1993-05

Using Transformations in Specification-Based Prototyping

Berzins, Valdis

IEEE

Using Transformations in Specification-Based Prototyping
Valdis Berzins, Luqi, Member, IEEE, and Amiram Yehudai, Member, IEEE

Abstract—We explore the use of software transformations for software evolution. Meaning-preserving program transformations have been widely used for program development from a fixed initial specification. We consider a wider class of transformations to support development in which the specification evolves, rather than being fixed in advance. We present a new and general classification of transformations based on their effect on system interfaces, externally observable behavior, and abstraction level of a system description. This classification is used to rearrange chronological derivation sequences containing meaning-changing transformations into lattices containing only meaning-preserving transformations. This paper describes a process model for software evolution, utilizing prototyping techniques, and shows how this class of transformations can be used to support such a process. A set of examples illustrates our ideas. Software tool support and directions for future research are discussed.

Index Terms—Executable specification, program transformations, software evolution, software prototyping.

I. INTRODUCTION

SOFTWARE transformations have been studied extensively in the contexts of program synthesis and program optimization. These efforts have sought to derive efficient implementations from nonconstructive specifications or inefficient programs via meaning-preserving transformations. Transformations provide a potentially economical road to reliable software: if a library of transformations can be certified to preserve program behavior and can be applied reliably via trusted software tools that check all of the applicability conditions for the transformations, then any derived program is known to conform to the original specification. Transformational implementation can thus eliminate explicit verification if the underlying system software can be trusted to operate as specified.

However, accurately specifying the desired behavior of a software system before the implementation is developed can be quite difficult in practice. Constructing correct specifications is hard because a set of informal ideas must be turned into a formal model via incomplete and imprecise communication between people. A validation process is needed because correctness of the software depends on whether the original specification corresponds to the customers' real needs, in addition to whether the implementation conforms to the specification. Executable prototypes of the specification are useful for obtaining user confirmation that a proposed specification correctly represents the problem, and for guiding the reformulation of the specification in cases where it misrepresents the problem. Prototyping is most effective if the scenarios for demonstrations are carefully chosen to expose the most likely weaknesses of the requirements. Methods for doing this are beyond the scope of this work. We focus on the evolution of proposed specifications and prototype designs.

The purpose of this paper is to explore how software transformations can support formulation of requirements based on software prototyping. The prototyping process repeats a guess/check/modify cycle until the users agree that the demonstrated behavior is acceptable. Most of the modifications seek to change the behavior of the system to reflect new requirements, rather than to preserve the behavior while improving efficiency. We therefore emphasize transformations that change the behavior of a system.

Transformations can enhance the prototyping process by capturing the conceptual dependencies in a design. The derivation of a specification exposes the structure of the design decisions leading to the proposed system, which can be used to record and guide systematic exploration of the design space. Such a representation is necessary if we are to develop software tools for managing this process and extracting useful information from the design history. Such tools should help coordinate the efforts of analysts and designers faced with a changing set of requirements, to avoid repeated effort and inconsistent parallel refinements.

The rest of the paper is organized as follows. Section II describes some previous work on program transformations, and suggests some classes of transformations for supporting the prototyping process. Section III discusses rapid prototyping, and describes the role of transformations in the prototyping process. Section IV introduces some notation, and illustrates our ideas with an example. Section V discusses tool support for the methodology, and outlines some research problems that must be tackled to make our approach practical. Section VI contains conclusions.

II. SOFTWARE TRANSFORMATIONS

A. Previous Work

Program transformations have been studied extensively since the 1970's. This section briefly reviews some of the pre-
vions results relevant to our goals. A comprehensive overview can be found in [30], [18].

A transformation is a function that maps programs or program schemes into other programs or program schemes. The domain and range of a transformation are usually assumed to be the same formal language, which may be a closure of the union of several simpler languages.

Most work on transformations assumes that the transformed program $P'$ must meet the requirements for the original program $P$. Such transformations are referred to as meaning-preserving transformations or correctness-preserving transformations. The most common meaning-preserving transformations require $P$ and $P'$ to be equivalent. Other suitable semantic relations include weak equivalence (where $P'$ must be equivalent to $P$ only in cases where $P$ terminates), and inclusion (where the possible results of $P'$ must be a subset of the possible results of a nondeterministic program $P$). These relations have been used in a study of formal semantics [12]. Two principal kinds of meaning-preserving transformations have been distinguished: vertical (going from a higher abstraction level to a lower one) and lateral (producing a result at a similar level of abstraction) [34, p. xv].

Transformation systems have been used to achieve a variety of goals. The first use of transformations was based on a natural tradeoff between efficiency and clarity in most programs. It is often possible to write a self-evidently correct program, but the resulting program may be very inefficient. Lateral transformations have been used to convert a clear but inefficient program into an efficient one [13], [15].

Transformations have also been used to construct a program from a formal specification. For example, the CIP project in Munich [6], [4], [5] treats program development as a formal process that transforms the problem specification into an efficient program for the target machine in a discrete series of steps. Each step in this process corresponds to the application of a meaning-preserving transformation. 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. This language may contain several different sublanguages for representing programs at different levels of abstraction. Program representations may include black-box specifications, dataflow decompositions, functional programs, imperative programs, or even machine code, depending on the nature of the application.

Transformational programming is usually supported by an appropriate programming environment. The level of support may vary from full automation to merely assisting the user in selecting and applying transformation rules. Existing transformation 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 [11] and PDS [14] 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 [22] 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 [24] 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 [35]. A key focus of these efforts has been an attempt to automate the choice of transformations as much as possible [17], [20], [38].

The sequence of transformations chosen to implement a program is a valuable record of the development process [38]. It should be possible to replay at least part of these transformations when the original specification is modified. This idea motivates work on applying transformations to software maintenance. A method called the transformation-based maintenance model (TMM) was proposed in [1]. The model is based on the assumptions that a program has been derived from a specification using a transformation system, and that the sequence of transformations applied has been recorded. This sequence may be viewed as a path from the source of a directed acyclic graph (representing the specification) to a terminal node (representing the current program). A design decision is made at each internal node, resulting in the application of a particular transformation. The structure is a directed acyclic graph rather than a tree because it is often possible to reach a node by more than one path. In the maintenance process, we can traverse the derivation path backwards, and change some of the decisions to choose different transformations. TMM takes advantage of some specific properties of the particular transformational programming paradigm used in Draco [28].

B. Characterizing Transformations

If we wish to take a transformational approach to the specification phase of software construction, we must allow for changes in the semantics. To quote [37].

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.

Real software systems and their operating environments are complex, and the full impact of a proposed system on its environment is very difficult to predict. Constructing specifications for such systems involves an inherently uncertain process of human communication that requires feedback from prototyping to achieve stability and customer satisfaction.

The goal of our work is to lay the foundation for a transformational approach in software construction that does not assume a fixed and completely correct initial specification. Instead, we expect requirements and specifications to be modified based on experience with prototype implementations in an iterative process that gradually converges on
appropriate requirements. A key question in such a context is how to ensure the appropriateness of transformations that are intended to change the meaning of a specification and design. This section characterizes some useful types of meaning-changing transformations, and suggests some restrictions on these transformations that provide a useful structure for validating a derivation relative to user feedback derived from demonstrations of prototype behavior. The characterizations are expressed in terms of the event model of computation, which covers a wide range of applications and provides a language-independent reference point. A specific language (Spec) based on this model is used in the examples in Section IV.

1) The Event Model: The event model is an extension of the actor model that can represent real-time systems as well as concurrent and distributed systems [9]. The primitives of the simplified event model\(^1\) used in this paper are modules, messages, and events.

A module is a black box that interacts with other modules only by sending and receiving messages. Modules can represent software systems, such as Ada tasks or packages, as well as people and hardware devices.

A message is a data packet that is sent from one module to another. Messages are classified into message types based on the name of the message and the signature of the associated data. For example, each overloaded variant of an Ada task entry corresponds to a message type, as does each variant of a function or procedure declared in an Ada package specification. Each call of such an entry or subprogram corresponds to an individual message. Messages can also be realized by other mechanisms, such as I/O and exceptions.

An event occurs when a module receives a message at a particular instant of time. Every event has a target module, an arriving message, and an occurrence time, and each event is uniquely identified by these three attributes. Events represent both stimuli and responses, and serve as reference points for timing constraints.

A computation history (or trace) is a set of events that is partially ordered by a causes relation. The causes relation connects each stimulus event to each of the response events caused by the stimulus.

Requirements expressed in terms of the event model are constraints on the causes relation for all possible traces. Most of these constraints\(^2\) have the form “every event satisfying pre must cause a corresponding event satisfying post,” where the precondition pre and the postcondition post are predicates on events and states. The state of a module is the set of all previous events at the module. The events occurring at each module are totally ordered by their occurrence times.

System structures are represented in the event model via the contains relation. The contains relation connects each module to each of its subcomponent modules. The contains relation for a hierarchically described system specifies its internal structure relationships. In particular, the contains relation distinguishes the modules inside a system from those outside the system. The

\(^1\)The full event model has an additional primitive for modeling temporal events.

\(^2\)Synchronization constraints have a different form.

image under the contains relation is empty for any module that is a primitive at the chosen level of abstraction (see granularity below). A system that is primitive at one level can be viewed at lower levels of abstraction by introducing its subsystems and specifying the interactions between the system and its subsystems. A trace at the higher level of abstraction can be recovered from a lower level trace by removing all events at the subsystems as well as the events at the decomposed system that consist of messages arriving from its subsystems.

2) Classification: We characterize transformations in terms of these orthogonal attributes of a program: its vocabulary, its behavior, and its granularity. These attributes can be formalized using the event model as follows.

- The vocabulary of a program is the set of all external stimuli recognized by the program. An external stimulus is a message that is received by the program and originates from (is caused by an event at) a module outside the program. The vocabulary is usually infinite, and can be finitely represented as the union of a set of message types. The set of message types recognized by a program can be made finite via a suitable representation for generic message types (e.g., generic subprograms declared in Ada packages). The vocabulary of a program represents the set of functional capabilities provided by the program.

- The granularity of a program is the set of all internal stimuli recognized by the program. An internal stimulus is a message that originates from a module inside the program and is received by a module inside the program. The granularity represents the amount of detail in which the computation has been specified. At one extreme is a black-box specification, which does not mention any internal stimuli at all. At the other extreme (if we avoid the hardware level) is machine code, which specifies internal events corresponding to individual machine instructions.

- The behavior of a program is the set of all possible traces for the program relative to a given vocabulary and granularity. The behavior of a program contains each of the possible responses of the program to each stimulus in the given vocabulary and granularity. The behavior is usually infinite.

Note that each of these three attributes is a set. Our thesis is that each meaning-changing transformation should be decomposed into substeps, each of which preserves two of the three attributes and makes a monotonic change (either $\subset$ or $\supset$) in the third. The part of the behavior that must be preserved by transformations that change the vocabulary or abstraction level is determined by the intersection of the vocabularies of the initial and transformed programs, and the intersection of the granularities. This says that transformations that add new message types or remove previously defined messages should not affect the behavior of the system with respect to the other previously defined message types. The proposed restrictions lead to the classification of primitive transformations shown in Fig. 1. The symbol $S_P$ represents the attribute $S$ of the original program $P$, and $S_{P'}$ represents the attribute $S$ of the transformed program $P'$.

Extending and constraining transformations are meaning-preserving, given our convention that each primitive transformation preserves two of the three attributes. Abstracting and
refining transformations are meaning-preserving if we do not consider performance requirements. Contracting and relaxing transformations are, in general, meaning-removing, and can be used to construct meaning-changing transformations when combined with the other types of transformations. There are two ways to add information to a design: by adding message types via extending or refining transformations which add elements to the vocabulary or granularity, and by adding new constraints via constraining transformations which reduce the set of legal behaviors.

These restrictions enable the rearrangement of a sequential derivation containing meaning-changing transformations into a tree-like rooted directed acyclic graph whose paths consist solely of meaning-preserving transformations that add information via compatible extensions, constraints, or refinements. The requirements at the root of the graph can be derived from the oldest set of requirements in a chronological derivation history by deleting all parts that were contradicted in later versions. Each path in the graph represents a series of refinements of the requirements, and branching points represent design decisions. The benefit of the proposed rearrangement is to identify design variations that were explored and later abandoned, and to factor them out of the actual chronological derivation, to expose a clear path to the final formulation. The requirements at the root of the derivation graph are set of the previous version that can be reused for the new version.

We propose this mechanism as a concrete means to document software as if it had been developed using a rational process [29], and we conjecture that such structures will be useful for choosing demonstration scenarios, guiding requirements reviews, and summarizing past history for analysis formulating the next version. The early parts of the development, in which the requirements are evolving, must be guided by people because the transformations add information to the requirements in a creative process that involves formalizing informal desires and criticisms. This makes it unrealistic to expect that the real chronological derivation can be composed only of meaning-preserving transformations because that would require the analysts never to make any mistakes in an activity that is dominated by educated guesswork and experimental validation. It is also unrealistic to expect that the modifications can all be accomplished merely by returning to a previous version and making a completely new refinement because most of the mistaken transformations must be only partially undone: skilled analysts guess right most of the time, and often only a relatively small part of an imperfect refinement must be undone. The transformation that undoes part of an information-adding refinement materializes a new version of the system, which did not appear earlier in the chronological derivation. Such a version is not explicitly constructed by the designer, who usually makes a single incompatible change that corrects the error, rather than first removing the faulty decision and then making a new refinement. Automated support for the proposed rearrangement is thus needed to gain the well-established benefits of meaning-preserving transformations prior to the point where the formalized requirements can be assumed to completely capture user needs since we do not expect analysts and designers to accept new working styles that require them to spend more effort to accomplish the same end.

The requirements at the root of the derivation graph are usually not strong enough to meet all of the user's needs, although they are consistent with those needs. The requirements get increasingly restrictive along each path in the derivation, and each point along the path satisfies all of the requirements at preceding points. Parallel paths represent alternative formulations of the requirements that are incompatible with each other. The purpose of exploratory analysis is to find a path to a version of the requirements that does meet the users' needs. The final requirements need to be validated once they are found, even though they have been derived from the root of the graph via meaning-preserving transformations because the root requirements are usually too weak to meet all of the users' needs. We believe that the intermediate points in the path are useful for the validation because the differences between neighboring points in a path are relatively small and can be checked independently of each other. Once the path is validated, its endpoint can provide a stable and reliable starting point for implementation via meaning-preserving transformations applied in the conventional way. We note that a substantial amount of research and development is needed to support such a process in practical contexts. In particular, work is needed to automatically identify the parts of a specification left invariant by a meaning-changing transformation, and to map this into the parts of the transformational implementation of the previous version that can be reused for the new version.

3) Examples of Transformation Types: We illustrate these concepts via examples.

- Extending transformations add new types of messages to a program. For example, adding a new command to a graphical editor is an extending transformation because it creates a new message type. This kind of transformation corresponds to the creation of a new requirement, unrelated to the requirements for the previous version of the program. Some subtler extending transformations extend previous message types. This can be done by making previously required input parameters optional, by introducing new optional input parameters, and by replacing the type of an input parameter by one of its supertypes. These types of transformations generally extend previous requirements by analogy. Some examples are extending a maximum function from a binary operation to an operation that acts on an arbitrary number of parameters, e.g., max(2, 5, 8, 3) = 8, and extending a mod function from mod(x: integer, y: integer) to mod(x: number, y: integer), e.g., mod(3.14159, 2) = 1.14159. On a larger scale, extending transformations can expand the set of stimuli by...
adding new modules. Such modules can represent subsystems with completely new purposes. A different example transforms an abstract data type, which is an individual module in the event model, into a family of abstract data types by adding generic parameters.

- **Contracting transformations** are inverses of extending transformations: they remove possible types of interactions from a program. A message type may be dropped because it is deemed useless by customers. More often, such a change may be a prelude to an extending transformation that will extend the system in a different way to meet a requirement that has been substantially redefined. An example of a contracting operation that adds information is the transformation of an event list into a set of message types. Since an event list specifies message names but not the types of input parameters, this kind of transformation limits the set of stimuli by replacing an undefined type signature with a specified type signature. An undefined type signature is consistent with all possible sets of actual parameters. A different use of contracting transformations is to restrict the set of stimuli in order to support a more efficient implementation. For example, the generic lookup table type map{domain: type, range: type} can be contracted to map{domain: ordered_type, range: type} by restricting a generic type parameter to a subtype. This transformation enables efficient balanced-tree implementations by requiring the domain type to supply an ordering operation. A related example transforms an operation such as divide(\(x:\) integer, \(y:\) integer) into a partial operation divide(\(x:\) integer, \(y:\) integer SUCH THAT \(y > 0\)), thus simplifying implementation and eliminating an exception condition at the expense of introducing a precondition that must be guaranteed by the context of each call.

- **Constraining transformations** restrict the behavior of a program by placing constraints on legal responses to a stimulus. For example, adding a postcondition to a message specification consisting only of a type signature is a constraining transformation, as is strengthening a postcondition (replacing the postcondition \(P\) with \(P'\) where \(P' \Rightarrow P\&P \not\Rightarrow P\')). Constraining transformations reduce nondeterminism. We are mostly concerned with nondeterminism due to incomplete specifications, although valid implementations that retain some nondeterminism in the code are possible. Such nondeterminism may arise from concurrency, from the use of nondeterministic programming language constructs such as guarded commands [16], or from implementations of abstract data types that do not have unique representations for each abstract value.

- **Relaxing transformations** are inverses of constraining transformations. They remove some restrictions on the behavior that exists in the previous version. An important class of relaxing transformations is applied because the more restrictive requirement is not logically satisfiable, cannot be implemented with existing technology, or cannot be realized within given constraints on budget and computing resources. An example is relaxing a 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. This transformation is needed because perfect control of physical systems is beyond existing technology. Another example is relaxing a postcondition for the function square_root(\(x:\) real) \(\Rightarrow (y:\) real) from \(y^2 = x\) to \(approximates\ (y^2, x)\). This transformation is necessary because exact square roots do not always exist for machine representations of real numbers. A different application of relaxing transformations is removing a postcondition as part of replacing a previously proposed requirement with an incompatible alternative formulation during an exploratory design process. This situation is a consequence of our recommendation that each nonmonotonic change in system behavior should be decomposed into a relaxing transformation followed by a constraining transformation. Relaxing transformations can also be used to enable efficient implementations. For example, optimization problems are sometimes relaxed so that results within a given tolerance of the global optimum are accepted.

- **Refining transformations** constrain the internal details of the computation specified by a program without changing its vocabulary or its externally visible behavior. Such a transformation may decompose a module into a network of submodules or choose algorithms and data structures for implementing a module. In the terminology of [34, p. xv], these are called **vertical** transformations. They occur often in the literature about transformations systems for program construction, such as CIP, where they have been called correctness-preserving. Refining transformations are used primarily to make nonconstructive specifications executable.

- **Abstracting transformations** are inverses of refining transformations. Abstracting transformations abstract away some details of a computation, without affecting the vocabulary or the externally visible behavior. They may occur in reverse engineering processes, such as the TMM approach to software maintenance [1]. Explicit abstracting transformations are not needed very often in software development because refinements are usually reversed completely if an alternative implementation is to be explored. Thus, if the sequence of versions produced by a derivation is stored, the effect of a refinement can usually be undone by restoring a saved previous version, without the need to apply a reversing transformation explicitly. Refinements can also be modified directly by meaning-preserving transformations, thus reducing the need for explicit reversals.

4) Comparisons:

**Meaning-Changing Versus Meaning-Preserving Transformations**: Meaning-changing transformations are different because they change the requirements. This implies that validation, and hence explicit explanation and understanding are needed. We contend that the derivation history helps formulate understandable explanations. The explanation of the original requirements is often similar to the explanation of the transformed requirements because requirements changes tend to be localized, so that the part of the explanation corresponding to the invariant part of the requirements remains the same. One of the reasons we suggest separating nonmonotonic changes into contractions and reexpansions, or relaxations and new constraints is that this process identifies which parts of the requirements have remained the same, which constraints have been removed, and which constraints have been added. A review process can examine the constraints that have been
removed, compare them to the constraints serving as the replacements, and ask why this was done and whether it makes sense.

Explanations of meaning-changing transformations differ from explanations of meaning-preserving transformations because the relation between the two versions is usually not one of simulation: the meanings are incompatible, and have been deliberately changed in response to a perceived deficiency. Therefore, the principles of operation of the two versions may be quite different. However, both versions of the requirements are usually different attempts to satisfy the same higher level goal. For example, one purpose of flight plans is to make the motions of airplanes predictable so that suitable restrictions on the approval process for the plans can keep airplanes from colliding. Such a goal must be identical in validating a change such as the transformation that introduces an error tolerance for the distance an airplane is allowed to stray off its course, and the new formulation must be checked against the higher level goal. Such checking may also suggest additional constraints, such as limits on the error tolerance.

Comparison with Algebraic Ideas: In our approach, we define the semantics of a software system entirely in terms of observable behavior. This differs from the usual practice in studies of algebraic specifications for data types, where the semantics of a type is considered to be characterized by the set of models. The algebraic approach is a viable alternative which we have not followed because algebraic specifications with different sets of models can have the same observable behavior. Two models can differ in their identity relations on the values of the types, which affects the algebraic structure of the models, even though these structural differences may not be observable by any computation. Even if we exclude algebras with elements that are not finitely constructible in terms of the operations in the signature, it is possible for one algebra to have several distinct elements that cannot be distinguished by any computation over the signature. In such a case, there is an equivalent model in which a single element represents all of the distinct but indistinguishable elements of the first model. The structures induced by the presence or absence of such distinctions are of theoretical interest, but they do not affect requirements analysis; any two models that are behaviorally indistinguishable are completely interchangeable from the point of view of the user and the requirements analyst. We prefer to work in the simplest possible framework, which contains only the features required by the problem at hand. In the context of this paper, we are concerned with the formulation of valid requirements, and consideration of the range of possible models is not necessary in that context, although consideration of the range of possible behaviors is necessary.

The distinction between an incomplete specification that admits only functional behavior but is consistent with several behaviorally distinguishable functions and a specification that admits truly nondeterministic behavior in addition to the same set of functions has also been extensively studied in the context of data algebras. This distinction is sometimes of importance in requirements analysis, when an operation is required to be repeatable, but the analyst does not want to unnecessarily constrain the specification to leave implementation freedom for the designer. For example, a hash function must be repeatable to be useful. The requirement that the response to a stimulus be determinate can be formulated on behavioral grounds in the event model, and a transformation that adds such a requirement to an incompletely specified interaction is a constraining transformation in our framework because it rules out behaviors with different responses to two instances of the same stimulus. Consideration of algebraic models is not necessary to describe such a constraint.

Comparison with the Gist Approach: Our classification of transformations is similar in spirit to the conceptual view presented in [21]. The two formulations are based on different computational models. The Gist project is concerned with describing behavior prior to defining system boundaries, and hence uses a global state model in which system behavior is represented as a sequence of states. Such a model does not represent parallel activity directly. Our work has been formulated using the event model, which is inherently parallel and distributed, but which requires system boundaries to be determined before system behavior is defined. This difference makes detailed comparisons somewhat difficult. We agree with the author's main thesis: it is best to present a complex specification in terms of its derivation from a series of simpler ones.

The Gist work [21] characterizes decisions in the development of a specification along three dimensions: structural granularity, temporal granularity, and coverage. These dimensions are defined informally and via examples, and are not used to define an explicit classification of transformations. Our classification is based on set-theoretic properties of the event model, and rests on a formal structure. Structural granularity deals with the amount of detail of the proposed process that is encoded in the state model. This dimension corresponds roughly to our vocabulary attribute, which determines the properties of a system represented in the model. The vocabulary corresponds directly to the behavioral properties of a system, and applies to modules representing the environment as well as to modules representing the proposed software. Structural granularity does not distinguish between properties that interact with the software and properties of the environment that may be relevant to the requirements but are not directly observable by the software. The event model represents only observable properties of systems. The specification language Spec represents both kinds of properties, and distinguishes between them. Observable properties are represented by Spec messages, and are defined using the event model. Nonobservable properties are represented by Spec concepts, and are defined using logic, independently of the event model. A meta-vocabulary can be defined based on concepts, analogous to the way the vocabulary is defined based on messages. We note that the meta-vocabulary is a property of the description of the system, rather than a behavioral property of the proposed system itself. There is a class of reformulating transformations significant in requirements analysis, whose purpose is to replace descriptions based on the meta-vocabulary with equivalent formulations that are expressed in its vocabulary, and hence depend only on observable properties...
of the proposed system. It is straightforward to develop a classification for meta-transformations that is analogous to the one proposed in Fig. 1. However, we prefer to clearly separate properties of the proposed system from properties of its description. Temporal granularity deals with the amount of change between successive states revealed by the specification. This dimension corresponds roughly to our granularity attribute, which determines how many of the internal events of a computation appear in the model. Our granularity attribute is directly related to the decomposition structure in a proposed software design via the contains relation. Because of the difference in underlying models, our formulation extends this concept from sequential systems to parallel systems. Coverage deals with the range of behaviors permitted by a specification. This dimension corresponds roughly to our behavior attribute.

The work on an evolution transformation library in ISL [23] shares some of the premises that were the basis of our work. In particular, transformations that do not preserve meaning are used to capture the evolution of software specification. Our approach differs from that of [23] in the way we view transformations. We focus on the semantic effects of transformations on system interfaces and system behavior, and base our classifications on such properties. The work on the transformation library has described transformations in terms of the semantic networks describing the proposed system rather than properties of the proposed system itself. A substantial portion of the semantic networks involved are syntax trees for the specification language, and the effects of the transformations are described largely in terms of the meta-vocabulary and the structure of the description rather than the vocabulary and behavior of the proposed system.

III. PROTOTYPING VIA TRANSFORMATIONS

A. The Prototyping Process

Prototyping enhances communication with the user community by providing an executable model of the system early in the development process. Early feedback from the user community leads to software systems that are more likely to meet user needs, and reduces life cycle costs because changes made at the early stages of development are much cheaper than changes made after the system has been delivered. Prototyping can also be useful for streamlining software evolution. In this section, we discuss specification-based prototyping.

There are two phases in our model of the prototyping process, prototype evolution and production code generation [25]. The purpose of prototype evolution is to stabilize the software requirements before effort is invested in detailed implementation and optimization. The purpose of production code generation is to generate an efficient implementation when the requirements are stable. If there is a need to modify the requirements after delivery, we can return to prototype evolution, followed by another instance of production code generation. The prototyping process is illustrated in Fig. 2.

The prototype evolution phase consists of the activities labeled "analyze requirements," "construct prototype," and "execute prototype." The process starts with requirements analysis to determine an initial version of the requirements.

Next, a prototype is constructed based on the requirements. For complex systems, this process usually requires decomposition of the system into simpler subsystems. In our approach, a prototype consists of a hierarchy of modules, where each module has a behavioral specification augmented with optional implementation information, such as a decomposition or a reusable program. The prototype execution activity demonstrates some typical cases of prototype behavior and generates a series of prototype adjustments based on the customer's quick feedback. This feedback is used in the requirements analysis activity to produce an updated set of requirements and trigger the next prototyping cycle. The analysis traces cases of unwanted behavior to identify incorrect or incomplete requirements, and proposes one or more plausible ways to modify the requirements. The next round of prototype execution tests the validity of the proposed changes, provides guidance to choose between alternative formulations, and explores previously unvalidated aspects of system behavior. This process continues until the requirements have been thoroughly exercised and the customers are satisfied with the demonstrated behavior of the prototype. The results of the prototype evolution phase are formal specifications for the proposed system, and a system decomposition that includes formal specifications for the subsystems.

The production code generation phase consists of the activities labeled "verify structure" and "implement (optimize)." The deliverable version of the envisioned system is constructed based on the specifications, decomposition, and other attributes determined in the prototype evolution phase. The verification of the structure of the prototype is an optional process whose purpose is to prevent integration problems, especially in cases where different subsystems are produced by different groups or subcontractors. This verification seeks to prove that the proposed decomposition of a system will meet the specifications for the entire system whenever the subsystems identified in the decomposition meet their specifications. If the decompositions are developed entirely via meaning-preserving transformations whose application has been checked by reliable software tools, then this verification step may not be necessary. However, in the near term, it is rational to assume that the analysis and design process is potentially imperfect. We therefore expect the highest levels of the decomposition to be verified before the decomposition is used as a starting point for detailed implementation, especially if different subsystems are to be implemented by different contractors. This process ensures
consistency between the specifications at all levels of the decomposition.

The implement/optimize activity produces efficient implementations of the subsystems. Optimization via traditional meaning-preserving transformations is one of the methods for carrying out this part of the process. If the optimization needs to alter the verified decomposition, these changes should be carried out via mechanically checked meaning-preserving transformations. This should also be done for critical aspects of system behavior, for which software failures can have very serious consequences. The individual bits of reasoning involved in a formal transformational approach are simpler and more tractable than those required to verify an optimized implementation directly. This approach also avoids problems associated with attempting to verify implementations that have bugs.

Our procedure for realizing a prototype is illustrated in Fig. 3, which provides an exploded view of the “construct prototype” box in Fig. 2.

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:

- Convert the specification into the executable subset of the Spec language. This step may be trivial if the original specification 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 conversion into executable form must also be feasible. In cases where the conversion is too difficult for an automatic process, we can use one of the methods listed below to realize the specification.

- Produce code in a programming language such as Ada. This can be done by retrieving and adapting reusable components based on the specifications, or by creating new code, either manually or via meaning-preserving transformations.

- 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 [27]). 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 frequently executed 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).

A key issue for realizing the benefits of prototyping in practice is rapid and correct construction and modification of prototypes via computer-aided tools such as the Computer Aided Prototyping System [26].

A prototype demonstration often results in user requests for adjustments to the behavior of the prototype. These adjustments can be classified as Spec adjustments, which modify the specified behavior of a module in the prototype, and as structural adjustments, which rearrange the modules in the prototype 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 lower levels. One of the goals of computer aid for prototype evolution is to help propagate intended changes from the highest level of the structure to the lower levels, and to ensure that this propagation is complete. Some transformations achieve one goal at the expense of destroying the integrity of the design, so that other transformations must follow to fix things up. Transformations should be packaged together in transactions that guarantee the design is free from some classes of faults, such as references to undefined variables. This is illustrated in Section IV in the context of a transformation that changes an input to a state variable.

The structure of the prototype must also be periodically simplified to maintain intellectual control and enable future modifications to be carried out with speed and accuracy. This cleanup function involves restructuring the prototype, and removing old features that are no longer needed to support current requirements. It can be considered part of the “modify prototype” activity shown in Fig. 3. This process is generally done between demonstration sessions, based on design reviews or computer-aided dependency analysis rather than on customer feedback. The next section discusses how software transformations can support the construction and evolution of prototypes. Tool support is discussed in Section V.

B. The Role of Transformations

The initial formulation of a prototype may involve contracting and relaxing transformations to the original requirements. These transformations are used to simplify the problem and speed up the prototyping process by ignoring less important stimuli, or to relax some of the requirements for stimuli that are represented. Prototyping usually focuses on critical subsystems or particular aspects of the system that cause
uncertainty. For example, if human factors are dominant, then formatting requirements may be preserved, while requirements on system semantics may be relaxed for the prototype. Partially developing and then relaxing some of the requirements is often preferable to delaying the elaboration of the less critical requirements because different aspects of a system are rarely completely independent, although they may be weakly coupled. Access to the complete requirements is sometimes necessary to make sensible choices in a prototyping effort, even if some of those requirements are not intended to be realized by the prototype.

The prototype evolution phase is dominated by a series of nonmonotonic changes to the behavior of the prototype. These changes are realized via contracting and extending transformations or via relaxing and constraining transformations, as illustrated in Section IV. Meaning-preserving transformations are applied at this stage mainly for adjusting the structure of the prototype to make it easier to understand or modify, and to completely or partially implement nonconstructive specifications. Improving efficiency is a major goal only if feasibility of hard real-time constraints must be established or if prototype demonstrations take impractically long to run.

We illustrate some additional uses of meaning-changing transformations to realize nonconstructive specifications. One of the ways to carry out the "product Ada code" activity shown in Fig. 3 is to retrieve a reusable software component from a software base. Relaxing transformations are useful in this context. If a component that satisfies the specification given by the designer cannot be found, the software base management system can make a query broader by applying relaxing transformations to the specification. Some candidates are relaxing transformations that drop some of the postconditions. For example, if the output of an operator must be a sequence that contains all elements satisfying a given set of constraints, and which must be in monotonically increasing order with respect to a given ordering, the software base management system can seek modules that satisfy only the first constraint, or only the second constraint. The original specification and the two relaxed specifications are shown in Fig. 4.

In this case, the retrieval will succeed for the second relaxed specification, yielding an instance of a generic module that sorts a sequence with respect to an ordering defined by a generic procedure parameter. The retrieved module can then be used to suggest a realization of the original design via decomposition. This example is an instance of a filter decomposition, which succeeds because the postcondition of the first relaxed specification is invariant under the function defined by the second relaxed specification. Strategies for meeting complex goals via operations that interfere with some subgoals, such as those developed in the context of robot task planning, can be useful for enhancing this approach.

Fig. 4. Relaxing transformations for approximate component retrieval.

Matching relative to extending and contracting transformations is also useful for retrieving reusable software components. Accepting stored components with supertypes for specified arguments, or with optional parameters that were not specified in the query are some simple examples of useful extending transformations. Retrieving partial operations with the same postcondition is a useful contracting transformation that results in a partial operation. This application of transformations differs from the use of relaxing transformations outlined above because the transformations extend the matching criterion, rather than modifying the query, as did the relaxing transformations above. In applications to matching, the transformations are constructed as part of the matching process, rather than being given a priori. The transformation that enables the match can contribute to the synthesis of the design. For example, the contracting transformation used to establish a partial match identifies a domain predicate for the partial operation, which is then used as a goal for the partial operation in a modified decomposition. In general, contracting transformations be applied both to the query and to the stored component. The stored component may be a total operation that does not satisfy the query, but it may have a partial subfunction that satisfies a contraction of the query. An example is using a remainder function to partially satisfy a query seeking a modulo function: the two are identical for nonnegative arguments, but differ for negative arguments. The result of the query is a contraction of the remainder operation that is limited to nonnegative arguments via a synthesized guard predicate.

Reformulating transformations are also useful for speeding up the specification matching process. These transformations realize several different normal forms that support signature indexing, fast semantic elimination procedures, and implication checking [36].

Constraining and relaxing transformations are useful when the designer discovers that it is possible to implement a stronger version of a component than was required by the original design. A simple example comes from the timing constraints associated with a time-critical operator. Suppose that the original design sought an operator with a maximum execution time of 100 ms, and a software base retrieval located an implementation for the operator with a maximum execution time of only 23 ms. The designer is likely to change the design to require a response for the operator in 23 ms, thus performing a constraining transformation by replacing a loose timing constraint by a tighter one, as illustrated in Fig. 5.

In this particular application, the constraining transformation can be followed by a relaxing transformation that reallocates computation time to some components that are yet to be implemented, making it easier to find implementations for those components. For this example and for other cases involving simple numerical constraints, the associated transformations can be efficiently implemented using a constraint maintenance
system. More difficult examples include cases where the retrieved component satisfies stronger behavioral properties than were requested by the designer. Such components may subsume functions allocated to other parts of a decomposition, if these additional properties were in fact desired by the designer. If this is not the case, and the requirements were initially incomplete, then such a retrieval may suggest corresponding constraining transformations on the specifications. This, in turn, suggests new requirements to be evaluated by customers in a prototype demonstration session.

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 meaning-preserving transformations for optimizing the design and implementation (the “Implement” activity of Fig. 2). Meaning-preserving transformations have been studied extensively [6], [4], [5], [20], [24], [22], [38]. However, meaning-changing transformations are sometimes also needed to optimize a design.

Efficient algorithms are often applicable only in special cases, so that their use may constrain the set of problems that can be solved. This makes the operation partially defined. We do not want to leave such an operation partial (the result of a contracting transformation) because that would put the burden on the user to avoid inputs outside the domain of the function. We prefer to remove all constraints on the behavior of the operation outside the domain of the efficient implementation (via a relaxing transformation), and then to reconstrain the specification by defining safe 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. Such an optimization adds a class of potential overflow errors to the specification, or at least changes the circumstances under which an overflow error will occur. Since meaning-changing transformations can change the observable behavior of the system, they require a validation step, possibly via an additional prototyping cycle focused on demonstrating the proposed change.

IV. CASE STUDY

This section illustrates our ideas via a simple case study, after a brief explanation of our notation.

A. Notation

In this section, we represent specifications using the Spec language [8]. Spec is a formal notation for expressing black-box descriptions of system behavior that can be applied to both the external interfaces of a system and to internal interfaces introduced by decomposition.

Spec is based on the event model of computation (see Section II), and uses (second-order, temporal) predicate logic for the precise definition of desired behavior. The emphasis in the design of the Spec language was to provide ease of expression to the analyst. The language includes unrestricted quantifiers, and provides a mechanism that allows users to define new quantifiers. The impossibility of executing a language with such powerful constructs has been resolved by requiring only a subset of the language to be executable. This choice was made because execution is not the only purpose of a specification. Sometimes it is necessary to reason about infinite processes: for example, to establish the accuracy of an approximation that is introduced to turn an intractable infinite process into an implementable finite one. Representations of the infinite processes defining the ideal goals are necessary to support such reasoning.

The event model underlying Spec extends the familiar precondition/postcondition style of specification to concurrent, distributed, and real-time systems in a natural way. Spec combines this model with language features supporting applications to complex systems, such as controlled name spaces. The most important ideas of this language are modules, messages, events, localized state models, atomic transactions, parameterization, inheritance, and defined concepts. The examples in this paper use only a small subset of the Spec language. A complete description of the language and larger examples of its use can be found in [9].

In Spec, modules are classified as functions, machines, and types. Modules represent systems that can be realized by any combination of software, hardware, and people. System behavior is defined using Spec MESSAGE declarations. Each MESSAGE declaration defines the required responses for all events in which a message of the declared form arrives at the module. A response contains a set of outgoing messages that correspond to required future events. Responses of modules with internal states can also include an optional state transition, which is defined via the local state model. An event can have several different responses that are guarded by preconditions. Requirements on the contents of outgoing messages and the next state of the module are defined by postconditions. Preconditions and postconditions are logical assertions marked by the keywords WHEN and WHERE, respectively. For modules with internal states, the part of the postcondition specifying the requirements on state transitions is separated and marked with the keyword TRANSITION to improve readability and to syntactically distinguish intended state transitions.

In this paper, we use Spec to define the required behavior of interfaces. We use augmented data flow diagrams to describe the interconnection between the Spec modules in a decomposition. This notation is from the prototyping language PSDL [27], and is easily readable without further explanation.

B. Example: Spelling Checker

We now illustrate the use of transformations in the evolution of a prototype for a spelling correction system, emphasizing the role of meaning-changing transformations. The initial focus of the prototyping effort is on the required behavior of the
be interacting with the proposed software through a single models that represent the required information without regard
The initial specification is expressed in terms of abstract data interface, as illustrated in Fig. 6, and results in the initial the problem are the main contributions of the initial analysis.

system rather than on display formats and human factors issues.

The initial requirements analysis determines that a user will
be interacting with the proposed software through a single interface, as illustrated in Fig. 6, and results in the initial
specification for the behavior of the proposed software given
in Fig. 7.

Identifying and modeling the aspects of the data relevant to
the problem are the main contributions of the initial analysis.
The initial specification is expressed in terms of abstract data models that represent the required information without regard for format or efficiency. The format of the data is hidden by the module labeled "user" in Fig. 6, which represents a software encapsulation of the human user. The initial version of this module uses default methods and formats for reading input data and displaying output data, and does not require any explicit description until the prototyping focuses changes from functional behavior to human interface factors. The types set, sequence, string, and type are predefined in the standard Spec library, which can be found in [9].

The behavior of the spell function is specified via a postcondition describing the required output. There is no precondition because the specified output is required for all possible inputs. The specification refers to selected reusable concepts from the Spec library, such as the predicates sorted and distinct, via IMPORT declarations.

The instance module defines the initial interpretation for the type word, documenting an assumption made by the analyst. The type word is declared as a subtype of string rather than as a new abstract data type because at this point, there are no apparent operations on words other than the standard string operations.

This completes the initial requirements. The next step of the process illustrated in Fig. 3 is to choose the implementation method for the top level module. The designer does not find a reusable software component realizing the entire spell function, and chooses to realize the specification via the decomposition shown in Fig. 8, using the subcomponents specified in Fig. 9. Sort_words is declared as an instance of the generic function sort, which is a standard building block well known to the designer.

After realizing the above components via reusable software components and meaning-preserving transformations, the proto-
type is demonstrated to a group of customers. A customer
reminds that many terms commonly used in his business are reported as spelling errors, such as names of products and suppliers. The 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 Fig. 10. This is the result of the "modify specification" activity shown in Fig. 3. The added text is boxed to highlight the changes.

The modified specification is produced by an extending transformation that adds an optional argument, followed by a relaxing transformation that removes the previous postcondition and a constraining transformation that rerestricts the behavior by adding the new postcondition. The inheritance mechanism of Spec is used to record the structure of transformational developments. Meaning-changing transformations are syntactically highlighted by the HIDE clauses that list all of the messages and concepts affected by nonmonotonic changes. The Spec representation of the transformed module spell_2 inherits the previous version spell_1, but hides the spell message to indicate that the transformed definition replaces the previous definition, rather than being combined with it. In this case, only the imported concepts "sorted" and "distinct" are inherited. Hiding the previous definition is necessary because the new postcondition is incompatible with the previous postcondition in cases where a private_dictionary is given explicitly, although the previous behavior is preserved whenever the private_dictionary takes its default value. If the previous version of the spell message was not hidden, the
new requirement would include the conjunction of the old
postcondition and the new postcondition, which would not be
satisfiable for any report containing a word in the private-
dictionary.

An initial modified design is obtained by noting that the new
version of spell can be implemented in terms of the old one by
passing the union of the dictionary and the private-dictionary
as the second parameter. This is illustrated in Fig. 11. This is
an example of a case in which partial replay of the derivation
of an implementation for a previous version is possible. The
situation can be recognized via a pattern matching process that
attempts to express the new requirements in terms of primitives
that have appeared in the previous version.

The second round of demonstrations exposes several differ-
ent issues: the users notice it is awkward to explicitly supply
a dictionary each time the system is used, and they want
the system to be able to learn new specialized words. The
analyst responds to the first concern by changing the dictionary
from an input parameter to a constant, built into the system.

At this point the new requirement would include the
conjunction of the old postcondition and the new postcondition,
which would not be satisfiable for any report containing a word in the private-
dictionary.

An initial modified design is obtained by noting that the new
version of spell can be implemented in terms of the old one by
passing the union of the dictionary and the private-dictionary
as the second parameter. This is illustrated in Fig. 11. This is
an example of a case in which partial replay of the derivation
of an implementation for a previous version is possible. The
situation can be recognized via a pattern matching process that
attempts to express the new requirements in terms of primitives
that have appeared in the previous version.

The second round of demonstrations exposes several differ-
ent issues: the users notice it is awkward to explicitly supply
a dictionary each time the system is used, and they want
the system to be able to learn new specialized words. The
analyst responds to the first concern by changing the dictionary
from an input parameter to a constant, built into the system.

The second round of demonstrations exposes several differ-
ent issues: the users notice it is awkward to explicitly supply
a dictionary each time the system is used, and they want
the system to be able to learn new specialized words. The
analyst responds to the first concern by changing the dictionary
from an input parameter to a constant, built into the system.

The analyst also notes that a learning function introduces a
requirement for long-term memory, so that the spell program
must be a state machine rather than a function. The state of
this machine corresponds to the private dictionary, as shown
in Fig. 12. The * in the transition is a temporal operator that
refers to the previous state.

The changes are the combination of a constraining trans-
formation that requires the dictionary to have a specified
contents, a pair of contracting and extending transformations
that removes all input from the spell message and replaces
them with just the report, a transformation that adds the
concept and the state variable to the meta-vocabulary of the
system (see Section II-B.4)), an extending transformation that
adds the learn message, and a constraining transformation that
restricts the behavior of the learn message via a postcondition.

We note that some of these transformations are needed to
restore the integrity of the specification after a desired change.
For example, the postcondition of spell becomes unsatisfiable
for all but one value of the input parameter "dictionary" as a
result of the first transformation. The contracting and extending
transformations solve part of this problem by removing all of
the unsatisfiable cases from the vocabulary of the system, but
they make the postcondition of the spell message undefined
because two of the variables are left unbound. This is corrected
by the addition of the state variable and concept. At the
conceptual level at which the analyst is working, we might
describe the whole process as changing an input to a constant
and making another input into a state variable, where the
second change implies that the module must become a state
machine and an operation to manage the state variable must
be added. It would be useful to package these two kinds of
changes as higher level transformations, and a remaining
challenge is how to do this at a more abstract level than the
constructs of a particular specification language.

The designer notes that the new message is expressed in
terms of the executable subset of the Spec language, so that
further refinement is not needed. The decomposition of the
spell message can also remain the same: the only changes are
in the nature of the sources of the input values. However, the
designer decides to simplify the decomposition as shown in
Fig. 13. The reformulation expands the old version of the spell
function, thus eliminating the reference to the previous version
of spell, and reduces the number of component types by
replacing the union function with another copy of check. This
meaning-preserving transformation depends on the property
"(x IN union(s1,s2)) \iff (x IN s1) \& (x IN s2). The purpose of this
reformulation is to simplify the design and to facilitate
future changes.

A complete exploration of the spelling checking system
would have many more aspects, such as correcting spelling er-
ners, suggesting corrections, and refining the concrete interface
formats. Due to lack of space, we leave the example incom-
plete, and consider instead the representation of derivation
histories.

C. Derivation Lattices

Section II suggests that it is useful to arrange derivation
histories in graph structures in which arcs represent monotonic
transformations. Such a graph is a lattice with respect to
the partial ordering \( \subseteq \) over programs that is defined by the following:

\[
p \subseteq q \iff \text{vocabulary}(p) \subseteq \text{vocabulary}(q) \&
\quad \text{granularity}(p) \subseteq \text{granularity}(q) \&
\quad \text{behavior}(p) \supseteq \text{behavior}(q).
\]

The vocabulary, granularity, and behavior of a program are
defined in terms of the event model in Section II-B2). The ordering
\( p \subseteq q \) means that \( q \) is a refinement of \( p \) along any of
these three attributes; there may be additional external stimuli
recognized by \( q \), its behavior may be specified at a more
detailed level of abstraction, and its behavior may be subject
to stricter constraints. The significance of the relation \( p \subseteq q \) is that \( q \) satisfies the specification of \( p \), so that from the point of view of a user, it is just as good, and it may be strictly better in the sense that it may provide some services that \( p \) does not. The second half is particularly important in the context of prototype evolution, where a meaning-preserving derivation should steadily strengthen the requirements until they become acceptable to the users. In this section, we illustrate this idea in terms of the spelling checker example.

Examining the initial part of the prototype evolution shown in the previous section, we see that this process is characterized by conceptual changes in the purpose of the proposed system, which are manifested as changes in its vocabulary. The externally visible behaviors of different versions of the system are not directly comparable because the set of potential stimuli is different for different versions. Therefore, we suggest organizing the derivation history first based on the effects of transformations on the vocabulary of the proposed system, then based on behavior within classes that share the same vocabulary, then based on granularity within classes that share the same external behavior, and then based on detailed computational behavior within classes that share the same granularity. Previous work on meaning-preserving transformations has mostly been restricted to the last three of these ranges, with emphasis on the last two.

We want to separate the effects of the transformations on orthogonal attributes of the system as much as possible so that independent changes can be automatically recombined in different combinations. The problem of automatically combining different versions of programs has been formally studied in several different contexts [10], [31], [7], and has been informally discussed in terms of the development of requirements in [19], where the independence of elaborations was assessed manually. However, the problem has not yet been solved completely, particularly for the case where the requirements are subject to change. This section considers the problem in the context of prototyping, and makes a step towards automating the detection of independent elaborations by proposing a formal model for refinement structures. In particular, is it potential for parallel elaborations whenever the lattices can be decomposed in a cross-product structure because different components of the cross product can be refined independently. This is usually the case for different messages in a system, for example. Previous methods for software merging have assumed that the vocabulary is fixed and common to all versions to be merge. The model proposed here is a possible basis for extending some previous work on merging via lattice structures [10], [7] to cases where the vocabulary changes. We believe that these models can support algorithms for rearranging chronological derivation histories with meaning-changing transformations into equivalent derivation lattices containing only mean-preserving lattices, but the details remain to be worked out.

It is possible to factor the vocabulary of the system based on the set of modules in the system, the set of messages recognized by each module, and the type signatures associated with each message. An independent structure of proposed versions is associated with each module and with each message. These structures are illustrated in Fig. 14. Each of the boxes is labeled with a version number, where version 0 corresponds to the empty program representing the state of the project before the beginning of the effort, and the other version numbers correspond to the numbers in the module names. The left diagram shows the set of modules for each version, ordered by the subset relation. The spelling checker is a very simple system, for which the set of modules is stable. The set of modules might change during the prototyping of a larger system if the analyst discovers that the proposed software must interact with an external system that was previously believed to be unaffected. In such a case, a module representing the affected external system would be added to the next version of the prototype. The attributes of each module are orthogonal, and can be refined independently.

The middle diagram shows just the messages recognized by the spell module. Sets of messages are also ordered by the subset relation. The set of messages changes when the requirement for learning is added in version 3. The signatures of each message are independent, and can be refined independently.

The diagram on the right represents the sets of signatures for the spell message, where \( r, d, \) and \( pd \) are abbreviations for "report," "dictionary," and "private dictionary," respectively. Since messages can be overloaded, each message can correspond to a set of signatures. Such sets are sometimes compactly represented via optional parameters. The signature for version 2 is really a set consisting of two signatures: \{\( (r, d) \), \( (r, d, pd) \)\}. Signature sets are also ordered by the subset relationship. If the state model of a module is fixed, including invariant restrictions, initial value restrictions, and semantic interpretations, then the descriptions of the behaviors corresponding to each signature of each message can also be refined independently.

The signature of the first version is a subtype of the second version because of the optional third parameter. Note that the third version lies on an alternative branch from the first two versions, and hence is independent of them: the first two versions represent a dead-end path whose only purpose was to provide enough insight into the problem to formulate version 3. A "rational explanation" of the process would proceed straight to version 3, although versions 1 and 2 were necessary in practice to elicit the communication between the users and the analyst that allowed the analyst to determine that version 3 was, in fact, necessary. This communications gap is what prevents practical requirements acquisition efforts from following only meaning-preserving transformations.
In a bigger example, the final formulations of different messages could be developed at different times, and might be parts of different versions of the entire system. There is also no guarantee that the final formulation is the most recently developed: it is entirely possible for a proposed enhancement to turn out badly, and for some aspects of the design to be reset to older versions due to newly discovered advantages. The chronological link to the past versions is useful for recording the justification for choosing the final version over other versions that have been explored. The refinement structure helps to bring related decisions together, even though they may have a large separation in the chronology, and helps extract the evolution structure of individual messages from the evolution structure of the system as a whole.

The previous example shows the lattices just for the vocabularies of the different versions. These lattices can be constructed based just on syntactic properties of the specifications, and the process is readily automatable. Constructing the behavioral lattices is considerably more difficult in the general case because of the need to decide implications and equivalences for logical statements, and partial or approximate methods will be needed. However, we note that in the early stages of prototyping, many of the changes affect the vocabulary, and there is a separate behavioral lattice for each version of the vocabulary because behaviors of systems with different vocabularies are not directly comparable. Hence, the behavioral lattices will be small. All of the behavioral lattices for the case study are single-point lattices because each version of the prototype has a different vocabulary.

V. ACHIEVING AUTOMATION

Automated support is needed to realize the potential benefits of transformational software prototyping. This section discusses some of the issues that must be resolved to provide such support: determining which meaning-changing transformations to apply, creating a library of higher-level transformations, and replaying a derivation from a different starting point.

A. Choosing Transformations

Choosing which transformations to apply in a particular situation in a development is a difficult problem. Some work on automating this process has been done in the context of meaning-preserving transformations [20]. In this context, the top-level goal is fixed: to implement and optimize a given specification. However, user interaction appears to be required because the search spaces involved in finding transformations and in checking their applicability conditions are very large and may not be finite. Also, the system may not have enough information about the application to resolve tradeoffs between conflicting design goals.

In contrast, the scope of potential automation for meaning-changing transformations is limited by the ability to formulate the goal rather than by computational resources: there are many possible ways to change the meaning of a program, and the user must distinguish between them before the system can even try to realize the change. This leads to the question of what information-adding or meaning-changing transformations should be provided by a transformation system.

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 to work in a more abstract space that corresponds more closely to the designer's conceptual universe, or to work with larger units than the primitives of the notation.

The relation between these two approaches can be understood by analogy with syntax-directed editors [32]. In such systems, there are two corresponding 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 mechanisms, 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.

Monotonic transformations are the semantic analog to the operations of a syntax-directed editor. Realizing the semantic analog to template transformations requires developing a characterization of the design space analogous to the grammar for the source language of a syntax-directed editor. In cases where the set of semantic choices available to the designer is finite, a choice can be picked from a menu. The initial choices of values for these attributes are monotonic transformations, and changes in the attributes are composite transformations. Some examples of design attributes for which this is possible are shown in Fig. 15. We believe that developing a better formal understanding of the design space is the key issue for developing better transformation libraries that can be embodied in menu-driven tools. The semantic design space is considerably less well understood than the syntax of a context-free language, and is likely to be more complex. In some situations, the set of choices available to the designer is not finite, and may not even have a closed description.

A process of free text entry is therefore likely to be necessary for semantic decisions, in addition to being a practical aid to the efficiency of design entry in some situations. For example, free text entry is likely to be used for transformations in which the designer adds a new module or a new message. A transformation system supporting such a mode should have an analog to the parsing process, which attempts to reconstruct a set of primitive monotonic 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 derivation lattice for the design provides a (sanitized) record of the designer’s thoughts, which may be useful for analyzing and changing the design, as indicated in Section IV-C. The structure is also relevant to the evolution scenarios outlined in [38], [1]. More work on models of software design decisions is needed to enable the implementation of such a facility.

B. The Transformation Library

We conjecture that the derivation lattice may be useful in identifying and suggesting new general transformation schemes based on instances of meaning-changing transformations entered by designers using free text and mechanically decomposed into the lattice representation. In particular, such manual transformations may not be monotonic. Reconstructing the lattice structure can help the system to identify which information is removed by a manual transformation, and which information is added. These characterizations can be used to match up several transformations with similar structures, and the mappings between the instances can be used to suggest generalizations.

The work on the requirements apprentice [33] is relevant here. The requirements apprentice gains its power from domain-specific cliches, which are similar to the concepts in the Spec model library [9]. Cliches have been used in the requirements apprentice work to represent commonly occurring patterns in models of application domains, and in earlier work on the programmer’s apprentice to represent common patterns in programs and software designs. Schemes in meaning-changing transformations are just patterns of commonly occurring changes to designs. We conjecture that mechanisms similar to those that support reasoning about cliches, such as the combination of frame-based reasoning, truth maintenance techniques, special-purpose methods for dealing with congruence closures, and general predicate calculus reasoning, will be useful for analyzing derivations and suggesting general transformations based on actual concrete derivations. We suggest this as a way of building up and managing a repertoire of meaning-changing transformations.

Transformations that remove a conjunct from a condition and information-removing transformations in general are easier to automate than information-adding transformations. Some of the information-adding transformations informally used by designers do have simple characterizations, and could be applied by automated procedures. For example, [23] describes some transformations of this type. Some of the transformations in [23], such as generalizing a concept by adding a parameter, are applicable in any context, although others are specific to the Gist language. Some additional examples of design-level transformations that have systematic characterizations follow.

- Implement an output sequence via a time series (incremental generator).
- Implement a Spec concept as a lower level software component (indicated for concepts appearing in preconditions). The specification for the lower level component can be automatically constructed in such a case.
- Implement a data type using a direct storage representation (no pointers).
- Implement an input value and an output value as a single in–out parameter (limited lifetime of input data).

These kinds of design decisions can be represented as Spec pragmas [9], which provide a compact way to document such decisions as annotations on the resulting design. Cliches are a potential source of information for information-adding transformations. Finding a generalized reusable component and then specializing it to match the application is a powerful approach to software specification that is relevant to transformational development.

C. Replay

Representing a derivation as a refinement lattice, as described in Section IV-C, rather than as a linear sequence of transformations has the advantage of separating independent refinements. This implies that a change to a feature of a system (such as the response to a particular stimulus) need only reconsider the transformations on the path in the lattice that leads to that particular feature, rather than the entire derivation. An informal version of this approach was suggested in [19]. The structures described in Section IV-C should be sufficient for automating the derivation and maintenance of the part of this lattice related to changes in the vocabulary of the proposed system. More work is needed to do the same for the part of the derivation lattices related to behavior (constraining transformations), and on methods for identifying independent components. In particular, better characterizations of the dependencies between specifications of the operations of a machine that are induced by a common state model and the analogous dependencies between the operations of an abstract data type that are induced by a common instance model are needed to realize this goal.

VI. CONCLUSIONS

One of the reasons software evolution is difficult is that realistic software designs are complicated by optimizations. These optimizations are meaning-preserving transformations that improve the efficiency of the software by introducing additional constraints that complicate the implementation and introduce dependencies between parts of the system that might otherwise be independent. We would like to apply transformations that change the behavior of a system before it is complicated by the optimizations. Our approach is a complement to the use of meaning-preserving transformations to achieve optimizations.

We have characterized some classes of software transformations relevant to software evolution, and have pro-
posed a framework for integrating such transformations into a computer-aided approach to prototyping and software development. We have emphasized transformations that modify the behavior of a software system because such transformations are needed in software evolution, and because they are not as well studied as meaning-preserving transformations. The goal of a meaning-preserving transformation is to improve some aspect of the software, usually its efficiency, without affecting its behavior. The goal of a transformation that changes the behavior of a software system is harder to characterize.

We have proposed that software modifications should be decomposed into combinations of monotonic transformations, and that these decompositions should be managed by software tools. Monotonic transformations are categorized based on their effects on interfaces, externally observable behavior, and internal computations, with examples, in Section II. We have extended previous formulations of transformations from serial computations to parallel computations via the event model explained in Section II-B-1, and have used this model to provide a construction of derivation lattices in Section IV-C. Transformations of these types should provide faster and more reliable ways for designers to modify software systems.

Our vision of software evolution is a process that operates on a structure representing the design decisions that lead to a software system. These design decisions correspond to transformations on partial or complete representations of the specifications, designs, and code. The product of software evolution is a structure that represents an idealized history of a system. This structure records which design decisions contribute to which versions. This is a simplified and idealized history because it represents the conceptual differences between versions, but not necessarily the actual sequence in which the versions were created or the order in which transformations were originally applied. The benefit of this structure is to bring together all of the changes related to the same design decision, and to provide an explicit representation for all the alternatives for each design decision that has been considered in an exploratory development such as a prototyping effort, or in the evolution of a deployed software system in response to changing circumstances. Recording the design history in a processable form is practically important because of personnel turnover in development projects. The proposed structure should help designers make better use of the history of a development.

Our research goal is to create conceptual models and software tools that allow automatic generation of variations on a software system with human consideration of only the highest level decisions that must change between one version and the next. Realization of this goal will lead to more flexible software systems and should make prototyping and exploratory design more effective.

Challenges facing future research on meaning-changing transformations are to span the software design space with a set of manageable transformations with precise and expressive representations, to provide automatic procedures for suggesting applicable transformations, and to construct automatic or computer-aided procedures for decomposing manual design changes into sequences of primitive transformations. Successful research in this direction and its future applications will support software design automation with great scientific and economic impact.

REFERENCES


Luqi (S'86–M'86) received the B.S. degree from Jilin University, China, and the M.S. and Ph.D. degrees in computer science from the University of Minnesota.

She is currently an Associate Professor at the Naval Postgraduate School. She worked on software research and development for the Science Academy of China, Computer Center at University of Minnesota, International Software Systems, etc. Her research interests include rapid prototyping, real-time systems, specifications, and software development tools. Her research is supported by the National Science Foundation and the Office of Naval Research. She is an Associate Editor for IEEE Expert and the Journal of Systems Integration.

Amiram Yehudai (M'85) received the B.Sc. degree in electrical engineering from the Technion, Israel Institute of Technology, Haifa, in 1974, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1974 and 1977, respectively.

In 1977 he joined the Faculty of the Computer Science Department, Tel-Aviv University, and is currently an Associate Professor there. He served as the Chairman of the Department from 1982 to 1983, and again from 1985 to 1987. He held visiting positions at the University of California at Berkeley and Los Angeles, at the University of Maryland, at GMD, and at the Naval Postgraduate School. His main research areas are software engineering and programming languages.

Dr. Yehudai is a member of the IEEE Computer Society and ACM.

Valdis Berzins received the B.S., M.S., E.E., and Ph.D. degrees from M.I.T.

He served as an Assistant Professor at the University of Texas, and as an Associate Professor at the University of Minnesota. He is currently a Professor of Computer Science at the Naval Postgraduate School. His research interests include software engineering, computer-aided design, and theoretical foundations of software maintenance. His recent work includes papers on software merging, specification languages, and engineering databases. He has developed several specification languages, software tools for computer-aided software design, and a fundamental theory of software merging.