A Cost-Benefit Approach to Recommending Conflict Resolution for Parallel Software Development

Nan Niu*, Fangbo Yang*, Jing-Ru C. Cheng†, and Sandeep Reddivari*
* Department of Computer Science and Engineering, Mississippi State University, USA
† Information Technology Laboratory, U.S. Army Engineer Research and Development Center, USA
niu@cse.msstate.edu, fy47@msstate.edu, Ruth.C.Cheng@usace.army.mil, srr159@msstate.edu

Abstract—Merging parallel versions of source code is a common and essential activity during the lifespan of large-scale software systems. When a non-trivial number of conflicts is detected, there is a need to support the maintainer in investigating and resolving these conflicts. In this paper, we contribute a cost-benefit approach to ranking the conflicting software entities by leveraging both structural and semantic information of the source code. We present a study by applying our approach to a legacy system developed by computational scientists. The study not only demonstrates the feasibility of our approach, but also sheds light on the future development of conflict resolution recommenders.

Keywords—software merging; conflict resolution; recommendation; cost-benefit analysis;

I. INTRODUCTION

Parallel changes, in which separate lines of development are carried out by different developers, are a basic fact of developing and maintaining large-scale software systems [1]. An optimistic version control mechanism [2] allows every developer to work on a local copy of the software artifact independently. Thus, a fundamental challenge 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.

The literature on software merging is extensive, e.g., Mens [3] provided an excellent summary of all but the most recent work in the field, showing the diverse range of techniques employed. The software merging process considered in our work is shown in Fig. 1 in which boxes represent entities and diamonds represent activities. Currently, there exist many approaches that facilitate the identification and classification of inconsistencies, such as discovering the differences at syntactic [4] or semantic [5] levels. The dimensions of recommendation tools for detecting software conflicts are discussed in [6].

In contrast to the considerable support for conflict detection, there has been modest support for conflict resolution. For example, earlier work on Infuse [7] proactively modularized the code base into workspaces so that developers working in separate workspaces would encounter few conflicts caused by parallel development. More recently, Palantir [8] informed code changes across workspaces by calculating a simple and coarse-grained measure of severity of those changes. While these approaches help separate the concerns and raise developer’s awareness, little work has been done on recommending a fine-grained order for the maintainer to investigate and resolve multiple conflicts. Lack of this support is a serious problem because addressing software conflicts in a random order can be inefficient and costly [6].

In this paper, we shorten the gap by presenting a cost-benefit approach to ranking the conflicting software entities in an ordered list. The bottom path of Fig. 1 highlights this recommendation in the software merging process. Our goal is to provide relevant and valuable information to improve the efficiency when a maintainer resolves multiple conflicts. To that end, we leverage Semantic Diff [5] to quantify the cost of conflict resolution for each procedure. We then characterize the resolution benefit of a procedure according to the change impact caused by the global variables. The final recommendation is made by sorting the conflicting procedures in a decreasing order.

The contributions of our work lie in the development of a conflict investigation mechanism by synthesizing structural and semantic information of the source code. Our approach is particularly applicable for situations where: 1) a single maintainer is directly responsible for resolving all the conflicts resulted from parallel changes; and 2) the primary source of conflict resolution is the code base (e.g., change intent is not well documented, original developers become unavailable, etc.). We collaborate with computational scientists to address their software merging needs, and apply our approach to a legacy project that exhibits the two characteristics mentioned above. In what follows, we present our cost-benefit recommender in Section II. In Section III, we describe the application of our approach in Section III. In Section I, we discuss related work in Section IV and conclude the paper in Section V.

II. A COST-BENEFIT RECOMMENDER

A. Cost Estimation

To estimate the effort required to fix conflicts, we adopt Semantic Diff [5], a tool that takes two versions of a procedure and reports the semantic differences between them. Because Semantic Diff outputs fewer false positives when compared with commercial, text-based diff tools [3],...
adopter it in our approach makes the recommendation more reliable. Key to Semantic Diff is the concept of dependence pair defined to capture a procedure’s semantic effect. Specifically, a pair of variables, \((x, y)\), forms a dependence pair if \(x\)‘s value after execution of the procedure depends on \(y\)‘s value before the procedure is executed [5].

Consider the procedure \(\text{node}\_\text{get}\_\text{local}\) of Version \(n.1\) in Fig. 2 as an example. The assignment statement gives rise to the dependence pair, \((\text{node}\_\text{pair}[\text{ind}].\text{sd}, \text{gn}.\text{sd})\). We extract a procedure’s dependence pairs based on the formulas defined in [5], and use \(DP(p)\) to denote the set of all dependence pairs generated from the procedure \(p\). A procedure is considered as a conflicting procedure if its parallel versions result in different sets of dependence pairs. In another word, a conflicting procedure exhibits different semantic effects during parallel changes.

We estimate the cost of fixing a conflicting procedure according to the number of inconsistent dependence pairs identified between the procedure’s parallel versions. Take the procedure in Fig. 2 as an example:

\[
\text{cost}(\text{node}\_\text{get}\_\text{local}) = \frac{|DP(\text{node}\_\text{get}\_\text{local}, n.1) \cup DP(\text{node}\_\text{get}\_\text{local}, n.2)|}{|DP(\text{node}\_\text{get}\_\text{local}, n.1) \cap DP(\text{node}\_\text{get}\_\text{local}, n.2)|} - 1.
\]

This operation is based on our conjecture that the maintainer needs to inspect and evaluate all the identified semantic differences. Because Semantic Diff is accurate in detecting software conflicts [5], our cost approximation represents a conservative estimate at the intra-procedural level.

B. Benefit Estimation

In contrast to the intra-procedural cost estimation, the benefit of conflict resolution in our approach is considered at the inter-procedural level. When changing a procedure from inconsistent to consistent, the effect of the changes may not be local (i.e., within the procedure only), but can propagate to the rest of the system [9]. The control of such a ripple effect [10] requires both the recognition of system interconnections and the coordination of modifying interdependent entities.

In our work, we focus on the interconnections caused by global variable references; however, other types of dependency (e.g., API call usages [11]) can also be incorporated. Fig. 3 outlines the algorithm for calculating the benefit of resolving conflicting procedures. The key idea is to leverage the intra-procedural dependence pairs to track the inter-procedural semantic dependencies connected by some global variable.

The algorithm listed in Fig. 3 considers every conflicting procedure \(p\) and its dependence-pair set \(DP(p)\). For a dependence pair \((x, y)\) that causes \(p\) to be inconsistent, i.e., \((x, y)\) contributes to the calculation of \(\text{cost}(p)\), the algorithm checks whether \(x\) is a global variable. If it is, then a local modification within \(p\) can propagate the change effect throughout the system via \(x\). In particular, procedure \(q\) will be influenced by the ripple effect if \((z, x) \in DP(q)\). In this sense, \(q\) benefits from the resolution of \(p\) in that changing \((x, y)\) from inconsistent to consistent saves the maintainer from investigating \((z, x)\) in \(q\). The benefit of resolving \(p\) increments each time \(q\) is identified. Note that \(q\) is looped over the set of all the procedures since conflict resolution can affect those procedures originally found to be consistent.

C. Recommendation List

The recommendation list is made by computing the \(\frac{\text{benefit}}{\text{cost}}\) ratio and ranking the conflicting elements in a descending order of the ratio. Fig. 4 shows our recommender’s prototype implementation. The maintainer specifies the merging scope by opening the root folder of each parallel version in the tool. The scope can range from the entire project to specific
modules or libraries. The prototype tool currently compares two parallel versions and relies on Semantic Diff [5] to detect semantic inconsistencies within the chosen scope.

As shown in Fig. 4a, the conflicting procedures are initially displayed alphabetically. The recommender takes the source code and the parallel change information as input, computes the cost and benefit of conflict resolution as described earlier in this section, and outputs a ranked list as a reference point for the maintainer to resolve conflicts. In case of a tie (e.g., node_packi and node_unpacki in Fig. 4b), the procedures are listed in a random order. In addition to the default cost-benefit ratio ranking, the maintainer can (re-)order the procedures by other attributes (columns) like name, cost, and benefit (cf. Fig. 4b).

To support further investigation of a particular procedure, the tool shows the detailed explanation in a separate window (cf. Fig. 4b). In this view, the dependence pairs inconsistent between the parallel versions are listed. These pairs contribute directly to the cost calculation in our approach. Meanwhile, the benefit of resolving the chosen procedure is explained by providing the change propagation information along with the global variable(s) behind each interconnection. The explanation viewer integrates both semantic (i.e., cost estimation) and structural (i.e., benefit estimation) information to offer insights into the recommendations.

III. A PRELIMINARY EVALUATION

We conducted an initial evaluation of our approach by collaborating with the computational scientists and software engineers at the U.S. Army Engineer Research & Development Center (ERDC). The objective is to assess the feasibility of our approach, and more importantly, to identify areas for improvement. In addition, we would like to explore how software conflicts are handled in practice.

The subject system of our study is a large-scale scientific application written in C. In order to honor confidentiality agreements, we use the pseudonym “HDT” to refer to the system. Table I lists some basic information of the two parallel versions in our study. These HDT variants are very similar in terms of the statistics reported in Table I. It is interesting to note that the DP:SLOC ratio per procedure in HDT is roughly 1:1, whereas a 4:1 ratio was reported (about 400 dependence pairs for a 100-SLOC procedure) when an industrial software system written in C from the real-time domain was analyzed [5]. The relatively low ratio suggests that fewer semantic dependencies exist in HDT than the system studied in [5]. This could be due to the modular design and regular refactoring of HDT, allowing process libraries (e.g., sediment transport, multiple elemental cycles, etc.) to be accessed as required for a specific application.

HDT is a legacy software system that has been evolved for several decades. Parallel changes are common and core HDT development activities. Currently, a single maintainer takes primary responsibility for merging HDT’s multiple versions and variants. Although automated tools such as DiffMerge (http://www.sourcegear.com/diffmerge) are employed, the challenge of conflict resolution remains. Based on an informal interview with the HDT maintainer, when 5 to 10 inconsistent procedures are identified, deciding which one to resolve first can be challenging. In some situations, the maintainer resolves conflicts based on the order in which they show up in DiffMerge, e.g., by following an alphabetic order. Part of the problem is that there exists insufficient documentation of change intent. Therefore, the code base typically serves
as the main source for conflict investigation and resolution. We believe our approach can improve the HDT project’s merging process (cf. Fig. 1) by recommending a systematic order as well as the detailed explanations about each of the recommendations (cf. Fig. 4).

We applied our approach to one of HDT’s core modules specified by the ERDC experts. In this module (M), 6 conflicting procedures (p_a to p_f) were identified – a subset of the 144 procedures reported in the bottom row of Table I. The domain experts also shared with us a set of test cases used to verify the correctness of M. We ran these test cases and recorded the execution traces that logged the sequence of procedures being activated at runtime. We then adopted the dynamic analysis technique described in [12] to determine the impact set of each conflicting procedures. For instance, if p_g was executed after p_a, then p_g would belong to the impact set of p_a. Compared to our use of static information, dynamic analysis could better predict a procedure’s change impact to the whole system [12]. Therefore, we used the size of the impact set as a surrogate measure of “benefit” considered in our approach: the larger the size, the more beneficial the effect of conflict resolution.

Table II compares two conflict resolution orders: alphabetic and our recommended order. While the alphabetic order leads to a random distribution of impact set size, our recommended order exhibits a trend of decreasing impact set size. This implies that our approach can contribute to a more systematic and efficient conflict resolution strategy. The “outlier” procedure in Table II p_f whose impact set size is 10, warrants further investigation. In addition to carrying out more evaluations, we have also identified areas for improvement by collecting the feedback from the ERDC maintainer. One primary issue is to explicitly annotate the version information in our recommender. Also of interest would be integrating the recommendations more seamlessly into software development environments (SDEs).

### Table III

<table>
<thead>
<tr>
<th>Approach</th>
<th>Description</th>
<th>Nature of Context</th>
<th>Recommendation Engine</th>
<th>Output Mode</th>
<th>Explanation</th>
<th>User Feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infuse [7]</td>
<td>Partition workspaces to reduce interferences</td>
<td>Hybrid</td>
<td>Source Code &amp; Change</td>
<td>No</td>
<td>Push</td>
<td>None</td>
</tr>
<tr>
<td>Flexible [14]</td>
<td>Offer possible resolution paths for users to choose</td>
<td>Hybrid</td>
<td>User Interface</td>
<td>No</td>
<td>Push</td>
<td>Inline</td>
</tr>
<tr>
<td>Treemap [16]</td>
<td>Visualize artifact states from other workspaces</td>
<td>Implicit</td>
<td>Source Code &amp; Change</td>
<td>No</td>
<td>Push</td>
<td>Inline</td>
</tr>
<tr>
<td>Palantir [8]</td>
<td>Raise awareness across workspaces along with severity of conflicts</td>
<td>Hybrid</td>
<td>Source Code &amp; Change</td>
<td>No</td>
<td>Push</td>
<td>Inline</td>
</tr>
<tr>
<td>Rational Mgmt [17]</td>
<td>Detect and resolve conflicts in models</td>
<td>Implicit</td>
<td>Model</td>
<td>No</td>
<td>Pull</td>
<td>Batch</td>
</tr>
<tr>
<td>Cost-benefit Recommender (our approach)</td>
<td>Rank conflicting entities based on resolutin’s cost and benefit</td>
<td>Implicit</td>
<td>Source Code &amp; Change</td>
<td>Yes</td>
<td>Pull</td>
<td>Batch</td>
</tr>
</tbody>
</table>

In an attempt to classify the design dimensions of recommendation systems for software engineering (RSSE), Robillard et al. discussed three major components (nature of the context, recommendation engine, and output mode), along with two crosscutting features (explanation and user feedback) [13]. Table III uses these dimensions to compare our recommender with other conflict resolution approaches.

Our approach establishes the merging context implicitly in that the tool parses and analyzes the parallel programs without maintainer’s explicitly specifying contextual information such as change intent. Other approaches like Palantir [8] require a hybrid of implicit and explicit context gathering. We feel that automatically extracting the context for RSSE, though challenging, is promising as researchers and tool builders can take advantage of the ever-growing quantities and types of software development data [18].

RSSE must analyze more than context data to make recommendations [13]. In our approach, the recommendation engine takes the source code and the concurrent changes as input. Although the code base serves as primary source for conflict resolution for most approaches shown in Table III taking into account of high-level artifacts (e.g., models [17]) can potentially increase the power of the recommendation engine. However, it is important to synchronize heterogeneous artifacts collected throughout the software lifecycle.

Despite the key role that ranking plays in RSSE [13], it is surprising to note that, among the existing approaches surveyed in Table III only TUKAN [15] supports a fine-grained ranking mechanism. Our approach quantifies the cost and benefit of conflict resolution and puts the entities most valuable to the maintainer at the top of the ranked list. This feature enables a systematic way to investigate and resolve software conflicts.

Most software merging recommenders (e.g., [7, 8, 14, 15, 16]) operate in push mode in which the tools deliver results
continuously. These tools notify relevant users when a conflict emerges. Pull mode is different from push mode in that users explicitly request for recommendation generation [13]. For example, the model management tool presented in [17] requires user interaction to choose from the conflicting operations. Similarly, users receive recommendations from our tool simply by clicking a button. Some tools integrate into other environments, following an inline presentation style, e.g., Palantir [8] is built upon version control systems and TUKAN [15] is integrated in the VisualWorks/EVNY Smalltalk environment. In contrast, our approach outputs the set of recommendations in separate views (cf. Fig. 4), working in a batch mode. As the prototype tool is only a proof-of-concept, we plan to improve the implementation by shifting our recommender’s output toward a more non-obstructive pull and inline mode.

In order to justify the recommendations made, we give detailed explanations about each recommended procedure. We not only provide rankings like TUKAN [15] does, but also offer rationales behind them. Some recommendation engines take user feedback into account, so that the interaction between user and the system can affect the recommendation results. For those collaborative software development tools like Palantir [3], the recommenders rely on users’ inputs across multiple workspaces. Thus, they incorporate globally adaptive user feedback. We plan to incrementally develop feedback mechanism, maybe by starting at the locally adjustable level [13], in order to extend our tool along the collaborative conflict resolution dimension [6].

V. Conclusions

The ability to merge parallel changes is needed during the development and maintenance of large-scale software systems. In this paper, we have presented a cost-benefit approach to ranking the conflicting software entities by exploiting both structural and semantic information from the code base. We developed a prototype tool and carried out a preliminary evaluation by applying our recommender to an industrial-strength software project.

From our initial experiences with the approach, we feel that it has rich value in helping maintainers to understand, justify, and manage conflict resolution, and software merging in general. In the future, more in-depth empirical studies are needed to lend strength to the preliminary findings reported here. We also plan to enhance our tool by annotating the version information and by inlining the recommendations into the SDEs.

ACKNOWLEDGEMENT

The work is supported by the U.S. Army Engineer Research & Development Center (Award W912HZ-10-C-0101). Any opinions, findings, and conclusions expressed herein are the authors’ and do not necessarily reflect those of the sponsor.

REFERENCES


