1990-08

Transformations in Specification-Based Software Evolution

Berzins, V.; Kopas, B.; Luqi; Yehudai, A.

Naval Postgraduate School

V. Berzins, B. Kopas, Luqi, and A. Yehudai, "Transformations in Specification-Based Software Evolution", Technical Report NPS 52-90-034, Computer Science Department, Naval Postgraduate School, 1990.

https://hdl.handle.net/10945/65207

T58

NPS52-90-034

NAVAL POSTGRADUATE SCHOOL

Monterey, California

TRANSFORMATIONS IN SPECIFICATION-BASED SOFTWARE EVOLUTION

V. Berzins
R. Kopas
Luqi
A. Yehudai

August 1990

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943-5100

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared for the National Science Foundation.

Reproduction of all or part of this report is authorized.

Valdis Berzins
VALDIS BERZINS
Associate Professor
of Computer Science

Luqi
LUQI
Associate Professor
of Computer Science

Reviewed by:

[illegible]
ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:

[illegible]
GORDON E. SCHACHER
Dean of Faculty
and Graduate Studies

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-90-034	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School 6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION National Science Foundation
6c. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5100	7b. ADDRESS *(City, State, and ZIP Code)* Washington, D.C. 20550
8a. NAME OF FUNDING/SPONSORING ORGANIZATION National Science Foundation 8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER CCR-8710737
8c. ADDRESS *(City, State, and ZIP Code)* Washington, D.C. 20550	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
TRANSFORMATIONS IN SPECIFICATION-BASED SOFTWARE EVOLUTION (U)

12. PERSONAL AUTHOR(S)

13a. TYPE OF REPORT Progress	13b. TIME COVERED FROM Mar TO Aug 90	14. DATE OF REPORT *(Year, Month, Day)* August 90	15. PAGE COUNT 28

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Program transformations. Software prototyping. Executable specification. Software Evolution.

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This paper presents a classification schema for the concepts and applications of software transformation in software evolution. Correctness preserving program transformations have been widely used for program development from an initial specification. We consider a more realistic case, where the specification evolves, rather than being fixed in advance. We outline a transformational software prototyping methodology, and develop an associated model to describe the process. An example illustrates the ideas. Software tool support and directions for future research are discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] SAME AS RPT. DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Luqi	22b. TELEPHONE *(Include Area Code)* (408) 646-2735	22c. OFFICE SYMBOL 52Lq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

[illegible]

Transformations in Specification-Based Software Evolution*

V. Berzins R. Kopas Luqi
A. Yehudai [†]
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

August 10, 1990

Abstract

This paper presents a classification schema for the concepts and applications of software transformatior in software evolution. Correctness preserving program transformations have been widely used for program development from an initial specification. We consider a more realistic case, where the specification evolves, rather than being fixed in advance. We outline a transformational software prototyping methodology, and develop an associated model to describe the process. An example illustrates the ideas. Software tool support and directions for future research are discussed.

Key Words Program transformations, Software prototyping, Executable specification, Software Evolution.

*This research was supported in part by the National Science Foundation under grant number CCR-8710737.

[†]Computer Science Department, Tel-Aviv University. Work done while visiting Naval Postgraduate School

[illegible]

[illegible]

*This research was [illegible]
number OCR [illegible]
*Computer [illegible]
recognition [illegible]

1 Introduction

Program transformations have been used extensively as a basis for automated software construction. Typically, one starts with a specification of the desired program, and by applying transformations that do not change the semantics, create an executable program realizing the specification. This process is at least partially automated. This approach is predicated on the assumption that a precise specification is written before the software engineer starts the implementation.

Such a separation is not always possible. Specification is the hardest phase of software development. In order to get user confirmation that the suggested specification is suitable, it is useful to have an executable version of the specification, or a prototype.

The purpose of this paper is to show how software transformations can be used for software prototyping. Transformations that keep the semantics unchanged are no longer sufficient if the specification is to evolve during the prototyping process. Hence we need to consider new kinds of transformations.

The rest of the paper is organized as follows. Section 1.1 describes some of the previous work in the area of program transformations. Then, in Section 2, we present our classification of transformations that are useful when the specification cannot be assumed fixed. Section 3 presents the prototyping methodology we propose and a concrete model in which we characterize the transformations, and illustrates our ideas with an example. Finally, in Section 4, we discuss tool support for the methodology, and outline future the research problems that must be tackled in order to make our approach feasible.

1.1 Previous work

Program transformations have been studied extensively since the 1970s. A more comprehensive overview can be found in [25, 16].

A program transformation is a relation between two programs or program schemes P and P'. Normally one is interested only in *valid* transformations, which must satisfy some semantic relation between P and P'. Most of the work to date concentrates on transformations under which the programs are *equivalent*. Other semantic relations considered were [10] *weak equivalence* (in which the programs P and P' may act differently for erroneous data), and *descendent relation* (which is relevant for nondeterministic programs; here the possible results of P' are a subset of the possible results for P). These relations are used in [10] in a study of formal semantics.

Program transformations have been used in various applications to construct or modify programs. These applications have concentrated on transformations satisfying a semantic equivalence. Such transformations are referred to as *semantic preserving transformations* or *correctness preserving transformations*. Two principal kinds of correctness preserving transformations have been distinguished: *vertical* (going from a higher abstraction level to a lower one) and *lateral* (specifying the equivalence between two expressions at a similar level of abstraction) [27, p. xv].

Transformations are used in the context of *transformational programming*, which is a methodology of program construction via successive application of transformation rules. Usually such a methodology is supported by an appropriate programming environment, or a collections of tools. The level of support may vary from full automation, to merely assisting the user in selecting and applying a transformation rule.

Transformation systems have been used to achieve a variety of goals. The first use of transformations was based on a natural trade-off between efficiency and clarity in most programs. It is often possible to write a self-evidently correct program, but the resulting program is usually very inefficient. Lateral transformations can then be used to convert a clear but inefficient program into an efficient one [11, 13].

The most common goal of transformations is the construction of a program from a formal specification. As an example, the CIP project in Munich [6, 4, 5], treats program development as an evolutionary process, that starts with the problem specification, and ends with an executable program for the target machine. The transitions between the various versions of the program are effected by applying semantic preserving transformations. The programmer has to choose an appropriate transformation rule at each step, and an interactive tool applies the rule, while checking its applicability. Both the language used (CIP-L) and the transformation system (CIP-S) are based on algebraic specifications.

Existing transformations implementation systems can be divided into two classes: those that are relatively limited in power but require no user guidance and those that are capable of very complex implementations but only under user guidance. TAMPR [9] and PDS [12] use simple control strategies and restrictions on the kinds of transformations that can be defined in order to eliminate the need for user guidance.

The PSI system [19] was one of the first systems to use a transformational implementation. PSI's transformational module [3] operated without guidance, generating all possible low-level programs. It was assumed that another component [21] would provide guidance as to which transformation to use. More recent work on complex transformational implementation systems has been done at ISI [2], and at the Kestrel Institute [29]. A key focus of these efforts has been an attempt to automate the choice of transformations as much as possible [15, 17, 31].

The sequence of transformation chosen to implement a program is a valuable record of the development process [31]. It is should be possible to replay at least part of these transformations when the original high level (specification) is modified. This idea motivates work on software maintenance with the aid of transformations. A method called transformation-based maintenance model (TMM), was proposed in [1]. The model is based on the assumption that a program has been derived from a specification using a transformational system, and that the sequence of transformations applied has been recorded. This sequence may be viewed as a path from the root of a tree (representing the specification), to a leaf (representing the current program). In each internal node of the tree, a design decision has been made, resulting in the application of a particular transformation. In the maintenance process we can then traverse the path backwards, and change some of the decisions to choose a different transformation. It should be noted that since it is possible to reach a node by more than one path, the structure is a directed acyclic graph, rather than a tree. TMM takes advantage of some specific properties of the particular transformational programming paradigm used in Draco [24].

An important issue is the representation of transformations. Transformations are represented as *transformation rules*, which may be *procedural* or *schematic.*

As can be seen from the above discussion, much of the work on program transformations is done in the context of using artificial intelligence techniques for software construction. Here transformations are used to represent knowledge about programming, as well as application domain specific knowledge.

We have recently learned about the work on evolution transformation library in ISI [20]. This work shares some of the premises that were the basis of our work. In particular, transformations that do not preserve correctness are used to capture evolution of software specification. Our approach differs from that of [20] in the way we view transformation. Our goal is to utilize transformations that correspond to design decision. Transformations are viewed at a semantic, rather than syntactic level.

2 Classification of Transformations

It is sufficient to use correctness preserving transformation if we are starting with a fixed specification and wish to generate a program that implements it. If we wish to take a transformational approach to the specification phase of software construction, we must allow for changes in the semantics. Moreover, it has become clear [30] that one cannot truly separate specification from implementation. Indeed, to quote [30],

> The standard software development model holds that each step of the development should be a "valid" realization of the specification. By "valid" we mean that the behaviors specified by the implementation are a subset of those defined by the specification. However, in actual practice, we find that many development steps violate this validity relationship between specification and implementation. Rather than providing an implementation of the specification, they knowingly redefine the specification itself. Our central argument is that these steps are a crucial mechanism for elaborating the specification and are necessarily intertwined with the implementation.

The goal of our work is to lay the foundation for using a transformational approach in a process of software construction that does not assume a fixed initial specification. In this section we characterize different kinds of transformations that we believe are useful in such a context.

A transformation is simply a function taking some document in a given language, and mapping it onto another document, possibly in another language. Normally, the input and output document are close to each other in terms of their meaning or content. This is a rather informal and imprecise characterization, as will be some of the definitions in this section. More precise definitions are provided in Section 3.1, where we refer to a concrete model. For the sake of simplicity, we assume that we are dealing with one language. This may require a union of the languages under consideration. Also, we will often think of a transformation as "modifying" a given document. Here we refer to the change or the difference between the input document and the output document.

A transformation may be either *global* or *incremental*. An incremental transformation is one that changes only a small portion of the input document. In particular, since we are concerned with documents that have a certain underlying structure, and can be viewed as composed of modules, an incremental transformation will leave most of the modules unchanged. Since we view documents as structured entities, an incremental transformation may also change some of the connections between modules, while preserving most of the content of the modules themselves. Simply making textual comparison between the input and the output can therefore not be used as the criterion for the amount of change done. Global transformations are those that change most or all of the document and possibly its structure. As an example of a global transformation one can consider a compiler that transforms an input program in some high level language into another program in assembly or machine language. Note that our assumption that a transformation does not alter the meaning of a document much holds for this example. In fact, the input and output programs in the case of a compiler should be semantically identical.

In what follows, we consider mostly incremental transformations. We believe that they have an important role throughout the software life cycle. However, our characterization of the different kinds of transformations applies to global transformations as well.

We consider two essentially orthogonal attributes of a transformation. One relates to the level of abstraction of the documents in question, and the other considers the meaning of the documents.

A transformation may leave the abstraction level of a document basically unchanged. It may lower the abstraction level by introducing more detail. Finally, a transformation may raise the level of abstraction by removing some details.

Independently of what changes are made to the abstraction level, transformations may have different effects on the meaning of the document in question. We will restrict our attention to a particular kind of meaning - that of the input-output behavior of the program being specified or coded. In general, one can associate with each document a collection of possible behaviors. During the early stages of the development, documents only approximate the behavior. For example, a requirement specification document is supposed to prescribe what we want the software to do, possibly leaving some freedom as to the exact input and output sequences involved. We can think of such a document as matching a (possibly infinite) collection of possible behaviors. A sequential program usually has one behavior. (We can think of an uncompilable program as having an empty collection of behaviors.) Some languages allow nondeterministic sequential programs (which imply a collection of behaviors larger than one), even though the object code that is produced by the compiler is deterministic at the bit level. For example, Dijkstra's guarded command construct [14] allows non-disjoint conditions in a choice. The effect of a conventional sequential program can appear to be nondeterministic when viewed as a function on abstract data types if those types can have multiple bit-level representation for the same abstract value. A choose operation on a set data type is an example. For concurrent software, and particularly for distributed systems, even the actual code may still have a larger collection of behaviors associated with it.

We can categorize transformations according to how they affect the collection of meanings of the document being transformed. In particular, we can assign categories according to the set-theoretic relation between the collection of behaviors of the input document, which we will denote B_I, and the corresponding collection for the output document, denoted by B_O. We can thus have the following cases :

- (i) B_I is equal to B_O,
- (ii) B_I (properly) contains B_O,
- (iii) B_I is (properly) contained in B_O,
- (iv) B_I is disjoint from B_O,
- (v) B_I and B_O have nonempty intersection, but none contains the other,

We believe that it is useful to restrict our attention to the first 3 kinds only. Transformations that fall under case (iv) or (v) have less useful structure, but they may be relevant e.g. in debugging: When we correct an error in a program, we may transform it in a way that takes us from a (single) meaning, to another one, different from it. Here we have disjoint (singleton) sets involved. We believe it will usually be desirable to view such a correction as an application of a type (iii) transformation, followed by a type (ii) transformation. This is the desired situation particularly when making corrections at an early phase of the life cycle. Note that in the terminology of [10], the semantic relation in case (i) is *strong equivalence*, whereas in case (ii), the output document is a *strong descendent* of the input document.

The two orthogonal attributes describing the effect of a transformation on abstraction level and on behavior will now be combined to present our classification. We describe each kind in detail (see Figure 1).

Abstraction level	Collections of behavior		
	$B_I \subset B_O$	$B_I = B_O$	$B_I \supset B_O$
Down (more detail)	—	refining	—
same level	relaxing	reformulating	constraining
Up (less detail)	—	retracting	—

Figure 1: Types of Transformations

- *Refining transformations*, also called *refinements*, are those that add detail to a document without changing its behavior. In the terminology of [27, p. xv], these are called a *vertical* transformations. They occur often in the literature about transformations systems for program construction, such as CIP, where they have been called correctness preserving. Typically, such a transformation may choose a particular algorithm or data structure for the implementation.

- *Retracting transformations*, also called *retractions*, are essentially the inverse of refining transformations. Retractions may be used to abstract away some detail in a document. They may occur in reverse engineering processes, useful for maintenance, as for instance in TMM [1]. In a development that is guided by a systematic, transformation based process, we would expect retractions to follow backward a path previously generated as a refinement.

- *Reformulating transformations*, also called *reformulations*, are those that make a local change to a document without changing its behavior, and leaving it in essentially the same abstraction level. (These are the *lateral* transformations in the terminology of [27, p. xv].) A typical example is a source level optimization, such as loop unrolling or tail recursion removal, applied to a program in some programming language. Other examples are any changes done to a document to improve its readability, clarity etc.

  So far we have discussed the transformations that left the behavior of a document unchanged. We now turn our attention to transformations changing the behavior of a document. These are useful when we explore the possible behavior of the software system being constructed in order to decide on the desired functionality. As noted above, we restrict ourselves to changes such that the input behavior and the output behavior are contained in each other.

- *Constraining transformations* restrict the behavior of a document, by making a particular decision. For example, replacing a bounded buffer with unspecified bound by one of a certain size (as in a generic package with the bound being a parameter), is a constraining transformation. A transformation specifying the way two modules will communicate with each other, where a previous version left this unspecified, is another example.

- *Relaxing transformations*, are the inverse of constraining transformations. They relax some restrictions on the behavior of the software that exist in the previous version. For example, relaxing the requirement to keep an airplane exactly on course to the requirement for corrective steering when the airplane strays off its course, thus keeping it within some tolerance of the expected position. A different kind of an example is when a module (such as an Ada subprogram or package) is changed to a generic one, replacing a constant or a specific data type by a parameter. Another example is when real numbers are replaced by floating point numbers of a particular precision. This is a relaxation since it replaces a single acceptable output by the interval of values that are acceptable with respect to the given precision.

In reality, not all changes to a specification are captured by relaxing transformations and constraining transformations. However, we can view a modification as a relaxing transformation, removing some restrictions that currently exist (implicitly or explicitly), followed by a constraining transformation, that introduces the new, modified restrictions. This approach provides a better way to record the evolution of the document, and yields a process that is more likely to be reused elsewhere.

As can be seen in Figure 1, we have disallowed 4 possible combinations of the two attributes of a transformation. In particular, we do not admit transformations that change both the behavior and the abstraction level of a document. We believe such a restriction is useful in better capturing the development process. Consider, for example, the process of generating a design document from a specification document. During this process, we make specific design decisions that restrict the behavior of the software, and also introduce more detail. We claim that it is better to describe this process as a sequence of transformations following our conventions. First, a series of constraining transformations introduce the design decisions, then we apply refinements to introduce the implementation details. In fact, it may be more desirable to alternate between these two kinds of transformation, as the more details are added following each step (or several steps) of design decisions reflected in constraining transformations. In practice we may actually see some relaxing transformations appear also along the way as better understanding of the problem introduces a need to undo some previous decisions, or to remove some other restrictions before continuing with new restrictions.

Our classification of transformations is somewhat similar to the conceptual view of specification evolution presented in [18]. The author's main thesis is that it is better to present a specification as evolving from simpler one. The evolution is characterized by three dimensions. *Coverage* deals with the range of behaviors permitted by a specification, where behavior is a sequence of states. This dimension is similar to our notion of changes in the behavior, but [18] does not characterize the desired or allowed changes. (Behavior modeling is also different.) *Structural granularity* and *temporal granularity* deals with the amount of detail of each individual state of the process, and the amount of change between successive states, respectively revealed by the specification. Both these dimensions have to do with abstraction level, but temporal granularity is useful only when behavior is modeled by state change.

3 Using Transformations for Prototyping

Prototyping has been suggested as an approach to software development that enhances communication with the user community by providing an executable model of the system early in the development process. In this section we discuss the prototyping methodology we are advocating, and show how transformation may be applied to aid in this process. We then give more precise definitions of the kinds of transformations we are dealing with, in the context of a particular model and language. Finally, an example is used to illustrate our ideas.

There are two phases in the prototyping process, *prototype evolution* and *production code generation* [22]. The purpose of prototype evolution is to firm up the software requirements. When the requirements are stable, production code generation can generate an efficient implementation. If at a later time there is a need to modify the requirements, we can return to prototype evolution, and again follow it up with production code generation.

In the prototype evolution phase the designer constructs and modifies a prototype based on feedback from the customer until the prototype matches the needs of the users. This part of the process is dominated by a series of incompatible changes to the behavior of the prototype. These changes are realized via relaxing and constraining transformations. Reformulating transformations are applied at this stage mainly for adjusting the structure of the prototype to make it easier to understand or modify. Efficiency is considered only if the requirements include hard real-time constraints or prototype demonstrations take impractically long to run.

In the production code generation phase of the process, the desired behavior of the system is relatively stable, and the major concern is improving efficiency, capacity, or robustness. This part of the process is dominated by behavior preserving transformations for optimizing the design and implementation. Behavior preserving transformations have been studied extensively [6, 4, 5, 17, 21, 19, 31]. However, relaxing and constraining transformations are sometimes also needed to improve efficiency. Such transformations are applied in practice to optimize a design, because efficient algorithms are often applicable only in special cases, and may constrain the set of problems that can be solved. In general such implementation strategies introduce additional preconditions, which are relaxing transformations because they remove constraints on system behavior in the cases where the new preconditions are not satisfied. Such a design is then constrained by defining responses for the remaining cases, such as exception conditions or error messages. A common example of an optimization that speeds up an algorithm by introducing constraints is static memory allocation, which puts a fixed bound on the size of a data structure.

3.1 A Formal Model

In order to make our definitions more precise, we need a concrete model. We will use a model in which programs and subprograms are specified using preconditions and postconditions. We use the common notation $P\{S\}Q$ to denote a program (segment) S, with precondition P and postcondition Q.

We restrict our attention to the transformations that leave the abstraction level of a document unchanged. We view changes in a program as reflected in changes in its precondition and postcondition. In particular, suppose a transformation T takes us from program $P\{S\}Q$ to another program $P'\{S'\}Q'$. Consider the case that the postcondition Q' is stronger than Q, while the postcondition remain unchanged [1]:

$$Q' \Rightarrow Q \& Q \not\Rightarrow Q' \& P' \Leftrightarrow P.$$

In particular, this may be the case when $Q' = Q\&C$, where C is some additional condition that the program must now satisfy upon termination. In this case we have constrained the possible behavior of the program, thus T is a constraining transformation.

Conversely, suppose the precondition P' is stronger than P, while the postcondition is unchanged:

$$P' \Rightarrow P \& P \not\Rightarrow P' \& Q' \Leftrightarrow Q.$$

In particular, this may be the case when $P' = P\&C$, where C is some additional condition that the input must satisfy. Here, by restricting the possible inputs, we have applied a relaxing transformation.

We can also obtain a relaxing transformation by weakening the postcondition, rather than strengthening the precondition. It also follows that a transformation that both weakens the postcondition and strengthens the precondition is also relaxing. Similarly, we can fill the other entries in Figure 2. Note that for two entries we cannot determine the effect of the transformation in general, because of the opposing effects that the changes on the precondition and the postcondition create.

We will use Spec [7] as our specification language. Spec is a formal language for writing specifications for components of software systems. It is used in functional specification for recording black-box specifications of the external interfaces of the proposed system, and in the architectural design stage for recording black-box specifications of the internal interfaces of the proposed system.

[1] $\Rightarrow$ and $\Leftarrow$ denote logical implication (in the indicated direction). $\Leftrightarrow$ denotes logical equivalence

Precondition	Postconditions		
	$Q \Rightarrow Q'$	$Q \Leftrightarrow Q'$	$Q \Leftarrow Q'$
$P \Rightarrow P'$	?	constraining	constraining
$P \Leftrightarrow P'$	relaxing	reformulating	constraining
$P \Leftarrow P'$	relaxing	relaxing	?

Figure 2: Classifying Transformations by the relation between pre- and postcondition

Spec is based on the event model of computation, and uses predicate logic for the precise definition of the desired behavior of modules. The most important ideas of this language are modules, messages, events, parameterization, and defined concepts.

A module is a black box that interacts with other modules only by sending and receiving messages. A message is a data packet that is sent from one module to another. An event occurs when a message is received by a module at a particular instant of time.

The response of a module to a message can be defined with several cases characterized by preconditions that are written as WHEN clauses. To describe a postcondition that must be satisfied by an outgoing message, a WHERE clause is used. The WHERE keyword is followed by a set of statements in predicate logic describing the relation between the contents of the message that was received and the contents of the reply message.

Spec is a specification language, and is not fully executable. However, there is a subset of Spec that may be translated into executable form, which may be viewed as a prototype.

We augment Spec with augmented data flow diagrams to describe the interconnection between Spec modules. This notation is borrowed from PSDL [23].

3.2 The Prototyping Process

The prototyping process starts from requirements analysis as shown in Figure 3. A computer aided prototyping system is helpful for constructing a software prototype based on the requirements. The validation activity demonstrates some typical cases of prototype execution and generates a series of requirements adjustments based on the customer's quick feedback. Such adjustments can be used to correct or change the set of requirements in the requirement analysis activity before the new set of requirements is used to construct the next version of the prototype in the iterative process of prototyping. This process is used to refine, adjust, and firm up the requirements. The feedback related to problems in the Spec modules or the structure of the prototype goes back to the "construct prototype" activity through different paths as one of the iterative activities in the evolutionary prototyping process. The adjustments to the structure of the prototype or corrections to the Spec modules are sent back to the "construct prototype" activity to be used in the next iteration of the prototype construction activity. The adjustments to the requirements are sent to the "analyze requirements" activity to trigger the next stage of the evolutionary process.

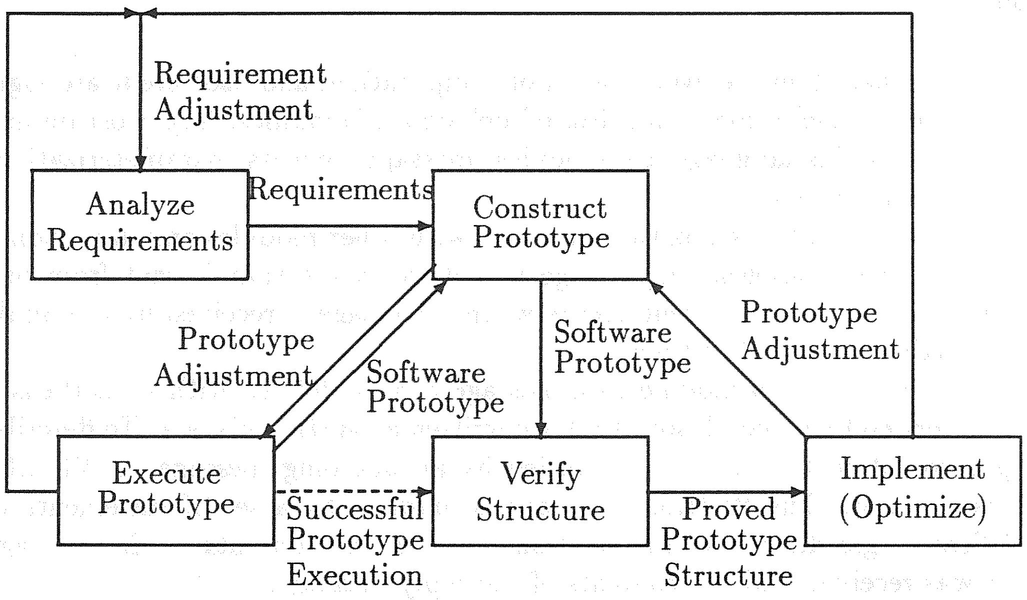

Figure 3: Prototyping Life Cycle Process Model

An acceptable execution of the prototype triggers the production of the envisioned system based on the structure and other attributes determined in the prototyping from the user and the designer. Verification or proof of the structure of the prototype can ensure the correctness of the implementation or optimization phase with a sound foundation, provided the implementations of the individual Spec module are correct. This can be useful in cases where different subsystems of the final product will be implemented by different groups, because it prevents integration problems at the end if the subcontractors stick to the specifications for their subsystems. In critical subsystems where software failures can have very serious consequences, the implementations of the individual subsystems can be proven correct at lower levels of the design and code as needed. Different kinds of transformations can be applied to the activities in the evolutionary prototyping process. For example, the meaning preserving transformation can be used in terms of executable specification context as well as in the automatic code generation and optimization of the production code. The refining and retracting transformations are useful in the activity of converting requirements to prototype specifications in Spec. Transformation techniques are very important in the "construct prototype" activity. It can be seen more directly if we examine the activity with more details. Figure 4 illustrates the process to use Spec in the prototype construction.

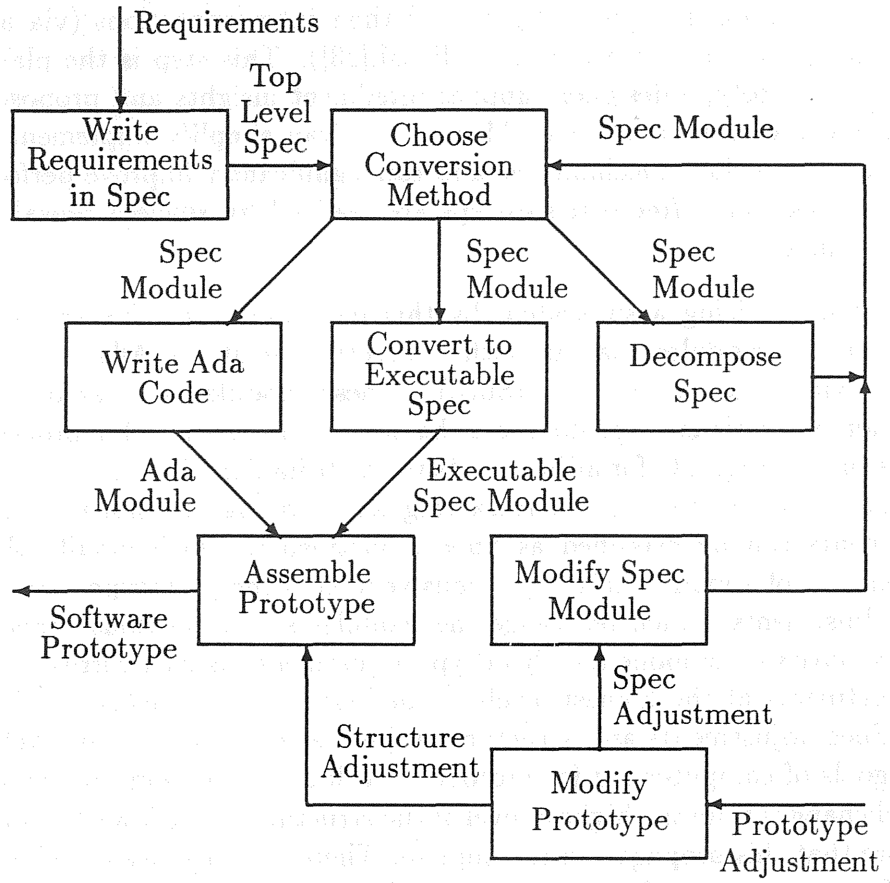

Figure 4: Constructing a Prototype

Our procedure for realizing a prototype works as follows. For each subsystem in the prototype, the results of the requirements analysis are used to propose a system interface, and the behavior of the interface is expressed in the Spec language. The specification is then converted to executable form. There are three ways to do this:

- Transform the specification into the executable subset of the Spec language. This step proceeds using meaning-preserving transformations, and may be trivial if the original spec is already in an executable form. This step is necessary because the full Spec language includes unbounded quantifiers and is strong enough to specify functions that are not computable. However, if the requirements are feasible, then the transformation into executable form must also be feasible, and we conjecture that in most practical cases it will be straightforward and partially automatable.

- Produce code in a programming language such as Ada. This can be done by retrieving and adapting reusable components, or by creating new code.

- Decompose the module into lower level components. This requires specifying the components (using Spec) and their interconnections (via an augmented data flow diagram, as in PSDL[23]). This step is the place where the prototype designer supplies intelligent insights and proposes useful lower-level abstractions. This process can simplify implementation via the previous mechanisms, and can significantly improve performance, especially if frequent substeps are realized by efficient reusable Ada modules.

The result of realizing a subsystem by this mechanism is a hierarchical decomposition into modules that are either directly executable(Ada) or can be simulated via symbolic execution(Spec). These modules are assembled to provide demonstrations of prototype behavior to the users. This process often results in user requests for adjustments to the behavior of the prototype. These adjustments are realized via retracting and refining transformations. The adjustments can be classified as Spec adjustments, which modify the specified behavior of a module in the previous version of the prototype, and as structural adjustments, which rearrange the modules in the previous version and add or remove subcomponents. Prototype adjustments usually correspond to Spec adjustments at the highest levels of the hierarchical structure, and a mixture of Spec adjustments and structural adjustments in the lowest levels. One of the goals of computer aid for prototype evolution is to help propagate the intended changes from the highest level of the structure to the lowest levels, and to ensure that this propagation is complete. There is also a cleanup phase in which reformulating transformations are used to simplify the structure and to remove features that are no longer needed to support the new specifications. This cleanup phase is not shown in the Figure 4 because it is done after a demonstration session is over.

3.3 Example: Spelling Checker

We now illustrate the use of transformations in prototyping by means of an example. We outline a process of specification and design of a spelling correction system. A specification for the initial version of a prototype is shown in Figure 5.

The spelling checking function is specified by giving a postcondition describing its required output. Since the function is supposed to produce the required output for all possible inputs, there is no precondition given. The concept is an auxiliary definition describing the intended interpretation of the

```
FUNCTION spell
   IMPORT sorted FROM sequence{word}
   IMPORT Subtype FROM type

   MESSAGE(report: sequence{word},
               dictionary: set{word})
      REPLY(errors: sequence{word})
      WHERE ALL(w: word :: w IN errors <=>
         w IN report & ~(w IN dictionary)),
         sorted{less_or_equal@word}(errors)

   CONCEPT word: type
      WHERE Subtype(word, string),
         ALL(c: character, w: word :: c IN w =>
            c IN ({a .. z} U {A .. Z}))
END
```

Figure 5: Specification of Initial Spelling Checker

type *word*. The initial prototype is written in terms of abstract inputs and outputs, and the default methods are used for reading input data and displaying output data. The types *set*, *sequence* and *type* are pre-defined in the standard Spec library, which can be found in [8]. Definitions of selected reusable concepts such as the predicate *sorted* from these standard specification modules are incorporated into the example via IMPORT declarations.

This specification can be realized by a design that decomposes it into two more primitive functions as shown in Figure 6.

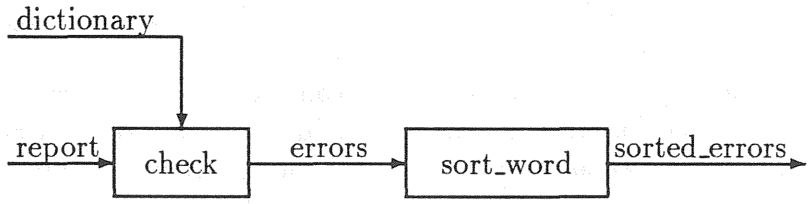

Figure 6: Initial Decomposition

The specifications for the subfunctions check and sort_words are given in Figure 7. Sort is defined as a generic module, and sort_words is declared as an instance of that generic module.

After building the prototype according to this design, by utilizing reusable software components in a software base, the prototype is demonstrated to a potential user. A user remarks that many terms commonly used in his business are reported as spelling errors, such as names of products and suppliers. The

```
FUNCTION check
   IMPORT word FROM spell

   MESSAGE(report: sequence{word}, dictionary: set{word})
       REPLY(errors: sequence{word})
       WHERE ALL(w: word :: w IN errors <=>
           w IN report & ~(w IN dictionary))
END

FUNCTION sort{t: type,
                 le: function{from:: [t, t], to:: boolean}}
   WHERE total_ordering(le)

   IMPORT total_ordering FROM total_order{t}
   IMPORT sorted permutation FROM sequence{t}

   MESSAGE(in: sequence{t})
       REPLY(out: sequence{t})
       WHERE sorted{le}(out), permutation(in, out)
END


INSTANCE sort_words = sort{words, less_or_equal@word} END
```

Figure 7: Specifications for Subfunctions

customer does not like this and wants it fixed. The designer notices that such terms are likely to be different for different installations and suggests augmenting the design with a private dictionary that can be augmented by each user to fit local needs. The specification for the modified design is shown in Figure 8. The added text is placed inside boxes.

The modified design is produced by a constraining transformation which adds an additional conjunct to the postcondition, further constraining the output and requiring an additional input for the function. An initial modified design is obtained by noting that to implement the new version of spell, one can simply call the old one and pass to it as a second parameter the union of the dictionary and the private_dictionary. This is illustrated in Figure 9.

A more concrete design, in terms of the subfunctions is obtained by adding an intermediate subfunction for checking the additional constraint is shown in Figure 10.

One can characterize the various stages of the process as follows. We consider the document in each version as a collection of Spec modules and design graphs.

1. A specification of spell (Figure 5).

```
FUNCTION spell
   IMPORT sorted FROM sequence{word}
   IMPORT Subtype FROM type

   MESSAGE(report: sequence{word},
              dictionary  private_dictionary  : set{word})
      REPLY(errors: sequence{word})
      WHERE ALL(w: word :: w IN errors <=>
         w IN report  & ~(w IN dictionary)
                      & ~(w IN private_dictionary)  ),
         sorted{less_or_equal@word}(errors)

   CONCEPT word: type
      WHERE Subtype(word, string),
         ALL(c: character, w: word :: c IN w =>
            c IN ({a .. z} U {A .. Z}))
END
```

Figure 8: Transformed Specification for the Spelling Checker

2. A specification of spell, and initial decomposition of it (Figure 5,6).
3. A specification of spell, initial decomposition of it, and specification of the components (Figure 5,6,7).
4. A specification of the new version of spell (Figure 8).
5. A specification of the new version of spell, and its decomposition in terms of the old one (Figure 8,9).
6. A specification of the new version of spell, and a redesign, using the

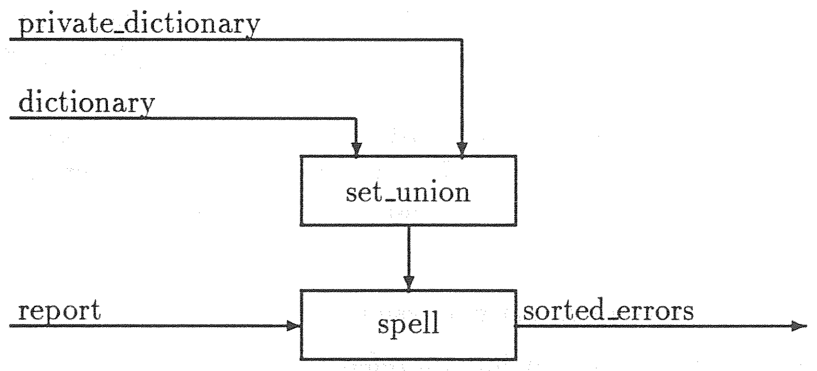

Figure 9: New Spell

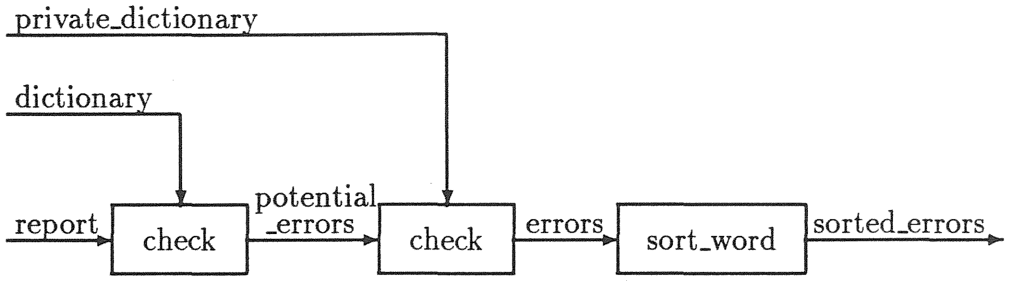

Figure 10: Transformed Decomposition

components of the old spell (Figure 8,10).

7. A specification of the new version of spell, the redesign, using the components of the old spell, and the specification of these subfunctions (Figure 8,10,7).

The particular steps taken can be described as follows:

1 $\leadsto$ **2** This step introduces more details – a design, so the transformation applied is a refinement. The resulting document has a lower abstraction level.

2 $\leadsto$ **3** This step also introduces more details – a specification for the subfunctions that appear in the design of the input document, so it is also a refinement. Note that this will probably be realized as a sequence of two refining transformations, each one introducing the specification of one of the subfunctions. The order of application is arbitrary, as the two transformations are independent of each other.

1 $\leadsto$ **4** This step changes the behavior of the system. The modified specification is produced by a constraining transformation which adds an additional conjunct to the postcondition, further constraining the output and requiring an additional input for the function. The transformation applies to stage 1, and not stage 3, because we are not dealing with the lower level information. The resulting output therefore does not include the information of Figure 5,6,7. These modules are conceptually still in our "library", representing the old version of spell, which has not been removed.

4 $\leadsto$ **5** This is a refinement, since a design is added.

Since this new version includes a reference to the old version of spell, we could at this stage apply the transformations that were applied in step 1 $\leadsto$ 2 and step 2 $\leadsto$ 3. However, we have decided to proceed with the following step.

5 $\rightsquigarrow$ 6 This is a reformulating transformation which does not affect the behavior but simplifies the implementation. The transformation reduces the number of distinct types of components in the design without introducing any new types, and therefore reduces the implementation cost.

6 $\rightsquigarrow$ 7 This is an application of the same transformation that was applied in the step $2 \rightsquigarrow 3$.

4 Tool Support and Future Work

The previous sections give the foundation for a transformational software prototyping methodology. This approach needs automated support to achieve a practical impact. In this section we explore some tools and conceptual advances needed to make this approach work smoothly. The main components of a system to provide this support are the transformations themselves, a semantic framework for representing and analyzing the transformations, and tools for:

- finding relevant transformations,
- applying transformations to software components,
- deducing transformations from software components,
- recording derivation histories,
- and managing design changes.

Any transformation system must supply a collection of transformation rules from which the user and/or system can choose [25]. These can be in the form of a finite *catalog*, or a countable *generative set* from which transformation rules are generated on demand. The choice of which transformations to include in the system determines both the capabilities of the system and the amount of effort the designer must spend to use the system. The collection of transformations encapsulates knowledge about the software development process, as well as domain specific knowledge. Much work is needed to establish the specific transformations and transformation structures that are needed. Initial investigations should focus on transformations applicable to a particular application domain, to build better models of the design process and determine the most effective structure for the space of transformations without getting bogged down in the analysis of large application domains.

Transformations can be classified as those applied automatically by the system and those chosen interactively by the system designer. It is desirable for a transformation system to apply some transformations without user interaction, to reduce the burden on the designer. Transformations chosen automatically by the system must have practically computable criteria for applicability, and these criteria must correspond to the intentions of the designer. Only universally desirable criteria such as simplicity and efficiency can be implicit in a set of transformations. In practice, many design decisions involve tradeoffs between conflicting design goals. Consequently, a transformation system should provide facilities for describing design goals and criteria for judging their relative importance, so that the designer can guide the system without having to specify each individual transformation.

The transformations chosen by the designer should correspond directly to design decisions at a level of detail natural for a human designer, so that the designer can efficiently construct a design and the system can keep a useful record of the design history. The primitives of common design notations such as specification languages and diagrams are at a much lower level of detail than the decisions commonly made by system designers. Two possible approaches to this problem are either to work with larger units than the primitives of the notation, or to work in a more abstract space that corresponds more closely to the designer's conceptual universe, and to generate the detailed design representations via automated translation steps.

The relation between these two approaches can be understood by analogy with syntax-directed editors [26]. In such systems there are two ways to add syntactic detail to a partial design: via a template transformation, or by entering free text. The template transformations are determined by the grammar of the source language - there is one alternative for each production whose left hand side matches the currently selected syntactic category. The grammar thus defines a finite number of choices, which can be chosen from an automatically constructed menu. Although all sentences in the source language can be generated by template expansion, this can be tedious in cases where templates have few terminal symbols, such as infix operators in arithmetic expressions. The other input mechanism, free text entry, is used to handle such situations more efficiently. The templates implicit in the free text are identified by a parsing operation which builds a syntax tree according to the grammar of the source language.

Refining transformations are the semantic analog to the text entry operations of a syntax-directed editor. In cases where the semantic choices available to the designer is finite and can be predicted by the system, a choice can be picked from a menu. This requires developing a characterization of the design space analogous to a grammar for the source language of a syntax-directed editor. However, the semantic design space is less well understood than the syntax of a context free language, and is likely to be more complex. In particular, it may not be possible to construct a closed description of the set of choices available to the designer in some situations, and in some situations the number of choices may not be finite. The process of free text entry is therefore likely to be necessary for semantic decisions, in addition to being a practical aid to efficiency of design entry in some situations. A transformation system supporting such a mode should have an analog to the parsing process, which attempts to reconstruct the set of primitive refining transformations that lead to the result of the free text entry step. Such a matching process is likely to be computationally expensive, and may be practical only for relatively small refinements. The advantage of providing such a process is that the internal representation of the derivation tree for the design provides a record of the designer's thoughts, which may be useful for mechanical aid in later changes to the design. More work on models of software design decisions is needed to enable the implementation of such a facility.

A tool with such a facility for reconstructing primitive design decisions would have full knowledge of the sequence of transformations by which we reached the current version of the prototype for each stage of the process. This will help us to go back to a previous stage, and change some of our decisions. This is reminiscent of the scenario outlined in [31, 1]. It is therefore desirable to make each transformation correspond to a single design decision. It is believed that incremental transformations can be almost self documenting in this respect.

Incremental transformations are also likely to have another advantage. If two transformations applied to different parts of the system are independent, which is often the case, they can be reordered. Consider the case in which a transformation T_1 was performed, followed by T_2, which deals with a different part or aspect of the system. Suppose that at a later time, we wish to change the design decision that triggered the application of T_1. We back up to that point, and apply T_1' instead. It is very likely that we can now perform T_2 again, since the local changes due to T_1' probably do not affect the applicability of T_2. We would like our tool to identify the cases where the transformation may be dependent, and alert the user to check if this is the case. In situations where the tool is able to ascertain independence, automatic re-application of the transformation should follow.

Clearly, not all desired transformations can be automated. Whenever the user adds a module, for instance, he is applying a refinement. This may be automated in some existing transformation systems designed to construct a program from specification. However, we do not wish to assume such a transformation can always be deduced by the tool. We want to allow the user to freely write such a module. The tool will then be required to keep track of this application, so that a full path is recorded. In addition, it may be possible to deduce from it a general transformation rule and some conditions on when it may be applied, and add this rule to the transformation library. Sometimes a single design step the user performs can be represented as a sequence of simpler transformations. Since incremental transformations are desirable, we would like the tool to help decompose this step into its constituent steps, and record them as such. Transformations that are the result of the user adding a conjunct to a precondition (or a postcondition) may automated, though probably not always. The tool should be able to suggest canonical ways to constrain a condition, depending on the module being handled. Needless to say, transformations that remove a conjunct from a condition are much easier to automate.

Some examples of design level transformations for creating refinements which have systematic characterizations follow:

- Implement a concept as a lower level component (indicated for concepts appearing in preconditions). The specification for the lower level component can be automatically constructed in such a case. Generalize (make a constant or an expression into a generic parameter). Even though this represents a single design decision, it will be reflected in many places in most design representations (i.e. in the definition of the module with the extra parameter, and in each use of that module).
- Extend a function from individuals to a collection (sequence, set, vector).
- Extend a function by generalizing the type of an input parameter (replacing it by a supertype).
- Strengthen a function by specializing the type of an output parameter (replacing it by a subtype or adding a new postcondition).
- Transform a predicate into a generator of values satisfying the predicate.
- Implement a data type using a direct storage representation (no pointers). This kind of a design decision, as well as the ones that follow, can be recorded as a Spec pragma [8], and some of the features of the programming language level interface implied by such a decision can be constructed automatically.
- Implement an input value and an output value as a single in-out parameter (limited lifetime of input data).
- Implement an output sequence via a time series (incremental generator).

One of the challenges facing future research on meaning-adding transformations is to span all or most of the software design space with a manageable set of transformations such as the ones listed, to provide automatic procedures for applying them, and providing automatic or computer-aided procedures for decomposing manually entered design changes into sequences of primitive transformations.

Our approach is different from that of [20]. The transformations presented in [20] are characterized by their effect on the specification when viewed as a *semantic network*. This view seems to be at a syntactic, rather than semantic level. For example, several splicing transformations are given. All of them introduce a new node in a graph, between two adjacent nodes. These transformations include, among others one that adds a statement between two adjacent statements in sequential composition, and another that adds an intermediate type in a type hierarchy (between two existing types). When viewed semantically, these are completely different kinds of transformations. Moreover, the nature of the change in the particular graph gives very little insight to the nature of the change of the meaning in the specification.

In contrast, our goal is to provide transformations that clearly reflect design decision. We would like to have the designer work in the semantic level that is appropriate for the job at hand.

It is clear that the transformation in [20] are intimately connected with the specifics of Gist, the specification language used.

References

[1] G. Arango, I. Baxter, P. Freeman, C. Pidgeon, TMM: Software Maintenance by Transformation, *IEEE Software*, Vol. 3, No.3, May 1986, pp. 27–39.

[2] R. Balzer, A 15 Year Perspective on Automatic Programming, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1257–1267.

[3] D.R. Barstow, An Experiment in Knowledge-Based Automatic Programming, *Artificial Intelligence*, Vol. 12 No. 2, Aug 1979, pp. 73–119. Also appears in [27], pp. 133–156.

[4] F.L. Bauer et al., *The Munich Project CIP. Volume I: The Wide Spectrum Language CIP-L*, Lecture Notes in Computer Science 183, Springer 1985.

[5] F.L. Bauer et al., *The Munich Project CIP. Volume II: The Program Transformation System CIP-S*, Lecture Notes in Computer Science 292, Springer 1987.

[6] F.L. Bauer, B. Möller, H. Partsch, P. Pepper, Formal Program Construction by Transformations – Computer-Aided, Intuition-Guided Programming, *IEEE Transactions on Software Engineering*, Vol. 15 No. 2, Feb 1989, pp. 165–180.

[7] V. Berzins, Luqi, An Introduction to the Specification Language Spec, *IEEE Software*, Vol. 7 No. 2, Mar 1990, pp. 74–84.

[8] V. Berzins, Luqi, *Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada*, Addison-Wesley Publishing Company, 1990.

[9] J. Boyle, M. Muralidharan, Program Reusability through Program Transformation, *IEEE Transactions on Software Engineering*, Vol. 10 No. 5, Sep 1984, pp. 574–588.

[10] M. Broy, H. Partsch, P. Pepper, M. Wirsing, Semantic Relations in Programming Languages, in *Information Processing 80*, S.H. Lavington, Ed. Elsevier North-Holland, 1980, pp. 101–106.

[11] R. Burstall, J. Darlington, A Transformation System for Developing Recursive Programs, *Journal of the ACM*, Vol. 24 No. 1, Jan 1977, pp. 44–67.

[12] T.E. Cheatham, Jr., Reusability through Program Transformations, *IEEE Transactions on Software Engineering*, Vol. 10 No. 5, Sep 1984, pp. 589–594. Also appears in [27], pp. 185–190.

[13] J. Darlington, An Experimental Program Transformation and Synthesis System, *Artificial Intelligence*, Vol. 16 No. 1, Mar 1981, pp. 1–46. Also appears in [27], pp. 99–121.

[14] E.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[15] M.S. Feather, A System for Assisting Program Transformation, *ACM Transactions on Programming Languages and Systems*, Vol. 4 No. 1, Jan 1982, pp. 1–20.

[16] M.S. Feather, A Survey and Classification of some Program Transformation Approaches and Techniques, in *Program Specification and Transformation (Proceedings of the IFIP TC2/WG 2.1 Working Conference)*, L.G.L.T. Meertens, Ed., North-Holland, 1987, pp. 165–198.

[17] S.F. Fickas, Automating the Transformational Development of Software, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1268–1277.

[18] N.M. Goldman, Three Dimensions of Design Development, *National Conference on Artificial Intelligence*, Washington DC, Aug 1983. Also Tech. Rept. ISI/RS-83-2, USC Information Sciences Institute, 1983.

[19] C. Green, A summary of the PSI Program Synthesis System, *5th International Joint Conference on Artificial Intelligence*, 1977, pp. 380–381.

[20] W.L. Johnson, M. Feather, Building an Evolution Transformation Library, *12th International Conference on Software Engineering*, 1990, pp. 238–248.

[21] E. Kant, On the Efficient Synthesis of Efficient Programs, *Artificial Intelligence*, Vol. 20 No. 3, May 1983, pp. 253–36. Also appears in [27], pp. 157–183.

[22] Luqi, Software Evolution via Rapid Prototyping, *IEEE Computer*, Vol. 22, No. 5, May 1989, pp. 13–25

[23] Luqi, V. Berzins, R. Yeh, A Prototyping Language for Real-Time Software, *IEEE Transactions on Software Engineering*, Vol. 14 No. 10, Oct 1988, pp. 1409–1423.

[24] J.M. Neighbors, The Draco Approach to Constructing Software from Reusable Components, *IEEE Transactions on Software Engineering*, Vol. 10 No. 5, Sep 1984, pp. 564–573. Also appears in [27], pp. 525–535.

[25] H. Partsch, R. Steinbrüggen, Program Transformation Systems, *ACM Computing Surveys*, Vol. 15 No. 3, Sep 1983, pp. 199–236.

[26] T. Reps, T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New-York, 1988.

[27] C. Rich, R.C. Waters, Eds., *Readings in Artificial Intelligence and Software Engineering*, Morgan Kaufmann, 1986.

[28] D.R. Smith, Top-Down Synthesis of Divide-and-Conquer Algorithms, *Artificial Intelligence*, Vol. 27 No. 1, Sep 1985, pp. 43–96. Also appears in [27], pp. 35–61.

[29] D.R. Smith, G.B. Kotik, S.J. Westfold, Research on Knowledge-Based Software Environments at Kestrel Institute, *IEEE Transactions on Software Engineering*, Vol. 11 No. 11, Nov 1985, pp. 1278–1295.

[30] W. Swartou:, R. Balzer On the Inevitable intertwining of Specification and implementation, *Communication of the ACM*, Vol. 25 No. 7, July 1982, pp. 438–440. Also appears in *Software Specification techniques*, N. Gehani, A.D. McGettrick, Eds., 1986, pp. 41–45.

[31] D.S. Wile, Program Developments: Formal Explanations of Implementations, *Communication of the ACM*, Vol. 26 No. 11, Nov 1983, pp. 902–911. Also appears in [27], pp. 191–200.

DISTRIBUTION LIST

(1)	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
(2)	Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
(3)	Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
(4)	Director of Research Administration Attn: Prof. Howard Code 012 Naval Postgraduate School Monterey, CA 93943	1
(5)	Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943-5100	1
(6)	Chief of Naval Research 800 N. Quincy Street Arlington, Virginia 22217	1
(7)	National Science Foundation Division of Computer and Computation Research Attn. K. C. Tai Washington, D.C. 20550	1
(8)	Naval Postgraduate School Code 52Lq Computer Science Department Monterey, CA 93943	50