1996

Software evolution in prototyping

Berzins, V.

Allerton Press, Inc.

http://hdl.handle.net/10945/51507

Downloaded from NPS Archive: Calhoun
SOFTWARE EVOLUTION IN PROTOTYPING *

V. Berzins and Lu Qi

(Computer Science Department, Naval Postgraduate School, Monterey, CA 93943, USA)

Manuscript received August 10, 1995

Abstract This paper proposes a model of software changes for supporting the evolution of software prototypes. The software evolution steps are decomposed into primitive substeps that correspond to monotonic specification changes. This structure is used to rearrange chronological derivation sequences into structures containing only meaning-preserving changes. The authors indicate how this structure can be used to automatically combine different changes to a specification. A set of examples illustrates the ideas.

Key words Software Engineering, Prototyping, Software Evolution.

1. Introduction

Evolutionary prototyping provides an efficient approach to formulating accurate software requirements[1]. 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 available. 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[2-5]. However, it has been recognized that in realistic contexts, many changes do not preserve the observable behavior of the

---

* Project supported in part by the National Science Foundation of USA under grant number CCR-9058453 and by the Army Research Office of USA under grant number ARO 111-95.

©1996 by Allerton Press, Inc. Authorization to photocopy individual items for internal or personal use, or the internal or personal use of specific clients, is granted by Allerton Press, Inc. for libraries and other users registered with the Copyright Clearance Center (CCC) Transactional Reporting Service, provided that the base fee of $50.00 per copy is paid directly to CCC, 222 Rosewood Drive, Danvers, MA 01923.
Most of the work on the area of meaning-changing transformations has been concerned with classifying the types of semantic modifications that are actually used in practice. 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.

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 introduces some notation and 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

2.1 Classification

We characterize changes to a system specification in terms of three orthogonal attributes of a system: its vocabulary, its behavior, and its granularity. 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 usual 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 Ref. [11].

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Effect of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocabulary</td>
<td>extending, contracting</td>
</tr>
<tr>
<td>Granularity</td>
<td>refining, abstracting</td>
</tr>
<tr>
<td>Behavior</td>
<td>relaxing, constraining</td>
</tr>
</tbody>
</table>

Fig. 1. Types of changes.

A decomposition of the chronological evolution history into primitive substeps conforming to these restrictions enables the rearrangement of a sequential derivation containing arbitrary changes into a tree-like rooted directed acyclic graph whose paths consist solely of meaning-preserving changes 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, to factor them out of the actual chronological derivation, and 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 and explained further in section 3.3.

We propose this mechanism as a concrete means to document software as if it had been developed using a rational process [12], 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 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 user 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 user 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.

2.2 Incompatible changes vs. refinements

Incompatible changes are different from refinements because they change the requirements. This implies that validation, and hence explicit explanation and understanding are needed. A properly structured derivation history should help to formulate understandable explanations. The explanation of the original requirements is often similar to the explanation of the changed 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 non-monotonic changes into contractions and re-expansions, 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 incompatible changes differ from explanations of refinements 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 identified in validating a change such as the change 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.

3. Case Study

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

3.1 Notation

In this section we represent specifications using the Spec language\textsuperscript{[13]}. 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\textsuperscript{[14]} 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 Ref.[14].
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 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\textsuperscript{[15]}, and is easily readable without further explanation. PSDL is the prototyping language used in the CAPS (computer aided prototyping system)\textsuperscript{[16]}.

3.2 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.

![Fig.2. Context diagram.](image)

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 to 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 Ref.[14].

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 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.

![Fig.4. Initial decomposition.](image)

After realizing the above components via reusable software components and meaning-preserving modifications, 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 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 the 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.

![Diagram](image)

Fig. 7. Decomposition of spell version 2.

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 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. 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.

We note that some of these changes are needed to restore the integrity of the specification after a desired change. For example, the contracting and extending changes 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 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 changes, 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.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 
\[(x \in \text{union}(s_1, s_2)) \iff (x \in s_1) \& (x \in s_2)\]. 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.

3.3 Conceptual derivation histories

It can be useful to arrange derivation histories in graph structures in which arcs represent monotonic changes. Such a graph is a partially ordered set with respect to the partial ordering \( \sqsubseteq \) over specifications that is defined as follows.

\[
p \sqsubseteq q \iff \text{vocabulary}(p) \subseteq \text{vocabulary}(q) \land \text{granularity}(p) \subseteq \text{granularity}(q) \land \text{behavior}(p) \supseteq \text{behavior}(q)
\]

The vocabulary, granularity, and behavior of a specification are defined in section 2.1. The ordering \( p \sqsubseteq q \) means that \( q \) is 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 \( p \sqsubseteq q \) 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 latter possibility 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 changes 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 changes 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 changes on orthogonal attributes of the system as much as possible, so that these independent changes can be automatically re-combined in different combinations. The problem of automatically combining different versions of programs has been formally studied in several different contexts\cite{10,17-22}, and has been informally discussed in terms of the development of requirements in Ref.[23], where the independence of elaborations was assessed manually. However, the problem has not yet been solved completely, particularly
Software Evolution in Prototyping

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, this is potential for parallel elaborations whenever the structures 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 merged. The model proposed here is a possible basis for extending some previous work on merging\textsuperscript{19,22} to cases where the vocabulary changes. Section 4 sketches some of the main ideas for this extension.

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 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 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.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig10.png}
\caption{Vocabulary refinement structure of the derivation history.}
\end{figure}
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 restrictions, and semantic interpretations, then the description 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 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.

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 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 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 different vocabularies are not directly comparable. Hence the behavioral structures will be small.

4. Combining Changes

The Boolean algebra structure of the vocabulary, granularity, and behavior of a specification identified in section 2.1 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.

$$A[B]C = (A - B) \cup (A \cap C) \cup (C - B)$$

Here $\cup$ denotes the least upper bound and $\cap$ denotes the greatest lower bound with respect to the ordering defined in section 3.3. The difference is defined by

$$A - B = A \cap \overline{B}$$

where the bar denotes the complement operation. This operation is closely related to set difference, but a bit of care is needed because the set inclusions for the system behavior go in the opposite direction to those for the vocabulary and the granularity. It is common to represent sets of behaviors by logical assertions representing postconditions. If we let $\text{post}(A)$ denote the postcondition for a specification $A$, then the concrete interpretations of these abstract operations are given by the following.

\[
\begin{align*}
\text{vocabulary}(A \cup B) &= \text{vocabulary}(A) \cup \text{vocabulary}(B) \\
\text{granularity}(A \cup B) &= \text{granularity}(A) \cup \text{granularity}(B) \\
\text{behavior}(A \cup B) &= \text{behavior}(A) \cap \text{behavior}(B) \\
\text{post}(A \cup B) &= \text{post}(A) \land \text{post}(B) \\
\text{vocabulary}(A \cap B) &= \text{vocabulary}(A) \cap \text{vocabulary}(B) \\
\text{granularity}(A \cap B) &= \text{granularity}(A) \cap \text{granularity}(B) \\
\text{behavior}(A \cap B) &= \text{behavior}(A) \cup \text{behavior}(B) \\
\text{post}(A \cap B) &= \text{post}(A) \lor \text{post}(B) \\
\text{vocabulary}(A - B) &= \text{vocabulary}(A) - \text{vocabulary}(B) \\
\text{granularity}(A - B) &= \text{granularity}(A) - \text{granularity}(B)
\end{align*}
\]
behavior(A - B) = behavior(B) - behavior(A)
post(A - B) = post(A) \lor \neg post(B)

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_1 from the starting vocabulary \{m_1, m_2\} and the change that adds m_3 to the same starting vocabulary is calculated as follows:

\{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\}) \cup
  (\{m_1, m_2, m_3\} - \{m_1, m_2\})
= \{m_1, m_3\}

The corresponding calculations on postconditions representing behaviors are a bit less obvious. If P, Q, and R are assertions representing postconditions, we can apply the general definition and make simplification to give the following rule:

P[Q] R = (P \lor \neg Q) \land (P \lor R) \land (R \lor \neg Q)
= (P \land R) \lor ((P \lor R) \land \neg Q)

We illustrate the consequences of this rule for some common change patterns. The combination of two different constraining changes to a behavior is the result of applying both constraints:

a \land b[a]a \land c = a \land b \land c

The combination of a relaxing change and an incompatible change shows the effects of both:

a[a \land b]a \land c = a \land (c \lor \neg b)

Two relaxing changes similarly show the effects of both changes:

a[a \land b] = a \lor b

All of the above examples follow directly from the definition and some simplification using the laws of ordinary propositional logic. The implementation of the change merging definitions for specifications is straightforward, just as the implementation of weakest preconditions for loop-free code. The difficulty in automating specification merging is the simplification step: since most logics that are useful for specification are not decidable, and 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".

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\textsuperscript{[24]}. This model has been applied to automate configuration management and a variety of project management functions\textsuperscript{[25]}. 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-changing 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


