequently.

Figure 3 shows a sample execution of FSQ. The top meaning window shows the desired result from the execution, the middle program trace window indicates each partial result, and the bottom window highlights the section of the source program that is under study.

FSQ executes over the covered portion of Figure 3 as follows:

- (1) For a: = 'x' the system derives the conditional T → a: = '*x*'. (This is similar to the LISP 'cond' and means 'True implies a: = '*x*'.')
- (2) For b: = 'y' the system derives the conditional T → b: = '*y*'.
- (3) For c: = a the system derives the conditional T → *c: = a*.
- (4) For c: = b the system derives the conditional T → *c: = b*.
- (5) For the if statement, FSQ combines steps (3) and (4) to produce:

  not (a < b) → c: = b;
  (a < b) → c: = a

- (6) Finally, for the entire sequence, FSQ combines the results from steps (1) through (5) to produce the function described in Figure 3.

Note that this process is simpler than general program verification (and potentially less accurate) as the programmer can override the system and insert arbitrary definitions. For example, in the program of Figure 3, the user, in the process of deriving the meaning of the if statement at step (5), could have either substituted the correct simplification

c: = min (a.b)

or any other correct or incorrect expression for the **if**. Thus the user must trade off between 'absolute' but extremely difficult correctness using a verifier and a system like FSQ, which performs efficient, but possibly imperfect, verification. The tool is truly interactive, with FSQ performing all the tedious bookkeeping procedures, and by having the user required provide for the creative program derivation activities. This avoids the general undecidability issues of general verifiers and permits the data-intensive functional verification mechanism to be used practically.

CONCLUSIONS

In this paper the basic features of syntax-directed editors have been described and possible reasons why such editors have not become more popular outlined. The author believes that their benefits do not increase productivity sufficiently to compensate for their deficiencies. Source-code generation, although labour intensive, is not a major cost factor in system development.

However, syntax editors can provide a consistent interface when system specification is integrated with source-code generation. To experiment with this, two specification projects have been described as extensions to an existing PASCAL development environment. In these extensions both algebraic specifications and functional correctness models of development were applied as extensions of automated tool support. Further work is needed to test the eventual applicability of this form of environment.

ACKNOWLEDGEMENTS

This work was partially supported by Air Force Office of Scientific Research grant 87-0130, Office of Naval Research grant N00014-87-K-0307, and NASA grant NSG-5123, all to the University of Maryland. Individuals who have contributed include: for Support: Bonnie Kowalchack, David Itkin, Jennifer Drapkin, Michael Maggio, and Laurence Herman; for AS*: Sergio Antoy (of Virginia Tech), Sergio Cardenas, Paola Forcheri and Maria Teresa Molfino (of I.M.A., Genoa, Italy), Stuart Pearlman, and Lifu Wu; and for FSQ: Victor Basili and Sara Qian.

REFERENCES

1 **Hansen, W J** 'User engineering principles for interactive systems' in *Proc. Full Joint Comp. Conf.* Vol 39 (1971) pp 523–532

2 **Donzeau-Gouge, V, Kahn, G, Huet, B, Lang, B and Levy, J** 'A structure assisted program editor: a first step towards computer assisted programming' in *Proc. Int. Computer Symp.* North-Holland, Amsterdam, The Netherlands (1975)

3 **Teitlebaum, T and Reps, T** 'CPS: the Cornell Program Synthesizer' *Commun. ACM* Vol 24 No 9 (1981) pp 563–573

4 *Proc. ACM SIGPLAN Symp. Language Issues in Programming Environments* Seattle, WA, USA (June 1985)

5 *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984)

6 **Zelkowitz, M V** 'A small contribution to editing with a syntax directed editor' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 1–6

7 **Zelkowitz, M V, Kowalchack, B, Itkin, D and Herman, L** 'A support tool for teaching computer programming' in **Fairley, R and Freeman, P (eds)** *Issues in software engineering education* Springer-Verlag, Berlin, FRG (1989) pp 139–167

8 **Fischer, C, Pal, A, Stock, D, Johnson, G and Mauney, J** 'The POE language-based editor project' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 21–29

9 **Habermann, N and Notkin, D** 'Gandalf: Software development environments' *IEEE Trans. Soft. Eng.* Vol 12 No 12 (December 1986) pp 1117–1127

10 **Standish, T and Taylor R.** 'Arcturus: a prototype advanced Ada programming environment' in *Proc. ACM SIGSOFT Practical Software Development Environment Conf.* Pittsburgh, PA, USA (April 1984) pp 57–64

11 **Lang, B** 'On the usefulness of syntax directed editors' in *Proc. IFIP Workshop on Advanced Programming Environments* Trondheim, Norway (June 1986) pp 45–51

12 **Knuth, D and Bendix, P** 'Simple word problems in universal algebras' in *Computational problems in abstract algebra* Pergamon Press, New York, NY, USA (1970) pp 263–297

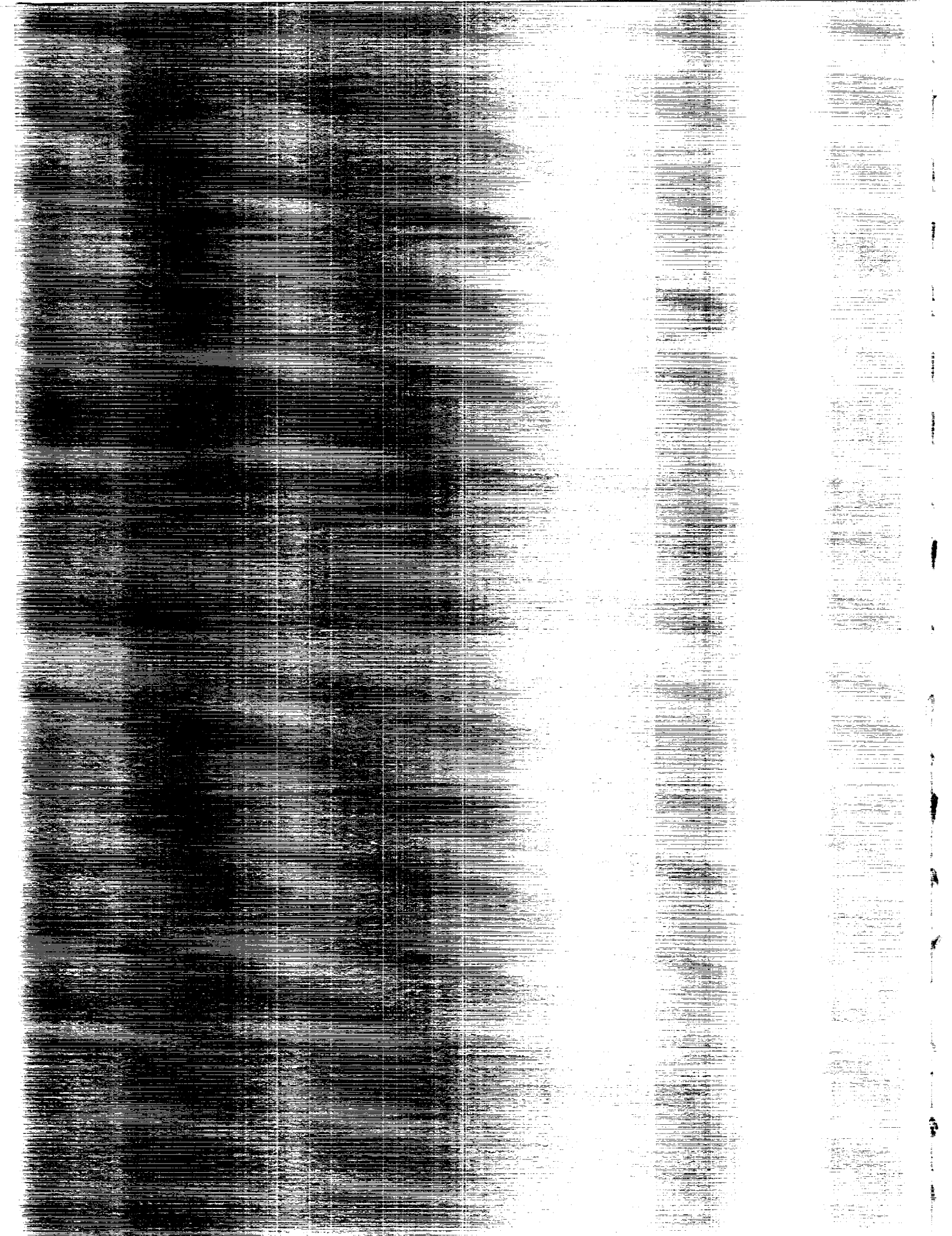
13 **Wing, J** 'Writing Larch interface specifications' *ACM Trans. Prog. Lang. Syst.* Vol 9 No 1 (1987) pp 1–24

14 **Antoy, S, Forcheri, P, Molfino, T and Zelkowitz, M** 'Rapid prototyping of system enhancements' in *Proc. 1st Int. Conf. System Integration* (April 1990)

15 **Gannon, J D, Hamlet, R G and Mills, H D** 'Theory of modules' *IEEE Trans. Soft. Eng.* Vol 13 No 7 (July 1987) pp 820–829

16 **Mills, H D, Basili, V R, Gannon, J D and Hamlet, R G** *Principles of computer programming: a mathematical approach* Allyn Bacon (1987)

SECTION 5—ADA TECHNOLOGY STUDIES



SECTION 5 - ADA TECHNOLOGY STUDIES

The technical papers included in this section were originally prepared as indicated below.

- "On Designing Parametrized Systems Using Ada," M. Stark, Proceedings of the Seventh Washington Ada Symposium, June 1990
- "PUC: A Functional Specification Language for Ada," P. Straub and M. Zelkowitz, Proceedings of the Tenth International Conference of the Chilean Computer Science Society, July 1990
- "Software Reclamation: Improving Post-Development Reusability," J. Bailey and V. Basili, Proceedings of the Eighth Annual National Conference on Ada Technology, March 1990

On Designing Parametrized Systems Using Ada

Michael Stark

Goddard Space Flight Center

1. Introduction

A parametrized system is a software system that can be configured by selecting generalized models and providing specific parameter values to fit those models into a standardized design. This is in contrast to the top-down development approach where a system is designed first, and software is reused only when it fits into the design. The term reconfigurable is used interchangeably with parametrized throughout the paper. This concept is particularly useful in a development environment such as the Goddard Space Flight Center (GSFC) Flight Dynamics Division (FDD), where successive systems have similar characteristics.

The FDD's Software Engineering Laboratory (SEL) has been examining reuse issues associated with Ada from the beginning of its Ada research in 1985. The lessons learned have been applied to operational Ada systems, leading to an immediate trend towards greater reuse than is typical for FORTRAN systems [McGarry 1989]. In addition, the Generic Simulator prototyping project (GENSIM) was a first effort at designing a parametrized simulator system. The lessons learned through the use of Ada and the GENSIM prototype are being applied to the Combined Operational Mission Planning and Attitude Support System (COMPASS), which is to be a reconfigurable system for a much larger portion of the flight dynamics domain. This paper will discuss the lessons learned from the GENSIM project, some of the reconfiguration concepts planned for COMPASS, and will define a model for the development of reconfigurable systems. This model provides techniques for realizing the potential for "Domain-Directed Reuse", as defined by Braun and Prieto-Diaz [Braun 1989].

The major motive for reconfigurable systems in the FDD is cost reduction. Having a well-tested set of reusable components may also increase reliability and shorten development schedules, but cost is the primary factor in this environment. Research done by the SEL indicates that verbatim software reuse (reuse without modification) can produce major cost savings. The cost of integrating a component that is reused verbatim is approximately 10 per cent of the cost of developing a new component from scratch [Solomon 1987]. Analysis done for GENSIM indicated that approximately 70 to 80 per cent of the code could be reused verbatim, and that this should cut simulator development costs in half [Markley 1987].

2. Reconfigurable Systems

This section focuses on the approaches taken and lessons learned from the GENSIM and COMPASS projects. These lessons influenced the reuse concepts and techniques defined in the subsequent sections of the paper.

2.1 GENSIM Overview

The GENSIM project was started in late 1986, and divided into two major phases. The first phase lasted until mid-1988, with the major products being the cost analysis cited above, mathematical specifications, and the high level system design. From mid-1988 to mid-1989 a small development team started implementing prototype software. The project was terminated before the prototype system was completed and evaluated, as COMPASS incorporates simulation requirements into its broader domain. Nonetheless, enough development work was done to learn some useful lessons.

The generic simulator design consists of a set of "modules" that plug into a standardized simulator architecture. Each of these modules was expected to have a corresponding mathematical specification, design data (object diagrams and Ada package specifications), and source code. The use of standardized specifications was intended to prevent the slight differences in specifications that often impede verbatim reuse. In addition, the GENSIM project intended to maintain test plans, data, and software for each module, so that changes in standard modules could be tested rapidly.

The simulator architecture is based on the designs of the first two Ada simulators developed in the FDD. The enhancements

Simulator Architecture

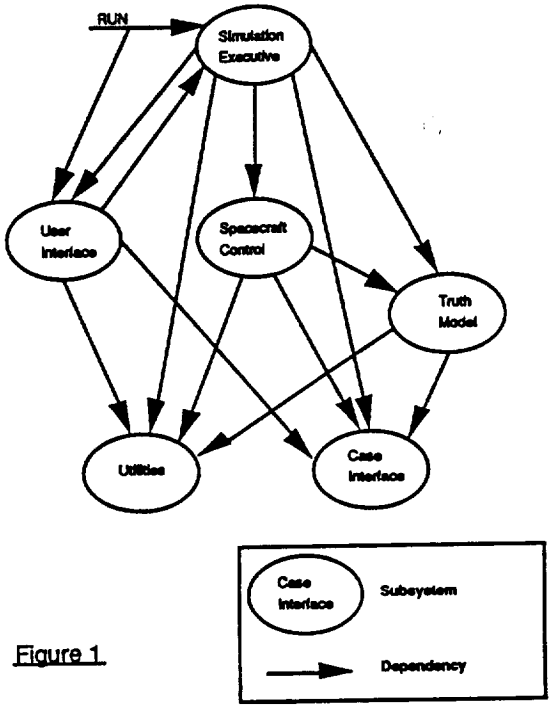


Figure 1

and changes to this architecture were intended to allow different sets of modules to be configured into a system, depending on the simulation requirements for a given satellite. It was possible to generalize the early designs, but because these were early designs, GENSIM incorporated some design flaws, even as others were removed. The major results of GENSIM were

1) The concept of reusing products from all life cycle phases presented no problems, and provided the anticipated benefit of standardizing mathematical specifications. The GENSIM team thoroughly specifies the individual simulator modules. However, the connections between modules were made at design time, despite the fact that they represented dependencies inherent in the problem. Note capturing these dependencies in the specification was not a problem, since the GENSIM team happened to be knowledgable enough to assure that a function needed by one module was provided by another. Nonetheless, problem domain dependencies should preferably be captured in the specifications, so that developers with less domain expertise will have the information they need. The COMPASS team is representing problem domain dependencies in their standardized specifications.

2) The configuration of a system is done by instantiating all the necessary generic Ada packages in the correct order. The GENSIM team instantiated each package as a library unit. In cases where the same set of packages are used in each system, generics can be combined so that a subsystem can be "instantiated" through the instantiation of a single generic package.

3) The legacy of the previous simulator architectures made the implementation of standardized components more difficult. In particular, the storage of inputs and results for a given simulation scenario could not be adequately generalized. This lesson is discussed in more detail in the next section.

2.2 GENSIM as a Standardized Architecture

The purpose of the flight dynamics simulators generalized by GENSIM is to test the flight dynamics control algorithms for a satellite before it is launched. Figure 1 shows the architecture for a spacecraft simulator built from GENSIM modules. This diagram shows the dependencies between major simulator subsystems. The Truth Model represents the "true" response of a spacecraft to its control system, and is configured using the components needed for a specific satellite. The Spacecraft Control subsystem contains new code that implements a particular satellite's control laws. The remaining subsystems are built to support these two subsystems, and must also be configurable to support varying sets of modules. This reconfigurability became especially cumbersome for the Case Interface, which is the subsystem that manages input data and results for simulation scenarios (cases). Figure 2 shows the two major parts of Case Interface. All simulation inputs are managed by Parameter Interface, and all results are managed by Results Interface. These two subsystems are accessed by both the user and the two simulation subsystems.

Case Interface Subsystem

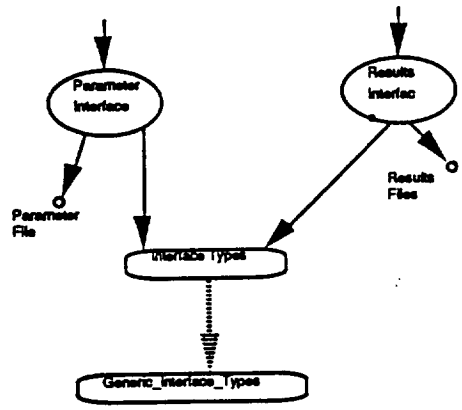


Figure 2

FSS Module's View of the Case Interface

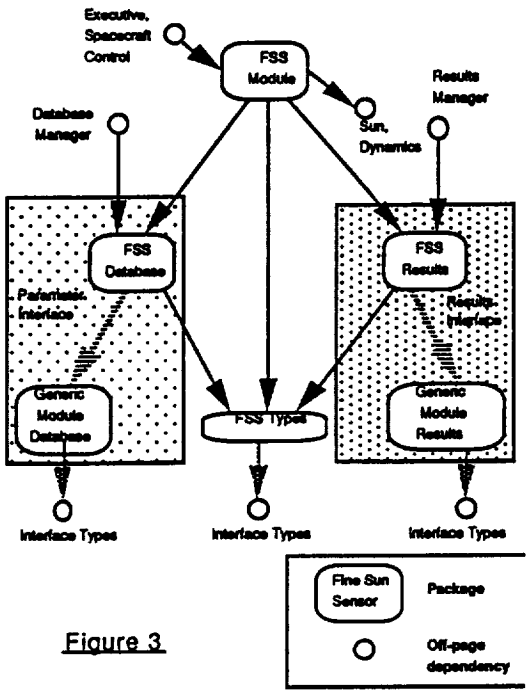


Figure 3

The GENSIM configuration concept called for the subsystems of the Case Interface to be built from components associated with each module. Figure 3 shows how a parameter and results database is created for a Fine Sun Sensor (FSS) module by instantiating standardized generics. The "FSS_Database" package is used by the module's initialization routine to get initial parameters, and the "FSS_Results" package is used by the module's computation routines to store simulated results. The shaded areas show that the individual components fit into the Case Interface packages. Figure 4 shows how several module databases fit into the Parameter Interface subsystem.

The advantage of this approach is that the packages Interface_Types and FSS_Types contain all the declarative information needed to include a module in a simulator configuration, and that standard types and protocols are used to achieve this. The configuration parameters include default values for module input parameters, flags indicating which parameters a user is allowed to change, and similar flags indicating what results a user may display during a simulation or print after a simulation. The disadvantages of this design approach are

1) the developer of a flight dynamics module has to be aware of all the complexities inherent in the simulator architecture, and all the dependencies shown in Figures 3 and 4, and

2) the parameters passed in and out of a package are limited to the data types defined by Interface Types. Module specific enumeration types (such as "type FSS_POWER is (OFF,ON)") cannot passed to the user except by using the 'POS attribute to convert to an integer which is then displayed.

Figure 5 shows an improvement to the architecture that addresses the first disadvantage. The package FSS_ADT exports an abstract data type (ADT) that implements all the modeling of the fine sun sensor. Now the state of the FSS module is based on this abstract data type, and the module's functionality is implemented by calling the operations on the type. This allows package FSS_ADT to be implemented by a developer who is aware of all the nuances of fine sun sensor modeling, and the FSS module can be implemented by a developer who is aware of all the nuances of the simulator architecture. In addition, FSS_ADT and all the other abstract data types defined for the flight dynamics simulation domain can be used to build a system with a completely different architecture, without changing a line of code in the packages that implement the modeling of the flight dynamics problem. An architecture that addresses the limitations imposed by Interface_Types can be built around such abstract data types, as is shown in section 4. The separation of problem domain and system architecture considerations is a key element of the reuse models described in section 3.

Parameter Interface Design

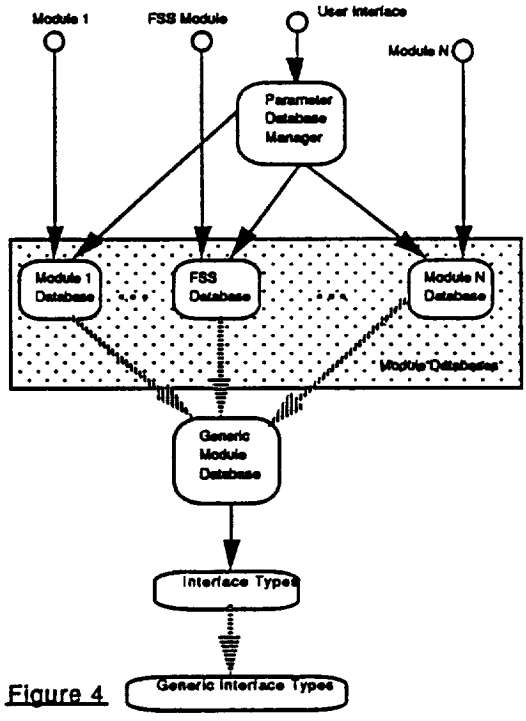


Figure 4

FSS Module's View of the Case Interface

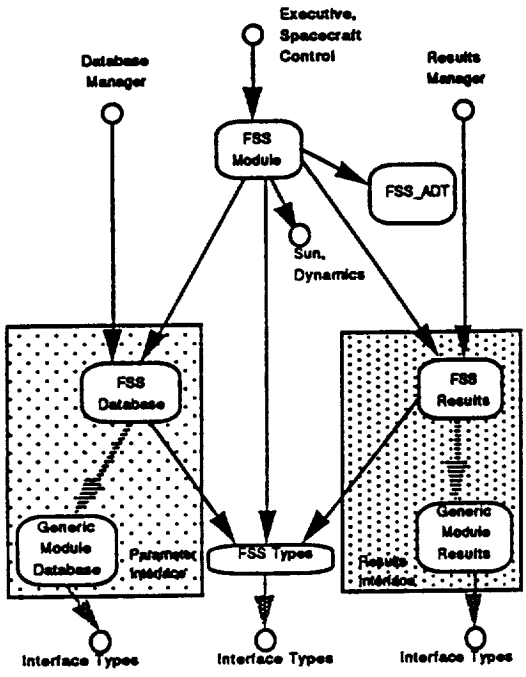


Figure 5

2.3 COMPASS

COMPASS is the second FDD project that is developing reconfigurable software. It has the same cost reduction goal as GENSIM, but covers a much larger problem domain.

COMPASS is intended to support the flight dynamics simulations area, mission planning and analysis both before and after launch, and spacecraft attitude support systems for mission operations. The estimated size of COMPASS is over a million lines (counting all carriage returns) of Ada source code, and is targeted to run on several different computers. This implies both being able to configure systems to run as distributed systems, and to be able to target the same functions to different platforms. These considerations have prompted refinements to the reuse model defined in [Booth 1989].

COMPASS has also involves defining standardized specifications to promote verbatim reuse. Unlike GENSIM, a standard specification methodology has been defined for COMPASS [Seidewitz 1989]. The COMPASS specification concepts are object-oriented, but contain restrictions tied to both reconfigurability and to project standards. For example, there is a restriction on the number of levels of superclasses and subclasses allowed in an inheritance hierarchy.

3. Reuse Concepts

To be able to design reconfigurable systems, it is necessary to have some underlying principles that can be used as design guidelines. The major concept defined in this paper is a Layered Reuse Model that categorizes components by function and defines dependencies among these components. The initial model was developed as a result of the work done on GENSIM and on an operational system, the Upper Atmosphere Research Satellite (UARS) Telemetry Simulator (UARSTELS) [Booth 1989]. This model was primarily driven by the need to separate problem domain and system architecture considerations, as is discussed in section 2. This model does not address how to incorporate very general components that have potential use across several problem domains and/or architectures, nor does it address the separation of system dependent features from potentially portable code. The latter omission became obvious when a multiplatform system such as COMPASS was considered.

The Layered Reuse Model

Major Layers	Levels	Examples
Architecture Levels	System Architecture Templates	Case_Interface
	System Modules	FSS_Module
Problem Domain Levels	Domain Definition Classes	FSS_ADT (Fine sun sensor abstract data type)
	Domain Language Classes	Linear_Algebra
Service Levels	System Independent Services	Booch Components (TM)
	System Dependent Services	DEC math library package

Figure 6

To address the above issues, a "services" layer was added to the model. This services layer is split into a system dependent and a system independent layer. The updated reuse model is shown in Figure 6. A component in a given layer can only depend on components in layers below it, as is the case in any good layered model. The layers are defined as follows:

System Architecture Templates -- Components at this level provide a template into which modules fit. These can be reconfigurable subsystems such as the GENSIM Case Interface discussed above, or they can be standard components that do not depend on the particular configuration. In GENSIM the Display Interface and the Plot Interface were designed to be

standard software, with any needed configuration data being provided by input files, rather than generic instantiation.

System Modules – This layer contains components that are designed to fit into a standard design. These modules are built from components at the problem domain and service levels.

Domain Definition Classes – These components define classes in the problem domain that are identified through domain analysis. They are generally implemented as Ada packages exporting abstract data types, as is discussed above.

Domain Language Classes – Components at this level capture the vocabulary of a particular domain, in other words, these classes capture the knowledge and language that domain experts use to express the specifications for domain definition classes. In the flight dynamics domain, such classes would include "vector", "matrix", "orbit", and "attitude". The domain analyst would use these simpler classes to define more complex classes such as "Fine Sun Sensor".

System Independent Services – This layer contains components that can be used in implementing both the problem domain layer and architecture layer components. They are usually usable in more than one problem domain and/or more than one system architecture. Components at this level include the generic data structures and tools provided by the Booch Components (TM) [Booch 1987], as well as portable interfaces to general services such as DEC's screen management routines. These portable interfaces can be moved to different computers, and new code or a different commercial product can be used to implement the same functions. Thus one ends up with multiple non-portable implementations of a single abstraction. Calls to this package should act the same, even if they are implemented in a machine dependent manner.

System Dependent Services – This layer contains all the components that are dependent on a particular computer or operating system. This generally includes all non-Ada code, as most other languages have different non-standard extensions on different machines. This also includes Ada code that incorporates system dependent features such as Direct_IO files created with a non-null FORM parameter. These system dependent features should have system independent interfaces at a higher level.

The improved model takes an object-oriented approach to specifying the problem domain. The domain definition classes and domain language classes form the two major groupings within the problem domain. Each of these two groups are also organized with the more domain specific classes depending on the more general classes. For example, the flight-dynamics classes "orbit" and "attitude" depend on the more general classes "vector" and "matrix".

The layered reuse model does not depend on Ada, but the Ada language contains features that support this model well. The use of generic packages allows each of the problem domain classes to be implemented as a generic unit that is completely decoupled from all other classes. In addition, the generic formal definitions associated with a package capture all the information about dependencies in a single location, as well as distributing external references throughout the code. Another useful feature is the separation of package specifications (and subprogram and task specifications as well) from their implementations. This is useful in hiding system dependent services, which can then have the system independent part defined at the appropriate layer. For example, the interface to a system dependent math library would be classified within the problem domain, and the interface to system-dependent screen management routines could be system independent services. The 5 top levels in this model would then contain system independent Ada code, which would be expected to be completely portable. This is not a consequence of attempting to make the highest layers portable, but rather is a benefit of isolating the known system dependencies, and using a standardized programming language. Using Ada leads naturally to having most reusable components also be portable. Similar portability may be attainable using C. It is almost certainly not attainable with FORTRAN, as the dialects vary too greatly between machines.

4. Example

This section presents an improved GENSIM design as an example of how to use the layered model. This new design is presented at the same level of detail as the original GENSIM design presented in section 2. Figure 7 shows the improved simulator design.

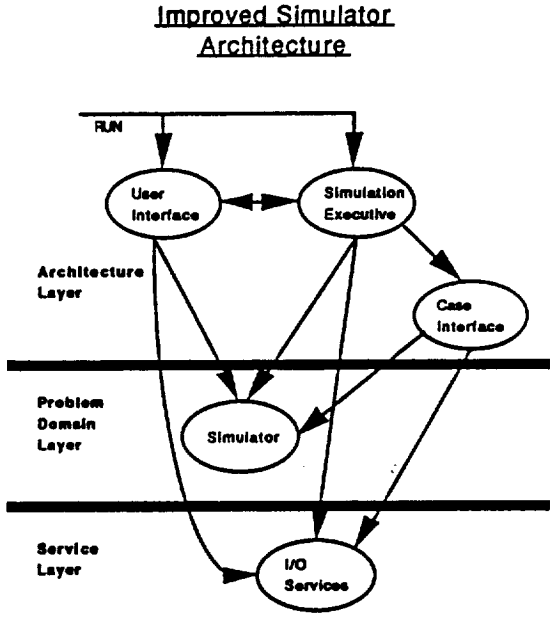


Figure 7

The key differences in this design are the location of the Case Interface subsystem and the new I/O services subsystem. In additon, the Spacecraft Control, Truth Model, and Utilities subsystem are combined into the Simulator subsystem. Figure 8 shows that the dependencies between these three subsystems are the same as in the original architecture (Figure 1), but that now none of these subsystems depends on Case Interface.

This extra design level is not carried through to implementation. Subsystems may be implemented as a single package which provides an interface to all the subsystem's components, but in this case the Simulator subsystem is merely

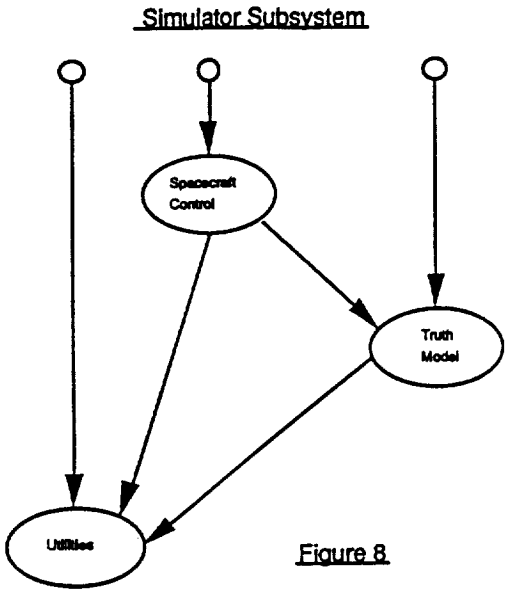


Figure 8

a logical grouping intended to reduce the design complexity.

Figure 7 also shows the three major layers of the reuse model. In this design, the I/O services consist of standard Ada packages such as Text_IO or Direct_IO, and an interface to DEC's Screen Management Guidelines (SMG) routines. Figure 9 shows the

interrelationship between the FSS module and the simulator architecture. Here the abstract data type for a sensor is created by instantiating a generic package. The generic ADT is designed so that all external dependencies are captured in the generic formal part. These dependencies include types provided by the simulator's Math_Types package, and functions to select information from the Sun and Dynamics modules. The FSS_Objects package uses the ADT (private type) exported by the FSS_ADT package to define its package state, and the FSS_Parameters_Display package uses visible types exported by FSS_ADT to define parameter screens. The FSS_Parameter_Displays package also instantiates Enumeration_IO using "type FSS_POWER is (OFF,ON)" as the actual parameter. This removes the reliance on using the 'POS attribute of enumerated types that has been a feature of all FDD simulators up until now.

Figure 10 shows how the FSS_Parameter_Display package fits into the design of the Case Editor subsystem. The Case Editor subsystem is the part of the User_Interface that allows a user to change any of the initial parameters for a simulation. The Parameter_Editor package tracks which displays the user has selected and calls the appropriate parameter display package. The difference is that now the User_Interface controls the initialization of simulation parameters, rather than the simulator components requesting intial values from a database contained within the Case Interface.

In this example, the use of the layered model removes the Truth Model's complex dependencies on the Case Interface packages shown in Figure 3. This enables the Simulator subsystem components to be usable within more than one architecture. The placing of the system architecture subsystems above the Simulator subsystem also allows general purpose service layer components to be enhanced as needed to integrate a given module into the system architecture. The FSS_Parameter_Display demonstrates this concept by using Enumeration_IO to add to the general IO services.

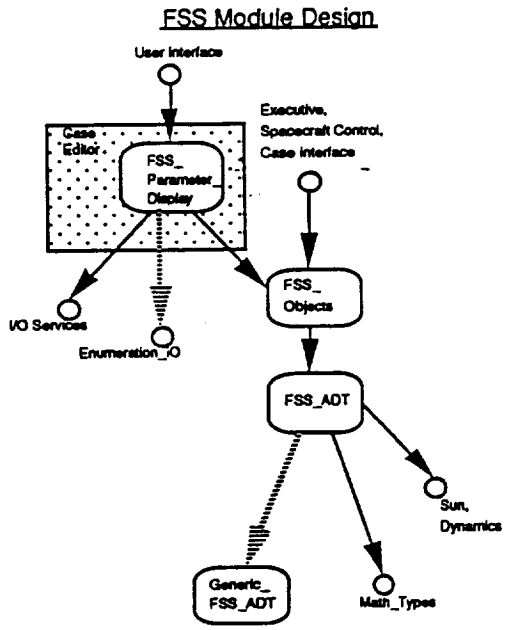


Figure 9

5. Future Directions

This paper describes a general reuse model for designing reconfigurable systems. The next step is to map the layered reuse model to Ada design and implementation concepts. The high-level designs presented in this paper use generic packages to help parametrize systems. There are many possible ways to incorporate generic packages into a larger design. These "reuse in the small" techniques include nesting generic instantiations, nesting generic definitions, and creating dependencies between library instantiations [Booth 1989]. This paper has used the last technique so that while generic instantiations are coupled, each of the generic units is completely decoupled from the others.

The layered reuse model provides a sound basis for project management. By strictly separating the problem domain issues from the system architecture issues, a manager can assign the appropriate experts to implement packages within each layer of the model. Improving the allocation of personnel to tasks should

improve both productivity and software quality. As this model is used, an understanding of what proportion of a system falls into

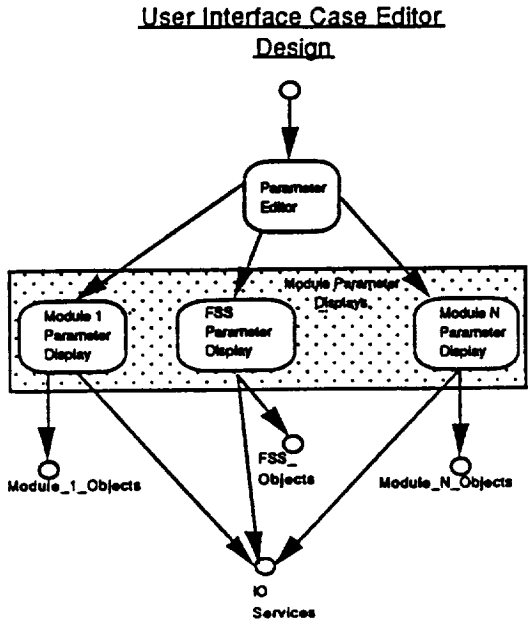


Figure 10

which layer will evolve.

The layered reuse model also can be used to understand which software is most critical. Layered models have seen the most use in operating system design. The kernel of an operating system typically requires the most attention, despite the fact it is a relatively small proportion of the code. This is because all other layers depend on its correctness and efficiency. The analogous layers in the reuse model are the service layers and the domain language layer. Additional evidence for the assertion is that the FDD has observed performance degradation in its Ada simulators due to the inefficient implementation of mathematical utilities packages.

In addition to the performance problems observed above, there is a concern that layered implementation models may be inherently slow due to the addition of extra levels of procedure calls to accomplish the same work. The FDD encountered this problem with a commercially provided graphics interface that provides the same FORTRAN interface routines on a VAX or an IBM mainframe. Whether this is due to extra procedure calls or generally inefficient implementation is unclear. Ada addresses the former problem by providing pragma Inline. The latter problem must be addressed by improving the software. If the software design and implementation is done properly, the layered reuse model should not degrade performance.

6. Conclusion

In "Domain-Directed Reuse", Braun and Prieto-Diaz extract properties that are common to applications (such as compiler design) where a high degree of reuse is already being obtained [Braun 1989]. These properties are a focus on a particular application domain, assumptions about system architecture constraints, and a set of generalized and well defined interfaces. The layered reuse model provides design concepts for examining applications domains and defining standardized architectures. These techniques will help realize the potential inherent in the concept of domain directed reuse.

References

[Booch 1987] Booch, G. Software Components With Ada, Menlo Park, Calif., Benjamin/Commings, 1987.

[Braun 1989] Braun, C. and R. Prieto-Diaz, "Domian-Directed Reuse," Proceedings of the Fourteenth Annual Software Engineering Workshop, November, 1989.

[Booth 1989] Booth, E. and M. Stark, "Using Ada Generics to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada '89, October 1989.

[Markley 1988] Markley, F. L., C. Mendelsohn, M. Stark and M. Woodard, "Impact Study of Generic Simulator Software (GENSIM) on Attitude Dynamics Simulator Development Within The Systems Development Branch," Unpublished FDD Study, 1988.

[McGarry 1989] McGarry, F., S. Waligora, and T. McDermott, "Experiences in the Software Engineering Laboratory (SEL) Applying Software Measurement," Proceedings of the Fourteenth Annual Software Engineering Workshop, November, 1989.

[Seidewitz 1989] Seidewitz, E. Combined Operational Mission Planning and Attitude Support System (COMPASS) Specification Concepts, Goddard Space Flight Center Flight Dynamics Division, COMPASS-102, 1989.

[Solomon 1987] Solomon, D. and W. Agresti, "Profile of Software Reuse in the Flight Dynamics Environment," Computer Sciences Corporation, CSC/TM-87/6062, November 1987.

PUC: A Functional Specification Language for Ada*

Pablo A. Straub†
Computer Science Department
University of Maryland

Marvin V. Zelkowitz
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland

Abstract

Formal specifications can enhance the quality, reliability, and even reusability of software; they are precise, can be complete in some sense, and are mechanically processable. Despite these benefits, formal specifications are seldom used in practice for several reasons: programmers lack an adequate background; both concepts and notations in specification languages appear obtuse to programmers; formal specifications are sometimes too high-level, providing too large a gap from the specification to the implementation; methods are not tailored to the environment; and fully formal methods are expensive and time consuming.

In this paper we present PUC (pronounced POOK), a specification language for Ada that addresses programmers' concerns for understandability. PUC is a functional language whose syntax and data types resemble Ada's, although it has features like parametric polymorphism and higher-order functions. The paper shows the requirements for the language PUC; presents an overview of the language and how it is used in the specification of Ada programs; and gives the requirements and strategies for a semi-automatic translator from PUC to Ada.

1 Introduction

The practical use of formal specifications in program development is an important goal in software engineering because formal specifications can enhance the quality, reliability, and reusability of software. Formal specifications are precise, can be complete (in some sense), and are mechanically processable (e.g., consistency checks). Since one of the main problems in software reusability is determining what is the functionality of a subprogram or module, having a precise description will also improve software reuse, lowering costs and improving quality by using already tested components.

Despite the benefits of formal methods, few are used in practice. There are several reasons: programmers do not have adequate background; both concepts and notation in specification languages are usually mathematically oriented; there is a big conceptual gap from a very high level specification down to the details of the implementation; methods cannot be tailored to the environment; fully formal systems are very expensive and time consumming, and much software is not critical enough to justify this cost.

A precise mathematical specification is useful only if it is understood by the persons involved in the development, so notational considerations are very important. Two aspects of the specification language have to be considered: the conceptual or semantic level and the syntactic level. The concepts represented in the language have to be very high level, like the concepts in the domain area, but not too high level, or else there will be a big conceptual gap from the specification to the implementation. Hence, there is a trade-off in the design of a specification language: if it is not very high level, the program analysts and designers have a hard time; if it is too high level, the implementors must figure out the algorithms from scratch. This trade-off is summarized by the question of how much design should be implicit on the specifications [13].

The second aspect of notation is syntax. Syntactic issues are sometimes dismissed as *syntactic sugar*; this is fine for a researcher who knows many programming languages and can learn another one very fast, but for most professional programmers syntax *is* important. In particular, a syntax that is similar but conflicting with the implementation language is confusing.

*Research supported in part by NASA Goddard Space Flight Center grant NSG-5123.
†Additional support from ODEPLAN and Catholic University of Chile.

This work grew out of studies within the Software Engineering Laboratory (SEL) of NASA Goddard Space Flight Center. The SEL has been monitoring the development of ground support software for unmanned spacecraft since 1976. Our goal is to improve the quality of software specifications within the SEL, to improve both software development and testing [13]. We approach this goal by incresing the use of formal methods in software specifications. The SEL environment is characterized by large (tens of K lines of code) scientific software with complex functions and complex structure; potential reuse of products and processes; programs written in Ada using object-oriented design; few critical timing constraints; and programmers without background in logic and abstract algebra. To increase formality of specifications, we designed the especification language PUC suitable to this kind of environment; in particular, users of PUC are not required to know advanced logic or abstract algebra.

Overview of the paper. The next section discusses the need for a new language. Section 3 presents the principal aspects of the PUC language, along with examples. Section 4 shows how Ada programs can be developed using PUC as the specification language. The example presented is a simplified telemetry processor for satellite data. The final section contains a summary, conclusions, and a description of further work.

2 Why Another Language

There are many specification languages, yet we have not found any that is suitable for our needs. This section motivates the design of PUC, by presenting previous work, design objectives and rationale.

2.1 Previous work

Specification languages proposed specifically for the Ada programming language are based either on first order predicate logic, Horn clauses, algebraic abstract data types, or procedural description.

Booch proposes to use Ada itself as a specification language for Ada programs. "Not only is Ada suitable as an implementation language, but it is expressive enough to serve as a vehicle for capturing our design decisions." [1, page 50] However, most design decisions that can be written in Ada are of syntactic nature. This includes functional decomposition, but the meaning of subprograms cannot be expressed in Ada without writing them in whole.

Anna (ANNotated Ada) is a specification language designed to provide machine-processable explanations of Ada programs [9]. Anna programs are Ada programs with formal comments, that describe the functional requirements for the program; properties of its components (variables, subprograms, modules); and how these components interact. Formal comments are in the form of pre- and post-conditions, module invariants, type constraints, and other assertions. Anna programs are executable because they are Ada programs, but the specifications themselves are only executable in the form of run-time testing for consistency.

The PLEASE specification language for Ada is based on logic restricted to Horn clauses [14]. PLEASE borrows from Anna the idea of writing formal comments in Ada programs. Programs in PLEASE are executable so they can be used to build prototypes, in which incomplete Ada programs call some procedures specified in PLEASE. Unfortunately, pure Horn clauses are so inefficient, that operational semantics (order of evaluation and PROLOG cut command) have to be explicitly declared complicating the specification.

The specification language Larch/Ada-88 also uses formal comments within Ada programs [11]. This language is one of the interface languages of the Larch family of specification languages. Larch specifications are done at two levels: the meaning of the abstractions used by the program are defined using the Larch shared language [5], and then one of the Larch interface languages is used to state what the program does in terms of these abstractions. The Larch shared language is based on algebraic abstract data types. Using Larch/Ada-88 and the prototype tools described in [11] it will be possible to develop verified Ada programs, hence this method is fully formal.

2.2 Design objectives

We set several specific goals in the design of the language to make it useful in the SEL environment. These goals are sometimes conflicting with each other.

- The language should bridge the usual gap between very high level logical specifications and the detailed data and control management in Ada.

- The language should be expressive and extensible. It should be easy to code domain specific concepts in a specification library.
- Specifications should be easily translated into Ada programs; only rarely used constructs are allowed not to have a simple Ada representation.
- The language should be easy to learn for an Ada programmer. It should have few concepts and very few concepts not present in Ada. Syntax should be Ada-like.
- The language should be executable, so that the specifications can be used as a prototype and in preparing test data for the final application.

2.3 Design rationale

Our first design decision is the semantic model on which PUC is based; that is, whether PUC specifications will consist of Horn clauses, procedural descriptions, etc., either purely or in combination with other semantic models.

Some researchers in formal specifications have advocated using both purely functional languages [3, 6, 16] and logic-based languages [8] for specifications, based mainly on the separation of concerns between what is intended and how it is achieved, especially in the management of data structures. The expressive power of logic languages and functional languages is not comparable, because logic languages can accomodate non-determinism whereas functional languages can be higher order [16]. Even though both logic and functional languages can be executable, we agree with Hoare in that "a modern functional programming language can provide a nice compromise between the abstract logic of a requirements specification and the detailed resource management provided by procedural programming" [7, page 90]. These arguments have influenced our decision to design a purely functional programming language to specify Ada programs.

Most functional languages have mathematical notation which makes them amenable to formal proofs; however, they have been developed for programmers with extensive mathematical background. Our goal in the design of PUC has been to make a specification language for Ada programmers who do not necessarily have this background. If formal proofs are needed, PUC specifications can be easily translated into recursion equations to prove properties of them.

Hence, both syntax and semantics of PUC are similar to familiar programming constructs. For example, instead of free algebras and pattern matching, in PUC there are variant records and case expression. The few constructs of PUC that are not present in Ada are explicit. For example, polymorphism is explicit in the declaration of polymorphic objects, and Curring (i.e., creating a higher order function by partial parameterization) is accomplished using predefined functions instead of just omitting parameters.

3 The Specification Language PUC

This section presents the main aspects of PUC. A technical report gives further details and a BNF description of the grammar [12].

3.1 Overview

PUC is a purely functional programming language with parametric polymorphism [2] and Ada-like syntax designed to serve as a specification language for Ada programs. Because PUC programs are executable, we will call PUC either a specification language or a programming language appropriate for prototyping.

A PUC program consists of a sequence of declarations of types and objects (functions and data). Type declarations give a name to a type and are needed to create new types. Object declarations give a name to an object, which represents a function or data object; they are either like Ada function definitions or like Ada assignments, where the defining symbol := is read as *is equal by definition* and represents the relationship of that object to other objects. This results in implied execution sequences by virtue of the partial ordering of these object relationships.

Example The following program consists only of data object declarations. The value of `root` is computed from the values of `a`, `b`, and `c`.

```
root := (- b + sqrt(b*b - 4*a*c)) / 2;
a := 2.0;
b := -4.0;
c := 2.0;
```

Example The program below defines `result` to be the factorial of 5. The program consists of two object declarations: `fact` and `result`.

```
result := fact(5);
function fact (n: integer) return integer is
begin
  if n = 0 then 1 else n * fact(n-1) end
end;
```

3.2 Data types

PUC is a strongly typed language, like Pascal or Ada. However, PUC types are higher level that Ada types. For example there are lists instead of arrays; recursive records instead of records and accesses. That means that PUC is easier to use, but not as efficient as Ada. There are four kinds of data types in PUC: scalar types, list types, record types, and function types.

The scalar data types in PUC are: `integer`, `real`, `boolean`, `character`, and enumerated types. Numeric types have the usual arithmetic operators (`+ - * / rem`); the boolean type has the operators: `not`, `and`, and `or`; and relational operators (`= /= < <= > >=`) are defined for scalar types. Precedence rules are the same as Ada.

Lists are unbounded sequences of objects of the same type. Constant lists are represented using square brackets. The catenate operator is `&`, subscripting and slicing (sublist) is done using parentheses. Strings are simply lists of characters.

Example Given the definition of `nums`, the following equalities hold.

```
nums := [10,20,30,40,50,60,70,80,90,100];

nums(4)      = 40
nums(2..3)   = [20,30]
[20,30,10]   = nums(2..3) & nums(1..1)
nums(8..8)   = [nums(8)]
length(num)  = 10
"string"     = ['s','t','r','i','n','g']
```

PUC records are very similar to Ada records; component selection uses the typical dot notation. Records can have variant parts and can be recursive. Variant records have components that depend on a tag, whose type must be boolean or an enumerated type. For example, type `expr` is a recursive record with variants to represent arithmetic expressions of integers.

```
type expr_kind is (number, plus, minus,
                   multiply, divide);
type expr is
  record
    case kind: expr_kind is
      when number => val: integer;
      when others => left, right: expr;
    end;
  end record;
```

The `null` record—compatible with all record types and similar to Ada's `null` access—is used to build finite recursive records without variant parts [10]. For example, the recursive record type `int_tree` represents binary trees of integers. Note the use of the type name as a constructor for constant records.

```
type int_tree is
  record
    datum: integer;
    left, right: int_tree;
  end;
a_tree := int_tree'(5, null,
                    int_tree'(8,null,null));
```

In addition to the arithmetic, list, and record expressions, there are two structured expressions, `if` and `case`. The syntax for these expressions is similar to the corresponding statements in Ada; the difference is that in place of a sequence of statements, a single expression is expected.

3.3 Functions

Functions in PUC behave like mathematical functions, mainly due to their declarative—as opposed to imperative—nature. Table 1 shows the main differences between Ada and PUC functions. Although functions are declared using a syntax similar to Ada, the text between the `begin` and the `end` is not a sequence of commands, but an expression. Usually this expression will involve conditionals and recursion.

Example Function `eval` evaluates an expression represented with the type `expr` from Section 3.2.

```
function eval (exp: expr) return integer is
  function eval_oper (exp: expr) is
    l := eval(exp.left);
    r := eval(exp.right);
  begin
    case exp.kind is
      when plus     => l + r
      when minus    => l - r
      when multiply => l * r
      when divide   => l / r
    end
  end eval_oper;
begin
  if exp.kind = number then exp.val
  else eval_oper(exp) end
end eval;
```

Ada Functions	PUC Functions
Can cause side effects	No concept of side effects in PUC
Can be generic	Can be polymorphic and higher order
Cannot return a function as result	Fully higher order
Can have local types, functions, procedures, variables, constants, ...	Can have local types, functions and constants
Body expressed using control flow statements	No concept of control flow; only conditionals and recursion

Table 1: Differences between Ada and PUC functions.

3.4 Polymorphism

An object is polymorphic if it can have more than one type. PUC has parametric polymorphism, where the type of an object can depend on another type [2]. This is similar to generic type parameters, although more general. PUC has polymorphic functions and polymorphic record types. Polymorphic functions are declared by preceeding the types of the parameters by a question mark (this declares an implicit type parameter).

Example The following polymorphic functions operate on lists of any base type.

```
function length (L: list of ?element) is
begin
  if L = [] then 0 else 1 + length(rest(L)) end
end;

function cons (elem: ?a; L: list of ?a)
              return list of a is
  ( [elem] & L );

function find (value: ?a; L: list of ?a)
              return list of a is
begin
  if L = [] then L
  elsif L(1) = value then L
  else find(value, rest(L))
  end
end;
```

Polymorphic records are used to define different records given a base type; they are also called type constructors. The example below defines a type constructor for binary trees which is used in the definition of a binary tree of integers.

```
type tree of elem is
  record
    datum      : elem;
    left, right: tree of elem;
  end;

type int_tree is tree of integer;
```

Polymorphic types are usually used in conjunction with polymorphic functions that operate on the type. For example, function `traverse_tree` builds a list from the in-order traversal of a binary tree.

```
function traverse_tree (t: tree of ?elem)
                       return list of elem is
begin
  if t = null then []
  else traverse_tree(t.left) & [t.datum] &
       traverse_tree(t.right)
  end
end;
```

3.5 Higher order functions

Higher order functions are those that have functions as parameters or compute a function as a result. A limited form of higher order functions is present in languages like FORTRAN or Pascal, where it is possible to specify a subprogram passed as a parameter to another subprogram. PUC is fully higher order because it imposes no restrictions on the kinds of higher order functions (e.g., a function can return a higher order function). A very limited form of higher order functions can be simulated in Ada using generics.

Usually higher order functions are polymorphic because they operate on polymorphic data structures (e.g., lists), but these two language features are independent. Figure 1 shows the definition of some standard higher order functions which are useful in defining other functions without explicitly writing the whole functions; that is, the use of higher order functions enhances reusability. Figure 2 shows several functions defined in terms of polymorphic functions; some of them were previously defined explicitly.

```
-- APPLY - a list with the application of f to the elements of L
function apply (f: function(?a) return ?b; L: list of ?a) return list of b is
begin
  if L = [] then [] else [f(L(1))] & apply(f, rest(L)) end
end;

-- FOLD_R - the right folding of list L with function f
function fold_r (f: function(?a,?b) return ?b; init: ?b; L:list of ?a) return b is
begin
  if L = [] then init else f(L(1), fold_r(f,init,rest(L))) end
end;

-- FOLD_R_1 - the right folding of nonempty list L with function f
function fold_r_1 (f: function(?a,?a) return ?a; L:list of ?a) return is
begin
  fold_r (f, L(1), rest(L))
end;

-- CURRY - a function like f, but with the first parameter fixed
function curry (f: function(?a,?b) return ?c; param1: ?a) is
begin
  function (param2:b) return c is f(param1,param2)
end curry;

-- FOLD_TREE - the folding of binary tree t with function f
function fold_tree (f: function(?b,?a,?b) return ?b; init: ?b; t: tree of ?a) return b is
begin
  if t = null then init
  else f( fold_tree(f,init,t.left), t.datum, fold_tree(f,init,t.right) ) end
end fold_tree;
```

Figure 1: Some standard polymorphic functions.

```
function traverse_tree (t: tree of ?a) return list of a is
  function combine (l:list of a; elem:a; r:list of a) return list of a is (l & [elem] & r);
begin
  fold_tree(combine, [], t)
end traverse_tree;

function sum_of_nodes (t: tree of integer) return integer is
  function add3 (l, elem, r: integer) return integer is (l + elem + r);
begin
  fold_tree(add3, 0, t)
end sum_of_nodes;

function concat (L: list of list of ?a) return list of a is
begin
  fold_r("&", [], L)
end;
```

Figure 2: Functions defined using polymorphic functions.

4 Developing Ada Programs with PUC

There are several approaches for developing Ada programs using PUC. One is to use PUC only as a formal documentation aid, taking advantage of its defined semantics, but not its executability. Using PUC simply as a notation requires in principle no software tool, but this is very limited; at least a parser and consistency checker has to be provided. But if there is a parser then it is relatively easy to build a translator or interpreter, so that specifications in PUC can be used as prototypes.

Another way of using PUC specifications, is to generate Ada implementations by means of a semi-automatic translation, in which a programmer decides implementation issues and can even modify the generated code. This choice seems to be more attractive than the others, because it provides a smooth transition from specifications to programs, but the caveat is that not all PUC constructs have a simple representation in Ada (e.g., Ada has no higher order functions).

These approaches are not fully formal development systems in the sense that it is still possible to write a program inconsistent with its specification. While this is not optimal, we think that our software engineering environment is not mature enough for a fully formal system, and that experience with semi-formal specifications (and development) is required before a fully formal development system can be used effectively.

In order to provide a translator from PUC to Ada it is first necessary to determine a set of translation rules that will preserve the semantics of the specification. Although this set will not be sufficient to translate any PUC program into Ada, we need to be able to translate most PUC programs, or else the method is impractical. There is an additional restriction we impose on the system: to facilitate manual modification of the generated Ada code (e.g., for optimization or maintainance) we want the generated Ada code to resemble the PUC specification.

Since PUC is syntactically similar to Ada, some PUC constructs require simple translations or even no translation at all. For example, enumerated types and simple record types are almost identical in both languages; recursive record types are translated into an access type and a record containing access fields. However, not all translations are so simple, because the semantics of PUC and Ada are quite different. It is particulary difficult to provide general and efficient translations for the use of (garbage collected) heap memory, lists, higher order functions, and polymorphism.

4.1 Memory management and functions

PUC functions can be translated to Ada functions or procedures. If procedures are used, there are choices in the parameter modes used (i.e., `IN`, `OUT`, `IN OUT`). It is not always possible to select any of the choices, though, because they depend on the way data is manipulated in the calling functions.

This brings up the issue of how memory is managed. The semantics of functional languages with automatic allocation and deallocation of memory is quite different from that of Ada. In Ada only local variables are allocated and deallocated automatically, because of the activation stack model used, whereas in functional languages all memory is allocated and deallocated automatically. An immediate consequence is that we will try to allocate as much memory as possible in the form of local variables, avoiding the use of the Ada heap. To do that we have to recognize when data can be stored safely in the stack (i.e., when we can be sure that data will not outlive the function call where the value was declared). One of the problems of this approach is that it complicates sharing.

Another important issue in the management of memory is when to use variables. In functional languages there are no updatable variables and that means that every value computed needs newly allocated memory. We want to take advantage of Ada variables to avoid these allocations, even if they ocurr in the activation stack. For example, tail recursion can be translated into loops that will use variables for the information that is passed to the next activation (i.e., iteration).

4.2 Translating lists

There are several ways to translate lists into Ada, based on arrays or linked lists. When lists have a fixed known length, they can be translated into Ada arrays. If the length is not fixed but there is a reasonable upper bound, lists can be represented by a record with an array and a count of used elements. When the length of the lists is highly variable or not bounded then a linked list representation is used, using a predefined generic package. In the case of strings, it is desirable to use Ada strings, so that

string variables are compatible with string literals. Array representations have advantages over linked lists because they can be more efficient and generated Ada code resembles closely the PUC code.

It is very difficult for the translator to detect whether a list can be represented by an array or not. On the other hand, if an array representation is chosen some upper bound has to be provided, so this translation cannot be done automatically. One solution to this problem is to provide a default representation with linked lists and let the programmer change that default. The default representation for strings are Ada strings. For each type that requires a non-default representation, the programmer has to specify which translation is desired. This translation applies to all objects of the type.

4.3 Translating higher order functions

Higher order functions are used often in specifications, because they are useful in representing abstract operations. Ada generics can represent uses of higher order functions in the particular case of functions passed as parameters, provided that all function parameters are statically known. This translation requires defining the function as generic and providing the corresponding instantiations.

It is hard to make a general translator for higher order functions. However, most programs use higher order functions that either satisfy the restrictions in the above paragraphs, or belong to a standard predefined set (e.g., the examples in Figure 1.) For the first case, the translation scheme described suffices. For the second case it is possible to have a set of ad-hoc translation rules for the standard higher order functions. These rules are semantic-preserving transformations coded into the translator [15], hence programs written in terms of standard higher order functions can be automatically translated. The system can be extended by adding translation rules for domain-specific higher order functions.

Example Function poly evaluates a polynomial represented by the list $[a_0, a_1, \ldots, a_n]$ of its coefficients, using the factorization

$$P(x) = a_0 + x(a_1 + x(a_2 + \ldots x a_n \ldots)).$$

```
function poly (as: list of real; x: real) is
  function combine (a_i, accum: real) is
    (a_i + x * accum);
begin
  fold_r_1( combine, as )
end;
```

The use of function `fold_r_1` (defined in Figure 1) can be transformed into a loop using the rule for `fold_r_1`. From the definition of `fold_r_1`, if as has only one element, then the result is equal to this element. If as has more than one element (say `as = [first] & rest`) then the result is equal to

```
combine( first, fold_r_1(combine,rest) )
```

That is, we can first compute the folding of the `rest` and then `combine` the result with the `first` element. This can be achieved by a loop that examines the elements in reverse order and accumulates the results of the folding. The first time the list will have only one element that is used to initialize the accumulator. Since we know that the loop will iterate `length(as)-1` times we can use a for-loop.

```
accum := as(length(as));
for i in reverse 1 .. length(as)-1 loop
  result := combine(as(i), result);
end loop;
```

To write the above loop in Ada we need to provide an implementation for lists and perform the corresponding translation on them. Note that this loop will be inefficient with linked implementations for lists, because `fold_r_1` accesses the elements in reverse order. Now we can expand the call to `combine` and produce a complete Ada function.

4.4 Translating polymorphism

Some polymorphic functions can be translated into generic functions with type variables. This is not true of all polymorphic functions, because parametric polymorphism is a type system more powerful than generic types. The restriction is that all uses of a polymorphic function must be monomorphic (i.e., it should be possible to assign a static type to every use of a polymorphic function). That means that a polymorphic function cannot call another function using polymorphic parameters. This restriction is in principle rather severe, but does not apply to predefined operators and functions whose invocations are translated by ad-hoc rules.

The difficulty with this approach is that all functions on the polymorphic type have to be explictly declared. For example, if we have a function to operate on lists, all list primitives used have to be declared, and the function can be generic on both the

```
generic
  type a;
  type list_of_a;
  with function empty_list return list_of_a;
  with function first (l: list_of_a) return a;
  with function rest  (l: list_of_a) return list_of_a;
function find (value: a; L: list_of_a) return list_of_a is
  result: list_of_a;
begin
  result := L;
  while not((result = empty_list) or else (first(result) = value)) loop
    result := rest(result);
  end loop;
  return result;
end find;
```

Figure 3: `Find` the longest sublist containing `value`.

base type and the list type. Figure 3 is the translation to Ada of function `find` from Section 3.4.

Polymorphic records can be translated into several record declarations, one for each instantiation. As with functions, all uses of polymorphic records have to be monomorphic, or else the translation cannot be done automatically.

Example The polymorphic function `fold_l` folds a list into one value by combining values pairwise from the left of the list. Since `fold_l` is also higher order, the techniques discussed above apply as well.

```
function fold_l (f: function(?a,?b) return ?b;
         accum: ?b; L: list of ?a) return b is
begin
  if L = [] then accum
  else fold_l(f, f(L(1),accum), rest(L)) end
end;
```

`Fold_l` can be transformed into a while-loop (it is tail recursive). Consider the following call to `fold_l`

```
result := fold_l(f, value, a_list);
```

From the definition of `fold_l`, if `a_list` is the empty list `[]`, then `result` is equal to `value`. If `a_list` is not empty (say `a_list = [first] & rest`) then the result is equal to

```
fold_l(f, f(first,value), rest)
```

so that this is a call to the same function, in which both `value` and `a_list` are updated accordingly. Hence the following while-loop in pseudo-Ada is a valid translation:

```
result   := value;
aux_list := a_list;
while aux_list /= [] loop
  result   := f(aux_list(1), result);
  aux_list := rest(aux_list);
end loop;
```

An obvious efficiency improvement is to use an index variable, updating this variable instead of copying a list. Furthermore, since the loop will iterate `length(a_list)` times we can use a for-loop.

```
result := value;
for j in 1..length(a_list) loop
  result := f(a_list(j), result);
end loop;
```

To write the above loop in Ada we need to provide an implementation for lists and perform the corresponding translation on them. Unlike the translation for `fold_r_l`, this loop is efficient with linked implementations for lists because the elements are accessed in order.

4.5 Example: A simplified telemetry processor

A *telemetry processor* is a program that interprets telemetry data sent from a spacecraft. Satellite telemetry data is a sequence of samples, each containing a set of measures representing the status of the spacecraft systems [4]. Data is transmitted to a ground station in binary form, packed in fixed-size bit matrices called *master frames*.

The telemetry processor takes this coded data and produces *calibrated* data in engineering units

(e.g., meters, Watts) in floating point format. The calibration is done by extracting each measure from the master frame and evaluating a polynomial on its value. Besides, some measures require maximum and minimum limit check. The input to a telemetry processor is a master frame and a set of descriptions of measures. The output is a set of calibrated measures. These sets will be represented by lists.

The following PUC type declaration represents a master frame as a list of lists of bits.

```
type bit is (On, Off);
type row is list of bit;
type master_frame is list of row;
```

Each row in the master frame is a fixed-length bitstring considered to be divided into several bitstrings of various lengths representing measures. Measures are described by the following attributes: name, position in the master frame, and calibration parameters. The position in the master frame includes the row number and the first and last bit positions within the row. Calibration parameters for each measure are: coefficients for the polynomial, a check-range flag, and minimum and maximum values (used if the flag is true.)

```
type measure_description is
  record
    name      : string;
    row_num   : integer;
    first_bit : integer;
    last_bit  : integer;
    coeffs    : list of real;
    do_check  : boolean;
    min_value : real;
    max_value : real;
  end;
```

Calibrated measures are described by the name of the measure, the result of the polynomial evaluation, and a range check code that is either `Small`, `In_range`, `Large`, or `No_check`, depending on the range check of the value.

```
type range_code is (Small, In_range,
                    Large, No_check);
type calibrated_measure is
  record
    name  : string;
    value : real;
    range : range_code;
  end;
```

The main function of the specification is `calibrate_master`, which returns a list of calibrated measures given the master frame and a list of measure descriptions. It is defined `apply`'ing function `calibrate_measure` to each measure description (Figure 4.) Function `calibrate_measure` uses function `extract` to obtain the bitstring of the measure, function `to_number` to convert from binary to floating point, and function `poly_eval` to evaluate the corresponding polynomial. The range check code is computed with a nested `if` expression.

To generate an Ada program we need to provide translations for functions like `apply`, `curry`, etc. We also have to decide how each list will be implemented. Figure 5 is the main program in Ada. The list of measure descriptions in represented by an array because the number of measure decriptions is fixed for each satellite. The `apply` function is translated into a for-loop because the size of the list is a constant. The `curry` function is not explicitly translated: it is only a notation to provide the additional parameter within the loop. An explicit list of calibrated measures is built in local variable `result`, which is the returned value.

5 Conclusions

We have presented a specification language suitable for a specific class of software engineering environments using Ada. The main purpose of this language is to bridge the gap between very high level specifications and detailed algorithms and data structures, so we have attempted to define constructs that are similar to those in Ada, especially in data structures. On the other hand, the need to represent application level concepts has led us to include features like higher order functions and polymophism, to increase the reusability of the specifications.

We had to make several trade-offs in the design of PUC, because we wanted expressiveness, simplicity and similarity to Ada. We decided not to include algebraic data types and pattern matching (present in several functional languages); the more familiar concepts of variant record and case expression were used instead. We included parametric polymorphism, higher order functions, and Curring (i.e., partial parametrization of functions), but since these are advanced features, we wanted them to be explicit. Having these constructs complicated the process of translation from PUC to Ada, but they provided the abstraction mechanisms needed in a specification language. Hence, we studied semi-automatic methods of translation.

```
function calibrate_master (master  : master_frame;
                           measures: list of measure_description)
    return list of calibrated_measure is
begin
  apply( curry(calibrate_measure,master), measures )
end;

function calibrate_measure (master: master_frame; measure: measure_description)
    return calibrated_measure is

  bits  := extract(master, measure);
  value := poly_eval(measure.coeffs, to_number(bits));

  code  := if not (measure.do_check) then No_check
           elsif value < measure.min_value then Small
           elsif value > measure.max_value then Large
           else In_range end;
begin
  calibrated_measure'(measure.name, value, code)
end calibrate_measure;
```

Figure 4: Calibration functions of telemetry processor.

```
function calibrate_master (master   : in master_frame;
                           measures : in list_of_measure_description)
    return list_of_calibrated_measure is
  result: list_of_calibrated_measure;
begin
  for j in measures'range loop
    result(j) := calibrate_measure(master, measures(j));
  end loop;
  return result;
end;
```

Figure 5: Main function in Ada.

A specification language is not useful unless there is a software development method that will include its use. We have presented two non-exclusive methods: use the specifications as a prototype and transform the specification into an Ada program. Both approaches require the development of supporting tools. The language, along with its related methods and tools, will provide for a practical semi-formal software engineering environment. However, we have not tested extensively the use of functional languages in the specification of large scientific software in Ada.

Acknowledgements

Thanks to Sergio Cárdenas-García and Eduardo Ostertag for their helpful comments.

References

[1] Grady Booch. *Software Engineering with Ada*, The Benjamin/Cummins Publishing Company, Inc., Menlo Park, California, 1987.

[2] Luca Cardelli. Basic Polymorphic Type Checking, *Science of Computer Programming*, Vol. 8, 1987, pp. 147–172.

[3] William D. Clinger and Ralph L. London. A Role for Functional Languages in Specifications, *Proc. Fourth Int'l Workshop on Software Specification and Design*, CS Press, Los Alamitos, CA, 1987, pp. 2–7.

[4] General Electric Company, Valey Forge Space Division. *Software Specifications for Ada Re-Development Project (DSCS-III Application)*, Philadelphia, Pennsylvania, 17 June 1982.

[5] John V. Guttag, James J. Horning, and Jeanette M. Wing. The Larch Family of Specification Languages, *IEEE Software*, September 1985, pp. 24–36.

[6] Peter Henderson. Functional Programming, Formal Specification and Rapid Prototyping, *IEEE Trans. Soft. Eng.*, Vol. SE-12, No. 2, February 1986, pp. 241–250.

[7] C.A.R. Hoare. An Overview of Some Formal Methods for Program Design, *IEEE Computer*, September 1987, pp. 85–91.

[8] R. Kowalski. The Relation Between Logic Programming and Logic Specification, *Mathematical Logic and Programming Languages*, eds. C.A.R. Hoare and J.C. Shepherdson, Prentice-Hall, Englewood Cliffs, N.J., 1984, pp. 11–28.

[9] David C. Luckham and Freidrich W. von Henke. An Overview of Anna, a Specification Language for Ada, *IEEE Software*, March 1985, pp. 9–22.

[10] R. Morrison, A.L. Brown, R. Carrok, R.C.H. Connor, A. Dearle, and M.P. Atkinson. Polymorphism, Persistence and Software Re-Use in a Strongly Typed Object-Oriented Environment, *Software Engineering Journal*, November 1987, pp. 199–204.

[11] Norman Ramsey. Developing Formally Verified Ada Programs, *ACM SIGSOFT Engineering Notes*, Vol. 14, No. 3, May 1989, pp. 257–265.

[12] Pablo Straub and Marvin V. Zelkowitz. PUC: A Functional Specification Language for Ada. University of Maryland, Department of Computer Science, Technical Report CS-TR-2404, UMIACS-TR-90-17, February 1990.

[13] Pablo A. Straub. Bias and Design Decisions in Software Specifications. University of Maryland, Department of Computer Science, Technical Report CS-TR-2476, UMIACS-TR-90-72, May 1990.

[14] Robert B. Terrwilliger and Roy H. Campbell. An Early Report on ENCOMPASS, *10th International Conference on Software Engineering*, April 11–15, 1988, Singapore, IEEE Computer Society Press.

[15] Simon Thomson. Functional Programming: Executable Specifications and Program Transformation, *ACM SIGSOFT Engineering Notes*, Vol. 14, No. 3, May 1989, pp. 287–290.

[16] D.A. Turner. Functional Programs as Executable Specifications, *Mathematical Logic and Programming Languages*, eds. C.A.R. Hoare and J.C. Shepherdson, Prentice-Hall, Englewood Cliffs, N.J., 1984, pp. 29–54.

SOFTWARE RECLAMATION:
Improving Post-Development Reusability

John W. Bailey and Victor R. Basili

The University of Maryland Department of Computer Science
College Park, Maryland 20742

Abstract

This paper describes part of a multi-year study of software reuse being performed at the University of Maryland. The part of the study which is reported here explores techniques for the transformation of Ada programs which preserve function but which result in program components that are more independent, and presumably therefore, more reusable. Goals for the larger study include a precise specification of the transformation technique and its application in a large development organization. Expected results of the larger study, which are partially covered here, are the identification of reuse promoters and inhibitors both in the problem space and in the solution space, the development of a set of metrics which can be applied to both developing and completed software to reveal the degree of reusability which can be expected of that software, and the development of guidelines for both developers and reviewers of software which can help assure that the developed software will be as reusable as desired.

The advantages of transforming existing software into reusable components, rather than creating reusable components as an independent activity, include: 1) software development organizations often have an archive of previous projects which can yield reusable components, 2) developers of ongoing projects do not need to adjust to new and possibly unproven methods in an attempt to develop reusable components, so no risk or development overhead is introduced, 3) transformation work can be accomplished in parallel with line developments but be separately funded (this is particularly applicable when software is being developed for an outside customer who may not be willing to sustain the additional costs and risks of developing reusable code), 4) the resulting components are guaranteed to be relevant to the application area, and 5) the cost is low and controllable.

Introduction

Broadly defined, software reuse includes more than the repeated use of particular code modules. Other life cycle products such as specifications or test plans can be reused, software development processes such as verification techniques or cost modeling methods are reusable, and even intangible products such as ideas and experience contribute to the total picture of reuse [1,2]. Although process and tool reuse is common practice, life cycle product reuse is still in its infancy. Ultimately, reuse of early lifecycle products might provide the largest payoff. For the near term, however, gains can be realized and further work can be guided by understanding how software can be developed with a minimum of newly-generated source lines of code.

The work covered in this paper includes a feasibility study and some examples of generalizing, by transforming, software source code after it has been initially developed, in order to improve its reusability. The term software reclamation has been chosen for this activity since it does not amount to the development of but rather to the distillation of existing software. (Reclamation is defined in the dictionary as obtaining something from used products or restoring something to usefulness [3].) By exploring the ability to modify and generalize existing software, characterizations of that software can be expressed which relate to its reusability, which in turn is related to its maintainability and portability. This study includes applying these generalizations to several small example programs, to medium sized programs from different organizations, and to several fairly large programs from a single organization.

Earlier work has examined the principle of software reclamation through generic extraction with small examples. This has revealed the various levels of difficulty which are associated with generalizing various kinds of Ada dependencies. For example, it is easier to generalize a dependency that exists on encapsulated data than on visible data, and it is easier to generalize a dependency on a visible array type than on a visible record type. Following that work, some medium-sized examples of existing software were analyzed for potential generalization. The limited success of these efforts revealed additional guidelines for development as well as limitations of the technique. Summaries of this preceding work appear in the following sections.

Used as data for the current research is Ada software from the NASA Goddard Space Flight Center which was written over the past three years to perform spacecraft simulations. Three programs, each on the order of 100,000 (editor) lines, were studied. Software code reuse at NASA/GSFC has been practiced for many years, originally with Fortran developments, and more recently with Ada. Since transitioning to Ada, management has observed a steadily increasing amount of software reuse. One goal which is introduced here but which will be addressed in more detail in the larger study is the understanding of the nature of the reuse being practiced there and to examine the reasons for the improvement seen with Ada. Another goal of this as well as the larger study is to compare the guidelines derived from the examination of how different programs yield to or resist generalization. Several questions

are considered through this comparison, including the universality of guidelines derived from a single program and whether the effect of the application domain, or problem space, on software reusability can be distinguished from the effect of the implementation, or solution space.

Superficially, therefore, this paper describes a technique for generalizing existing Ada software through the use of the generic feature. However, the success and practicality of this technique is greatly affected by the style of the software being transformed. The examination of what characterizations of software are correlated with transformability has led to the derivation of software development and review guidelines. It appears that most, if not all, of the guidelines suggested by this examination are consistent with good programming practices as suggested by other studies.

The Basic Technique

By studying the dependencies among software elements at the code level, a determination can be made of the reusability of those elements in other contexts. For example, if a component of a program uses or depends upon another component, then it would not normally be reusable in another program where that other component was not also present. On the other hand, a component of a software program which does not depend on any other software can be used, in theory at least, in any arbitrary context. This study concentrates only on the theoretical reusability of a component of software, which is defined here as the amount of dependence that exists between that component and other software components. Thus, it is concerned only with the syntax of reusable software. It does not directly address issues of practical reusability, such as whether a reusable component is useful enough to encourage other developers to reuse it instead of redeveloping its function. The goal of the process is to identify and extract the essential functionality from a program so that this extracted essence is not dependent on external declarations, information, or other knowledge. Transformations are needed to derive such components from existing software systems since inter-component dependencies arise naturally from the customary design decomposition and implementation processes used for software development.

Ideal examples of reusable software code components can be defined as those which have no dependencies on other software. Short of complete independence, any dependencies which do exist provide a way of quantifying the reusability of the components. In other words, the reusability of a component can be thought of as inversely proportional to the amount of external dependence required by that component. However, some or all of that dependence may be removable through transformation by generalizing the component. A measure of a component's dependence on its externals which quantifies the difficulty of removing that dependence through transformation and generalization is slightly different from simply measuring the dependence directly, and is more specifically appropriate to this study. The amount of such transformation constitutes a useful indication of the effort to reuse a body of software.

Both the transformation effort and the degree of success with performing the transforms can vary from one example to the next. The identification of guidelines for developers and reviewers was made possible by observing what promoted or impeded the transformations. These guidelines can also help in the selection of reusable or transformable parts from existing software. Since dependencies among software components can typically be determined from the software design, many of the guidelines apply to the design phase of the life cycle, allowing earlier analysis of reusability and enabling possible corrective action to be taken before a design is implemented. Although the guidelines are written with respect to the development and reuse of systems written in the Ada language, since Ada is the medium for this study, most apply in general to software development in any language.

One measure of the extent of the transformation required is the number of lines of code that need to be added, altered, or deleted [4]. However, some modifications require new constructs to be added to the software while others merely require syntactic adjustments that could be performed automatically. For this reason, a more accurate measure weighs the changes by their difficulty. A component can contain dependencies on externals that are so intractable that removing them would mean also removing all of the useful functionality of the component. Such transformations are not cost-effective. In these cases, either the component in question must be reused in conjunction with one or more of the components on which it depends, or it cannot be generalized into an independently reusable one. Therefore, for any given component, there is a possibility that it contains some dependencies on externals which can be eliminated through transformation and also a possibility that it contains some dependencies which cannot be eliminated.

To guide the transformations, a model is used which distinguishes between software function and the declarations on which that function is performed. In an object-oriented program (for here, a program which uses data abstraction), data declarations and associated functionality are grouped into the same component. This component itself becomes the declaration of another object. This means the function / declaration distinction can be thought of as occurring on multiple levels. The internal data declarations of an object can be distinguished from the construction and access operations supplied to external users of the object, and the object as a whole can be distinguished from its external use which applies additional function (possibly establishing yet another, higher level object). The distinction between functions and objects is more obvious where a program is not object-oriented since declarations are not grouped with their associated functionality, but rather are established globally within the program.

At each level, declarations are seen as application-specific while the functions performed on them are seen as the potentially generalizable and reusable parts of a program. This may appear backwards initially, since data abstractions composed of both declarations and functions are often seen as reusable components. However, for consistency here, functions and declarations within a data abstraction are viewed as separable in the same way as functions which depend on declarations contained in external components are separable from those declarations. In use, the reusable, independent functional components are composed with application-specific declarations to form objects, which can further be composed with other independent functional components to implement an even larger portion of the overall program.

Figure 1 shows one way of representing this. All the ovals are objects. The dark ones are primitives which have predefined operations, such as integer or Boolean. The white ovals represent program-supplied functionality which is composed with their contained objects to form a higher level

object. The intent of the model is to distinguish this program-specific functionality and to attempt to represent it independently of the objects upon which it acts.

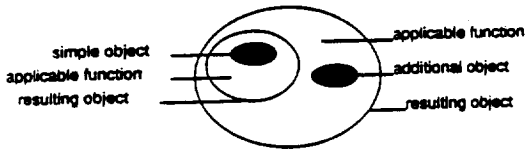


Figure 1.

Some Ada which might be represented as in the above figure might be:

```
package Counter is          -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
end Counter;

package body Counter is
  Count : Natural := 0;   -- simple object
  procedure Reset is
  begin
    Count := 0;
  end Reset;
  procedure Increment is
  begin
    Count := Count + 1;
  end Increment;
  function Current_Value return Natural is
  begin
    return Count;
  end Current_Value;
end Counter;


package Max_Count is        -- resulting object
  procedure Reset;          -- applicable function ...
  procedure Increment;
  function Current_Value return Natural;
  function Max return Natural;
end Max_Count;

with Counter;
package body Max_Count is
  Max_Val : Natural := 0;   -- additional object
  procedure Reset is
  begin
    Counter.Reset;
  end Reset;
  procedure Increment is
  begin
    Counter.Increment;
    if Max_Val < Counter.Current_Value then
      Max_Val := Counter.Current_Value;
    end if;
  end Increment;
  function Current_Value return Natural is
  begin
    return Counter.Current_Value;
  end Current_Value;
  function Max return Natural is
  begin
    return Max_Val;
  end Max;
end Max_Count;
```

In this example, the objects are properly encapsulated, though, they might not have been. If, for example, the simple objects were declared in separate components from their applicable functions, the result could have been the same (although the diagram might look different). In actual practice, Ada programs are developed with a combination of encapsulated object-operation groups as well as separately declared object-operation groups. Often the lowest levels are encapsulated while the higher level and larger objects tend to be separate from their applicable function. Perhaps in the ideal case, all objects would be encapsulated with their applied function since encapsulation usually makes the process of extracting the functionality at a later time easier. This, therefore, becomes one of the guidelines revealed by this model.

If the above example were transformed to separate the functionality from each object, the following set of components might be derived:

```
generic
  type Count_Object is (<>);
package Gen_Counter is     -- resulting object
  procedure Reset;         -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
end Gen_Counter;

package body Gen_Counter is
  Count : Count_Object       -- simple object
          := Count_Object'First;
  procedure Reset is
  begin
    Count := Count_Object'First;
  end Reset;
  procedure Increment is
  begin
    Count := Count_Object'Succ (Count);
  end Increment;
  function Current_Value return Count_Object is
  begin
    return Count;
  end Current_Value;
end Gen_Counter;

generic
  type Count_Object is (<>);
package Gen_Max_Count is   -- resulting object
  procedure Reset;         -- applicable function ...
  procedure Increment;
  function Current_Value return Count_Object;
  function Max return Count_Object;
end Gen_Max_Count;

with Gen_Counter;
package body Gen_Max_Count is
  Max_Val : Count_Object     -- additional object
            := Count_Object'First;
  package Counter is
              new Gen_Counter (Count_Object);
```

```
    procedure Reset is
    begin
      Counter.Reset;
    end Reset;
    procedure Increment is
    begin
      Counter.Increment;
      if Max_Val < Counter.Current_Value then
        Max_Val := Counter.Current_Value;
      end if;
    end Increment;
    function Current_Value return Natural is
    begin
      return Counter.Current_Value;
    end Current_Value;
    function Max return Natural is
    begin
      return Max_Val;
    end Max;
  end Gen_Max_Count;

  with Gen_Max_Count;
  procedure Max_Count_User is
    package Max_Count is
                    new Gen_Max_Count (Natural);
  begin
    Max_Count.Reset;
    Max_Count.Increment;
    ...
  end Max_Count_User;
```

Note that the end user obtains the same functionality that a user of Max_Count has, but the software now allows the primitive object Natural to be supplied externally to the algorithms that will apply to it. Further, the user could have obtained analogous functionality for any discrete type simply by pairing the general object with a different type (using a different generic instantiation).

This model is somewhat analogous to the one used in Smalltalk programming where objects are assembled from other objects plus programmer-supplied specifics. However, it is meant to apply more generally to Ada and other languages that do not have support for dynamic binding and full inheritance, features that are in general unavailable when strong static type checking is required. Instead, Ada offers the generic feature which can be used as shown here to partially offset the constraints imposed by static checking.

Applying this model to existing software means that any lines of code which represent reusable functionality must be parameterized with generic formal parameters in order to make them independent from their surrounding declaration space (if they are not already independent). Generics that are extracted by generalizing existing program units, through the removal of their dependence on external declarations, can then be offered as independently reusable components for other applications.

Unfortunately, declarative dependence is only one of the ways that a program unit can depend on its external environment. Removing the compiler-detectable declarative dependencies by producing a generic unit is no guarantee that the new unit will actually be independent. There can be dependencies on data values that are related to values in neighboring software, or even dependencies on protocols of operation that are followed at the point where a resource was originally used but which could be violated at a point of later reuse. (An example of this kind of dependency is described in the Measurement section.) To be complete, the transformation process would need to identify and remove these other types of dependence as well as the declarative dependence. Although guidelines have been identified by this study which can reduce the possibility for these other types of dependencies to enter a system, this work only concentrates on mechanisms to measure and remove declarative dependence.

More Examples

In a language with strong static type checking, such as Ada, any information exchanged between communicating program units must be of some type which is available to both units. Since Ada enforces name equivalence of types, where a type name and not just the underlying structure of a type introduces a new and distinct type, the declaration of the type used to pass information between units must be visible to both of those units. The user of a resource, therefore, is constrained to be in the scope of all type declarations used in the interface of that resource. In a language with a fixed set of types this is not a problem since all possible types will be globally available to both the resource and its users. However, in a language which allows user-declared types and enforces strong static type checking of those types, any inter-component communication with such types must be performed in the scope of those programmer-defined declarations. This means that the coupling between two communicating components increases from data coupling to external coupling (or from level two to level five on the traditional seven-point scale of Myers, where level one is the lowest level of coupling) [5].

Consider, for example, project-specific type declarations which often appear at low, commonly visible levels in a system. Resources which build upon those declarations can then be used in turn by higher level application-specific components. If a programmer attempts to reuse those intermediate-level resources in a new context, it is necessary to also reuse the low-level declarations on which they are built. This may not be acceptable, since combining several resources from different original contexts means that the set of low-level type declarations needed can be extensive and not generally compatible. This situation can occur whether or not data is encapsulated with its applicable function, but for clarity, and to contrast with the previous examples, it is shown here with the data and its operations declared separately.

For example, imagine that two existing programs each contain one of the following pairs of compilation units:

```
-- First program contains first pair:
package Vs_1 is
  type Variable_String is
    record
      Data   : String (1..80);
      Length : Natural;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_1;
```

```
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Pm_1;


-- Second program contains second pair:
package Vs_2 is
  type Variable_String is
    record
      Data  : String (1..250) := (others=>' ');
      Length: Natural := 0;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_2;

with Vs_2;
package Mm_2 is
  type Mail_Message is
    record
      From    : Vs_2.Variable_String;
      To      : Vs_2.Variable_String;
      Subject : Vs_2.Variable_String;
      Text    : Vs_2.Variable_String;
    end record;
  function Mail_Message_From_User
                        return Mail_Message;
end Mm_2;
```

Now, consider the programmer who is trying to reuse the above declarations in the same program. A reasonable way to combine the use of Mail_Messages with the use of Phone_Messages might seem to be as follows:

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
            Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
            Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From := Name;    -- illegal
end User;
```

This will fail to compile, however, since the types Vs_1.Variable_String and Vs_2.Variable_String are distinct and therefore values of one are not assignable to objects of the other (the value of Name is of type Vs_1.Variable_String and the record component Mm.From is of type Vs_2.Variable_String).

In the above example, note that the variable string types were left visible rather than made private to make it seem even more plausible for a programmer to expect that, at least logically, the assignment attempted is reasonable. However, the incompatibility between the underlying type declarations used by Mail_Message and Phone_Message becomes a problem. One solution might be to use type conversion. However, employing type conversion between elements of the low level variable string types destroys the abstraction for the higher-level units. For instance, the user procedure above could be written as shown below, but exposing the detail of the implementation of the variable strings represents a poor, and possibly dangerous, programming style.

```
with Vs_1;
with Pm_1;
with Mm_2;
procedure Type_Conversion_User is
  Name : Vs_1.Variable_String;
  Pm : Pm_1.Phone_Message :=
            Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
            Mm_2.Mail_Message_From_User;
begin
  Name := Pm.To;
  Mm.From.Data (1..80) := Name.Data;
  Mm.From.Length := Name.Length;
end Type_Conversion_User;
```

Notice that we had to be careful to avoid a constraint error at the point of the data assignment. This is one example of how attempts to combine the use of resources which rely on different context declarations is difficult in Ada.

Static type checking, therefore, is a mixed blessing. It prevents many errors from entering a software system which might not otherwise be detected until run time. However, it limits the possible reuse of a module if a specific declaration environment must also be reused. Not only must the reused module be in the scope of those declarations, but so must its users. Further, those users are forced to communicate with that module using the shared external types rather than their own, making the resource master over its users instead of the other way around. The set of types which facilitates communication among the components of a program, therefore, ultimately prevents most, if not all, of the developed algorithms from being easily used in any other program.

This study refers to declarations such as those of the above variable string types as *contexts*, and to components which build upon those declarations and which are in turn used by other components, such as the above Mail_Message and Phone_Message packages, as *resources*. Components which depend on resources are referred to as *users*. The above illustrates the general case of a context-resource-user relationship. It is possible for a component to be both a resource at one level and also a context for a still higher-level resource. The dependencies among these three basic categories of components can be illustrated with a directed graph. Figure 2 shows a graph of the kind of dependency illustrated in the example above.

A resource does not always need full type information about the data it must access in order to accomplish its task. In the above examples, it would be possible for the Mail and Phone message resources to implement their functions via the functions exported from the variable string packages without any further information about the structures of those lower level variable string types. Sometimes, even less knowledge

of the structure or functionality of the types being manipulated by a resource is required by that resource for it to accomplish its function.

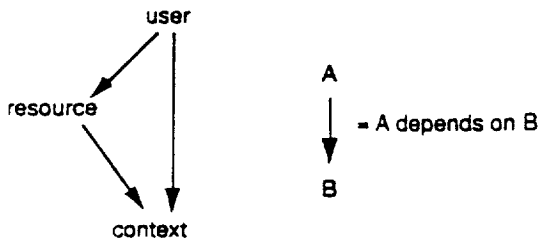


Figure 2.

A common example of a situation where a resource needs no structural or operational information about the objects it manipulates is a simple data base which stores and retrieves data but which does not take advantage of the information contained by that data. It is possible to write or transform such a resource so that the context it requires (i.e., the type of the object to be stored and retrieved) is supplied by the users of that resource. Then, only the essential work of the module needs to remain. This "essence only" principle is the key to the transformations sought. Only the purpose of a module remains, with any details needed to produce the executing code, such as actual type declarations or specific operations on those types, being provided later by the users of the resource. In languages such as Smalltalk which allow dynamic binding, this information is bound at run time. In Ada, where the compiler is obligated to perform all type checking, generics are bound at compilation time, eliminating a major source of run time errors caused by attempting to perform inappropriate operations on an object. Even though they are statically checked, however, Ada generics can often allow a resource to be written so as to free it from depending upon external type definitions.

Using the following arbitrary type declaration and a simplified data store package, one possible transformation is illustrated. First the example is shown before any transformation is applied:

```
-- context:
package Decls is
   type Typ is ...  -- anything but limited private
end Decls;

-- resource:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;
```

```
package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;
```

The above resource can be transformed into the following one which has no dependencies on external declarations:

```
-- generalized resource:
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Typ) is
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

Note that, by naming the generic formal parameter appropriately, none of the identifiers in the code needed to change, and the expanded names were merely shortened to their simple names. This minimizes the handling required to perform the transformation (although automating the process would make this an unimportant issue). This transformation required the removal of the context clause, the addition of two lines (the generic part) and the shortening of the expanded names. The modification required to convert the package to a theoretically independent one constitutes a reusability measure. A user of the resource in the original form would need to add the following declaration in order to obtain an appropriate instance of the resource:

```
package Store is new General_Store (Decls.Typ);
```

Formal rules for counting program changes have already been proposed and validated [4], and adaptations of these counting rules (such as using a lower handling value for shortening expanded names and a higher one for adding generic formals) are being considered as part of this work.

The earlier example with the variable string types can also be generalized to remove the dependencies between the mail and phone message packages (resources) and the variable string packages (contexts). For example, ignoring the implementations (bodies) of the resources, the following would functionally be equivalent to those examples:

```
-- Contexts, as before:
package Vs_1 is
  type Variable_String is
    record
      Data : String (1..80);
      Len  : Natural;
    end record;
  function Variable_String_From_User
                        return Variable_String;
end Vs_1;

package Vs_2 is
  type Variable_String is
    record
      Data : String (1..250) := (others=>' ');
      Len  : Natural := 0;
    end record;
  function Variable_String_From_User
                       return Variable_String;
end Vs_2;
```

```
-- Resources, which no longer depend upon
-- the above context declarations:
generic
  type Component is private;
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;

generic
  type Component is private;
package Gen_Mm_2 is
  type Mail_Message is
    record
      From : Component;
      To   : Component;
      Subj : Component;
      Text : Component;
    end record;
  function Mail_Message_From_User
                          return Mail_Message;
end Gen_Mm_2;
```

Now, the programmer who is trying to reuse the above declarations by combining the use of Mail_Messages with the use of Phone_Messages has another option. Instead of trying to combine both contexts, just one can be chosen (in this case, Vs_2):

```
with Vs_2;
with Gen_Pm_1;
with Gen_Mm_2;
procedure User is
  package Pm_1 is new
            Gen_Pm_1 (Vs_2.Variable_String);
  package Mm_2 is new
            Gen_Mm_2 (Vs_2.Variable_String);
  Name : Vs_2.Variable_String;
  Pm : Pm_1.Phone_Message :=
                Pm_1.Phone_Message_From_User;
  Mm : Mm_2.Mail_Message :=
                Mm_2.Mail_Message_From_User;
begin
  Name := Mm.From;
  Pm.To := Name;    -- now OK
end User;
```

An additional complexity is required for this example. The resources must be able to obtain component type values from which to construct mail and phone messages. Although this is not obvious from the specifications only, it can be assumed that such functionality must be available in the body. This can be done by adding a generic formal function parameter to the generic parts, requiring the user to supply an additional parameter to the instantiations as well:

```
generic
  type Component is private;
  with function Component_From_User
          return Component;
                        -- parameterless for simplicity
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;
      To   : Component;
      Data : Component;
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Gen_Pm_1;
```

Although the above examples show the context, the resource, and the user as library level units, declaration dependence can occur, and transformations can be applied, in situations where the three components are nested. For example, the resource and user can be co-resident in a declarative area, or the user can contain the resource or vice versa.

This reiterates the earlier claim that, at least for the purpose of this model, it does not matter if the data is encapsulated with its applicable function, it just makes it easier to find if it is. In the programs studied, the lowest level data types, which were often properly encapsulated with their immediately available operations, were used to construct higher level resources specific to the problem being solved. It was unusual for those resources to be written with the same level of encapsulation and independence as the lower level types, and this resulted in the kind of context-resource-user dependencies illustrated above.

For example, in the case of the generalized simple data base, the functionality of the data appears in the resource while the declaration of it appears in the context. The only place where the higher-level object comes into existence is inside the user component, at the point where the instantiation is declared. If desired, an additional transformation can be applied to rectify this problem of the apparent separation of the object from its operations. Instead of leaving the instantiation of the new generic resource up to the client

software, an intermediate package can be created which combines the visibility of the context declarations with instantiations of the generic resource. This package, then, becomes the direct resource for the client software, introducing a layer of abstraction that was not present in the original (non-general) structure.

For example, the following transformation to the second example above combines the resource General_Store with the context of choice, type Typ from package Decls. The declaration of the package Object performs this service.

```
generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is
  subtype Typ is Decls.Typ;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ)
          renames Store.Put;
  procedure Get_Last (Obj : out Typ)
          renames Store.Get_Last;
end Object;

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Note that no body for package Object is required using the style shown. If it were preferable to leave the implementation of Object flexible, so that users would not need to be recompiled if the context used by the instantiation were to change, the context clauses and the instantiation could be made to appear only in the body of Object. An alternate, admittedly more complex, example is shown here which accomplishes this flexibility:

```
package Object is
  type Typ is  private;
  function Initial return Typ;
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : in Typ);
private
  type Designated;
  type Typ is access Designated;
end Object;

with Decls;
with General_Store;
package body Object is
  type Typ is new Decls.Typ;
  function Initial return Typ is
  begin
    return new Designated;
  end Initial;
  package Store is new General_Store (Typ);
  procedure Put (Obj : in Typ) is
  begin
     Store.Put (Obj.all);
  end Put;
  procedure Get_Last (Obj : in Typ) is
  begin
     Store.Get_Last (Obj.all);
  end Get_Last;
end Object;
```

In the alternate example, note that the parameter mode for the Get_Last procedure needed to be changed to allow the reading of the designated object of the actual access parameter. Also, a simple initialization function was supplied to provide the client with a way of passing a non-null access object to the Put and Get_Last procedures. Normally, there would already be initialization and constructor operations, so this additional operation would not be needed. The advantage of this alternative is that the implementation of the type and operations can change without disturbing the client software. However, the first alternative could be changed in a compilation-compatible way, such that any client software would need recompilation but no modification.

It is also possible to provide just an instantiation as a library unit by itself, but this requires the user to acquire independently the visibility to the same context as that instantiation. This solution results in the reconstruction of the original situation, where the instantiation becomes the resource dependent on a context, and the user depends on both. The important difference, however, is that now the resource (the instantiation) is not viewed as a reusable component. It becomes application-specific and can be routinely (potentially automatically) generated from both the generalized reusable resource and the context of choice, while the generic from which the instantiation is produced remains the independent, reusable component. The advantage of this structure lies in the abstraction provided for the user component which is insulated from the complexities of the instantiation of the reusable generic. Since the result is similar to the initial architecture, the overall software architecture can be preserved while utilizing generic resources. The following illustrates this.

```
package Decls is
  type Typ is ...
end Decls;

generic
  type Typ is private;
package General_Store is
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);
```

```
with Decls;
with Object;
procedure Client is
  Item : Decls.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

By modifying the generic resource to "pass through" the generic formal types, the user's reliance on the context can be removed:

```
generic
  type Gen_Typ is private;
package General_Store is
  subtype Typ is Gen_Typ;   -- pass the type through
  procedure Put (Obj : in Typ);
  procedure Get_Last (Obj : out Typ);
end General_Store;

package Decls is
  type Typ is ...
end Decls;

with Decls;
with General_Store;
package Object is new General_Store(Decls.Typ);

with Object;
procedure Client is
  Item : Object.Typ;
begin
  Object.Put (Item);
  Object.Get_Last (Item);
end Client;
```

Measurement

In the above examples, the context components were never modified. Resource components were modified to eliminate their dependence on context components. User components were modified in order to maintain their functionality given the now general resource components, typically by defining generic actual parameter objects and adding an instantiation. In the case of the encapsulated instantiations, an intermediate component was introduced to free the user component of the complexity of the instantiation. It is the ease or difficulty of modifying the resource components that is of primary interest here, and the measurement of this modification effort constitutes a measurement of the reusability of the components. The usability of the generalized resources is also of interest, since some may be difficult to instantiate.

Considering the above examples again, the simple data base resource Store required the removal of the context clause and the creation of a generic part (these being typical modifications for almost all transformations of this kind). In addition, the formal parameter types for the two subprograms were changed to the generic formal private type, causing a change to both the subprogram specification and body. No further changes were required.

```
-- original:
with Decls;
package Store is
  procedure Put (Obj : in Decls.Typ);
  procedure Get_Last (Obj : out Decls.Typ);
end Store;

package body Store is
  Local : Decls.Typ;
  procedure Put (Obj : in Decls.Typ) is
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj : out Decls.Typ) is
  begin
    Obj := Local;
  end Get_Last;
end Store;

-- transformed:
generic
  type Typ is private;                          -- change
package General_Store is
  procedure Put (Obj : in Typ);                 -- change
  procedure Get_Last (Obj: out Typ);            -- change
end General_Store;

package body General_Store is
  Local : Typ;
  procedure Put (Obj : in Typ) is               -- change
  begin
    Local := Obj;
  end Put;
  procedure Get_Last (Obj: out Typ) is          -- change
  begin
    Obj := Local;
  end Get_Last;
end General_Store;
```

The Phone_Message and Mail_Message resources required the deletion of the context clause, the addition of a generic part consisting of a formal private type parameter and a formal subprogram parameter, and the replacement of three occurrences (or four, in the case of Mail_Message) of the type mark Vs_1.Variable_String with the generic formal type Component.

```
-- original:
with Vs_1;
package Pm_1 is
  type Phone_Message is
    record
      From : Vs_1.Variable_String;
      To   : Vs_1.Variable_String;
      Data : Vs_1.Variable_String;
    end record;
  function Phone_Message_From_User
                        return Phone_Message;
end Pm_1;

-- transformed:
generic
  type Component is private;        -- change
  with function Component_From_User
                return Component;   -- change
```

```
package Gen_Pm_1 is
  type Phone_Message is
    record
      From : Component;          -- change
      To   : Component;          -- change
      Data : Component;          -- change
    end record;
  function Phone_Message_From_User
                          return Phone_Message;
end Gen_Pm_1;
```

Generalizing the bodies of Gen_Pm_1 and Gen_Mm_2 would involve replacing any calls to the Variable_String_From_User functions with calls to the generic formal Component_From_User function. In the case of the simple bodies shown before, this would require three and four simple substitutions, for Gen_Pm_1 and Gen_Mm_2, respectively.

In addition to measuring the reusability of a unit by the amount of transformation required to maximize its independence, reusability can also be gauged by the amount of residual dependency on other units which cannot be eliminated, or which is unreasonably difficult to eliminate, by any of the proposed transformations. For any given unit, therefore, two values can be obtained. The first reveals the number of program changes which would be required to perform any applicable transformations. The second indicates the amount of dependence which would remain in the unit even after it was transformed. The original units in the examples above would score high on the first scale since the handling required for its conversion was negligible, implying that its reusability was already good (i.e., it was already independent or was easy to make independent of external declarations). After the transformation, there remain no latent dependencies, so the transformed generic would receive a perfect reusability score.

Note that the object of any reusability measurement, and therefore, of any transformations, need not be a single Ada unit. If a set of library units were intended to be reused together then the metrics as well as the transformations could be applied to the entire set. Whereas there might be substantial interdependence among the units within the set, it still might be possible to eliminate all dependencies on external declarations.

In the above examples, one reason that the transformation was trivial was that the only operation performed on objects of the external type was assignment (except for the mail and phone message examples). Therefore, it was possible to replace direct visibility to the external type definition with a generic formal private type. A second example illustrates a slightly more difficult transformation which includes more assumptions about the externally declared type. In the following example, indexing and component assignment are used by the resource.

Before transformation:

```
-- context
package Arr is
  type Item_Array is
        array (Integer range <>) of Natural;
end Arr;

-- resource
with Arr;
procedure Clear (Item : out Arr.Item_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Clear;

-- user
with Arr, Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
begin
  Clear (X);
end Client;
```

After transformation:

```
-- context (same)
package Arr is
  type Item_Array is
        array (Integer range <>) of Natural;
end Arr;

-- generalized resource
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
         array (Index range <>) of Component;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := 0;
  end loop;
end Gen_Clear;

-- user
with Arr, Gen_Clear;
procedure Client is
  X : Arr.Item_Array (1..10);
  procedure Clear is new Gen_Clear
                          (Natural,
                           Integer,
                           Arr.Item_Array);
begin
  Clear (X);
end Client;
```

The above transformation removes compilation dependencies, and allows the generic procedure to describe its essential function without the visibility of external declarations. As before, an intermediate object could be created to free the user procedure from the chore of instantiating a Clear procedure, which requires visibility to both the context and the resource. However, it also illustrates an important additional kind of dependence which can exist between a resource and its users, namely information dependence.

In the previous example, the literal value 0 is a clue to the presence of information that is not general. Therefore, the following would be an improvement over the transformation shown above:

```
generic
   type Component is range <>;
   type Index is range <>;
   type Gen_Array is
         array (Index range <>) of Component;
   Init_Val : Component := Component'First;
procedure Gen_Clear (Item : out Gen_Array);
procedure Gen_Clear (Item : out Gen_Array) is
begin
  for I in Item'Range loop
    Item (I) := Init_Val;
  end loop;
end Gen_Clear;
```

Note that the last transformation allows the user to supply an initial value, but also provides the lowest value of the component type as a default. An additional refinement would be to make the component type private which would mean that Init_Val could not have a default value. Information dependencies such as the one illustrated here are harder to detect than compilation dependencies. The appearance of literal values in a resource is often an indication of an information dependence.

A third form of dependence, called protocol dependence, has also been identified. This occurs when the user of a resource must obey certain rules to ensure that the resource behaves properly. For example, a stack which is used to buffer information between other users could be implemented in a not-so-abstract fashion by exposing the stack array and top pointer directly. In this case, all users of the stack must follow the same protocol of decrementing the pointer before popping and incrementing after pushing, and not the other way around. Beyond the recognition of it, no additional treatment of this form of dependence between components will appear in this study.

Formalizing the Transformations

The following is a formalization of the objectives of transformations which are needed to remove declaration dependence.

1. Let P represent a program unit.

2. Let D represent the set of n object declarations, $d_1 .. d_n$, directly referenced by P such that d_i is of a type declared externally to P.

3. Let $O_1 .. O_n$ be sets of operations where O_i is the set of operations applied to d_i inside P.

4. P is completely transformable if each operation in each of the sets, $O_1 .. O_n$ can be replaced with a predefined or generic formal operation.

The earlier example transformation is reviewed in the context of these definitions:

1. Let P represent a program unit.
   P = procedure Clear (Item : out Arr.Item_Array) is ...

2. Let D represent the set of n object declarations, $d_1 .. d_n$, directly referenced by P such that d_i is of a type declared externally to P.
   D = { Arr.Item_Array }.

3. Let $O_1 .. O_n$ be sets of operations where O_i is the set of operations applied to d_i inside P.
   O_1 =
{ indexing by integers, integer assignment to components }

4. P is completely transformable if each operation in each of the sets, $O_1 .. O_n$ can be replaced with a predefined or generic formal operation.

Indexing can be obtained through a generic formal array type. Although no constraining operation was used, the formal type could be either constrained or unconstrained since the only declared object is a formal subprogram parameter. Since component assignment is required, the component type must not be limited. Therefore, the following generic formal parts are possible:

```
type Component is range <>;
type Index is range <>;
```

followed by either:

```
type Gen_Array is array (Index) of Component;
```

or:

```
type Gen_Array is
        array (Index range <>) of Component;
```

Notice that some operations can be replaced with generic formal operations more easily than others. For example, direct access of array structures can generally be replaced by making the array type a generic formal type. However, direct access into record structures (using "dot" notation) complicates transformations since this operation must be replaced with a user-defined access function.

Application to External Software

Medium-Sized Projects

To test the feasibility of the transformations proposed, a 6,000-line Ada program written by seven professional programmers was examined for reuse transformation possibilities. The program consisted of six library units, ranging in size from 20 to 2,400 lines. Of the 30 theoretically possible dependencies that could exist among these units, ten were required. Four transformations of the sort described above were made to three of the units. These required an additional 44 lines of code (less than a 1% increase) and reduced the number of dependencies from ten to five, which is the minimum possible with six units. Using one possible program change definition, each transformation required between two and six changes.

A fifth modification was made to detach a nested unit from its parent. This required the addition of 15 lines and resulted in a total of seven units with the minimum six dependencies. Next, two other functions were made independent of the other units. Unlike the previous transformations which were targeted for later reuse, however, these transformations resulted in a net reduction in code since the resulting components were reused at multiple points within this program. Substantial information dependency which would have impaired actual reuse was identified but remained within the units, however.

A second medium-sized project was studied which exhibited such a high degree of mutual dependence between pairs of library units that, instead of selecting smaller units for generalizations, the question of non-hierarchical dependence was studied at a system level. The general conclusion from this was that loops in the dependency structure (where, for example, package A is referenced from package body B and package B is referenced from package body A) make generalization of those components difficult. The program was instead analyzed for possible restructuring to remove as much of the bi-directional dependence as practical. This was partially successful and suggests that this sort of redesign might appropriately precede other reuse analyses.

The NASA Projects

Currently, the research project is examining several spacecraft flight simulation programs from the NASA Goddard Space Flight Center. These programs are each more than 100,000 editor lines of Ada. They have been developed by an organization that originally developed such simulators in Fortran and has been transitioning to the use of Ada over the past several years. Because all the programs are in the same application domain and were developed by the same organization there is considerable opportunity for reuse. In the past, the development organization reported the ability to reuse about 20% of earlier programs when a new program was being developed in Fortran. However, since becoming familiar with Ada, the same organization is now reporting a 70% reuse rate, or better.

After gaining an understanding of the nature of the reuse accomplished in Fortran and later in Ada, and how similar or different reuse in the two languages was, we would like to test several theories about why the Ada reuse has been so much greater. We already know that the reuse is accomplished by modifying earlier components as required, and not, in general, by using existing software verbatim. Because of this reuse mode, one theory we will be testing is that the Ada programs are more reusable simply because they are more understandable.

For the current study, the programs were studied to reveal opportunities to extract generic components which, had they been available when the programs were being developed originally, could have been reused without modification. There is an additional advantage to working with this data, however, since, as mentioned above, the several programs already exhibit significant functional similarities which can be studied for possible generalization. In other words, whereas the initial discussion of generic extraction has focussed on attempts to completely free the essential function of a component from its static declaration context, this data gives examples of similar components in two or more different program contexts and therefore allows us to study the possibility of freeing a component from only its program-specific context and not from any context which remains constant across programs.

This gives rise to the notion of domain-specific generic extraction as opposed to domain-independent generic extraction. Given the problems associated with extracting a completely general component, as examined earlier, a case can be made to generalize away only some of the dependence, leaving the rest in place. The additional problem, then, becomes how to determine what dependence is permissible and what should be removed. The permissible dependence would be common across projects in a certain domain, and would therefore be domain-specific while the dependence to be removed would be the problem-specific context. When reused, then, these components would have their problem-specific context supplied as generic actual parameters.

This is currently a largely manual task, since the programs must be compared to find corresponding functionality and then examined to determine the intersection of that functionality. Interestingly, on the last project the developers themselves have also been devising generic components which are instantiated only one time within that program. This implied to us that some effort was being spent to make components which might be reusable with no, or perhaps only very little, modification in the next project. We have confirmed with the developers that this is in fact the case. By comparing the results of our generalizations with those done by the developers, we find that ours have much more complex generic parts but correspondingly much less dependence on other software. This is a reasonable result, since the developers already have some idea about the context for each reuse of a given generic; what aspects of that context are likely to change from project to project and what aspects are expected to remain constant across several programs. The program-specific context, only, appears in the generic parts of the generics written by the developers, while our generalizations have generic parts which contain declarations of types and operations which apparently do not need to change as long as the problem domain remains the same. In other words, when our generic parts are devised by analyzing only a single instance of a component, we cannot distinguish between program-specific and domain-specific generalizations.

One interesting question we would like to answer is whether we can derive the generic part that makes the most sense within this domain by comparing similar components from different programs and generalizing only on their differences, leaving the software in the intersection of the components unchanged. In this way, a component would be derived which would not be completely independent but, like the developer-written generics, would be sufficiently independent for reuse in the domain. Then, a comparison with the generics developed within the organization would be revealing. If the generics are similar then our process might be useful on other parts of the software that have not yet been generalized by the developers. However, if they differ greatly, it would be useful to characterize that difference and

understand what additional knowledge must be used in generalizing the repeated software. Unfortunately, there is not enough reuse of the developer's generics yet to make this final comparison but a project is currently in progress which should supply some of this data.

The following example illustrates the complexity of the generic parts which were required to completely isolate a typical unit from its context. Here, the procedure Check_Header was removed from a package body and generalized to be able to stand alone as a library level generic procedure.

```
generic
  type Time is private;
  type Duration is digits <>;
  with function Enable return Boolean;
  type Hd_Rec_Type is private;
  with procedure Set_Start
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Start
       (H : Hd_Rec_Type) return Duration;
  with procedure Set_Stop
       (H : in out Hd_Rec_Type; To : Duration);
  with function Get_Stop
       (H : Hd_Rec_Type) return Duration;
  type Real is digits <>;
  with function Get_Att_Int
       (H : Hd_Rec_Type) return Real;
  with function Conv_Time
       (D_Float : Duration) return Duration;
  Header_Rec : in out Hd_Rec_Type;
  Goesim_Time_Step : in out Duration;
  with function Seconds_Since_1957
       (T : in Time) return Duration;
  with procedure Debug_Write (Output : String);
  with procedure Debug_End_Line;
  type Direct_File_Type is limited private;
  with procedure Direct_Read
       (File : Direct_File_Type);
  with procedure Direct_Get
       (File : in  Direct_File_Type;
        Item : out Hd_Rec_Type);
  with function Image_Of_Base_10
       (Item : Duration) return String;
  with procedure Header_Data_Error;
procedure Check_Header_Generic
   (Simulation_Start_Time : in Time;
    Simulation_Stop_Time  : in Time;
    Simulation_Time_Step  : in Duration;
    History_File : in out Direct_File_Type);
```

The instantiation of this generic part is correspondingly complex:

```
procedure Check_Header_Instance is new
                          Check_Header_Generic
   (Abstract_Calendar.Time,
    Abstract_Calendar.Duration,
    Debug_Enable,
    Attitude_History_Types.Header_Record,
    Set_Start,
    Get_Start,
    Set_Stop,
    Get_Stop,
    Utilities.Read,
    Get_Att_Hist_Out_Int,
    Converted_Time,
    History_Data.Header_Rec,
    History_Data.Goesim_Time_Step,
    Timer.Seconds_Since_1957,
    Error_Collector.Write,
    Error_Collector.End_Line,
    Direct_Mixed_Io.File_Type,
    Direct_Mixed_Io.Read,
    Get_From_Buffer,
    Image_Of_Base_10,
    Raise_Header_Data_Error);
```

In contrast, a typical generic part on a unit which was developed and delivered as part of the most recent completed project by the developers themselves is shown here:

```
with Css_Types;
generic
  Number_Of_Sensors : Natural :=
       Css_Types.Number_Of_Sensors;
  with function Initialize_Sensor
     return Css_Types.Css_Database_Type is <>;
package Generic_Coarse_Sun_Sensor is
  ...
```

Note that by allowing the visibility of Css_Types, the generic part was simplified. Being unfamiliar with the domain, had we attempted to generalize Coarse_Sun_Sensor by examining only the non-generic version of a corresponding component in another program we would not be able to tell whether the dependence on Css_Types was program-specific or domain-specific. Here, however, the developer leads us to believe that Css_Types is domain-specific while the number of sensors and sensor initialization is program specific.

Guidelines

The manual application of the principles and techniques of generic transformation and extraction has revealed several interesting and intuitively reasonable guidelines relative to the creation and reuse of Ada software. In general, these guidelines appear to be applicable to programs of any size. However, the last guideline in the list, concerning program structure, was the most obvious when dealing with medium to large programs.

- Avoid direct access into record components except in the same declarative region as the record type declaration.

Since there is no generic formal record type in Ada (without dynamic binding such a feature would be impractical) there is no straightforward way to replace record component access with a generic operation. Instead, user-supplied access functions are needed to access the components and the type must be passed as a private type. This is unlike array types for which there are two generic formal types (constrained and unconstrained). This supports the findings of others which assert that direct referencing of non-local record components adversely affects maintainability [6].

- Minimize non-local access to array components.

Although not as difficult in general as removing dependence

on a record type, removing dependence on an array type can be cumbersome.

• Keep direct access to data structures local to their declarations.

This is a stronger conclusion than the previous two, and reinforces the philosophy of using abstract data types in all situations where a data type is available outside its local declarative region. Encapsulated types are far easier to separate as resources than globally declared types since the operations are localized and contained.

• Avoid the use of literal values except as constant value assignments.

Information dependence is almost always associated with the use of a literal value in one unit of software that has some hidden relationship to a literal value in a different unit. If a unit is generalized and extracted for reuse but contains a literal value which indicates a dependence on some assumption about its original context, that unit can fail in unpredictable ways when reused. Conventional wisdom applies here, and it might be reasonable to relax the restriction to allow the use of 0 and 1. However, experience with a considerable amount of software which makes the erroneous assumption that the first index of any string is 1 has shown that even this can lead to problems.

• Avoid mingling resources with application specific contexts.

Although the purpose of the transformations is to separate resources from application specific software regardless of the program structure, certain styles of programming result in programs which can be transformed more easily and completely. By staying conscious of the ultimate goal of separating reusable function from application declarations, whether or not the functionality is initially programmed to be generic, programmers can simplify the eventual transformation of the code.

• Keep interfaces abstract.

Protocol dependencies arise from the exportation of implementation details that should not be present in the interface to a resource. Such an interface is vulnerable because it assumes a usage protocol which does not have to be followed by its users. The bad stack example illustrates what can happen when a resource interface requires the use of implementation details, however even resources with an appropriately abstract interface can export unwanted additional detail which can lead to protocol dependence.

• Avoid direct reference to package Standard.Float

Even when used to define other floating point types, direct reference to Float establishes an implementation dependence that does not occur with anonymous floating point declarations. Especially dangerous is a direct reference to Standard.Long_Float, Standard.Long_Integer, etc., since they may not even compile on different implementations. Some care must also be taken with Integer, Positive, and Natural, though in general they were not associated with as much dependence as Float. Note that fixed point types in Ada are constructed as needed by the compiler. Perhaps the same philosophy should have been adopted for Float and Integer. Reference to Character and Boolean is not a problem since they are the same on all implementations.

• Avoid the use of 'Address

Even though it is not necessary to be in the scope of package System to use this attribute, it sets up a dependency on System.Address that makes the software non-portable. If this attribute is needed for some low-level programming than it should be encapsulated and never be exposed in the interface to that level.

• Consider the inter-component dependence of a design

By understanding how functionally-equivalent programs can vary in their degree of inter-component dependence, designers and developers can make decisions about how much dependence will be permitted in an evolving system, and how much effort will be applied to limit that dependence. For system developments which are expected to yield reusable components directly, a decision can be made to minimize dependencies from the outset. For developments which are not able to make such an investment in reusability, a decision can be made to allow certain kinds of dependencies to occur. In particular, dependencies which are removable through subsequent transformation might be allowed while those that would be too difficult to remove later might be avoided. A particularly cumbersome type of dependence occurs when two library units reference each other, either directly or indirectly. This should be avoided if at all possible. By making structural decisions explicitly, surprises can be avoided which might otherwise result in unwanted limitations of the developed software.

Acknowledgements

This work was supported in part by the U.S. Army Institute for Research in Management Information and Computer Science under grant AIRMICS-01-4-33267, and NASA under grant NSG-5123. Some of the software analysis was performed using a Rational computer at Rational's eastern regional office in Calverton, Maryland.

References

1. Basili, V. R. and Rombach, H. D. Software Reuse: A Framework. In preparation.

2. Basili, V. R. and Rombach, H. D. The TAME Project: Towards Improvement- Oriented Software Environments. IEEE Transactions on Software Engineering, SE-14, June 1988.

3. Funk & Wagnalls. Standard College Dictionary, New York, 1977.

4. Myers, G. Composite/Structured Design, Van Nostrand Reinhold, New York, 1978.

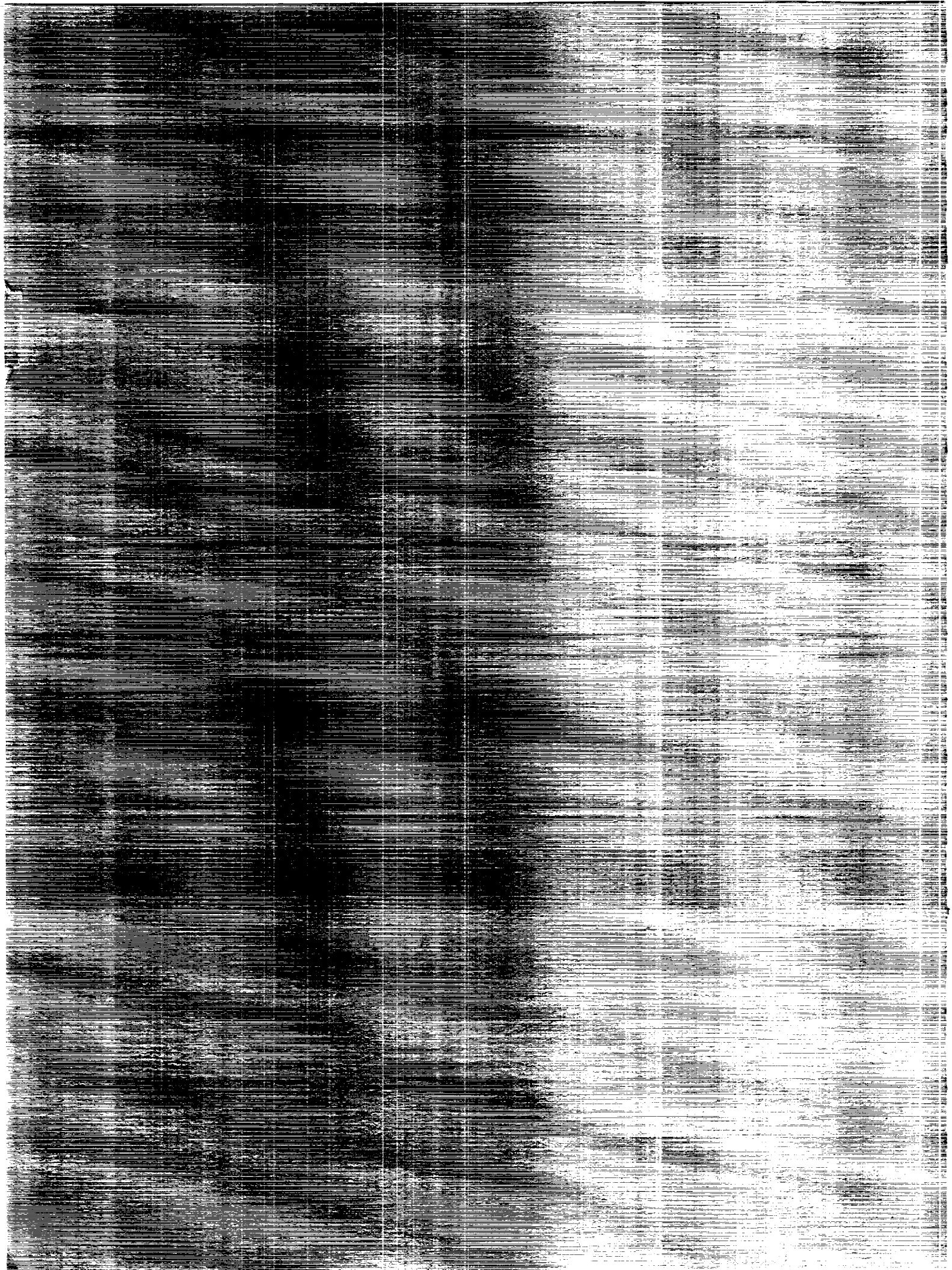
5. Dunsmore, H.E. and Gannon, J.D. Experimental Investigation of Programming Complexity. In Proceedings ACM/NBS 16th Annual Tech. Symposium: Systems and Software, Washington D.C., June 1977.

6. Gannon, J.D., Katz, E. and Basili, V.R. Characterizing Ada Programs: Packages. In Proceedings Workshop on Software Performance, Los Alamos National Laboratory, Los Alamos, New Mexico, August 1983.

John W. Bailey is a Ph.D. candidate at the University of Maryland Computer Science Department. He is a part-time employee of Rational and has been consulting and teaching in the areas of Ada and software measurement for seven years. In addition to Ada and software reuse, his interests include music, photography, motorcycling and horse support. Bailey received his M.S. in computer science from the University of Maryland, where he also earned bachelor's and master's degrees in cello performance. He is a member of the ACM.

Victor R. Basili is a professor at the University of Maryland, College Park's Institute for Advanced Computer Studies and Computer Science Department. His research interests include measuring and evaluating software development. He is a founder and principal of the Software Engineering Laboratory, which is a joint venture among NASA, the University of Maryland and Computer Sciences Corporation. Basili received his B.S. in mathematics from Fordham College, an M.S. in mathematics from Syracuse University and a Ph.D. in computer science from the university of Texas at Austin. He is a fellow of the the IEEE Computer Society and is editor-in-chief of IEEE Transactions on Software Engineering.

STANDARD BIBLIOGRAPHY OF SEL LITERATURE



STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, Tutorial on Models and Metrics for Software Management and Engineering, V. R. Basili, 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-906, Annotated Bibliography of Software Engineering Laboratory Literature, P. Groves and J. Valett, November 1990

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-106, Monitoring Software Development Through Dynamic Variables (Revision 1), C. W. Doerflinger, November 1989

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-84-101, Manager's Handbook for Software Development, Revision 1, L. Landis, F. McGarry, S. Waligora, et al., November 1990

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. W. Agresti, June 1987

SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, Data Collection Procedures for the Rehosted SEL Database, G. Heller, October 1987

SEL-87-009, Collected Software Engineering Papers: Volume V, S. DeLong, November 1987

SEL-87-010, Proceedings From the Twelfth Annual Software Engineering Workshop, December 1987

SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988

SEL-88-003, Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis, K. Quimby and L. Esker, December 1988

SEL-88-004, Proceedings of the Thirteenth Annual Software Engineering Workshop, November 1988

SEL-88-005, Proceedings of the First NASA Ada User's Symposium, December 1988

SEL-89-002, Implementation of a Production Ada Project: The GRODY Study, S. Godfrey and C. Brophy, September 1989

SEL-89-003, Software Management Environment (SME) Concepts and Architecture, W. Decker and J. Valett, August 1989

SEL-89-004, Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard, C. Brophy, November 1989

SEL-89-006, Collected Software Engineering Papers: Volume VII, November 1989

SEL-89-007, Proceedings of the Fourteenth Annual Software Engineering Workshop, November 1989

SEL-89-008, Proceedings of the Second NASA Ada Users' Symposium, November 1989

SEL-89-101, Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1), M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Spiegel, February 1990

SEL-90-001, Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide, M. Buhler and K. Pumphrey, March 1990

SEL-90-002, The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis, S. Green et al., March 1990

SEL-90-003, A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL), L. O. Jun and S. R. Valett, June 1990

SEL-90-004, Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary, T. McDermott and M. Stark, September 1990

SEL-90-005, Collected Software Engineering Papers: Volume VIII, November 1990

SEL-RELATED LITERATURE

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," Proceedings of the Eighth Annual National Conference on Ada Technology, March 1990

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[7]Basili, V. R., Maintenance = Reuse-Oriented Software Development, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., Software Development: A Paradigm for the Future, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., "Viewing Maintenance as Reuse-Oriented Software Development," IEEE Software, January 1990

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

Basili, V. R., and J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

[5]Basili, V., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proceedings of the 9th International Conference on Software Engineering, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," Proceedings of the Joint Ada Conference, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, June 1988

[7]Basili, V. R., and H. D. Rombach, Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes, University of Maryland, Technical Report TR-2446, April 1990

[2]Basili, V. R., R. W. Selby, Jr., and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," Proceedings of the NATO Advanced Study Institute, August 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[5]Basili, V. and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, December 1987

[2]Basili, V. R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," Proceedings of the Joint Ada Conference, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Washington Ada Technical Conference, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[5]Card, D., and W. Agresti, "Resolving the Software Science Anomaly," The Journal of Systems and Software, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," The Journal of Systems and Software, June 1988

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. Page, "Evaluating Software Engineering Technologies," IEEE Transactions on Software Engineering, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

[5]Doubleday, D., ASAP: An Ada Static Source Code Analyzer Program, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," Proceedings of the 1988 Washington Ada Symposium, June 1988

Hamilton, M., and S. Zeldin, A Demonstration of AXES for NAVPAK, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

Jeffery, D. R., and V. Basili, Characterizing Resource Data: A Model for Logical Association of Software Data, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," Proceedings of the Tenth International Conference on Software Engineering, April 1988

[5]Mark, L., and H. D. Rombach, A Meta Information Base for Software Engineering, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]McGarry, F., and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," Proceedings of the Sixth Washington Ada Symposium (WADAS), June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[5]Ramsey, C., and V. R. Basili, An Evaluation of Expert Systems for Software Engineering Management, University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," IEEE Transactions on Software Engineering, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," IEEE Software, March 1990

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," Proceedings From the Conference on Software Maintenance, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[7]Rombach, H. D., and B. T. Ulery, Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, University of Maryland, Technical Report TR-2252, May 1989

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," Proceedings of the 21st Hawaii International Conference on System Sciences, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," Proceedings of the CASE Technology Conference, April 1988

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[8]Stark, M., "On Designing Parametrized Systems Using Ada," Proceedings of the Seventh Washington Ada Symposium, June 1990

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada 1989, October 1989

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," Proceedings of the Joint Ada Conference, March 1987

[8]Straub, P. A., and M. Zelkowitz, "PUC: A Functional Specification Language for Ada," Proceedings of the Tenth International Conference of the Chilean Computer Science Society, July 1990

[7]Sunazuka, T., and V. R. Basili, Integrating Automated Support for a Software Management Cycle Into the TAME System, University of Maryland, Technical Report TR-2289, July 1989

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[5]Valett, J., and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[5]Wu, L., V. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," Proceedings of the Joint Ada Conference, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," Journal of Systems and Software, 1988

[8]Zelkowitz, M. V., "Evolution Towards Specifications Environment: Experiences With Syntax Editors," Information and Software Technology, April 1990

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

[1]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.

[5]This article also appears in SEL-87-009, Collected Software Engineering Papers: Volume V, November 1987.

[6]This article also appears in SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988.

[7]This article also appears in SEL-89-006, Collected Software Engineering Papers: Volume VII, November 1989.

[8]This article also appears in SEL-90-005, Collected Software Engineering Papers: Volume VIII, November 1990.