A Category-Theoretic Approach to Syntactic Software Merging

Nan Niu, Steve Easterbrook, and Mehrdad Sabetzadeh
Department of Computer Science, University of Toronto
Toronto, Ontario, Canada M5S 3G4
Email: {nn,sme,mehrdad}@cs.toronto.edu

Abstract

Software merging is a common and essential activity during the lifespan of large-scale software systems. Traditional textual merge techniques are inadequate for detecting syntactic merge conflicts. In this paper, we propose a domain-independent approach for syntactic software merging that exploits the graph-based structure(s) of programs. We use morphisms between fuzzy graphs to capture the relationships between the structural elements of the programs to be merged, and apply a truth ordering lattice to express inconsistencies and evolutionary properties as we compute the merge. We demonstrate the approach with a three-way consolidation merge in a commercial software system; in particular, we show how analyzing merged call structures can help developers gain a better understanding and control of software evolution.

1 Introduction

Parallel changes, in which separate lines of development are carried out by different software developers, are a basic fact of developing and maintaining large-scale software systems [17]. An optimistic version control mechanism [3] allows every developer to work on a local copy of the software artifact independently. Thus, a fundamental and important problem in building and evolving complex large-scale software systems is how to *merge* parallel versions and variants of a software product to yield a consistent shared view.

Traditional merge tools are built using textual merge techniques. This gives the tools high flexibility, since any program can be treated as a flat text file. An intrinsic disadvantage of textual merging is that only very basic conflicts can be identified due to the lack of structured syntactic and semantic knowledge. This considerably limits textual merge techniques' analytical power. Another problem is that textual merge techniques focus mainly on software systems at the source code level. This makes detecting inconsistencies at higher levels of abstraction very difficult. As a result, the software system's overall picture is overlooked in the textual merge process.

Graph-based representations are used frequently in Software Engineering as an aid to comprehension. Different graph-based notations are used to model various software artifacts, such as requirements, specifications, architecture, design, and so on. Such graphical representations can also be extracted from legacy code using reverse engineering tools, and these representations have been demonstrated to be useful for program understanding [15]. Hendrix and Cross [10] proposed a language-independent approach to generating graphical representations of source code that supports both forward and reverse engineering. Given that reverse engineering has improved continuously in handling legacy systems and that huge potential benefits can be gained from the effective use of graphical representations of software artifacts, we have focused on developing graph-based merge techniques in the context of software reverse engineering and reengineering.

In our previous work [20], we developed a framework for merging requirements models, based on Goguen's fuzzy set category-theoretic formalism [8]. The approach is robust in the face of inconsistency, and has the advantage that structural inconsistencies are explicitly flagged in the merged model, and can be traced to their sources. In this paper, we adapt the idea for merging source code. In our approach, we treat each program as a graph, and annotate its nodes and arrows with the elements of a lattice to specify how they have been modified in different versions of the software. By defining an appropriate complete lattice to express the evolutionary ordering, we construct a finitely cocomplete fuzzy set category. We then show that a set of interconnected software artifacts can be merged by constructing the colimiting graph in this category.

Our aim is to lay a foundation for the graph-based merge techniques that exploit categorical formalisms and lead to further automation. Our proposed approach to modeling inconsistencies and analyzing evolutionary properties is very general and does not depend on any specific programming language or particular type of application. To demonstrate

the idea, we present a study showing the approach's application to a commercial software system. Our purpose is to identify the graph-based syntactic software merging problem, demonstrate the feasibility of our approach in a real-world setting, discuss the preliminary results, share our experience, and open up new research avenues arising from our investigation.

The remainder of the paper is organized as follows: In Section 2, we motivate the paper by means of an illustrative example and further explore truth orderings in parallel software development and maintenance. Section 3 reviews basic concepts in category theory and fuzzy set categories. We propose a category-theoretic approach to syntactic software merging in Section 4. Section 5 presents an application of our approach. We discuss related work and conclude the paper with a summary and some directions for future work in Section 6.

2 Motivation and Evolutionary Orderings

In this section, we first identify the problems associated with textual merging. Then we briefly review the definitions of graphs, graph homomorphisms, posets, and lattices. The purpose is to explore evolutionary orderings in parallel software development and to build the context of the graph-based merge technique used in our approach.

2.1 Running Example

The example of Figure 1 shows a *three-way merge* problem for a C program: we need to merge two parallel versions that share a common ancestor. The program fragment implements the routine of entering the top score of a LAN-supported BBS door game. The original implementation (version 1) sends the player plain text emails informing her of potential awards if her score passes a pre-defined threshold. Parallel changes are made by versions 1a and 1b. To divide and conquer, tasks are assigned complementarily: 1a focuses on the modification of award policies, whereas 1b adds HTML email features to the program module.

The middle boxes of Figure 1 highlight only the changes made by 1a and 1b, not the entire code fragments. Version 1a deletes the function **get_point** and defines a new boolean function **pass_rank** to revise the award policy. **send_award** is rewritten by 1b, but the assignment of the newly introduced variable *p* is made by calling the function **get_point**.

A possible textual merge result is shown at the bottom of Figure 1, where differences from version 1 are underlined. Although the merged program is syntactically correct, it has semantic errors. The most obvious problem is that version 1a's discarding of the function **get_point** does not take effect in the merged result, because version 1b keeps this part of the program structure intact. Consequently, a wrong

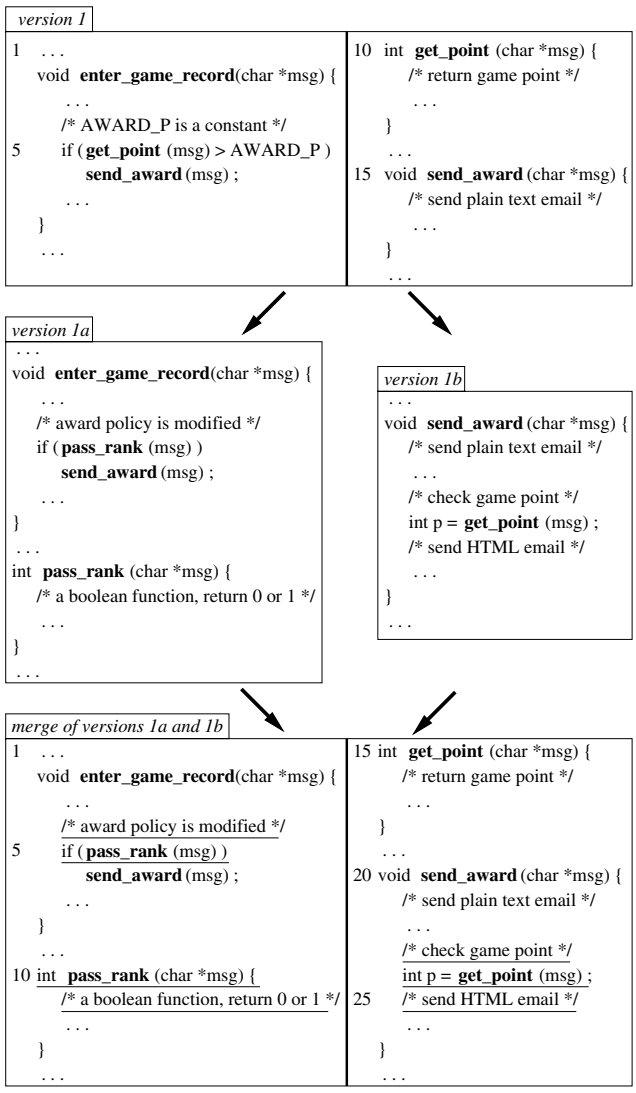


Figure 1. Three-way textual merge example

value is obtained for variable *p* (line 24) since the award policy is updated in **pass_rank**. Furthermore, the return type of **pass_rank** is essentially boolean. But the function declaration coincidentally remains *int* due to the lack of *bool* data type in C. This contingency, accompanied by the change of business logic, may cause later uses of *p* in **send_award** to behave inappropriately.

Figure 1 also demonstrates the importance of treating this as a three-way merge problem. If we attempt to compare and merge just versions 1a and 1b, without considering their common ancestor, we cannot determine how to treat lines 4 and 5 in the merged version. Version 1a contains "modified policy" and the function **pass_rank**, while the corresponding lines in 1b include "constant AWARD_P" and **get_point**. This generates nondeterminism for most two-way textual merge algorithms, and a syntactic conflict

is reported. Three-way merging overcomes this shortcoming: "constant AWARD_P" and **get_point** of version 1b are also present in the parent version, implying that only 1a makes modifications to those lines. This extra knowledge helps the algorithm to infer which later version's information should be included in the merged result. However, if 1a's changes take precedence during merging, we may encounter new problems. For example, domain knowledge reveals a crisp separation of concerns between the parallel versions 1a and 1b. Thus, in accordance with 1a's revision, the function **get_point** should be removed from the merged version because 1b is concerned only with modifying **send_award**. But such a removal will cause a syntactic error in the merged result, because p is defined by making an explicit invocation to **get_point**.

We are aware that the program structure of the above example could be refactored to achieve low couplings among functions so that some merge conflicts may be circumvented. However, we intentionally choose such a "lousy" structure in our running example for the following reasons: The source code shown in Figure 1 is derived from our study on a commercial system; and we believe this example is representative of many existing legacy systems.

2.2 Graphs and Graph Homomorphisms

To overcome the limitations of textual merge techniques, we treat programs as graphs, and the merge process as a process of merging graphs. To obtain a sensible merge, the relationships between the programs must be captured as interconnections between their graphs. We use graph homomorphisms as the basis for the interconnections, although, as we shall see, we need to go beyond homomorphisms to gain the appropriate expressive power.

Definition 2.1 A *graph* is a quadruple $\mathcal{G} = (N, A, s_\mathcal{G}, t_\mathcal{G})$ where N is a set of nodes, A is a set of arrows, and $s_\mathcal{G}, t_\mathcal{G} : A \to N$ are functions giving the source and the target for each arrow in $\mathcal{G}$ respectively.

Definition 2.2 A *graph homomorphism* ϕ from a graph $\mathcal{G}$ to a graph $\mathcal{G}'$ is a pair of functions $\phi_0 : N \to N'$ and $\phi_1 : A \to A'$ such that if u is an arrow of $\mathcal{G}$, then $\phi_1(u)$ is an arrow of $\mathcal{G}'$ with $s_{\mathcal{G}'}(\phi_1(u)) = \phi_0(s_\mathcal{G}(u))$ and $t_{\mathcal{G}'}(\phi_1(u)) = \phi_0(t_\mathcal{G}(u))$.

Figure 2 sketches the running example's merge scenario using call graphs. For each graph, nodes are functions and arrows are call relations. The source and the target of an arrow represent the caller and the callee of the call relation respectively. Note that if **pass_rank** in $\mathcal{G}_A$ is regarded as a newly added function by version 1a, then the morphism from $\mathcal{G}_C$ to $\mathcal{G}_A$ fails to be a graph homomorphism because, for example, the node **get_point** of $\mathcal{G}_C$ does not have an image in $\mathcal{G}_A$. Figure 2 provides only a background example

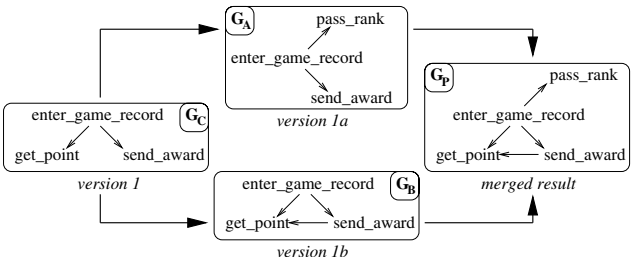


Figure 2. Graph-based merge example

of our graph-based merge technique, and we will elaborate the more rigorous construction in the following sections.

2.3 Partially Ordered Sets and Lattices

We now construct a series of posets and lattices to provide sets of labellings for nodes and arrows of our program graphs, which express how those graph elements evolve.

Definition 2.3 A *partial order* is a reflexive, antisymmetric, and transitive binary relation. A non-empty set with a partial order on it is called a *partially ordered set* or a *poset*. We use Hasse diagrams [4] to visualize finite posets.

Definition 2.4 Let $(A, \leq)$ be a poset and $S \subseteq A$. An element $v \in A$ is an *upper bound* of S if $\forall s \in S : s \leq v$. If v is an upper bound of S and $v \leq w$ for all upper bounds w of S, then v is called the *supremum* of S. *Lower bound* and *infimum* are defined dually. We write $\bigsqcup_A S$ (respectively $\bigsqcap_A S$) to denote the supremum (resp. infimum) of $S \subseteq A$, when it exists. If both $\bigsqcup_A \{a, b\}$ and $\bigsqcap_A \{a, b\}$ exist for any $a, b \in A$, then A is called a *lattice*. If both $\bigsqcup_A S$ and $\bigsqcap_A S$ exist for any $S \subseteq A$, then A is called a *complete lattice*.

Lemma 2.5 *(cf. e.g., [4]) Every finite lattice is complete.*

Since three-way merging can both detect more conflicts [14] and generate more decisive merge results (as shown in the running example) than its two-way variant, information about the common ancestor is taken into account in our construction. Figure 3 presents four ordered sets in Hasse diagrams. A_3 provides a set of labels to show how program elements from the common ancestor can evolve. In the ancestor, initial (I) [1] indicates no evolution yet; subsequent versions will either preserve (P) or remove (R) such elements. A_2 provides labels for the converse type of evolution, where new elements are added (A) that were nonexistent (N) in the ancestor.

If we attempt to merge two parallel versions that are created from a common parent, we need the Cartesian products of these posets. For example, the products $A_9 = A_3 \times A_3$ and $A_4 = A_2 \times A_2$ are shown in Figure 3. The first (resp.

[1] For succinctness, we just use the first letter of each element, where this does not cause ambiguity.

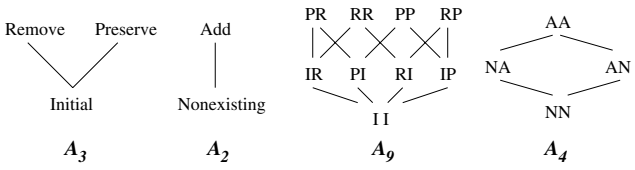


Figure 3. Orderings A_3, A_2, A_9, and A_4

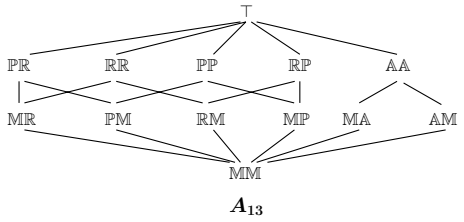


Figure 4. Evolutionary ordering A_{13}

second) component of each element in the product represents the first (resp. second) evolved version's knowledge. Based on this labeling mechanism, if we consider the running example of Figure 2, the node **send_award** in $\mathcal{G}_A$ and the arrow from **send_award** to **get_point** in $\mathcal{G}_B$ will be annotated with the values PI of A_9 and NA of A_4 respectively.

Finally, we need to combine both sets of labels to capture the full range of program element evolutions. In order to amalgamate A_9 and A_4, we introduce a single bottom value $\mathbb{M}$ (Maybe) that combines I and N. For reasons that will become clear in Section 3, we focus our discussion here on complete lattices. Therefore, a top element ($\top$, meaning "incompatible") is imposed to form a complete lattice. The result, A_{13}, is shown in Figure 4.

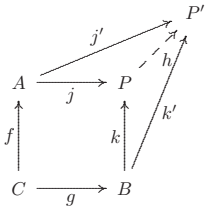
3 Category-Theoretic Preliminaries

3.1 Category Theory

The operation of merging an interconnected set of graphs corresponds to the colimit operation in category theory. We therefore appeal to category theory to provide the appropriate constructs, and proofs of their existence. In this paper, we will assume some familiarity with basic concepts of category theory and only make use of pushouts, rather than the more general colimits. An excellent introduction to category theory from a computer science perspective is [1].

Definition 3.1 $\mathbf{Set}$ is the category whose objects are sets and whose morphisms are total functions. $\mathbf{Grf}$ is the category whose objects are graphs and whose morphisms are graph homomorphisms.

Definition 3.2 A *pushout* of a pair of morphisms $f : C \to A$ and $g : C \to B$ in a category $\mathscr{C}$ is a $\mathscr{C}$-object P together with a pair of morphisms $j : A \to P$ and $k : B \to P$ such that: $j \circ f = k \circ g$; and for any $\mathscr{C}$-object P' and pair of morphisms $j' : A \to P'$ and $k' : B \to P'$ satisfying $j' \circ f = k' \circ g$, there is a unique morphism $h : P \to P'$ such that the following diagram commutes:



The canonical pushout construction of a pair of $\mathbf{Set}$-morphisms is described in [20]. An equivalent definition for graph and graph homomorphism can be given by noticing that a graph is a many-sorted algebra and a graph homomorphism is a many-sorted homomorphism [19]. This indicates that (many-sorted) set is the underlying construct for graph and graph homomorphism; and the pushout in $\mathbf{Grf}$ can be computed component-wise for nodes and arrows by applying the canonical construction of pushout in $\mathbf{Set}$.

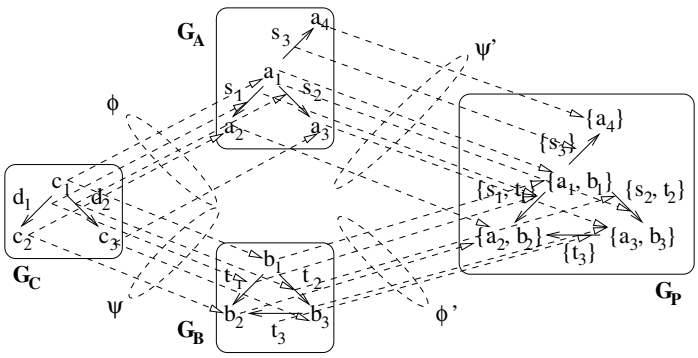


Figure 5. Pushout example in $\mathbf{Grf}$

Pushout formalizes the three-way merge technique discussed in this paper. For example, Figure 5 illustrates that the pushout of a pair of morphisms $\phi : \mathcal{G}_C \to \mathcal{G}_A$ and $\psi : \mathcal{G}_C \to \mathcal{G}_B$ in $\mathbf{Grf}$ can be considered as the combination of $\mathcal{G}_A$ and $\mathcal{G}_B$ with respect to a shared part $\mathcal{G}_C$. The program graphs of Figure 5, ($\mathcal{G}_C$, ϕ, ψ), are chosen from the running example's call graphs of Figure 2, with the following changes: Arrows are specified explicitly; a new node (a_2) is added to $\mathcal{G}_A$ to correctly render the graph homomorphism ϕ; nodes or arrows in different graphs do not share the same names; and the naming schema seeks concision and conformity with the canonical pushout construction's guidelines [19].

3.2 Categories of Fuzzy Sets

Since its inception in the 1960s, fuzzy set theory has received considerable attention from different computing disciplines. This section presents the definitions and results of fuzzy set categories [8] needed in the paper. We have ex-

tended the term "fuzzy" to posets and lattices so that the truth values in our framework differ in nature from the linearly ordered real interval $[0, 1]$.

Definition 3.3 Let A be a poset. An *A-valued set* is a pair (S, σ) that consists of a set S and a function $\sigma : S \to A$. We call S the *carrier set* of (S, σ) and A the *truth-set* of σ. For every $s \in S$, the value $\sigma(s)$ is interpreted as the *degree of membership* of s in (S, σ).

Definition 3.4 Let (S, σ) and (T, τ) be two A-valued sets. A *morphism* $\mathbf{f} : (S, \sigma) \to (T, \tau)$ is a function $f : S \to T$ such that $\sigma \leq \tau \circ f$, i.e., the degree of membership of s in (S, σ) does not exceed that of $f(s)$ in (T, τ). The function $f : S \to T$ is called the *carrier function* of $\mathbf{f}$.

Lemma 3.5 *For a fixed poset A, the objects and morphisms defined above, together with the obvious identities, give rise to a category, denoted* $\mathbf{Fuzz}(A)$.

Figure 6 (informally) shows two $\mathbf{Fuzz}(A_4)$ objects $(\mathcal{G}_C, \gamma)$ and $(\mathcal{G}_A, \alpha)$ along with the carrier function $\phi : \mathcal{G}_C \to \mathcal{G}_A$. Here, we adopt $\mathcal{G}_C$, $\mathcal{G}_A$, and ϕ from Figure 5 to emphasize the facts that graphs are special (many-sorted) sets and morphisms between fuzzy graphs are handled component-wise for nodes and arrows. In this paper, we omit the formal procedure for constructing fuzzy graphs. The interested reader should refer to [19] for further details. The truth-set A_4, which is adopted from Figure 3, is depicted in the same figure using Hasse diagram, with the lines connecting truth values widened.

Theorem 3.6 *All the pushouts exist for* $\mathbf{Fuzz}(A)$ *when* A *is a complete lattice.*

Proof of this theorem can be found in [20].

To guide our approach to graph-based software merging, we spell out the procedure for computing fuzzy graph pushouts without directly using the underlying category-theoretic constructs. Let A be a complete lattice. For computing the pushout of a pair of $\mathbf{Fuzz}(A)$-morphisms $\boldsymbol{\phi} : (\mathcal{G}_C, \gamma) \to (\mathcal{G}_A, \alpha)$ and $\boldsymbol{\psi} : (\mathcal{G}_C, \gamma) \to (\mathcal{G}_B, \beta)$, first compute the canonical $\mathbf{Grf}$-pushout of the carrier morphisms $\phi : \mathcal{G}_C \to \mathcal{G}_A$ and $\psi : \mathcal{G}_C \to \mathcal{G}_B$ in order to find a graph $\mathcal{G}_P$ along with graph homomorphisms $\psi' : \mathcal{G}_A \to \mathcal{G}_P$ and $\phi' : \mathcal{G}_B \to \mathcal{G}_P$. Then, compute a membership degree for every $p \in \mathcal{G}_P$ (component-wise for nodes and arrows) by taking the *supremum* of the membership degrees of all those elements in $(\mathcal{G}_A, \alpha)$ and $(\mathcal{G}_B, \beta)$ that are mapped to p. This yields an object $(\mathcal{G}_P, \rho)$ and lifts ψ' and ϕ' to $\mathbf{Fuzz}(A)$-morphisms, all of which constitute the pushout of $\boldsymbol{\phi}$ and $\boldsymbol{\psi}$ in $\mathbf{Fuzz}(A)$.

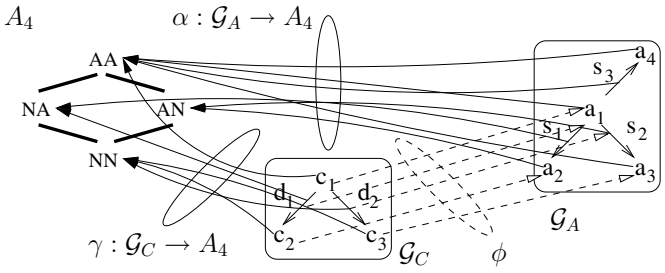


Figure 6. Example of fuzzy graphs

4 A Category-Theoretic Approach to Syntactic Software Merging

When merging arbitrary software artifacts, it is impossible to guarantee that there are no undesirable interactions, because any non-trivial property of a program's execution behavior is undecidable [14]. For this reason, we are compelled to make some assumptions about the general structure of the programs to be merged. We restrict our discussion to *graph-based software merging*. We assume that it is sufficient to represent the structure of the programs as graphs, and that the relationships between versions of a program can be expressed as structural mappings between these graphs.

In our approach, we represent the programs to be merged as objects of the category $\mathbf{Fuzz}(A)$ for some fixed complete lattice A. A diagram in $\mathbf{Fuzz}(A)$ can be regarded as a family of different versions or variants of a software system in which software artifacts are represented graphically by $\mathbf{Fuzz}(A)$-objects and artifacts' interconnections are represented by $\mathbf{Fuzz}(A)$-morphisms. The result given in Theorem 3.6 states that the pushout exists for any pair of morphisms with a common domain [2].

It is worth pointing out that software merging can range from a manual – and often time-consuming – process, over a semi-automated process that requires interaction with the user, to a fully automated approach. Only in very specific situations is it possible to fully automate the merge process [14]. This is why we consider our semi-automated approach to be useful. By applying graph-based software merging together with experience, domain knowledge, and common sense, the user can build intuition about the context, rationale, scope, and intent of software evolution. However, we do not explicitly address how this might be systematically achieved in this paper, as it is beyond of the scope of our current work.

[2] Note that this result generalizes to allow us to integrate any finite set of graph-based software artifacts with known interconnections by constructing the colimit.

4.1 Preprocessing

Preprocessing takes advantage of reverse engineering techniques to transform source codes into graph-based software artifacts, such as call graphs, attributed graphs, and the like. The process of analyzing a subject software to create graphical representations of the system's structure at higher levels of abstraction is well studied, and many tools are available in the reverse engineering literature.

Figure 7 highlights the process, with bi-directional arrows showing correspondences. The dashed "inclusion" arrow renders a graph homomorphism. Single-directed arrows represent actual operations. Rigi [15] is an interactive, visual tool designed to help understand and reengineer legacy systems. We used Rigi for our initial study, but we anticipate that any program comprehension tool that offers graphical representations of a system structure will fit into this process.

Coping with huge amounts of data is one of the major problems associated with software evolution, as several versions of the same software must be analyzed in parallel. Current approaches reduce this complexity by filtering out irrelevant information. The "projection" operation in Figure 7 facilitates abstraction through subsystem identification, which is end-user programmable in Rigi. This makes subsystem identification semi-automated by leveraging application or programming domain knowledge. For example, all nodes labeled with a common prefix according to some naming convention can be collapsed into a single subsystem. One can drill down into the subsystem of interest from source codes directly, but this disregards various graphical features provided by a visual reverse engineering tool. Projection is not mandatory, but it is often necessary in practice.

4.2 Pushout Diagram Construction

A typical three-way merge scenario is depicted in Figure 8a. Graphs $\mathcal{G}_1$, $\mathcal{G}'_A$ and $\mathcal{G}'_B$ are preprocessing results of versions 1, 1a, and 1b, respectively. Newly added parts by both later parallel versions are captured by the connector graph $\mathcal{G}_{new}$. Figure 8b sketches this concept, in which $\iota_{A'}$ and $\iota_{B'}$ are inclusion homomorphisms.

It is important to make $\mathcal{G}_1$ and $\mathcal{G}_{new}$ disjoint. This implies that a program entity (node or arrow) captured in $\mathcal{G}_{new}$ must be *really* new and should not be viewed as a renamed, moved, or otherwise changed version of an entity from $\mathcal{G}_1$. While simple renaming and moving of program entities are easy to define formally and fairly easy to detect, the more general concept of matching entities in different versions is not. Many techniques for identifying correspondence in software evolution have been proposed, among which *original analysis* [21] is a semi-automated process tailored for deciding newly introduced program entities among different versions. Tu and Godfrey's prototype tool, Beagle [21],

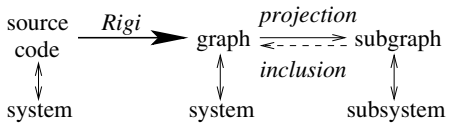


Figure 7. Preprocessing

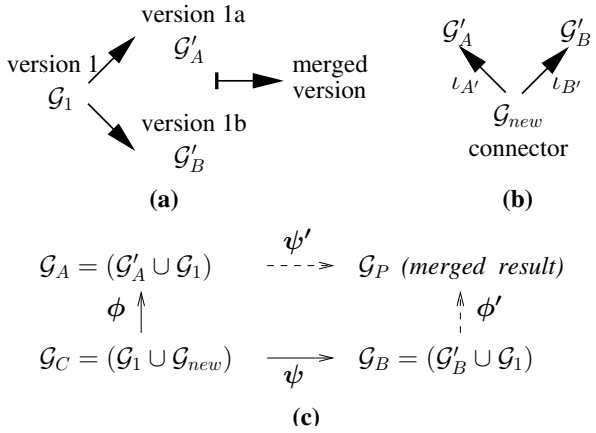


Figure 8. Pushout construction in $\mathbf{Fuzz}(A_{13})$

supports original analysis and can be used to construct $\mathcal{G}_{new}$.

Figure 8c shows the pushout diagram, $(\mathcal{G}_C, \phi : \mathcal{G}_C \to \mathcal{G}_A, \psi : \mathcal{G}_C \to \mathcal{G}_B)$, in the category of $\mathbf{Fuzz}(A_{13})$. We adopted the idea of code clone detection to help in building correspondences from the nodes and arrows of version 1 ($\mathcal{G}_1$) to later versions' ($\mathcal{G}_A$ and $\mathcal{G}_B$) counterparts. A code clone [11], which is a code portion in source files identical or similar to another code fragment, is considered a serious problem in industrial software because it creates difficulties in maintaining source files consistently. Various clone detection tools have been proposed and implemented. In our study, we used CCFinder [11] to gather clones from the candidate software systems and to help construct mappings from $\mathcal{G}_1$ to $\mathcal{G}_A$ and $\mathcal{G}_B$.

The morphisms to the truth-set A_{13} are omitted to keep Figure 8c concise. We now describe the sequence of the labeling operations of the elements (nodes and arrows) in $\mathcal{G}_C$ and $\mathcal{G}_A$ (resp. $\mathcal{G}_B$) to build the truth-set mapping for the $\mathbf{Fuzz}(A_{13})$-morphism ϕ (resp. ψ):

1. Label every element in $\mathcal{G}_1$ of $\mathcal{G}_C$ with $\mathbb{MM}$;
2. Label every element in $\mathcal{G}_{new}$ with $\mathbb{AA}$;
3. Label every element in $\mathcal{G}_A \setminus \mathcal{G}'_A$ (resp. $\mathcal{G}_B \setminus \mathcal{G}'_B$) with $\mathbb{RM}$ (resp. $\mathbb{MR}$);
4. Label every element in the image of the carrier function $\phi(\mathcal{G}_{new})$ (resp. $\psi(\mathcal{G}_{new})$) with $\mathbb{AA}$;
5. Label every element in $\mathcal{G}_A \setminus \mathcal{G}_C$ (resp. $\mathcal{G}_B \setminus \mathcal{G}_C$) with $\mathbb{AM}$ (resp. $\mathbb{MA}$); and
6. Label all remaining elements in $\mathcal{G}_A$ (resp. $\mathcal{G}_B$) with $\mathbb{PM}$ (resp. $\mathbb{MP}$).

The condition for all annotating operations defined in the sequence is the following: A subsequent operation *cannot* override the labeling effects of previous operations. It can be checked that the given labeling sequence and the carrier functions $\phi : \mathcal{G}_C \to \mathcal{G}_A$ and $\psi : \mathcal{G}_C \to \mathcal{G}_B$ give rise to a pushout diagram in $\mathbf{Fuzz}(A_{13})$.

4.3 Pushout Computation and Analysis

Given the pushout diagram $(\mathcal{G}_C, \boldsymbol{\phi}, \boldsymbol{\psi})$, the pushout object $\mathcal{G}_P$, along with morphisms $\boldsymbol{\phi'} : \mathcal{G}_B \to \mathcal{G}_P$ and $\boldsymbol{\psi'} : \mathcal{G}_A \to \mathcal{G}_P$, can be computed automatically in $\mathbf{Fuzz}(A_{13})$ by applying the procedure described in Section 3. The dashed arrows in Figure 8c indicate this automated construction for the merged result.

A fundamental problem in visualizing software changes is the choice of effective visual representations for data that are not inherently physical. The goal is an insightful rather than a faithful depiction of the data [5]. Multiple views are often shown side-by-side in current visual reengineering tools. This increases developers' cognitive overhead when they are in the process of understanding parallel software modifications. In our approach, the pushout result $\mathcal{G}_P$ in $\mathbf{Fuzz}(A_{13})$ characterizes much evolutionary information within one view. This provides developers with a centralized vision to comprehend and analyze the merge process more effectively and efficiently.

The truth ordering lattice A_{13} offers interpretations of software artifacts according to certain semantics. In our framework, a system of interconnected graphs in $\mathbf{Fuzz}(A_{13})$ is *syntactically inconsistent* if the colimit of the diagram corresponding to the system has some node or arrow with an inconsistent truth value.

It is up to the maintainers to designate (in)consistent values of A_{13}. For example, we may regard $\mathbb{RR}$, $\mathbb{PP}$, and $\mathbb{AA}$ as consistent and the rest of the values as inconsistent. This is a reasonable choice when the system we are modeling mandates the total agreement of both parallel versions in every aspect. If we are only interested in explicit conflicts and incompatibilities, we can relax this constraint and only designate $\mathbb{PR}$, $\mathbb{RP}$, and $\top$ as inconsistent. If a three-way consolidation merging occurs, in which most of the parallel revisions are complementary (e.g., when changes are made to different subsystems, workspaces, or modules) [16], $\mathbb{RR}$ and $\mathbb{AA}$ may be deemed inconsistent as well.

A limitation to our proposed approach is that nodes and arrows are treated equally when graph-based software artifacts are annotated. We deliberately choose to use the same evolutionary ordering lattice for labeling, as our intention is to explore the graph-based syntactic software merging problem. From this perspective, A_{13} is a suitable truth ordering candidate, since it provides the basic expressive and analytical power to capture the intuitive nature of software evolution. However, we have constructed pushouts in $\mathbf{Grf}$ by handling nodes and arrows component-wise. Following Theorem 3.6, nodes and arrows of graph-based software artifacts can have different complete lattices for annotation, where labels of nodes and arrows in the merged pushout graph must be constructed and interpreted separately in two fuzzy set categories.

Nevertheless, having two disparate evolutionary lattices for nodes and arrows does not address the orthogonal problem of modeling different software versions or variants by using different truth orderings. We anticipate that such a limitation could be tackled by introducing *institutions* [9] for consistently transforming signatures, sentences, and models from one logical system into another; however, we have not yet investigated this idea.

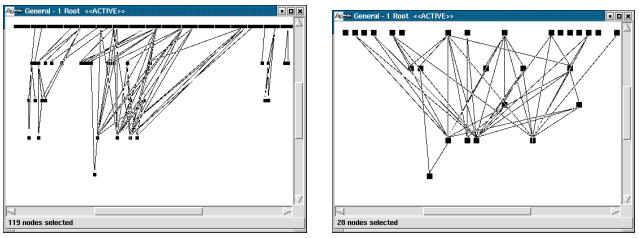
5 A Proof-of-Concept Example

We applied the technique described in this paper to the software produced by a software company in Beijing, PR China, which offered us access to parallel changes in an industrial software system. The company was highly cooperative and generous with regards to our research, sharing not only their data, but also staff time and other resources.

The subject system of our study is a commercial proprietary system developed in-house by a single company. In order to honor confidentiality agreements, we will call it *iBBS*. *iBBS* aims to turn a PC into a customizable online service that supports multiple simultaneous users with hierarchical message and file areas, multi-user chat, and the ever-popular BBS door games.

iBBS's development began in the early 1990s for single-tasking MS-DOS compatible computers and Hayes compatible modems. The program was commercialized and released for both the 16-bit DOS and 32-bit OS/2 platforms to contracted customers, which were mainly LAN-supported local companies. *iBBS* continued to evolve during the 1990s with a specific focus on the Internet community, embracing and integrating standard Internet protocols such as Telnet, FTP, SMTP, POP3, IRC, NNTP, and HTTP. *iBBS* has since been substantially maintained as a BBS package for Win32 and Unix-x86 platforms as both an Internet- and LAN-compatible system. *iBBS* is written in C, and Microsoft SourceSafe is used as the version control system.

In Burd and Munro's studies investigating software evolution [2], they pointed out that the calling structure was one of the most important units of understanding used within the composition of the maintainer's *mental model*. Figure 9a is a Rigi screen dump showing the overall calling structure of one variant of *iBBS*. This graph, in which 119 functions (nodes) and 75 calls (arcs) are presented, exemplifies the scalability and complexity problem encountered in the program comprehension and maintenance process. The projected and more manageable "util" subsystem's call graph



(a) overall system (b) "util" subsystem

Figure 9. Call graphs of *iBBS* (version 1b)

Table 1. Comparison of "util" programs

version	1	1a	1b
overview	WIN32	fix bugs	Linux86
LOC (lines of code)	2,159	2,476	2,722
number of functions	13	19	28
number of calls	10	27	43
cyclomatic complexity	7	22	31

is shown in Figure 9b.

Table 1 summarizes a brief comparison of the common ancestor and two later versions of the "util" subsystem of ***iBBS*** that are under study. Platform-dependent and security-related files are mainly included in the "util" module. Development of versions 1a and 1b took place in parallel, since time-to-market was among the top priorities for such a business application. Changes made by later versions were complementary, but unavoidably had some overlaps. This generated a rich and real environment for a software merging study – after all, one ***iBBS*** package that originated from version 1 was released by incorporating both modifications of 1a and 1b.

Figure 10 shows the interconnection of the call relations from version 1 ($\mathcal{G}_C$) to 1a ($\mathcal{G}_A$) of ***iBBS***'s "util" subsystem. For simplicity, we have adopted some labeling conventions: A rectangle with a natural number and an A_{13} element – such as 1 $\mathbb{MM}$ – denotes a "function of interest" and its corresponding truth value; call relations are also decorated with truth values from A_{13}. The morphism from $\mathcal{G}_C$ to $\mathcal{G}_B$ is analogous, but is omitted due to space constraints. Beagle [21] and CCFinder [11] were used to help extract facts from and build mappings among software artifacts. This examination and determination were performed independently by the first author and one of the original developers of ***iBBS***, and the differences were reconciled. It also should be noted that the results given by auxiliary tools had false negatives (e.g., mappings to some renamed and slightly modified functions were unable to be established). Under these circumstances, more human input was needed.

The pushout of Figure 11 shows the three-way merging result of "util" in ***iBBS***, and yields an insightful view for

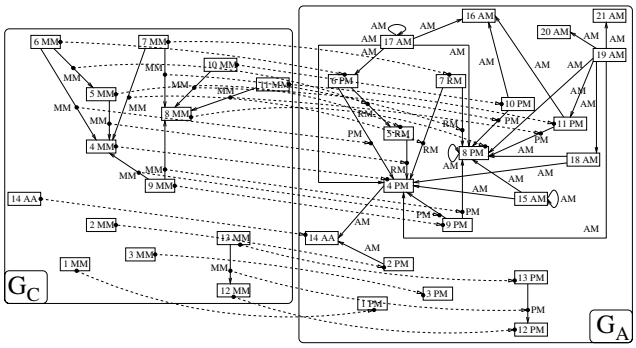


Figure 10. Interconnecting the call relations

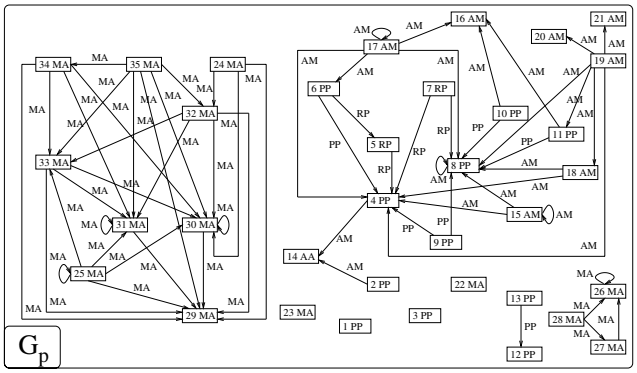


Figure 11. Pushout computation

understanding software evolution. Assuming $\mathbb{PR}$, $\mathbb{RP}$, and $\top$ are designated as inconsistent values in A_{13}, six instances of syntactic inconsistency occur in the pushout surrounding nodes 5 and 7, and four related arrows. Careful code inspection suggests that these functions partly fulfill the message censorship requirement. Version 1a refactored this implementation by removing node 5 and changing function 7 to 15. The security-related structure was preserved by version 1b, whose major task was to add Linux86-compatible files to the software package. The $\mathbb{RP}$-like inconsistencies detected in our framework, which would also catch the running example's problems, did reflect conflicts between parallel changes in ***iBBS***. Additional work was sought to handle these inconsistencies during the merge process.

The fact that many $\mathbb{AM}$- and $\mathbb{MA}$-labeled nodes and arrows appear in the pushout graph illustrates the complementary nature of ***iBBS***'s consolidation merging. Only one function (node 14) is annotated with $\mathbb{AA}$. In discussions with the developers, this revealed a *pre-determined* new function during maintenance. The function prototype was declared *a priori*, permitting both revisions to succeed in unit testing, but the actual implementation was assigned solely to 1a (as indicated by two incoming $\mathbb{AM}$-calls to function 14) to circumvent any inconsistencies. Our study implies the existence of "hot spots" for inconsistency in

$\mathbb{AA}$-annotated entities, so newly added components by both parallel revisions need to be treated with great caution.

Intuitively, $\mathbb{PP}$ captures the base structure of the software system, which is common across all variants. This generally holds for *iBBS*'s security-related components (e.g., nodes 4, 6, 8, 9, and the call relations among them). It also turns out, in hindsight, that some $\mathbb{PP}$-labeled elements in the pushout are *legacy code*. For example, function 3 contributed to supporting UC-DOS 4.0, a Chinese operating system. The review of SourceSafe records implied that *iBBS* stopped supporting UC-DOS two years ago. Therefore, function 3 became *dead code* (i.e., functions not called directly or indirectly).

Another possible A_{13} truth value that can arise in the pushout graph is $\mathbb{RR}$. But this example does not present any $\mathbb{RR}$-annotated nodes or arrows, even though dead code appears in the system. This demonstrates, not surprisingly, Lehman's laws of software evolution: A program hardly ever decreases in size and complexity, and its structure tends to deteriorate as the software system evolves [12].

It is important to point out that this is a single study on one application, rather than a controlled experiment on a number of representative subjects. An important threat to the validity of this study is that our results rely on constructing morphisms among fuzzy graphs. As noted earlier, matching entities in different software versions can be problematic. To address this threat, we used semi-automated tools to increase the accuracy of matching, performed the experiment independently by two different researchers, and reconciled the results, as described previously.

6 Conclusions

6.1 Discussion

In [14], Mens presented an excellent summary of all but the most recent work in software merging, showing the diverse range of techniques employed. Some of the syntactic merge techniques surveyed therein use graphs as the underlying data structure. Rho and Wu [18] use attributed graphs to represent software artifacts. The same is true for Mens [13], who additionally makes use of graph rewriting techniques in order to provide a formal foundation for software merging. We do not prescribe specific structures for graph-based software artifacts in general, and restrict ourselves to call graphs in the study to illustrate our ideas.

The fact that a pair of call graphs (corresponding to a pair of parallel versions of a piece of software) can be merged into a well-formed fuzzy graph in our approach does not guarantee that everything will behave correctly. For example, a $\mathbb{PP}$-annotated arrow is dangling if its target (callee) is labeled with $\mathbb{PR}$. Another semantic constraint inferred from our example is that call graphs must be *simple* [1]; i.e., at most one call relation exists between a given caller and a given callee. To handle such problems, semantic merge techniques need to be sought.

We have exploited original analysis [21] for the construction of morphisms among software artifacts. Godfrey and Zou [7] have extended original analysis to aid in the detection of merging and splitting of files and functions in procedural code. They identify a set of merge/split patterns, such as service consolidation, clone elimination, and pipeline contraction, to show how reasoning about call relationships can help software developers locate merge/split occurrences in software evolution. We note that their work primarily concentrates on the sequential development of a software system, whereas we have focused on merging parallel changes. Our approach enhances the overall competence in parallelism and inconsistency management for the merge techniques derived from original analysis.

Our introduction of partially ordered sets for constructing fuzzy set categories differs substantially from the commonly used linear ordering $[0, 1]$ in fuzzy logic. The existence of truth ordering posets and lattices is not due to mere chance. In fact, the poset A_9 in Figure 3 partly forms an instance of a family of multi-valued logics known as Kleene-like logics [6]. We have appealed only to the intuitive nature of such logics and sketched the informal construction of A_{13} in Section 2. The complete lattice A_{13} proposed in this paper is, to the best of our knowledge, the first attempt to characterize evolutionary orderings capable of modeling uncertainty and disagreement in the three-way software merging context. Multi-valued exploratory orderings can be fine-tuned to bear more analytical power. For example, the value $\mathbb{P}$ (meaning "preserved") may be refined to "unchanged", "renamed", "splitted", and the like.

The idea of using colimits as abstract mechanisms for putting structures together in (finitely) cocomplete categories is not new: Of the prominent industrial experience is the work of Williamson *et al.*, who have investigated the software synthesis problem using Specware™ [22]. In their study, colimits are employed to modularly combine specifications so that forward engineering is supported. We notice that Williamson's approach does not explicitly consider graph-based software merging that facilitates reverse engineering, which is the focus of our work here. We also note that the "general systems engineering" concept in Williamson's work bolsters our perception that our approach should be regarded as a semi-automated comprehension task. Therefore, practitioners are encouraged to incorporate knowledge from other disciplines to gain a more complete and accurate understanding of software merging.

6.2 Summary

The ability to merge parallel changes is needed during the development and maintenance of large-scale software systems. However, traditional textual merge techniques are

insufficient for detecting syntactic merge conflicts. In this paper, we have designed a category-theoretic approach for three-way syntactic merging that facilitates software reverse engineering and reengineering. Our mathematically rigorous approach is general, since graphs are used as an underlying representation for evolving software. Syntactic inconsistencies and evolutionary properties are captured through lattices, and an exploratory ordering A_{13} is defined to guide our study. Our use of supremum and colimit exposes a natural and correct-by-construction way to join parallel modifications.

From our initial experiences with the approach, we feel that it has rich value in helping maintainers understand, justify, and manage software merging, and software evolution in general. Empirical studies are needed to lend strength to the exploratory findings reported here. Systematic experiments in merging open source software may render results supplementary to our analysis, since reconciliation [16], instead of consolidation, is typically involved in the merge process. Our future work also includes adding support for hierarchical structures so that software merging can be studied at different architectural granularities and structural merge conflicts can be detected. Also of interest would be addressing typing constraints in such a way that graph-based software representations can contain heterogeneous nodes and arrows in one view; meanwhile, proper merge techniques can preserve both graphical integrity and the typing information. Finally, the constructive nature of category theory leads our approach to automation. We have implemented a proof-of-concept Java tool for merging fuzzy graphs. By performing more trials with this automated support, we can critically evaluate our approach, investigate the efficiency and scalability of graph-based merge algorithms, and explore a deeper understanding of the rudimentary principles of software merging.

Acknowledgments. *We thank the partner company for the generous support throughout our study, Hongying Zhao for help with extracting and analyzing iBBS systems, and Kenny Wong and Li-Shih Huang for providing valuable comments on drafts of this paper. Financial support was provided by NSERC and MITACS.*

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science*. Les Publications CRM Montréal, third edition, 1999.

[2] E. Burd and M. Munro. Investigating component based maintenance and the effect of software evolution: a reengineering approach using data clustering. In *Intl. Conference on Software Maintenance*, pages 199–207, 1998.

[3] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[4] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[5] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE Trans. on Software Engineering*, 28(4):396–412, 2002.

[6] M. Fitting. Kleene's logic, generalized. *Journal of Logic and Computation*, 1:797–810, 1992.

[7] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. on Software Engineering*, 31(2):166–181, 2005.

[8] J. Goguen. *Categories of Fuzzy Sets: Applications of Non-Cantorian Set Theory*. PhD thesis, University of California, Berkeley, 1968.

[9] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of ACM*, 39(1):95–146, 1992.

[10] T. Hendrix and J. Cross. Language independent generation of graphical representations of source code. In *Annual Conference on Computer Science*, pages 66–72, 1995.

[11] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale souce code. *IEEE Trans. on Software Engineering*, 8(7):654–670, 2002.

[12] M. Lehman and L. Belady. *Program Evolution: Process of Software Change*. Academic Press, 1985.

[13] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije University, 1999.

[14] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. on Software Engineering*, 28(5):449–462, 2002.

[15] H. Müller and K. Klashinsky. Rigi — a system for programming-in-the-large. In *Intl. Conference on Software Engineering*, pages 80–86, 1988.

[16] J. Munson and P. Dewan. A flexible object merging framework. In *ACM Conference on Computer Supported Cooperative Work*, pages 231–242, 1994.

[17] D. Perry, H. Siy, and L. Votta. Parallel changes in large scale software development: an observational case study. In *Intl. Conference on Software Engineering*, pages 251–260, 1998.

[18] J. Rho and C. Wu. An efficient version model of software diagrams. In *Asia-Pacific Conference on Software Engineering*, pages 236–243, 1998.

[19] M. Sabetzadeh. A category-theoretic approach to representation and analysis of inconsistency in graph-based viewpoints. Master's thesis, University of Toronto, 2003.

[20] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints: a category-theoretic approach. In *Intl. Conference on Automated Software Engineering*, pages 12–21, 2003.

[21] Q. Tu and M. Godfrey. An integrated approach for studying software architectural evolution. In *Intl. Workshop on Program Comprehension*, pages 127–136, 2002.

[22] K. Williamson, M. Healy, and R. Barker. Industrial applications of software synthesis via category theory – case studies using specware. *Automated Software Engineering*, 8(1):7–30, 2001.