**Calhoun: The NPS Institutional Archive**

**DSpace Repository**

NPS Scholarship | Publications

1998

# Recombining changes to software specifications

Berzins, V.

Elsevier

The Journal of Systems and Software 42 (1998) 165-174
https://hdl.handle.net/10945/51485

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

*Downloaded from NPS Archive: Calhoun*

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

ELSEVIER

The Journal of Systems and Software 42 (1998) 165–174

The Journal of Systems and Software

# Recombining changes to software specifications [1]

V. Berzins [2]

*Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, USA*

Received 10 December 1996; received in revised form 10 November 1997

**Abstract**

This paper proposes a model of software changes for supporting the evolution of software prototypes. We decompose software evolution steps into primitive substeps that correspond to monotonic specification changes. This structure is used to rearrange chronological derivation sequences into idealized conceptual derivation structures containing only meaning-extending changes, and to automatically combine different changes to a specification. A set of examples illustrates the ideas. © 1998 Elsevier Science Inc. All rights reserved.

*Keywords:* Software evolution; Change merging; Specifications; Logic

## 1. Introduction

Evolutionary prototyping provides an efficient approach to formulating accurate software requirements [19]. Simple models reflecting the main issues associated with the proposed system are constructed and demonstrated, and then reformulated to better match customer concerns, based on specific criticisms and the issues they elicit. This process aids understanding because independent issues are separated and treated in isolation as much as possible, via communication based on the simplest models possible. The models are refined only as needed to resolve open issues, and the issues arising at one level of detail are resolved as much as possible before considering the next level of detail, or the next aspect of the system. This helps to focus the attention of the customers, designers, and analysts because only a few selected aspects of the system are changing at any point in the process.

The focus of the current work is the evolution of proposed specifications and prototype designs. Much of the previous work on changes to software has focused on meaning-preserving transformations [2,14,16,24,25]. However, it has been recognized that in realistic contexts, many changes do *not* preserve the observable behavior of the system [26]. Most of the work on the area of meaning-changing transformations has been concerned with classifying the types of semantic modifications that are used in practice [12,11,15]. We investigate the relationships between different versions of the specifications and propose an abstract model of the design history to provide a more formal model for understanding the details of this subject.

Modeling the design history can enhance the prototyping process by capturing the conceptual dependencies in a design. A properly structured derivation of a specification can highlight 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. These 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, and to aid the designers in combining design choices from different branches of a parallel exploration of the design space.

In larger prototyping efforts, several explorations of the requirements that are focused on distinct aspects of the system may proceed in parallel. In such cases, the lessons learned from different branches of the effort must be combined and integrated. This is a specification-level instance of the software change-merging problem

[1] This research was supported in part by the National Science Foundation under grant number CCR-9058453, by the Army Research Office under grant number ARO 96-8, and by the Army AI Center under grant number 6GNPG00072.

[2] Tel.: +1 408 656 2610; fax: +1 408 656 2189; e-mail: berzins@cs.nps.navy.mil.

0164-1212/98/$19.00 © 1998 Elsevier Science Inc. All rights reserved.
PII: S0164-1212(98)10007-9

[8]. Solutions to this problem can also be used to propagate improvements to all affected versions.

The rest of the paper is organized as follows. Section 2 suggests some classes of primitive changes and sketches an associated representation for abstract derivation histories. Section 3 illustrates our ideas with an example. Section 4 discusses change merging for specifications and indicates how merged versions can be constructed. Section 5 contains conclusions.

## 2. Software changes

We characterize changes to a system specification in terms of three orthogonal attributes of a system: its vocabulary, its behavior, and its granularity [21]. These concepts are reviewed below.

- The *vocabulary* of a system is the set of all external stimuli recognized by the system.
- The *granularity* of a system is the set of all internal stimuli recognized by the system.
- The *behavior* of a system is the set of all possible traces for the system relative to a given vocabulary and granularity.

Each of these three attributes is a set, and is subject to an approximation ordering induced by the subset relation. The resulting partially ordered set becomes a Boolean algebra under the set union, set intersection, and set complement operations. As explained in Section 4, this structure can support a formal model of software change merging.

If we restrict primitive changes to be monotonic and to affect just one of the three attributes listed above, we get the classification of primitive changes shown in Fig. 1, which is repeated from [21].

The symbol $A_S$ represents the attribute $A$ of the original system $S$, and $A_{S'}$ represents the attribute $A$ of the modified system $S'$.

A decomposition of the chronological evolution history into primitive substeps conforming to these restrictions enables the rearrangement of a sequential derivation containing meaning-modifying changes into a tree-like rooted directed acyclic graph whose paths consist solely of meaning-preserving changes that add information via compatible vocabulary extensions, granularity refinements, or behavior constraints.

| | Effect of Change | |
|---|---|---|
| Attribute A | $A_S \subset A_{S'}$ | $A_S \supset A_{S'}$ |
| Vocabulary | extending | contracting |
| Granularity | refining | abstracting |
| Behavior | relaxing | constraining |

Fig. 1. Types of changes.

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 structures of chronological derivations produced by people are often obscured by interleaved sequences of changes that introduce and later remove inappropriate aspects of system behavior. This process is illustrated in Fig. 10 and explained further in Section 3.2.

We propose this mechanism as a concrete means to document software as if it had been developed using a rational process [22], and conjecture that such structures will be useful for choosing demonstration scenarios, guiding requirements reviews, and summarizing past history for analysts formulating the next version. The early parts of the development, in which the requirements are evolving, must be guided by people because these changes 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 monotonic changes, 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 changes 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.

A change 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 changes 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 a derivation graph usually do not capture 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 if they have been derived from the root of the graph via meaning-preserving changes, because the root requirements do not satisfy 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. We note that a substantial amount of research and development is needed to support such a process in practical contexts.

## 3. Case study

This section illustrates our ideas via a simple case study, after a brief explanation of our notation. We define required behavior of interfaces using the Spec language [4]. 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.

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 [18], and is easily readable without further explanation. PSDL is the prototyping language used in the Computer Aided Prototyping System CAPS [17].

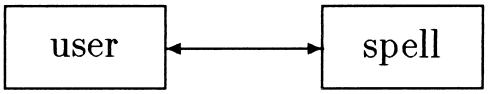


Fig. 2. Context diagram.

### 3.1. Example: Spelling checker

We now illustrate the use of specifications in the evolution of a prototype for a spelling correction system, emphasizing the role of monotonic changes. The initial focus of the prototyping effort is on the required behavior of the 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. 2, and results in the initial specification for the behavior of the proposed software given in Fig. 3.

Identifying and modeling the aspects of the data relevant to the problem is the main contribution 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. 2, 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 focus changes from functional behavior to human interface factors. The types *set*, *sequence*, *string* and *type* are pre-defined in the standard Spec library, which can be found in [5].

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

```
FUNCTION spell_1
  IMPORT sorted distinct FROM sequence{word}
  MESSAGE spell(report: sequence{word}, dictionary: set{word})
    REPLY(errors: sequence{word})
    WHERE ALL(w: word :: w IN errors <=> w IN report & ~(w IN dictionary)),
      sorted{less_or_equal@word}(errors),
      distinct(errors)
END

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

Fig. 3. Specification of initial spelling checker.

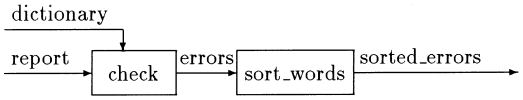


Fig. 4. Initial decomposition.

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 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. 4, using the subcomponents specified in Fig. 5. 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 by interconnecting some reusable software components, the prototype is demonstrated to a group of customers. A customer remarks 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. 6. The added text is boxed to highlight the changes.

The modified specification is produced by an extending change that adds an optional argument followed by

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

FUNCTION sort{t: type, le: function{from:: [t, t], to:: boolean}
              SUCH THAT total_ordering(le) }
  IMPORT total_ordering FROM total_order{t}
  IMPORT sorted permutation FROM sequence{t}
  MESSAGE(in: sequence{t}) REPLY(out: sequence{t})
    WHERE sorted{le}(out), permutation(in, out)
END

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

Fig. 5. Specifications for subfunctions.

```
FUNCTION spell_2 INHERIT spell_1 HIDE spell
  MESSAGE spell(report: sequence{word}, dictionary private_dictionary : set{word})
    DEFAULT private_dictionary = {}
    REPLY(errors: sequence{word})
    WHERE ALL(w: word :: w IN errors <=>
              w IN report & ~(w IN dictionary) & ~(w IN private_dictionary) ),
          sorted{less_or_equal@word}(errors),
          distinct(errors)
END
```

Fig. 6. Transformed specification for the spelling checker.

a relaxing change that removes the previous postcondition and a constraining change that re-restricts the behavior by adding the new postcondition. The inheritance mechanism of Spec is used to record the design history. Meaning-changing modifications are syntactically highlighted by HIDE clauses that list all of the messages and concepts affected by non-monotonic 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 were 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. 7. This is an example of a case in which partial reuse of the derivation of an implementation for a previous version is possible.

The second round of demonstrations exposes several different 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

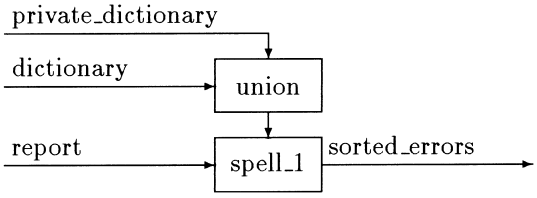


Fig. 7. Decomposition of spell version 2.

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 next version of 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. 8. The TRANSITION clause illustrates the use of temporal logic in Spec to specify requirements associated with state transitions in state machines. The * is a temporal operator that refers to the previous state.

The changes are the combination of a pair of contracting and extending changes that remove all inputs from the spell message and replace them with just the report, a change that modifies the type of the module from a function to a state machine, a change that adds the concept and the state variable to the meta-vocabulary of the system, an extending change that adds the learn message, and a constraining change that restricts the behavior of the learn message via a postcondition.

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

```
MACHINE spell_3 INHERIT spell_2 HIDE spell
  STATE(private_dictionary: set{word})
    INVARIANT true
    INITIALLY private_dictionary = {}

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

  MESSAGE learn(words: set{word})
    TRANSITION private_dictionary = *private_dictionary U words

  CONCEPT dictionary: set{word} -- The words in the Oxford English Dictionary.
END
```

Fig. 8. Transformed specification for the spelling checker.

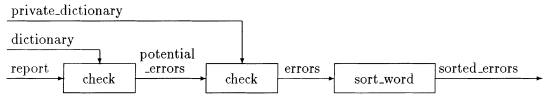


Fig. 9. Simplified decomposition of spell version 2.

decomposition as shown in Fig. 9. 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 change 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 errors, suggesting corrections, and refining the concrete interface formats. Due to lack of space, we leave the example incomplete, and consider instead the representation of derivation histories.

### 3.2. Conceptual derivation histories

A conceptual derivation history is a simplified version of the chronological history of an evolving system that includes only the decisions that were not undone in later steps. We model conceptual derivation histories as graphs whose nodes represent versions and whose arcs represent monotonic changes. Such a graph is a partially ordered set with respect to the refinement ordering $\sqsubseteq$ on specifications, defined as follows:

$$\begin{aligned} p \sqsubseteq q \iff & \text{vocabulary}(p) \subseteq \text{vocabulary}(q)\ \& \\ & \text{granularity}(p) \subseteq \text{granularity}(q)\ \& \\ & \text{behavior}(p) \supseteq \text{projection}(\text{behavior}(q), \\ & \qquad \text{vocabulary}(p) \cup \\ & \qquad \text{granularity}(p)) \end{aligned}$$

The vocabulary, granularity, and behavior of a specification are defined in Section 2. The projection is needed to ensure that we are comparing just the corresponding parts of the two behaviors; it removes all events in traces of $q$ that are outside the vocabulary and granularity of $p$. The ordering $p \sqsubseteq q$ means that $q$ satisfies the specification of $p$. From the point of view of a user $q$ is just as good as $p$, and it may be strictly better if it provides some services that $p$ does not. Enhancements can occur if $q$ responds to additional external stimuli, its behavior is specified at a more detailed level of abstraction, or its behavior is subject to stricter constraints. An idealized prototype evolution process should steadily strengthen the requirements in this sense, until they become acceptable to the users. In practice the path is often less direct. However, a reconstructed direct path should provide a useful summary of the relevant parts of the evolution of the requirements. We illustrate this idea in terms of the spelling checker example.

The initial part of the prototype evolution shown in the previous section contains 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 changes on the vocabulary of the proposed system, then based on behavior of subsystems that share the same vocabulary, then based on granularity for interactions that share the same external behavior, and then based on computational efficiency of interactions that share the same external behavior and granularity. Previous work on meaning-preserving changes has mostly been restricted to the last three of these ranges, with emphasis on the last two.

We would like to separate the effects of changes to orthogonal attributes of the system as much as possible, so that these 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 [6–10,23,3], and has been informally discussed in terms of the development of requirements in [13], where the independence of elaborations was assessed manually. However, the problem has not yet been solved completely, particularly for requirements.

We make a step towards automating the detection of independent elaborations by proposing a formal model for refinement structures. There is potential for parallel elaboration whenever the refinement ordering can be decomposed in a cross-product structure, because different components of the cross-product can be refined independently. For example, this is usually the case for changes to different messages in a system.

Previous methods for software change merging have assumed that the vocabulary is fixed and common to all versions to be merged. The model proposed here is a possible basis for extending some previous work on merging [7,3] to cases where the vocabulary changes. Such an extension adopts an open and extensible view of the vocabulary: the behavior of a system whose vocabulary does not contain a given stimulus is considered equivalent to the behavior of a modified system that extends its vocabulary with the extra stimulus and leaves its response to that stimulus undefined and unconstrained. This is appropriate for requirements exploration and prototyping, although it is not consistent with the closed-world view typically adopted in software products, where requests outside the vocabulary are expected to produce error messages and have no other effect. Section 4 sketches some of the main ideas for this extension.

We can separate different aspects of the vocabulary of a 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 version refinement structure is associated with each module and with each message. These structures are illustrated for the most abstract level of granularity in Fig. 10. Each of the boxes is labeled with a version number, where version 0 corresponds to the empty software model representing the initial state of the project, 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, 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: sequence{word}", "dictionary: set{word}", and "private_dictionary: set{word}", respectively. Since messages can be overloaded, each message can correspond to a set of signatures. Signature sets are sometimes compactly represented via optional parameters. For example, version 2 has a set 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.

Note that the third version lies on an alternative branch from the first two versions, and hence is independent from 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 proposing only meaning-extending changes. The main benefit of a monotonic representation for conceptual derivation histories is that such dead end paths are identified and separated out from the main line of the derivation history.

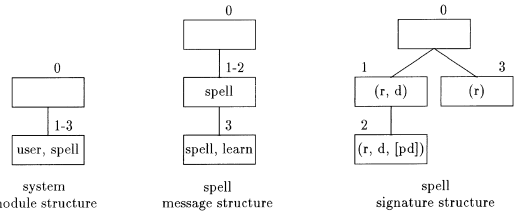


Fig. 10. Vocabulary refinement structure of the derivation history.

Fig. 10 shows the ordering structures just for the vocabularies of the different versions. These structures can be constructed based just on syntactic properties of the specifications, and the process is readily automatable. Constructing the behavioral structures is considerably more difficult in the general case, because of the need to decide implications and equivalences for logical statements. Consequently, 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 that there is a separate behavioral structure for each version of the vocabulary, because behaviors of systems with incompatible vocabularies are not directly comparable. Hence the ordering structures corresponding to behavioral changes with a common vocabulary and granularity will be small, particularly in the context of prototyping.

## 4. Combining changes

The Boolean algebra structure of the vocabulary, granularity, and behavior of a specification identified in Section 2 implies that the usual formulation of the change merging operation can be applied in the context of changes to software specifications. If $A$, $B$, and $C$ are specifications, the result of combining the change from $B$ to $A$ with the change from $B$ to $C$ is denoted by $A[B]C$, which is defined as follows.

$$A[B]C = (A - B) \sqcup (A \sqcap C) \sqcup (C - B).$$

Here $\sqcup$ denotes the least upper bound and $\sqcap$ denotes the greatest lower bound with respect to the ordering defined in Section 3.2. The difference is defined by

$$A - B = A \sqcap \overline{B},$$

where the bar denotes the complement operation. This operation is well defined for any Boolean algebra; in the special case of sets ordered by $\subseteq$, it is the set difference operation.

The interpretations of the above Boolean operations for different aspects of software specifications are

| Aspect | Operation | | | |
|---|---|---|---|---|
| | $X \sqsubseteq Y$ | $X \sqcup Y$ | $X \sqcap Y$ | $X - Y$ |
| Vocabulary | $X \subseteq Y$ | $X \cup Y$ | $X \cap Y$ | $X - Y$ |
| Granularity | $X \subseteq Y$ | $X \cup Y$ | $X \cap Y$ | $X - Y$ |
| Behavior | $X \supseteq Y$ | $X \cap Y$ | $X \cup Y$ | $\overline{Y - X}$ |
| Postcondition | $X \Leftarrow Y$ | $X \wedge Y$ | $X \vee Y$ | $X \vee \neg Y$ |

Fig. 11. Concrete interpretations of abstract operations.

summarized in Fig. 11. Since it is common to represent sets of behaviors by logical assertions representing postconditions, we include the postcondition representation as well.

The set inclusions in the definition of the specification refinement ordering (see Section 3.2) go in the opposite direction for the system behavior than for the vocabulary and the granularity. This is reflected in the interpretations of the Boolean operations for those aspects. Since the specification refinement ordering is derived from the orderings of the three different aspects according to the usual ordering construction for a cross-product domain, all of the operations extend componentwise. This implies that we can compute change merges for the three aspects independently, according to the interpretations summarized in Fig. 11.

Some examples illustrate the effects of these definitions. Suppose we represent vocabularies as sets of messages. Then the combination of the change that removes the message $m_2$ from the starting vocabulary $\{m_1, m_2\}$ and the changes that adds $m_3$ to the same starting vocabulary is calculated as follows:

$$\begin{aligned}&\{m_1\}\{m_1, m_2\}\{m_1, m_2, m_3\}\\ &= (\{m_1\} - \{m_1, m_2\}) \cup (\{m_1\} \cap \{m_1, m_2, m_3\})\\ &\quad \cup (\{m_1, m_2, m_3\} - \{m_1, m_2\}) = \{m_1, m_3\}.\end{aligned}$$

The corresponding calculations on postconditions representing behaviors may be bit less intuitive. If $P$, $Q$, and $R$ are assertions representing postconditions, we can apply the general definition and simplify to give the following rule:

$$\begin{aligned}P[Q]R &= (P \vee \neg Q) \wedge (P \vee R) \wedge (R \vee \neg Q)\\ &= (P \vee R) \wedge (Q \Rightarrow P) \wedge (Q \Rightarrow R).\end{aligned}$$

We illustrate the consequences of this rule for some common change patterns. Suppose that $a$, $b$, and $c$ are three assertions representing postconditions in the specification of the behavior of a system in response to a given stimulus.

The combination of two different constraining changes to a behavior includes both constraints:

$$(a \wedge b)[b](b \wedge c) = (a \wedge b \wedge c).$$

The first changes adds the constraint $a$ to the postcondition $b$ of the base version and the second change adds a different constraint $c$. The original constraint and both of the new ones are present in the combination.

The combination of two relaxing changes loosens both of the affected constraints:

$$a[a \wedge b]b = a \vee b.$$

Note that the combination of removing each of two constraints separately does not result in a vacuous requirement: either of the two relaxed versions of the requirements is acceptable, but the system must satisfy at least one of them.

The combination of a relaxing change and a constraining change selectively loosens and also tightens the requirements:

$$b[a \wedge b](a \wedge b \wedge c) = b \wedge (a \Rightarrow c).$$

The constraint $b$ is common to all three versions, and it appears in the combination as well. The first change drops the constraint $a$, while the second change adds the constraint $c$. In the combination, the new constraint $c$ must hold only in the cases where the original constraint $a$ is satisfied. This moderation of the constraining change is due to the presence of the relaxing change; if we do not remove the constraint $a$ then the new constraint $c$ is added unconditionally:

$$(a \wedge b)[a \wedge b](a \wedge b \wedge c) = a \wedge b \wedge c.$$

To illustrate the effects of these rules in a more realistic context, Fig. 12 shows the results of merging two changes to a simplified version of the spelling checker example of Section 3.

We focus on the spell message. The base version has only the most basic requirements: there is only one dictionary, and there are no constraints on the order of the words in the output. The first enhancement introduces the modified requirement that acronyms (which contain only capital letters) are never reported as spelling errors. The second enhancement adds a requirement for sorting the output. The result of merging the two changes includes the acronym modification, but requires sorting only in the cases where the acronym modification did not take effect. This is a consequence of the minimal change principle [8] implicit in the change merging formula. In this case, a review by an analyst concludes that the case where the sorting requirement is suspended is impossible: the dictionary (a constant in the specification) cannot be empty in any acceptable version of a spell checking system, as would be required by the second condition in the last quantifier of Fig. 12. In general, application of the change merging rules can highlight cases where requirements changes interact. These cases can then be reviewed by people to check whether a subtle interaction was missed or misjudged.

All of the change merges in the above examples follow directly from the definition, after simplification

```
-- base version:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) )

-- first modification:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) & ~acronym(w) )

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

-- result of change merging:
MESSAGE spell(report: sequence{word} ) REPLY(errors: sequence{word})
  WHERE ALL(w: word :: w IN errors <=>
            w IN report & ~(w IN dictionary) & ~acronym(w) ),
        sorted{less_or_equal@word}(errors) |
          ALL(w: word :: w IN report & ~(w IN dictionary) & acronym(w) & ~(w IN errors))
```

Fig. 12. Merging changes to the spelling checker.

using the laws of ordinary propositional logic. These simplifications were performed manually and then checked via an automatic simplifier for propositional logic that is implemented using term rewriting with respect to a canonical set of rewrite rules.

The implementation of the change merging definitions for specifications is straightforward, just as is the implementation of weakest preconditions for loop-free code. The difficulty of automatic application lies in the simplification step in both cases: since most logics that are useful for specification are not decidable, it is in general impossible to do a perfect job of simplification. For these logics, there is no computable canonical form in which all tautologies reduce to the logical constant "true" and all contradictory statements reduce to the logical constant "false". In the above examples, simplification using only the propositional structure of the formulas was sufficient to get useful results, even though human judgement was needed to recognize constraints that hold in the problem domain, but are not universally true in the logical sense. However, even for decidable systems such as propositional logic, the existence of a canonical form does not solve the problem completely, because the result produced by the simplifier does not resemble the original formulas and is typically hard to read. Manual simplification was needed in the above examples to make the results readable by people. Heuristic methods that try to match the original structures as far as possible would be useful for practical decision support. This is an area for further research.

## 5. Conclusions

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 changes 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 changes 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 have 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 previous research has explored formal models of the chronological evolution history [20]. This model has been applied to automate configuration management and a variety of project management functions [1]. The ideas presented in this paper are a promising basis for

improving these capabilities, particularly in the area of computer aid for extracting useful design rationale information from a record of the evolution of the system.

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

## References

[1] S. Badr, Luqi, Automation support for concurrent software engineering, in: Proceedings of the Sixth International Conference Software Engineering and Knowledge Engineering, Jurmala, Latvia, 20–23 June 1994, pp. 46–53.
[2] F. Bauer et al., The Munich Project CIP. Vol. II: The Program Change System CIP-S, in: Lecture Notes in Computer Science, vol. 292, Springer, Berlin, 1987.
[3] V. Berzins, On merging software enhancements, Acta Inform. 23 (6) (1986) 607–619.
[4] V. Berzins, Luqi, An introduction to the specification language Spec, IEEE Software 7 (2) (1990) 74–84.
[5] V. Berzins, Luqi, Software Engineering with Abstractions: An Integrated Approach to Software Development using Ada, Addison-Wesley, Reading, MA, 1991.
[6] V. Berzins, Software merge: Models and methods, J. Systems Integration 1 (2) (1991) 121–141.
[7] D. Dampier, Luqi, V. Berzins, Automated merging of software prototypes, J. Systems Integration 4 (1) (1994) 33–49.
[8] V. Berzins, Software merge: Semantics of combining changes to programs, ACM TOPLAS 16 (6) (1994) 1875–1903.
[9] V. Berzins, Software merging and slicing, Tutorial, IEEE Comput. Soc. Press, Silver Spring, MD, 1995.
[10] V. Berzins, D. Dampier, Software merge: Combining changes to decompositions, J. Systems Integration 6 (1/2) (1996) 135–150.
[11] M. Feather, A system for assisting program change, ACM Trans. Programming Languages Systems 4 (1) (1982) 1–20.
[12] M. Feather, A survey and classification of some program change approaches and techniques, in: L.G.L.T. Meertens (Ed.), Program Specification and Change, Proceedings of the IFIP TC2/WG 2.1 Working Conference, North-Holland, Amsterdam, 1987, pp. 165–195.
[13] M. Feather, Constructing specifications by combining parallel elaborations, IEEE Trans. Software Eng. 15 (2) (1989) 198–208.
[14] S. Fickas, Automating the transformational development of software, IEEE Trans. Software Eng. 11 (11) (1985) 1268–1277.
[15] W. Johnson, M. Feather, Building an evolution change library, in: Twelfth International Conference on Software Engineering, 1990, pp. 238–248.
[16] E. Kant, On the efficient synthesis of efficient programs, Artificial Intelligence 20 (3) (1983) 253–305; also appears in: C. Rich, R. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, Los Altos, CA, 1986, pp. 157–183.
[17] Luqi, M. Ketabchi, A computer aided prototyping system, IEEE Software 5 (2) (1988) 66–72.
[18] Luqi, V. Berzins, R. Yeh, A prototyping language for real-time software, IEEE Trans. Software Eng. 14 (10) (1988) 1409–1423.
[19] Luqi, Software evolution via rapid prototyping, IEEE Comput. 22 (5) (1989) 13–25.
[20] Luqi, A graph model for software evolution, IEEE Trans. Software Eng. 16 (8) (1990) 917–927.
[21] Luqi, Specifications in software prototyping, in: Proc. SEKE 96, Lake Tahoe, NV, 10–12 June 1996, pp. 189–197.
[22] D. Parnas, P. Clemens, A rational design process: How and why to fake it, IEEE Trans. Software Eng. 12 (2) (1986) 251–257.
[23] S. Horowitz, J. Prins, T. Reps, Integrating non-interfering versions of programs, ACM Trans. Programming Languages Systems 11 (3) (1989) 345–387.
[24] C. Rich, R. Waters (Eds.), Readings in Artificial Intelligence and Software Engineering, Morgan Kaufmann, Los Altos, CA, 1986.
[25] D. Smith, G. Kotik, S. Westfold, Research on knowledge-based software environments at Kestrel Institute, IEEE Trans. Software Eng. 11 (11) (1985) 1278–1295.
[26] W. Swartout, R. Balzer, On the inevitable intertwining of specification and implementation, Comm. ACM 25 (7) (1982) 438–440; also appears in: N. Gehani, A.D. McGettrick (Eds.), Software Specification Techniques, 1986, pp. 41–45.

**Prof. Berzins** received a Ph.D from MIT in 1979 and worked as a professor at the University of Texas, the University of Minnesota, and the Naval Postgraduate School. He is working on increasing productivity and software quality via automated decision support and has done pioneering research on computer-aided design and software evolution, resulting in many publications, student theses, and software systems. His work has been supported by NSF, ARO, and many computer industries. He designed two specification languages, and initiated and established software merging, and important research area in software engineering. He developed the first semantically sound method for automatically combining extensions to software systems, a comprehensive theory of modifications to software, and the first methods for combining changes to software that treat parallel programs and algorithm optimizations. He is the author of books on software specifications and computer-aided software maintenance.