Spork: Structured merge for Java with GumTree diff and 3DM-merge

Simon Larsén

November 16, 2020

KTH Royal Institute of Technology

Introduction

Fundamentals of merging

Spork

Experimental methodology

Results

Discussion and conclusions

Questions?

Introduction

Merging of source code

- Merging of different revisions - as in GIT-MERGE
- Distributed version control (DVCS): branching cheap and easy
- Parallel development on multiple branches ⇒ we must merge!

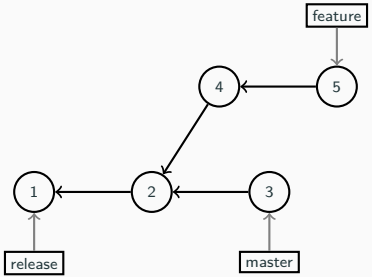


Figure 1: A branching commit history

Problems with the state of the practice

- State of the practice is *unstructured* merge, typically line-based
  - Coarse and often leads to conflicts [1], [2]
- Solving merge conflicts manually introduces bugs [1], [3]–[6]

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
<<<<<<< Left revision
    tot += i
=======
    tot = tot + i
  print(tot)
>>>>>>> Right revision
  return tot
```

Figure 2: Merge with conflict crossing syntactical boundaries

One solution: structured merge

- Structured merge works on a structural representation [1]
  - e.g. *abstract syntax tree* (AST) or some more general graph
- Understands the structure of the code
  - And perhaps semantics as well

Thesis goals

- Create a structured merge tool that
  - Furthers increased adoption
  - Improves upon the state of the art
- Target JAVA
  - Current best tool: JDIME [2], [7]–[9]
- In particular, improve upon:
  - Conflict handling
  - Runtime
  - Formatting preservation
  - Merge correctness

Research questions

- RQ1: How does SPORK compare to JDIME in terms of amounts and sizes of conflicts?
- RQ2: How does SPORK compare to JDIME in terms of runtime?
- RQ3: How does SPORK compare to JDIME in terms of preserving source code formatting?
- RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

Fundamentals of merging

VCS terminology

- Revision - same as version
  - May refer to a single file or an entire project
- Repository - the store of all version history
- Commit - a *saved state* in the repository
- Branch - an independent line of development within a repository

What is a merge?

- Combine branches to create one consistent state
- Preserve all changes from all branches

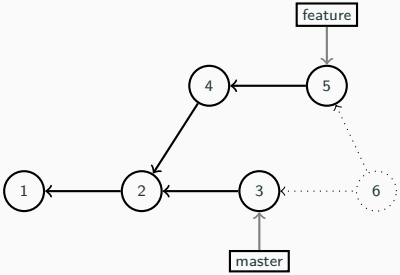


Figure 3: A branching commit history

File merge

- More than one branch edits the same file $\rightarrow$ file merge
- This is what SPORK does!
- Typically a two-step process:
  - File differencing
  - Merging

A simple file merge

- Assume that `master` and `feature` both edit the same file
- We must perform a file merge!

```
def arith_sum(n):
  tot = 0
  print(tot)
  for i in 1..n:
    tot += i
  return tot
```

(a) master revision (left)

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
    tot = tot + i
  print(tot)
  return tot
```

(b) feature revision (right)

Figure 4: Two revisions of the same file

File differencing

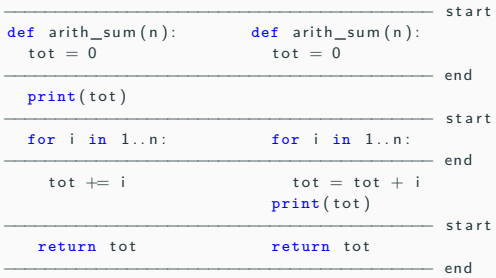


Figure 5: Visualization of a line-based matching of the left and right revisions. start and end in the right margin marks the start and end of a matching block.

- Matchings
- What happened where?

The merge base

- We need more context
- *Merge base*: Closest common ancestor of revisions under merged[1]

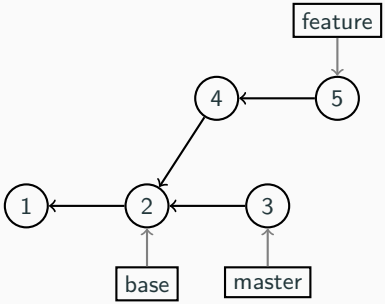


Figure 6: A branching commit history

[1] https://git-scm.com/docs/git-merge-base

The three-way merge

```
def arith_sum(n):
  tot = 0
  print(tot)
  for i in 1..n:
    tot += i
  return tot
```

(a) Left revision

```
def arith_sum(n):
  tot = 0
  print(tot)
  for i in 1..n:
    tot = tot + i
  return tot
```

(b) Base revision

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
    tot = tot + i
  print(tot)
  return tot
```

(c) Right revision

Figure 7: Three revisions of the same file, where left and right are derived from base

The three-way merge 2

- Match lines in common across all three revisions
- Unmatched lines represent edits

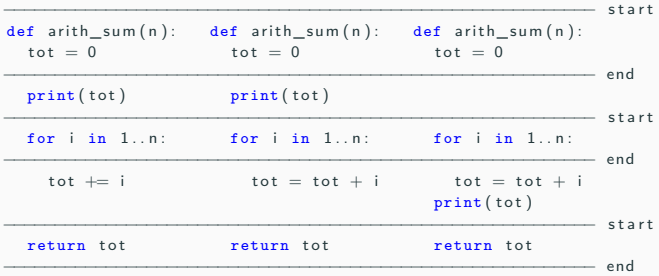


Figure 8: Line matchings across the left, base and right revisions

Merge resolution

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
<<<<<<< Left revision
    tot += i
=======
    tot = tot + i
  print(tot)
>>>>>>> Right revision
  return tot
```

(a) Line-based merge result, containing a boundary-crossing conflict

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
    tot += i
  print(tot)
  return tot
```

(b) Expected merge result

Figure 9: Actual and expected merge results

Structured merge: exploding the structure

- The fundamental difference is the increased granularity of an AST
- Structured merge of single child list $\approx$ unstructured merge of entire file

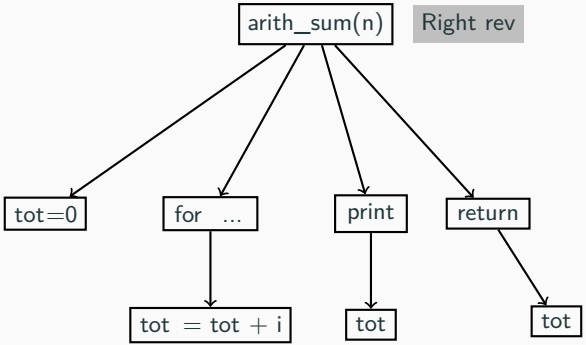


Figure 10: Simplified AST of the right revision

Structured merge: more granular, same principles

- Match AST nodes instead of lines
- Merge individual child lists instead of flat list of lines

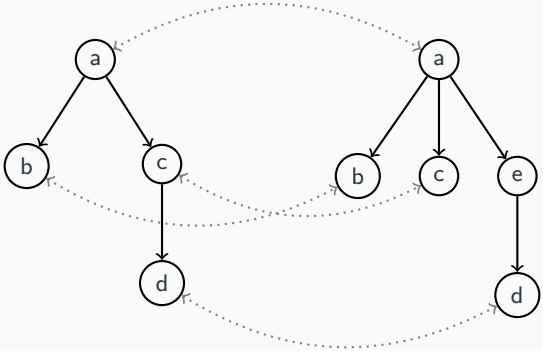


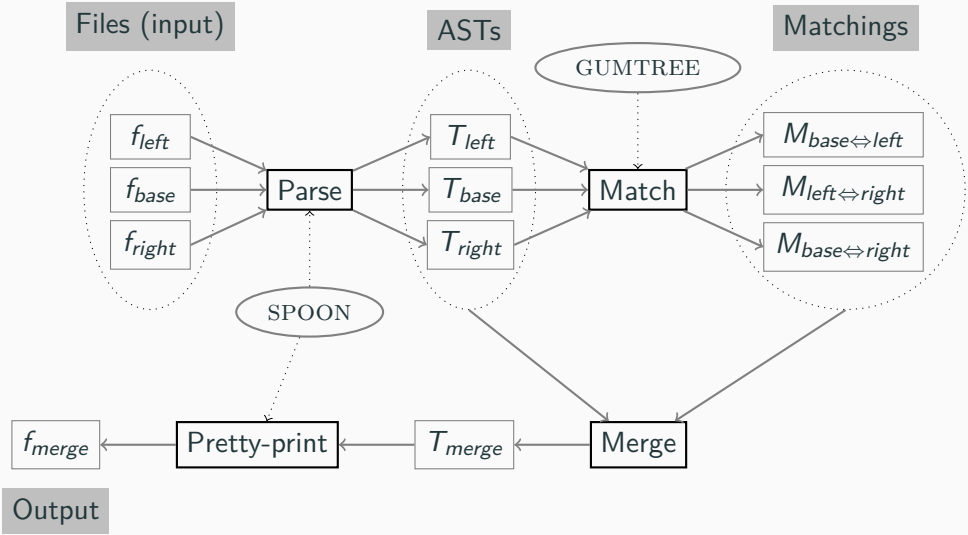
Figure 11: A matching between two trees

Spork

Three key pieces

- Uses SPOON [10] for AST
  - Name derived from SPOON + fork: SPORK!
- Uses GUMTREE [11] for tree diff
  - Can match moved and updated nodes, is *move-enabled*
- *Implements* 3DM-MERGE [12]
  - Modified version, called SPORK-3DM
  - Can act upon moved and updated matchings, is *move-enabled*

Spork architecture



Example benefit of move-enabled merge

- A rename-method refactoring

```
def sum_arith(n):
  tot = 0
  for i in 1..n:
    tot += i
  return tot
```

(a) Left revision

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
    tot += i
  return tot
```

(b) Base revision

```
def arith_sum(n):
  tot = 0
  for i in 1..n:
    tot += i
  print(tot)
  return tot
```

(c) Right revision

Figure 12: Three revisions of the same file, where left and right are derived from base

Experimental methodology

Research questions

- RQ1: How does SPORK compare to JDIME in terms of amounts and sizes of conflicts?
- RQ2: How does SPORK compare to JDIME in terms of runtime?
- RQ3: How does SPORK compare to JDIME in terms of preserving source code formatting?
- RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

Dataset

- Projects selected from the REAPER dataset [13]
- Clone projects from GITHUB
- In total:
  - 119 projects
  - 890 merge scenarios
  - 1740 file merges

Expected and replayed revisions

- *Expected revision* - revision commited by the developers
- *Replayed revision* - merge computed by the merge tool under test

Experiment protocols

- Protocal for RQ1, RQ2 and RQ3
  - Replay individual file merges
  - Measure conflicts, runtime and diff size with expected revision
  - Challenge: extracting the file revisions
- Protocol for RQ4
  - Replay entire merge scenario
  - Compare compiled bytecode with normalized bytecode diff
  - Challenge: everything

Results

RQ1

- RQ1: Amounts and sizes of conflicts?
- SPORK produces fewer conflicts
  - SPORK avoids renaming-related conflicts
  - JDIME finds some true conflicts that SPORK does not
  - File header bug caused 45/308 (~14%) of SPORK's conflicts
- Conflict sizes: JDIME produces smaller conflicts
  - Mostly due to bugs in SPORK
  - Move conflicts - SPORK falls back to line-based merge

RQ2

- RQ2: Runtime performance?

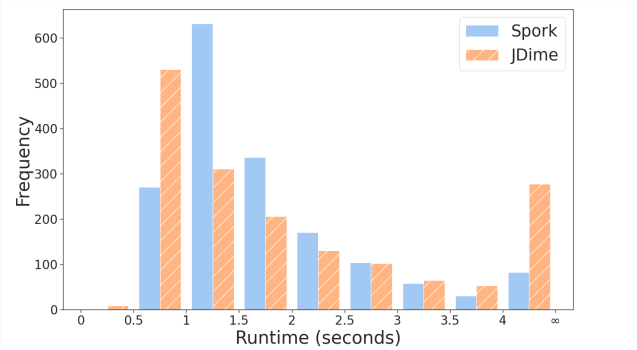


Figure 13: Histogram of file merge runtimes for for SPORK and JDIME

- JDIME faster for small merges, SPORK faster for larger

RQ3

- RQ3: Preservation of source code formatting?

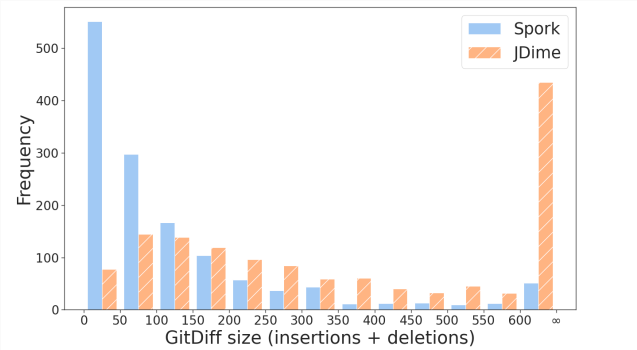


Figure 14: Histogram of GitDiff sizes for SPORK and JDIME

- SPORK is 4x better in the median case

RQ4

- RQ4: How does SPORK compare to JDIME in terms of producing merges that are semantically equivalent to the merges committed by developers?

Table 1: Contingency table showing the amount of source files for which JDIME's and SPORK's merges yield the same or opposite results

	JDIME fail	JDIME success
SPORK fail	621	176
SPORK success	154	783

- No significant difference in discordant case probability

RQ summary

- RQ1: Conflict quantity/size
  - A: SPORK produces fewer but slightly larger conflict hunks
- RQ2: Runtime performance
  - A: SPORK slower for small merges, but scales better
- RQ3: Formatting preservation
  - A: SPORK knocks it out of the park
- RQ4: Semantic equivalency with merges comitted by developers
  - A: No discernable difference

Discussion and conclusions

Imperfect experiments

- Merge committed by developers may be incorrect
- Conflict amount not the whole story: false negatives
- Bytecode equivalency stricter than actual semantic equivalency

Performance differences

- SPORK matches or exceeds JDIME's performance in most aspects
- JDIME has more robust conflict handling
  - But we know what to work on to catch up
- Limiting large runtimes arguably more important
  - 0.5s or 1s is not much of a difference, but 10s and 100s is substantial
- Hard to definitively state a cause for a performance diff due to tool differences

Conclusions

- SPORK works!
  - Find it at https://github.com/KTH/spork
- Move-enabled structured merge for JAVA is feasible
- Move conflicts are difficult to handle - needs more work
- SPORK is modular: can use to experiment with different diff algorithms

Questions?

References i

[1] T. Mens, "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462, May 2002.

[2] O. Leßenich, S. Apel, and C. Lengauer, "Balancing precision and performance in structured merge," *Automated Software Engineering*, vol. 22, no. 3, pp. 367–397, May 2014.

[3] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE international working conference on mining software repositories*, 2009.

References ii

[4] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering - FSE 12*, 2012.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, Oct. 2013.

[6] B. K. Kasi and A. Sarma, "Cassandra: Proactive conflict minimization through optimized task scheduling," in *Proceedings of the 2013 international conference on software engineering*, 2013, pp. 732–741.

[7] O. Leßenich, "Adjustable syntactic merge of java programs," Master's thesis, Universität Passau, 2012.

References iii

[8] O. Leßenich, S. Apel, C. Kästner, G. Seibt, and J. Siegmund, "Renaming and shifted code in structured merging: Looking ahead for precision and performance," in *Proceedings of the 32Nd ieee/acm international conference on automated software engineering*, 2017, pp. 543–553.

[9] G. Cavalcanti, P. Borba, G. Seibt, and S. Apel, "The impact of structure on software merging: Semistructured versus structured merge," in *2019 34th ieee/acm international conference on automated software engineering (ase)*, 2019, pp. 1002–1013.

[10] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "SPOON: A library for implementing analyses and transformations of java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, Aug. 2015.

[11] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on automated software engineering - ASE 14*, 2014.

[12] T. Lindholm, "A three-way merge for XML documents," in *Proceedings of the 2004 ACM symposium on document engineering - DocEng 04*, 2004.

[13] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Apr. 2017.