Published in IET Software
Received on 26th May 2012
Revised on 1st September 2012
Accepted on 25th September 2012
doi: 10.1049/iet-sen.2012.0089

ISSN 1751-8806

Conflict resolution support for parallel software development

Nan Niu[1], Fangbo Yang[1], Jing-Ru C. Cheng[2], Sandeep Reddivari[1]
[1]Department of Computer Science and Engineering, Mississippi State University, P.O. Box 9637, Mississippi State, MS 39762, USA
[2]Information Technology Laboratory, US Army Engineer Research and Development Center, Vicksburg, MS 39180, USA
E-mail: niu@cse.msstate.edu

Abstract: Parallel changes, in which separate lines of development are carried out by different developments, are a basic fact of developing and maintaining large-scale software systems. Merging parallel versions and variants of source code is a common and essential software engineering activity. 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 study, the authors present software conflict resolution recommender (scoreRec), a cost–benefit approach to ranking the conflicting software entities. The contributions of scoreRec lie in the leverage of both structural and semantic information of the source code to generate conflict resolution recommendations, as well as the hierarchical presentation of the recommendations with detailed explanations. The authors evaluate scoreRec through an industrial-strength legacy system developed by computational scientists. The results show that scoreRec offers relevant and insightful information and sound engineering support for conflict resolution. The authors' work also sheds light on the future development of recommendation systems in the context of software merging.

1 Introduction

As the size and complexity of software increase, finding the relevant information to carry out software engineering activities has become challenging. Without assistance, the developer can easily spend a disproportionate amount of time seeking information at the expense of such value-producing tasks as bug fixing and feature enhancement [1]. Recommendation Systems for Software Engineering (RSSEs) address the challenge by proactively tailoring suggestions that meet developers' particular information needs and preferences [2].

An RSSE is aimed at providing information items estimated to be valuable for a software engineering task in a given context, and is particularly useful in supporting decision making when the developer cannot consider all the data at hand [2]. A specific situation that RSSE can help is parallel software development [3], in which software changes are carried out by different programmers and the separate lines of development must be merged at regular intervals. The need for merging parallel versions and variants of a software system to yield a consistent shared view arises in the contexts of concurrent software development, global software engineering and computer-supported collaborative work.

The literature on software merging is extensive. For example, Mens [4] 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, which illustrates parallel software development that adopts the optimistic version control mechanism allowing every developer to work on a local copy of the code independently [5]. Currently, there exist many approaches that facilitate the identification and classification of inconsistencies, such as uncovering the differences at syntactic [6] or semantic [7] levels. The RSSE design dimensions for detecting software conflicts are discussed in [8].

In contrast to the considerable support for conflict detection, there has been only modest support for 'conflict resolution'. For example, earlier work on Infuse [9] proactively modularised the code base into workspaces so that developers working in separate workspaces would encounter few conflicts caused by parallel development. TUKAN [10], as another example, used a spatial representation to model the parallel changes with no direct support for relating the graphical representation to the code base where the actual modification and resolution would take place. More recently, Palantír [11] informed code changes across workspaces by calculating a simple and coarse-grained measure of severity of those changes. Note that only a few prominent sample approaches are mentioned here; a more comprehensive review of related work is given in Section 5. Whereas 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 [4].

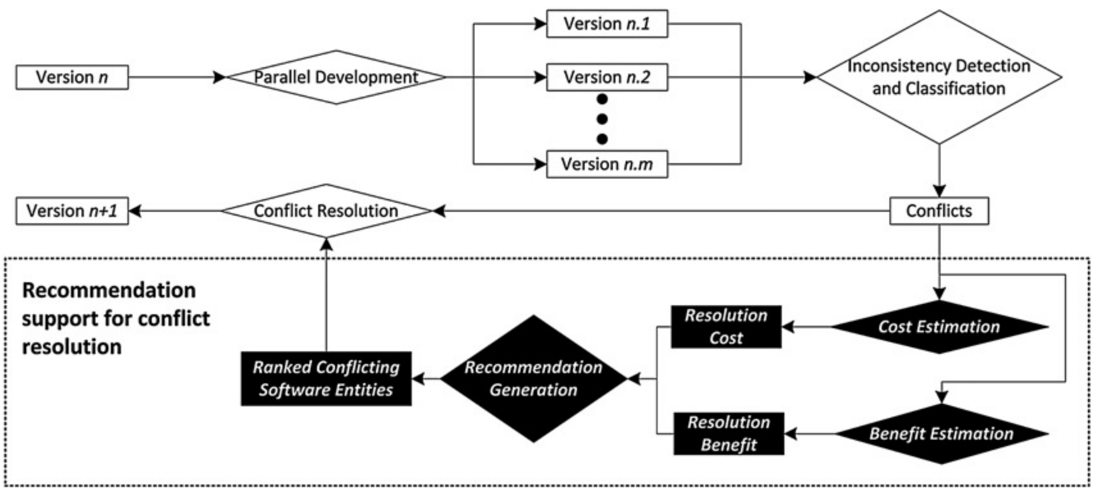


Fig. 1 *Software merging process depicted in a flow diagram, in which boxes represent entities and diamonds represent activities, the dotted box highlights our recommendation support for software conflict resolution*

In this paper, we shorten the gap by introducing software conflict resolution recommender (scoreRec), a cost–benefit approach to ranking the conflicting software entities in an ordered list and presenting the recommendations in a hierarchical fashion. Our goal is to provide relevant and valuable information to improve the efficiency when a maintainer resolves merge conflicts resulted from parallel software development. To that end, we leverage Semantic Diff [7] to quantify the 'cost' of conflict resolution for each procedure. We then characterise the resolution 'benefit' of a procedure according to the change impact it has on the other procedures in the system. The final recommendation is made by sorting the conflicting procedures in a decreasing (benefit/cost) order, coupled with detailed explanations showing the rationale of the recommendation at different levels: system, procedure and code segment.

The contributions of our work lie in the development of a conflict investigation mechanism and the prototype scoreRec tool by synthesizing structural and semantic information of the source code. Our approach is particularly applicable for situations where: (i) a single maintainer is directly responsible for resolving all the conflicts resulted from parallel changes; and (ii) 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 scoreRec to a legacy project that exhibits the two characteristics listed above. The results show that scoreRec offers remarkable insights to and sound engineering support for conflict resolution, and that scoreRec can be integrated into the existing practice to improve the overall software merging process.

Preliminary work on conflict resolution recommendation was published in [12]. The emphasis was to show the value of RSSE in parallel development and to report a feasibility study. This paper extends prior work by enhancing the underlying benefit estimation algorithm, creating an integrated hierarchy of recommendations and conducting a thorough assessment of our approach. The remainder of the paper is organised as follows. Section 2 lays out the background and provides the context of our research. Section 3 presents our cost–benefit recommender, scoreRec. Section 4 describes the evaluation of scoreRec through a large-scale, real-world software system. Section 5 reviews related work. Section 6 draws some concluding remarks and outlines future work.

2 Background

We have related our work in several areas: inconsistency management, software conflict resolution, and RSSE. We are not aware of other work focused exclusively on offering fine-grained and ranked conflict resolution recommendations in an integrated programming environment for parallel software development.

2.1 *Inconsistency management*

The development and evolution of large-scale software systems inevitably involve the detection and handling of inconsistencies. Inconsistencies arise because stakeholders may have varied expertise areas, distinct responsibilities, different terminologies and complementary conceptual models [13]. In [14], 'inconsistency' is referred to as any situation in which two descriptions do not obey some relationship that is prescribed to hold between them. In another word, it is against a set of consistency rules that software artifacts can be checked. In parallel development, for instance, a textual merging rule may require the two versions of the same procedure to have an equal number of non-empty lines. Although this rule's checking can be fully automated, the detected inconsistencies must be analysed, for example the unequal number of lines of comments is typically classified as a false positive.

It is clear that the detection of inconsistencies hinges strongly on the rules that specify the desired consistency relationships. However, it is argued that eradicating inconsistencies as soon as they are detected is counterproductive [14]. Experience shows that practitioners often learn to live with inconsistencies because they judge that any adverse impact of these inconsistencies is tolerable [14]. Therefore in our model shown in Fig. 1, the detected inconsistencies must be classified. In [6], a category-theoretic approach to tolerating syntactic inconsistencies is proposed in the context of merging parallel versions and variants of software.

A general framework for inconsistency management is presented in [14]. Depending on the characteristics of inconsistency, it can be ignored, circumvented, ameliorated or deferred. However, if the inconsistency causes adverse impacts or consequences (e.g. misunderstandings or errors in software development), then it is classified as a conflict and should be resolved.

2.2 *Software conflict resolution*

In parallel lines of development, resolving software conflicts can range from a manual and time-consuming process, over an interactive resolution mechanism, to a fully automated tool support. Much depends on the kinds of conflicts the stakeholders intend to solve and the level of accuracy that needs to be reached [4].

In many non-trivial situations, conflicts resulted from parallel development cannot be resolved in an automated way. For example, if a procedure is renamed differently in parallel modifications, it will be difficult for the merge algorithm to automatically decide which of the two renamings, is most appropriate. To deal with such situations, a tool could provide automated assistance for negotiating the resolution of conflicts in the style of the 'Programmer's Apprentice' [15]. Alternatively, the information of the ancestor version can be taken into consideration (also known as three-way merging) for determining a desired merging result [6]. When a three-way merging is performed, conflict resolution can be more deterministic than the two-way counterpart (i.e. no information about the ancestor version is available). For example, if the two revisions are complementary, 'consolidation' can be adopted; otherwise, 'reconciliation' could be applied [16].

In summary, the resolution of software merging conflicts may be as simple as adding or deleting a piece of information from the source code. However, it often relies on resolving fundamental conflicts and taking important design decisions. In such cases, relevant and sound recommendations can offer valuable insights into the decision-making process.

2.3 *Recommendations systems for software engineering*

Recommendation systems are software applications aimed to support users in their decision-making whereas interacting with large information spaces. Following the idea of recommendation systems in overcoming the information overload problem, Robillard *et al.* [2] identify the opportunities and challenges for recommendation systems specific to software engineering, that is, RSSEs. Key factors giving rise to practical RSSEs include large stores of publicly available source code for analysing recommendations, mature software-repository mining techniques, and mainstream adoption of common software development environments (SDEs) such as Eclipse and Bugzilla.

Most current RSSEs support developer while programming. For example, Burch *et al.* [17] support API selection by evaluating usage information, Hou and Pletcher [18] suggest automatic code-completion through a popularity-based ranking and McMillan *et al.* [19] recommend code examples through the API calls and documentation. Whereas more RSSEs begin to cover a wider spectrum of software engineering activities, such as requirements engineering [20] and software testing [21], no direct recommendation support is available for resolving software merging conflicts. We next present a cost–benefit approach to fill such a gap in the RSSE literature.

3 scoreRec: a cost–benefit recommender for conflict resolution

This section introduces the fundamental concepts, the underlying mechanisms, and the prototype tool implementation of scoreRec. The basic idea of scoreRec is to leverage the semantic information of the source code to estimate the cost of resolving the conflicting software entities, and to make use of the static structural dependencies to quantify the resolution benefit. The recommendations are then presented in a hierarchical manner, allowing the software maintainer to flexibly drill down or roll up along the contexts and explanations.

3.1 *Cost estimation*

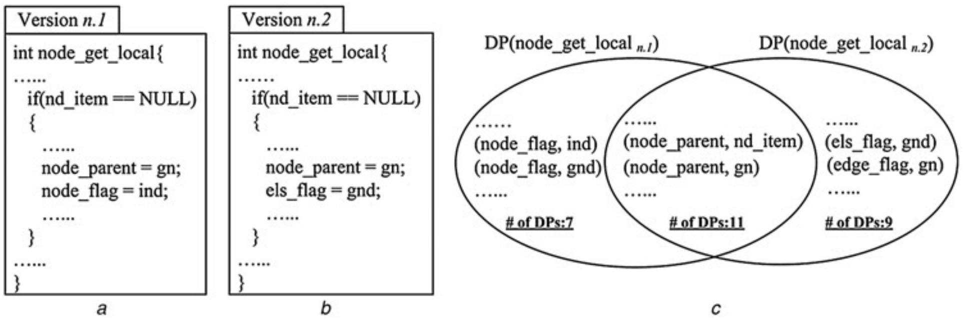
To estimate the effort required to fix conflicts, we adopt Semantic Diff [7], a tool that takes two versions of a procedure and reports the semantic differences between them. As the Semantic Diff outputs fewer false positives when compared with commercial, text-based diff tools [4], adopting it in our approach makes the recommendation more reliable. Key to Semantic Diff is the concept of 'dependence pair' (DP) defined to capture a procedure's semantic effect. Specifically, a pair of variables, (x, y), forms a DP if x's value after execution of the procedure depends on y's value before the procedure is executed [7].

We extract a procedure's DPs based on the compositional relations defined in [7], and use DP(p) to denote the set of all DPs generated from the procedure p. Fig. 2 illustrates the dependence pairs (Fig. 2*c*) generated from two parallel versions of the procedure node_get_local (Version n.1 in Fig. 2*a* and Version n.2 in Fig. 2*b*). Take node_get_local$_{n.1}$ as an example, the if-condition gives rise to the DP, (node_parent, nd_item) as the control flow indicates a semantic dependence of node_parent on nd_item. In this way, the semantic effect of the procedure node_get_local$_{n.1}$ is characterised by the set of DPs DP(node_get_local$_{n.1}$).

In Fig. 2*c*, the relationships between DP(node_get_local$_{n.1}$) and DP(node_get_local$_{n.2}$) are depicted as the set-intersection and set-differences. The set-intersection (overlap) shows the portion of the semantic effects which parallel versions agree upon, whereas the set-differences indicate merging conflicts. Thus we define a procedure to be a 'conflicting procedure' if its parallel versions result in different sets of DPs (i.e. the DP-set-differences are not empty). 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 DPs identified between the procedure's parallel versions. Take the procedure in Fig. 2 as an example (see equation at the bottom of the page)

$$\begin{aligned}\mathrm{cost}(\text{node\_get\_local}) = {} & |\mathrm{DP}(\text{node\_get\_local}_{n.1}) - \mathrm{DP}(\text{node\_get\_local}_{n.2})| \\ & + |\mathrm{DP}(\text{node\_get\_local}_{n.2}) - \mathrm{DP}(\text{node\_get\_local}_{n.1})|\end{aligned}$$



Version *n.1*

```
int node_get_local{
……
  if(nd_item == NULL)
  {
    ……
    node_parent = gn;
    node_flag = ind;
    ……
  }
……
}
```

Version *n.2*

```
int node_get_local{
……
  if(nd_item == NULL)
  {
    ……
    node_parent = gn;
    els_flag = gnd;
    ……
  }
……
}
```

Fig. 2 *Using Semantic Diff's DP to approximate the cost required to resolve a conflicting procedure*

where '-' represents the set-difference operator and $|S|$ denotes the size of set S. Fig. 2*c* shows that cost(node_get_local) = 7 + 9 = 16. This cost calculation is based on our conjecture that the maintainer needs to inspect and evaluate all the identified semantic differences. As the Semantic Diff is accurate in detecting software conflicts [7], our cost approximation represents a conservative estimate at the intraprocedural level.

3.2 Benefit estimation

In contrast to the intraprocedural cost estimation, the 'benefit' of conflict resolution in our approach is considered at the interprocedural 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 [22]. The control of such a ripple effect [23] requires both the recognition of system interconnections and the coordination of modifying interdependent entities.

In the current implementation of scoreRec, we focus on the interconnections caused by global variable references and procedure calls; however, other types of dependency (e.g. API call usages [19]) can also be incorporated. Fig. 3 outlines the algorithm for calculating the benefit of resolving conflicting procedures. The key idea here is to leverage the intraprocedural DPs to track the interprocedural semantic dependencies connected by some global variables or call relationships.

The algorithm listed in Fig. 3 considers every conflicting procedure p and its DP set, DP(p). For a DP (x, y) that causes p to be inconsistent, that is (x, y) contributes to the calculation of 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 through 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 algorithm also takes the procedure calls into account: If q calls p, the behaviour of p can affect that of q. A local modification that makes p consistent will also benefit q. Therefore in Fig. 3, 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.

```
Input: a set of all procedures $P$, a set of all global variables $G$, and a set of
       conflicting procedures $CP \subseteq P$
Output: benefit($p$) for all $p \in CP$

Main Procedure
  Initialise int[] benefit        //indexed by $p \in CP$
  For each $p \in CP$
    If [$(x, y) \in DP(p)$ AND $(x,y)$ contributes to cost($p$) AND $x \in G$]
      For each $q \in P$
        If [$(z,x) \in DP(q)$] OR [$q$ calls $p$]
          benefit($p$)++
          BREAK
  Return benefit
```

Fig. 3 *Using global variable references and procedure calls to calculate the benefit of conflict resolution*

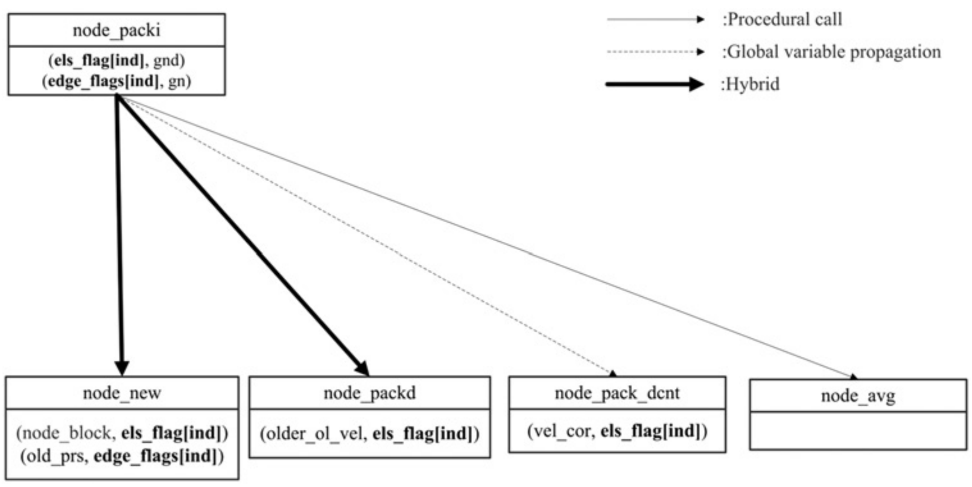


Fig. 4 *Illustration of the conflict resolution benefit*

Fig. 4 illustrates the benefit calculation for resolving the conflicting procedure, node_packi. In the figure, three types of connections are depicted: procedural call, global variable propagation and a hybrid of call relation and global variable reference. The benefit calculation is bound by the number of procedures in the system. Since the calculation relies on only static information, we compute the interprocedural connections a priori to generate recommendations in an efficient manner.

3.3 Recommendation generation

The objective of scoreRec is to provide the maintainer with relevant and insightful information, as well as a practical and flexible way to investigate and resolve software merging conflicts. To meet these goals, we develop scoreRec in C# as a plug-in to the common textual merging tools (e.g. DiffMerge [http://www.sourcegear.com/diffmerge] and SDEs (e.g. Visual Studio [http://www.microsoft.com/visualstudio]). This facilitates the integration of scoreRec into the existing software merging infrastructures and development environments. A critical design decision to enable flexibility is to display the recommendations by using multiple views that are hierarchically related. This not only eases the maintainer's understanding about the recommendations, but also equips the scoreRec tool with the capability of scaffolding the detailed explanations when desired. Three hierarchies are supported in the current scoreRec implementation: system, procedure and code segment.

Fig. 5 illustrates the system-level recommendation of scoreRec. 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 [7] to detect semantic inconsistencies within the chosen scope. The consistency rules include most common control flows such as sequential composition, if-condition, while-loop and for-loop, against which the parallel versions can be checked.

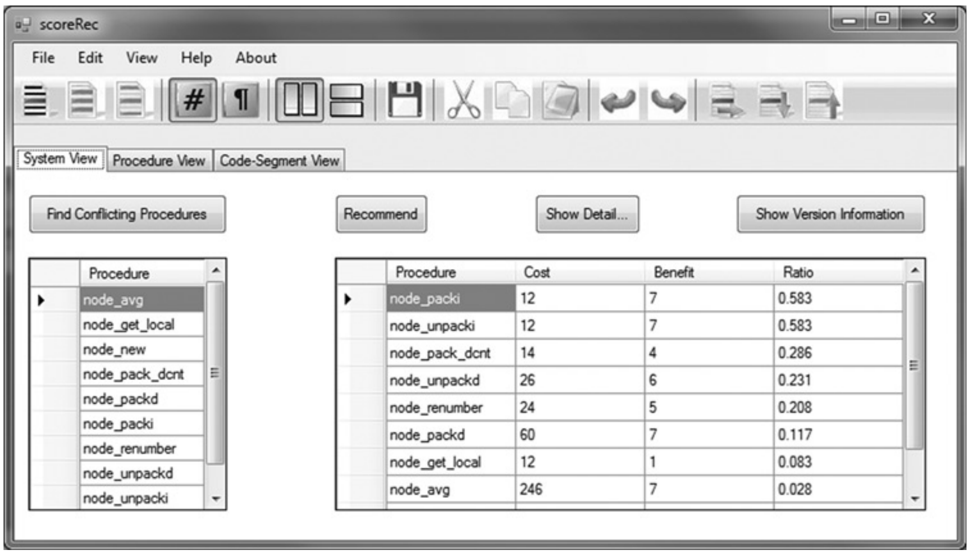
As shown in Fig. 5, the conflicting procedures are initially displayed alphabetically. The recommender takes the source code and the parallel change information as input, computes the cost and benefit of conflict resolution as described earlier in this section, and outputs a ranked list as a reference point for the maintainer to resolve conflicts. In case of a tie (e.g. node_packi and node_unpacki in Fig. 5), the procedures are listed in a random order. In addition to the default (benefit/cost) ratio ranking, the maintainer can (re-)order the procedures by other attributes (columns) like name, cost and benefit.

Fig. 6 illustrates the procedure-level recommendation of scoreRec, which provides detailed explanations of the system-level recommendation (Fig. 5). In this view, the inconsistent DPs 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) and/or the call relationships behind each interconnection. The procedure-level viewer integrates both semantic (i.e. cost estimation) and structural (i.e. benefit estimation) information to offer insights into the recommendations.

Fig. 7 illustrates the code-segment-level recommendation of scoreRec. In Fig. 7, the highlighted part indicates a difference, which is recognised in segment—a contiguous code fragment. In addition to the basic line number information, Fig. 7 also displays a dynamic status bar in the bottom of the viewer to show merging-relevant information. This includes the total number of inconsistent code segments, the position of the current inconsistency under investigation and the number of DPs resulted from that inconsistency. It is important to point out that the finer-grained code-segment viewer not only displays the version information side-by-side, but also offers direct support for code editing and conflict resolution.

4 Evaluation

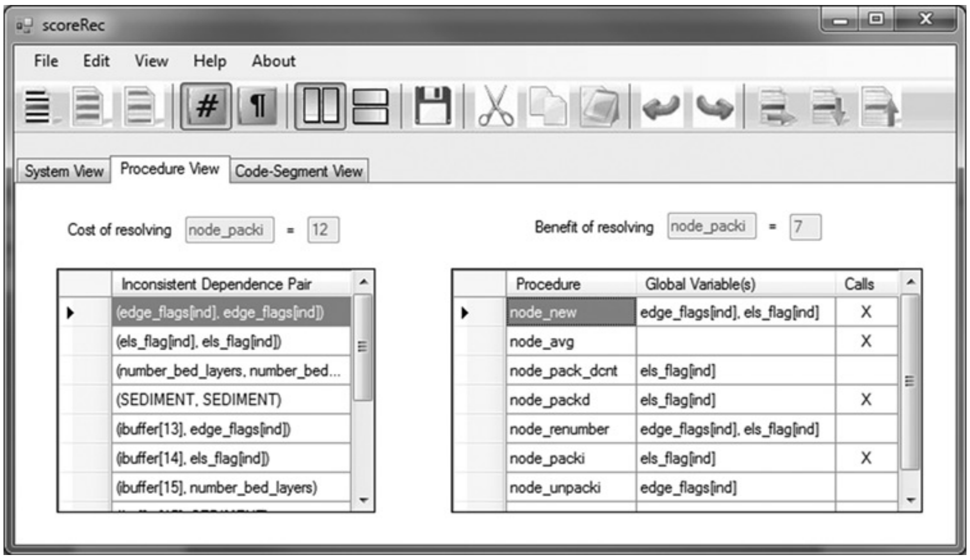
We use an 'exploratory case study' [24] as the basis for our empirical evaluation. We choose case study as the research methodology because the software merging phenomenon can be investigated within its real-life context. We employ a mix of qualitative and quantitative methods in our study: interviewing with software professionals to elicit their software merging strategies and conducting a validation



Procedure
node_avg
node_get_local
node_new
node_pack_dcnt
node_packd
node_packi
node_renumber
node_unpackd
node_unpacki

Procedure	Cost	Benefit	Ratio
node_packi	12	7	0.583
node_unpacki	12	7	0.583
node_pack_dcnt	14	4	0.286
node_unpackd	26	6	0.231
node_renumber	24	5	0.208
node_packd	60	7	0.117
node_get_local	12	1	0.083
node_avg	246	7	0.028

Fig. 5 *Screenshot of scoreRec's system-level recommendation*



Cost of resolving node_packi = 12

Benefit of resolving node_packi = 7

Inconsistent Dependence Pair
(edge_flags[ind], edge_flags[ind])
(els_flag[ind], els_flag[ind])
(number_bed_layers, number_bed...
(SEDIMENT, SEDIMENT)
(buffer[13], edge_flags[ind])
(buffer[14], els_flag[ind])
(buffer[15], number_bed_layers)

Procedure	Global Variable(s)	Calls
node_new	edge_flags[ind], els_flag[ind]	X
node_avg		X
node_pack_dcnt	els_flag[ind]	
node_packd	els_flag[ind]	X
node_renumber	edge_flags[ind], els_flag[ind]	
node_packi	els_flag[ind]	X
node_unpacki	edge_flags[ind]	

Fig. 6 *Screenshot of scoreRec's procedure-level recommendation*

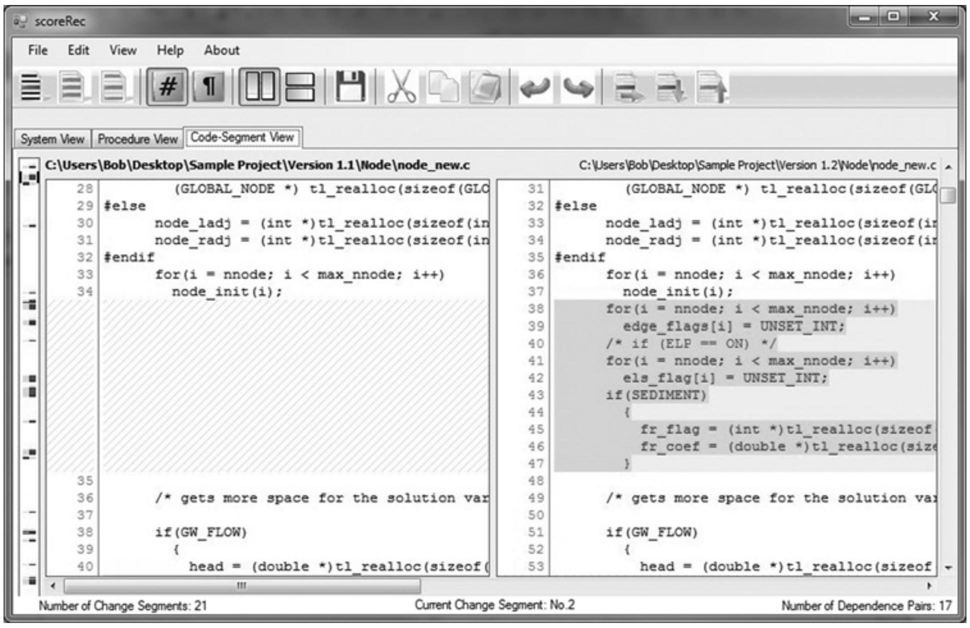
exercise to collect quantitative data on conflict resolution. Our objective is to explore how software conflicts are handled in practice, to uncover important factors affecting conflict investigation and resolution, to assess the usefulness of scoreRec and to identify potential improvements of our approach.

4.1 Background

We conduct the case study by collaborating with the computational scientists and software engineers at the US Army Engineer Research & Development Center (ERDC). The subject system of our study is a real-world, large-scale scientific application written in C. In order to honor confidentiality agreements, we use the pseudonym 'HDT' to refer to the system. Table 1 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 1. 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 DPs for a 100-SLOC procedure) when an industrial software system written in C from the real-time domain was analysed [7]. The relatively low ratio suggests that fewer semantic dependencies exist in HDT than the system studied in [7]. 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.

4.2 Eliciting factors affecting conflict resolution

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



C:\Users\Bob\Desktop\Sample Project\Version 1.1\Node\node_new.c

```
            (GLOBAL_NODE *) tl_realloc(sizeof(GLC
#else
      node_ladj = (int *)tl_realloc(sizeof(in
      node_radj = (int *)tl_realloc(sizeof(in
#endif
      for(i = nnode; i < max_nnode; i++)
        node_init(i);


      /* gets more space for the solution var

      if(GW_FLOW)
        {
          head = (double *)tl_realloc(sizeof(
```

C:\Users\Bob\Desktop\Sample Project\Version 1.2\Node\node_new.c

```
            (GLOBAL_NODE *) tl_realloc(sizeof(GL(
#else
      node_ladj = (int *)tl_realloc(sizeof(ir
      node_radj = (int *)tl_realloc(sizeof(ir
#endif
      for(i = nnode; i < max_nnode; i++)
        node_init(i);
      for(i = nnode; i < max_nnode; i++)
        edge_flags[i] = UNSET_INT;
      /* if (ELP == ON) */
      for(i = nnode; i < max_nnode; i++)
        els_flag[i] = UNSET_INT;
      if(SEDIMENT)
        {
          fr_flag = (int *)tl_realloc(sizeof
          fr_coef = (double *)tl_realloc(size
        }

      /* gets more space for the solution va

      if(GW_FLOW)
        {
          head = (double *)tl_realloc(sizeof
```

Number of Change Segments: 21   Current Change Segment: No.2   Number of Dependence Pairs: 17

Fig. 7 *Screenshot of scoreRec's code-segment-level recommendation*

Table 1 Characteristics of the parallel versions

	$\mathrm{HDT}_{n.1}$	$\mathrm{HDT}_{n.2}$
number of procedures	928	922
mean LOC[a] per procedure	78.8	82.0
mean SLOC[b] per procedure	68.6	71.6
mean |DP| per procedure	69.0	72.4
number of conflicting procedures	144	

[a]LOC: lines of code.
[b]SLOC: source LOC (neither comment nor empty line is considered).

versions and variants. Although automated tools such as DiffMerge are employed, the challenge of conflict resolution remains. We conducted an open-ended interview with the HDT maintainer. The primary concern focused on the challenges faced in practical conflict resolution. The main finding was that when 5–10 inconsistent procedures were identified, deciding which one to resolve first, and which order to follow in general, could 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 (The alphabetic order is adopted as a baseline for comparing scoreRec's effectiveness in Section 4.3.). 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 scoreRec can improve the HDT project's merging process (cf. Fig. 1) by recommending a systematic order as well as the detailed explanations (Figs. 5–7).

In order to determine the major factors affecting software conflict resolution, we carried out a validation exercise on one of HDT's core modules specified by the ERDC experts. A software engineer who is familiar with the HDT project was asked to resolve this particular module *M*. During the study, the engineer relied primarily on the code base for conflict resolution and was provided with the DiffMerge's output as the reference to the inconsistency detection result. The engineer was free to choose any strategy based on the working experience. The engineer's operations were logged in an unobtrusive manner and further validated by a researcher who was present during the study. At the end of the study, the researcher interviewed the engineer to elicit key decisions made and difficulties encountered during the entire conflict resolution session. For this exercise, scoreRec was not tested as our objective was to quantify the effort without the support offered by scoreRec, and more importantly, to gain an understanding about the underlying factors that impact a software engineer's conflict resolution.

Table 2 shows the conflict resolution results of the validation exercise. The module *M* under study contains nine conflicting procedures, as shown in the leftmost column of Table 2. This column, from top to bottom, also displays the sequence of the procedures resolved by the software engineer, namely the conflict resolution order. According to the engineer, the resolution strategy adopted in the resolution session was based on the perceived complexity of the procedure: the less complex the procedure, the earlier the engineers would like to resolve it. The rightmost column of Table 2 shows the actual time used by the software engineer to resolve each of the conflicting procedures.

It turned out that multiple procedures were investigated during the resolution of any given conflicting procedure. Such 'investigation set' information is provided in the second column of Table 2. For example, in order to resolve P_{17}'s conflicts, the software engineer had to examine three other procedures $\{P_9, P_{21}, P_{29}\}$. Not only was the size of the investigation set important, but the order in which the procedures were examined was a crucial factor influencing the conflict resolution. In Table 2, the procedures in bold represent those reexamined by the software engineer. For instance, during P_4's resolution, P_{21} and P_9 were checked even though both procedures were examined before by the engineer. Such reexaminations, though time-consuming, are indispensable when inherent dependency and high coupling exist among the procedures [4]. Table 2 also provides the size information about the source code. It can be summarised from our validation exercise that the following three factors have major impacts on the conflict resolution:

- *Size and complexity of the source code*: The less complex the code is perceived (e.g. indicated by the size of the code), the less effort is required to address its conflicts.
- *Size of the investigation set*: The larger the investigation set, the more time is spent on conflict resolution.
- *Resolution order*: This is the most important factor based on both the observation during the actual resolution session and the interview with the software engineer after the validation exercise. The finding is in line with the literature in that merging conflicts can only be resolved if they are addressed in a certain order because the inverse order may lead to further inconsistency or expensive extra rework [4]. For this reason, we believe that scoreRec can be of great practical value towards recommending an 'optimal' resolution order.

Table 2 Conflict resolution results of the validation exercise

Procedure	Investigation set	LOC		No. of inconsistent segments	LOC of segments	Time, min
		Version 1	Version 2			
P_{17}	P_9, P_{21}, P_{29}	41	43	1	2	23
P_{29}	P_{21}, P_9, P_{19}, P_1, $\boldsymbol{P}_{17}$	37	46	2	9	35
P_{19}	$\boldsymbol{P}_{29}$, P_{21}, P_9, P_{10}, P_{22}, P_8	17	68	4	8	48
P_{20}	P_{30}, P_{23}, $\boldsymbol{P}_{29}$, P_{21}, P_9, P_4, P_{24}, P_8	37	68	2	11	52
P_{30}	P_{18}, $\boldsymbol{P}_{20}$, P_9, P_1, $\boldsymbol{P}_{19}$, P_{21}, P_{24}, P_8	319	347	6	29	97
P_{21}	P_5, P_6, P_8, P_{13}, $\boldsymbol{P}_{19}$, P_{31}	320	348	6	37	93
P_9	$\boldsymbol{P}_{29}$, P_{21}, $\boldsymbol{P}_{30}$, P_4, P_1	496	515	22	66	64
P_4	P_1, $\boldsymbol{P}_{21}$, P_9, P_{20}	797	940	9	124	55
P_1	$\boldsymbol{P}_9$, P_{16}	894	993	21	116	45

4.3 Assessing scoreRec's effectiveness

In order to test the effectiveness of scoreRec's recommended order, we obtained a complete set of test cases that the ERDC domain experts developed to verify the correctness of the module M. Our purpose was to use dynamic analysis based on this real-world testing suite to determine the 'investigation set' of each procedure. This information could then be fed to our analysis of the usefulness of scoreRec. Specifically, we ran these test cases and recorded the execution traces that logged the sequence of procedures being activated at runtime. We adopted the dynamic analysis technique described in [25] to determine the investigation set (or impact set) of each conflicting procedure. For instance, if P_4 was executed after P_1, then P_4 would belong to the investigation set of P_1. Compared with static analysis, dynamic analysis could predict a procedure's change impact to the whole system more accurately and concisely [25]. Therefore the investigation set determined by the dynamic analysis represents a precise surrogate measure of 'benefit' considered in our approach.

Table 3 compares three conflict resolution orders: alphabetic order (baseline), the order adopted in the validation exercise (manual) and scoreRec's recommended order (scoreRec). Two effects are measured: the size of the investigation set and the occurrences of reexamination decided by a particular resolution order. As for the former, socreRec exhibits a trend of decreasing investigation set size, whereas both the alphabetic order and the manual resolution order lead to a rather random distribution of investigation set size. This implies that scoreRec can contribute to a more systematic and efficient conflict resolution strategy. As for the latter, only two reexaminations are required if scoreRec's resolution order is adopted, whereas the alphabetic order and the manual resolution order resulted in 14 and 11 reexaminations, respectively. This shows that the resolution order offered by scoreRec is indeed effective and beneficial.

4.4 Discussion

We now discuss the limitations that need to be taken into account when interpreting our results, and share the lessons learned from our evaluation. As one of the main objectives of our exploratory case study was to assess the usefulness of scoreRec, we carried out an interview and a validation exercise to identify the factors affecting conflict resolution in practice. Although the software engineer involved in the validation exercise is representative of the experts who are familiar with the subject software system, having multiple engineer participants and/or executing multiple resolution sessions would be worth pursuing. The results generated with the larger samples can be of more statistical relevance. Another limitation refers to our reliance on the post-study interview to collect resolution strategies and tactics. A more naturalistic inquiry, such as 'think aloud', may provide complementary findings about a diverse set of the decisions made during the resolution of various conflicts.

Although we feel that the dynamic analysis serves a good surrogate measure of the 'benefit' for conflict resolution, directly measuring the investigation set and reexamination effort is necessary to quantify the effectiveness of scoreRec. A proper study like this can also help to evaluate the usability aspects of scoreRec. In addition to plan and carry out more empirical studies, we have also identified areas for improvement for scoreRec. One primary issue is to visualise and trace the semantic DPs more explicitly. Also of interest would be categorising the semantic conflicts and the impacted procedures so that more tailored and dynamic resolution strategies and orderings can be generated.

In our ongoing collaboration with ERDC scientists, we have extended the initial interview with the HDT maintainer by interacting with more software engineers. In addition to the code size and complexity, size of the investigation set, and resolution order reported in Section 4.2, we found out that other factors also play important roles in conflict resolution. For example, one maintainer stated that he would always start resolving conflicts for the most familiar and/or well-documented code. In this way, he could be confident that little to no rework (reexamination) would be needed. As we evolve our understanding about how conflict resolution is handled in practice, the practitioners seem to be inspired by our research simultaneously. For instance, the hierarchy of views shown in Figs. 5–7 has led a couple of ERDC software professionals to suggest the integration of more visualisations to the scoreRec tool. In particular, static models like data and control flow diagrams are among the popular requests, together with execution traces that can help to dynamically locate the conflicting regions of code.

5 Related work

In an attempt to classify the design dimensions of RSSEs, Robillard *et al.* [2] discussed three major components (nature of the context, recommendation engine and output mode), along with two crosscutting features (explanation and user feedback). Table 4 uses these dimensions to compare scoreRec with other conflict resolution approaches.

Our approach establishes the merging context implicitly in that scoreRec parses and analyses the parallel programmes

Table 3 Comparing scoreRec's recommended order with the alphabetical order (baseline) and the resolution order used in the manual validation exercise (cf. Table 2)

Step no.	Size of the investigation set			No. of reexaminations		
	Baseline	Manual	scoreRec	Baseline	Manual	scoreRec
1	9	3	12	0	0	0
2	2	5	9	0	1	0
3	8	6	8	1	1	0
4	5	8	7	1	1	0
5	1	8	9	0	2	2
6	4	6	5	1	1	0
7	12	5	4	5	2	0
8	7	4	2	2	2	0
9	9	2	1	4	1	0

Table 4 Comparing scoreRec with other related approaches by the design dimensions discussed in [2]

Approach	Description	Nature of context	Recommendation engine		Output mode		Explanation	User feedback
		Input	Data	Ranking	Mode	Presentation		
Infuse [9]	partition workspaces to reduce interferences	hybrid	source code and change	no	push	batch	none	none
Flexible [26]	offer possible resolution paths for users to choose	hybrid	user interface	no	push	inline	none	individually adaptive
TUKAN [10]	assign weights to software conflicts	implicit	source code and change	yes	push	inline	eetailed	globally adaptive
Treemap [27]	visualise artifact states from other workspaces	implicit	source code and change	no	push	inline	none	globally Adaptive
Jazz [28]	eclipse IDEs additional feature for collaborative development	implicit	source code and change	no	push	inline	none	globally adaptive
Palantír [11]	raise awareness across workspaces along with severity of conflicts	hybrid	source code and change	no	push	inline	detailed	globally adaptive
Indirect conflict [29]	use cross-workspace method to recommend indirect conflict	implicit	source code and change	no	push	inline	none	globally adaptive
Semi-synchronisation [30]	model semi-synchronous distributed conflict detection and resolution	implicit	source code and change	no	push	inline	none	none
Rational management [31]	detect and resolve conflicts in models	implicit	model	no	pull	batch	none	none
Proactive detection [32]	precisely detect conflicts by using previously-unexploited information	Implicit	source code and change	no	pull	batch	none	none
scoreRec ('our approach')	rank conflicting entities based on resolution's cost and benefit	implicit	source code and change	yes	pull	inline	detailed	none

without maintainer's explicitly specifying contextual information such as change intent. Other approaches like Palantír [11] require a hybrid of implicit and explicit context gathering. We feel that automatically extracting the context for RSSEs, though challenging, is promising as researchers and tool builders can take advantage of the ever-growing quantities and types of software development data [33].

RSSEs must analyse more than context data to make recommendations [2]. In our approach, the scoreRec 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 4, taking into account of high-level artifacts (e.g. models [31]) can potentially increase the power of the recommendation engine. However, it is important to synchronise heterogeneous artifacts collected throughout the software lifecycle.

Despite the key role that ranking plays in RSSEs [2], it is surprising to note that, among the existing approaches surveyed in Table 4, only TUKAN [10] 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. [9–11, 26–30]) 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 recommendation generation. For example, the model management tool presented in [31] requires user interaction to choose from the conflicting operations. Similarly, users receive recommendations from scoreRec simply by clicking buttons and switching tabs. As far as the presentation style is concerned, some tools integrate into other environments, following an inline presentation style, for example Palantír [11] is built upon version control systems and TUKAN [10] is integrated in the VisualWorks/EVNY Smalltalk environment. A similar inline mode is adopted in our approach, although scoreRec is implemented on a textual merging system.

In order to justify the recommendations made, scoreRec gives detailed and hierarchical explanations. We not only provide rankings like TUKAN [10] 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 Palantír [11], 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 [2], in order to extend scoreRec along the collaborative conflict resolution dimension [8].

6 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 scoreRec, 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 an empirical evaluation through an industrial-strength software project. The results show that scoreRec offers sound engineering support by providing an efficient resolution order together with detailed explanations.

From our initial experiences with scoreRec, 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 conduct more thorough analysis to tease out and quantify the factors that influence the conflict resolution process. Finally, we would like to enrich scoreRec with a set of visualisations of the control and data flow diagrams, the execution traces, and the tailored resolution recommendations. The goal is to enhance scoreRec's practical value in supporting parallel software development.

7 Acknowledgment

The work is supported by the US 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.

8 References

1 Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H.: 'An exploratory study of how developer seek, relate, and collect relevant information during software maintenance tasks', *IEEE Trans. Softw. Eng.*, 2006, **32**, (12), pp. 971–987

2 Robillard, M.P., Walker, R.J., Zimmermann, T.: 'Recommendation systems for software engineering', *IEEE Softw.*, 2010, **27**, (4), pp. 80–86

3 Perry, D.E., Siy, H.P., Votta, L.G.: 'Parallel changes in large scale software development: an observational case study'. Proc. IEEE Int. Conf. Software Engineering, 1998, pp. 251–260

4 Mens, T.: 'A state-of-the-art survey on software merging', *IEEE Trans. Softw. Eng.*, 2002, **28**, (5), pp. 449–462

5 Conradi, R., Westfechtel, B.: 'Version models for software configuration management', *ACM Comput. Surv.*, 1998, **30**, (2), pp. 232–282

6 Niu, N., Easterbrook, S., Sabetzadeh, M.: 'A category-theoretic approach to syntactic software merging'. Proc. IEEE Int. Conf. on Software Maintenance, 2005, pp. 197–206

7 Jackson, D., Ladd, D.A.: 'Semantic Diff: a tool for summarizing the effects of modifications'. Proc. IEEE Int. Conf. on Software Maintenance, 1994, pp. 243–252

8 Dewan, P.: 'Dimensions of tools for detecting software conflicts'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2008, pp. 21–25

9 Perry, D.E., Kaiser, G.E.: 'Infuse: a tool for automatically managing and coordinating source changes in large systems'. Proc. ACM Annual Conf. on Computer Science, 1987, pp. 292–299

10 Schümmer, T.: 'Lost and found in software space'. Proc. Annual Hawaii Int. Conf. on System Sciences, 2001

11 Sarma, A., Zoroozi, Z., van der Hoek, A.: 'Palantír: raising awareness among configuration management workspaces'. Proc. IEEE Int. Conf. on Software Engineering, 2003, pp. 444–454

12 Niu, N., Yang, F., Cheng, J.-R.C., Reddivari, S.: 'A cost-benefit approach to recommending conflict resolution for parallel software development'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2012, pp. 21–25

13 Niu, N., Easterbrook, S.: 'So, you think you know others' goals? A repertory grid study', *IEEE Softw.*, 2007, **24**, (2), pp. 53–61

14 Nuseibeh, B., Easterbrook, S., Russo, A.: 'Making inconsistency respectable in software development', *J. Syst. Softw.*, 2001, **58**, (2), pp. 171–180

15 Rich, C., Waters, R.: 'The programmers' apprentice' (Addison-Wesley, 1990)

16 Munson, J.P., Dewan, P.: 'A flexible object merging framework'. Proc. ACM Conf. on Computer Supported Cooperative Work, 1994, pp. 231–241

17 Burch, M., Schafer, T., Mezini, M.: 'On evaluating recommender systems for API usage'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2008, pp. 16–20

18 Hou, D., Pletcher, D.M.: 'Towards a better code completion system by API grouping, filtering and popularity-based ranking'. Proc. Int.

Workshop on Recommendation Systems for Software Engineering, 2010, pp. 26–30

19 McMillan, C., Poshyvanyk, D., Grechanik, M.: 'Recommending source code examples via API call usage and documentation'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2010, pp. 21–25

20 Felfernig, A., Schubert, M., Mandl, M.: 'Recommendation and decision technologies for requirements engineering'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2010, pp. 11–15

21 Kpodjedo, S., Ricca, F., Galinier, P., Antoniol, G.: 'Not all classes are created equal: toward a recommendation system for focusing testing'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2008, pp. 6–10

22 Brudaru, I.I., Zeller, A.: 'What is the long-term impact of changes?'. Proc. Int. Workshop on Recommendation Systems for Software Engineering, 2008, pp. 30–32

23 Chen, K., Rajlich, V.: 'RIPPLES: tool for change in legacy software'. Proc. IEEE Int. Conf. on Software Maintenance, 2001, pp. 230–239

24 Yin, R.K.: 'Case study research: design and methods' (Sage Publications, 2003, 3rd edn.)

25 Apiwattanapong, T., Orso, A., Harrold, M.J.: 'Efficient and precise dynamic impact analysis using execute-after sequences'. Proc. IEEE Int. Conf. on Software Engineering, 2005, pp. 432–441

26 Edwards, W.K.: 'Flexible conflict detection and management in collaborative applications'. Proc. ACM Symp. on User Interface Software and Technology, 1997, pp. 139–148

27 Molli, P., Skaf-Molli, H., Bouthier, C.: 'State Treemap: An awareness widget for multi-synchronous groupware'. Proc. Int. Workshop on Groupware, 2001, pp. 106–114

28 Cheng, L.T., Hupfer, S., Ross, S., Patterson, J.: 'Jazzing up eclipse with collaborative tools'. Proc. OOPSLA Workshop on Eclipse Technology eXchange, 2003

29 Sarma, A., Bortis, G., van der Hoek, A.: 'Towards supporting awareness of indirect conflicts across software configuration management workspaces'. Proc. ACM/IEEE Int. Conf. on Automated Software Engineering, 2007, pp. 94–103

30 Dewan, P., Hegde, R.: 'Semi-synchronous conflict detection and resolution in asynchronous software development'. Proc. European Conf. on Computer-Supported Cooperative Work, 2007

31 Koegel, M., Helming, J., Seyboth, S.: 'Operation-based conflict detection and resolution'. Proc. ICSE Workshop on Comparison and Versioning of Software Models, 2009, pp. 43–48

32 Brun, Y., Holmes, R., Ernst, M.D., Notkin, D.: 'Proactive detection of collaboration conflicts'. Proc. ACM SIGSOFT Symp. on Foundations of Software Engineering, 2011, pp. 168–178

33 Zeller, A.: 'The future of programming environments: integration, synergy, and assistance'. Proc. Future of Software Engineering, 2007, pp. 316–325