Merge and Code Review

Paulo Borba
Informatics Center

pauloborba.cin.ufpe.br

Semantic merge

There are code integration problems beyond unnecessary merge conflicts…

Build conflicts

(static semantics/syntactic conflicts)

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```

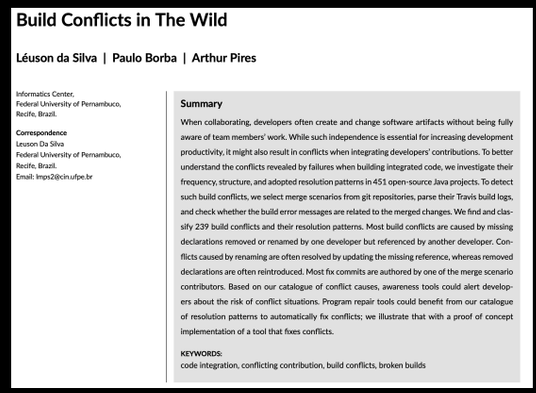
```
class Text {
  public String text;
  …
  void normalizeWhitespace() {…}

  void cleanText() {
    removeComments();
  }
}
```

merge

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    normalizeWhitespace();
  }
}
```

```
class Text {
  public String text;
  …
  void normalizeWhitespace() {…}

  void cleanText() {
    removeComments();
   normalizeWhitespace();
  }
}
```

https://pauloborba.cin.ufpe.br/publication/2022build_conflicts_in_the_wild/2022-Silva-Build conflicts in the wild.pdf



Build Conflicts in The Wild

Léuson da Silva | Paulo Borba | Arthur Pires

Informatics Center,
Federal University of Pernambuco,
Recife, Brazil.

Correspondence
Leuson Da Silva
Federal University of Pernambuco,
Recife, Brazil.
Email: lmps2@cin.ufpe.br

Summary

When collaborating, developers often create and change software artifacts without being fully aware of team members' work. While such independence is essential for increasing development productivity, it might also result in conflicts when integrating developers' contributions. To better understand the conflicts revealed by failures when building integrated code, we investigate their frequency, structure, and adopted resolution patterns in 451 open-source Java projects. To detect such build conflicts, we select merge scenarios from git repositories, parse their Travis build logs, and check whether the build error messages are related to the merged changes. We find and classify 239 build conflicts and their resolution patterns. Most build conflicts are caused by missing declarations removed or renamed by one developer but referenced by another developer. Conflicts caused by renaming are often resolved by updating the missing reference, whereas removed declarations are often reintroduced. Most fix commits are authored by one of the merge scenario contributors. Based on our catalogue of conflict causes, awareness tools could alert developers about the risk of conflict situations. Program repair tools could benefit from our catalogue of resolution patterns to automatically fix conflicts; we illustrate that with a proof of concept implementation of a tool that fixes conflicts.

KEYWORDS:
code integration, conflicting contribution, build conflicts, broken builds

Understanding and automatically resolving build conflicts

(57,065 merges from 451 Java projects with Travis CI)

6 conflict patterns

unavailable symbol is the most common (65%)

17 resolutions patterns

build conflict repair tool

covering 3 patterns

https://is.gd/TJnNcc

L. Da Silva. Detecting, Understanding and Resolving Build and Test Conflicts. ICSE Poster 2019.

L. Da Silva, P. Borba , A. Pires. Build Conflicts in The Wild. Submitted for publication, 2020.

These are caused by
syntactic or static
semantics incompatibilities

But not as bad as dynamic
semantic incompatibilities
(test or production conflicts)

Test or production conflicts

(dynamic semantics conflicts)

Detecting Semantic Conflicts via
Automated Behavior Change Detection

Leuson Da Silva*, Paulo Borba*, Wardah Mahmood+, Thorsten Berger+, and João Moisakis*

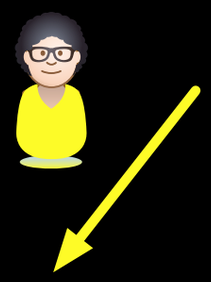
*Federal University of Pernambuco, Recife, Brazil
+Chalmers | University of Gothenburg, Gothenburg, Sweden

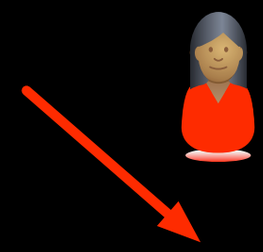
***Abstract*—Branching and merging are common practices in collaborative software development. They increase developer productivity by fostering teamwork, allowing developers to independently contribute to a software project. Despite such benefits, branching and merging comes at a cost—the need to merge software and to resolve merge conflicts, which often occur in practice. While modern merge techniques, such as 3-way or structured merge, can resolve many such conflicts automatically,**

changes to be merged. This can negatively affect development productivity, and even compromise software quality in case developers incorrectly fix conflicts [17], [6], [18]. To avoid dealing with merge conflicts, developers sometimes even adopt risky practices, such as rushing to finish changes first [19], [17] and partial check-ins [20]. Similarly, partially motivated by the need to reduce merge conflicts, development teams have been

https://pauloborba.cin.ufpe.br/publication/2020detecting_semantic_conflicts_via_automated_behavior_change_detection/

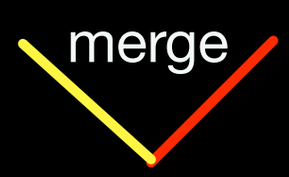
```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```





```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```




```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

```
class Text {
   public String text;
   …
   void cleanText() {
      normalizeWhitespace();
      removeComments();
      removeDuplicateWords();
   }
}
```

~~resulting text has no duplicate whitespace~~

resulting text has no duplicate words

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace()); FAILS!
```

Conflict terminology: process x language aspects

- merge conflict — textual conflict
- build conflict — static semantics/syntactic conflict
- test conflict — dynamic/behavioral semantics conflict
- production conflict — dynamic/behavioral semantics conflict

Conflict terminology

- merge conflict—not clear how to put together textual changes—textual conflict (but not quite that for semistructured merge tools, and eventually for semantic merge tools; so classification valid only for line based merge tools) (for s3m: not clear how to put together textual changes, or clear but they break some rule that is checked—duplicated declarations—and that would lead to a build issue)

- build conflict—clear how to put together textual changes, but they lead to a build error—often due to a static semantics incompatibility, but could also be a syntactic incompatibility (one guy removed a try-catch, the other added a finally after an existing comment)

- test conflict—clear how to put together textual changes, they don’t lead to a build error, but lead to a project test error— due to a dynamic/behavioral semantics incompatibility

- production conflict (conflict revealed as a bug, escaped defect)—clear how to put together textual changes, they don’t lead to a build error, they don't lead to a project test error, but lead to an untested execution error— due to a dynamic/behavioural semantics incompatibility

dependences
but no
coordination

knowledge
breakdown

integration
conflicts

syntactic,
static semantics

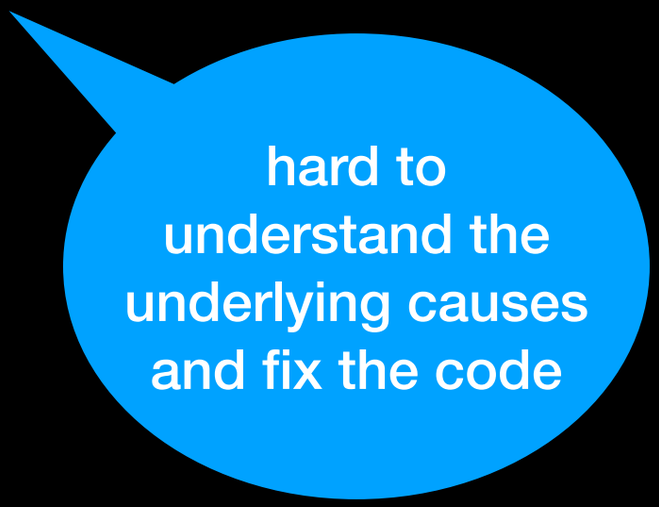
tedious and
risky resolution

hard to find
and fix bugs

dynamic semantics

Current merge tools are oblivious to the semantics of the code changes that they integrate

Missed conflicts are hardly detected by project tests or code reviews, and end up escaping to system users



We need semantic merge tools to detect and resolve dynamic semantics conflicts

But what exactly is a dynamic semantics conflict?

Undesired interference between integrated developers changes

G. Cavalcanti, P. Borba, P. Accioly. Evaluating and Improving Semistructured Merge. OOPSLA 2017.

But what exactly is interference?

Two procedures interfere when one can perform a global action which has a global effect up on the other

J. Reynolds. Syntactic Control of Interference.
POPL 1978.

One group of users, using a certain set of commands, is noninterfering with another group of users if what the first group does with these commands has no effect on what the second group of users can see

J. Goguen and J. Meseguer. Security Policies and Security Models. IEEE SSP 1982.

Separate changes L and R to a base program B interfere when the integrated changes does not preserve the changed behavior of L or R, or the unchanged behavior of B

S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. TOPLAS 1989 (POPL 1988).

J. Reynolds. Syntactic Control of Interference. POPL 1978.

J. Goguen and J. Meseguer. Security Policies and Security Models. IEEE SSP 1982.

We have interference because the integrated changes does not preserve the changed behavior of the yellow developer (version)

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

```
text = "the␣the␣␣dog"

text = "the␣the␣dog"

text = "the␣␣dog"
```

We have a semantic conflict because we have undesired interference,

assuming the intention captured in the boxes

resulting text has no duplicate whitespace

resulting text has no duplicate words

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

```
text = "the␣the␣␣dog"

text = "the␣the␣dog"

text = "the␣␣dog"
```

Partially expressing interference in terms of specifications

Separate changes L and R to a base program interfere if their behavior change specifications are not jointly satisfied by the program that integrates L and R

behavior change specifications constrain only the values of the state elements that have been affected by the change

```
{true}

normalizeWhitespace();
removeComments();

{no duplicate whitespace}
```

```
{true}

removeComments();
removeDuplicateWords();

{no duplicate words}
```

```
{true && true}

normalizeWhitespace();
removeComments();
removeDuplicateWords();

{no duplicate whitespace && no duplicate words}
```

{preL}

L

{postL}

{preR}

R

{postR}

{preL && preR}

merge(L,R,Base)

{postL && postR}

Implication, not equivalence: interference because the unchanged behavior of base is not preserved

```
{true}
x = 1;
skip;
y = 0;
if (x+y > 1) {z = 1;}
else {z = 0;}
{x == 1}
```

$$z = 0$$

```
{true}
x = 0;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
{y == 1}
```

```
{true && true}
x = 1;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
{x == 1 && y == 1}
```

$$z = 1$$

Partially expressing interference in terms of more concrete conditions

Separate changes L and R to a base program B interfere if there is a state element x such that B, L and R compute different values for x

S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. TOPLAS 1989 (POPL 1988).

variable instead of state element

B, L and R compute different values for `Text.text`

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

`text = "the␣the␣␣dog"` `text = "the␣the␣dog"` `text = "the␣␣␣dog"`

Implication, not equivalence: there is no such x but there is interference

(unchanged behavior of base is not preserved)

```
x = 1;
skip;
y = 0;
if (x+y > 1) {z = 1;}
else {z = 0;}
```

z = 0

```
x = 0;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
```

z = 0

```
x = 1;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
```

z = 1

Separate changes L and R to a base program B interfere if there is a state element x such that L (or R) computes a different value for x than B and Merge

S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. TOPLAS 1989 (POPL 1988).

variable instead of state element

negation

L compute different value for `Text.text` than B and Merge compute different

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

`text = "the_the__dog"` `text = "the_the_dog"` `text = "the__dog"`

Implication, not equivalence: there is no such x but there is interference

(unchanged behavior of base is not preserved)

```
x = 1;
skip;
y = 0;
if (x+y > 1) {z = 1;}
  else {z = 0;}
```

z = 0

```
x = 0;
skip;
y = 1;
if (x+y > 1) {z = 1;}
  else {z = 0;}
```

z = 0

```
x = 1;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
```

z = 1

Separate changes L and R to a base program B interfere if there is a state element x such that B, L and R compute the same value for x but Merge computes a different value

S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. TOPLAS 1989 (POPL 1988).

variable instead of state element

negation

B, L and R compute the same value for z but Merge computes a different value

```
x = 1;
skip;
y = 0;
if (x+y > 1) {z = 1;}
  else {z = 0;}
```

z = 0

```
x = 0;
skip;
y = 1;
if (x+y > 1) {z = 1;}
  else {z = 0;}
```

z = 0

```
x = 1;
skip;
y = 1;
if (x+y > 1) {z = 1;}
else {z = 0;}
```

z = 1

Detailing the auxiliary definitions

Analyzing methods with an open world perspective

State element stands for any part of the system state (global variables, object fields, system files, output stream, visible GUI elements, variables that hold method return values or raised exceptions, etc.)

Analyzing methods with a restricted open world (API) perspective

State element stands for any part of the system state (global variables, object fields, system files, output stream, visible GUI elements, variables that hold method return values or raised exceptions, etc.) that affect the execution of other methods of the same API

Analyzing the system with a closed world perspective

State element stands for any part of the system state that is externally visible during and after program execution (system files, output stream, visible GUI elements, etc.)

Computes captures reachability. So integrated changes might involve only statements with y, but end up impacting the final computed value of x

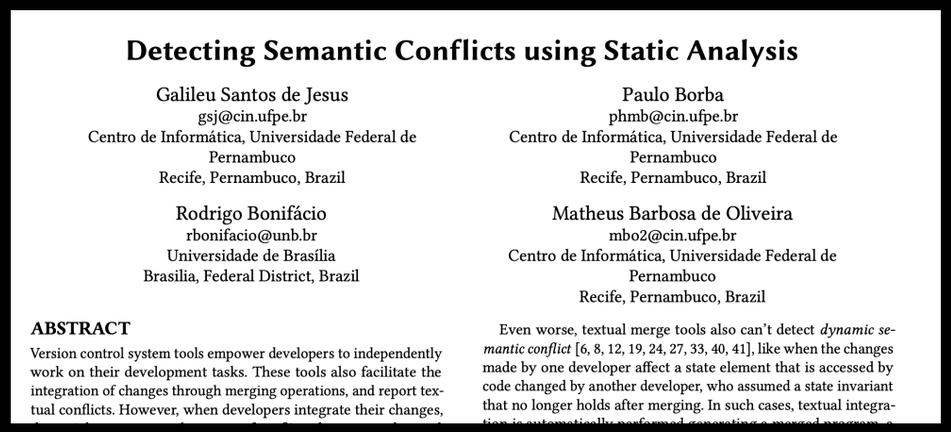
Assumptions

- Sequential execution
- No observation of system resources (CPU time, memory, power consumption, etc.) usage
- Changes that do not affect interfaces

But interference is not computable!

Are there approximations for automatically detecting interference (and semantic conflicts)?

Detecting interference with static analysis



Detecting Semantic Conflicts using Static Analysis

Galileu Santos de Jesus
gsj@cin.ufpe.br
Centro de Informática, Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil

Paulo Borba
phmb@cin.ufpe.br
Centro de Informática, Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil

Rodrigo Bonifácio
rbonifacio@unb.br
Universidade de Brasília
Brasilia, Federal District, Brazil

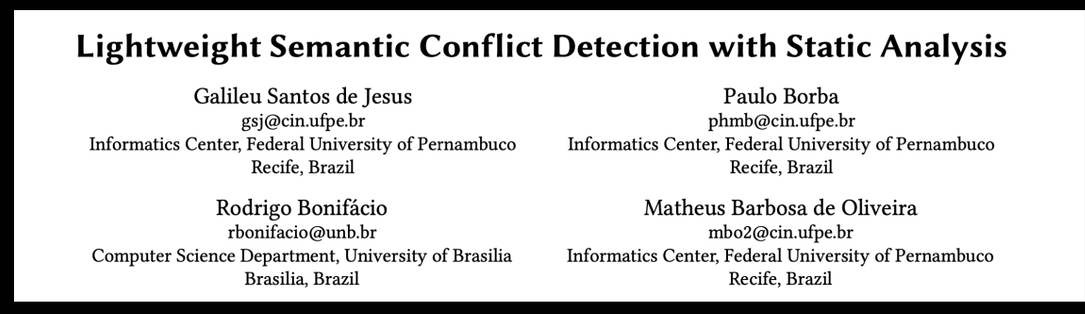
Matheus Barbosa de Oliveira
mbo2@cin.ufpe.br
Centro de Informática, Universidade Federal de Pernambuco
Recife, Pernambuco, Brazil

ABSTRACT

Version control system tools empower developers to independently work on their development tasks. These tools also facilitate the integration of changes through merging operations, and report textual conflicts. However, when developers integrate their changes,

Even worse, textual merge tools also can't detect *dynamic semantic conflict* [6, 8, 12, 19, 24, 27, 33, 40, 41], like when the changes made by one developer affect a state element that is accessed by code changed by another developer, who assumed a state invariant that no longer holds after merging. In such cases, textual integration is automatically performed generating a merged program a

https://arxiv.org/pdf/2310.04269



Lightweight Semantic Conflict Detection with Static Analysis

Galileu Santos de Jesus
gsj@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Paulo Borba
phmb@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

Rodrigo Bonifácio
rbonifacio@unb.br
Computer Science Department, University of Brasilia
Brasilia, Brazil

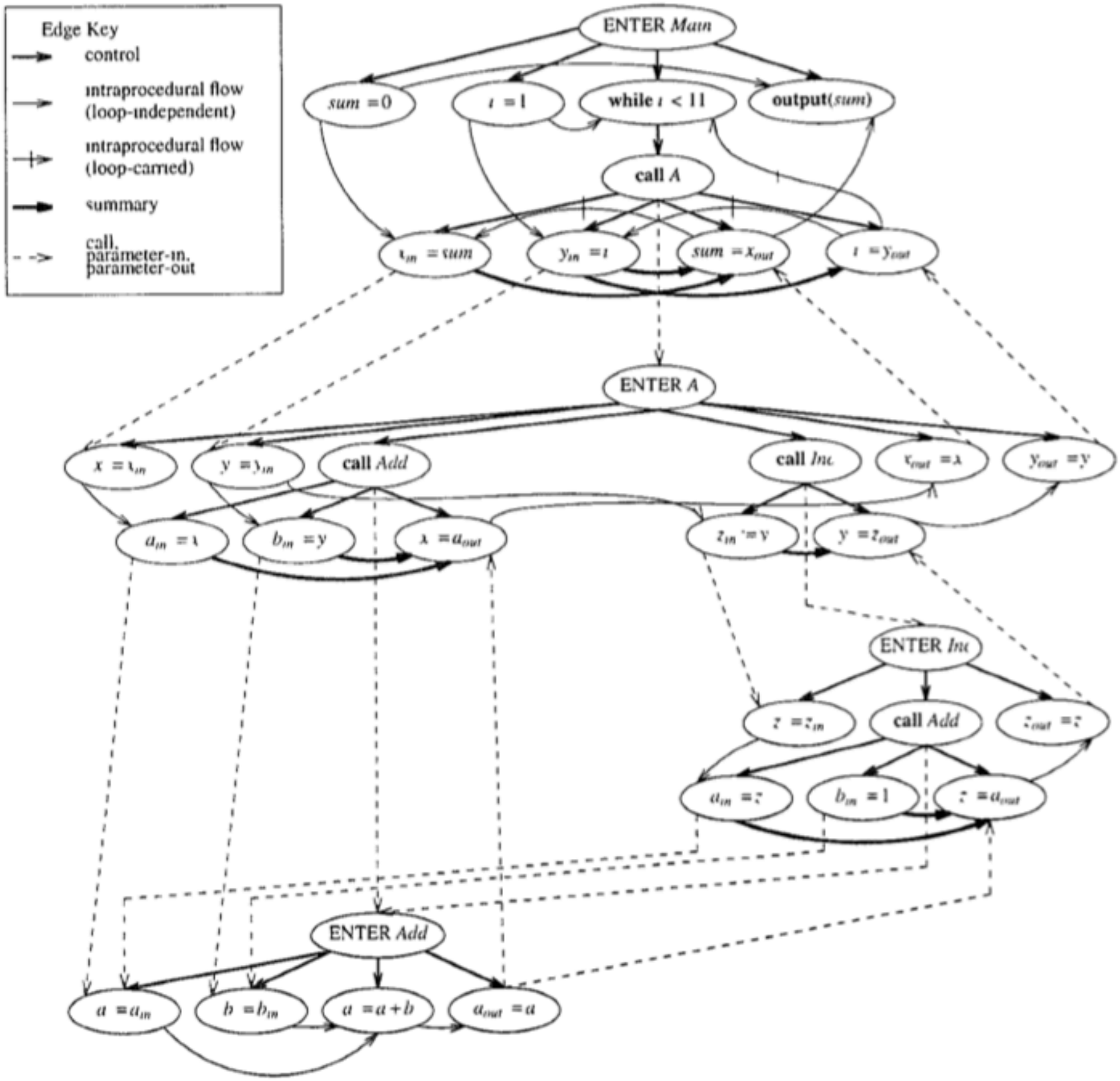
Matheus Barbosa de Oliveira
mbo2@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Recife, Brazil

https://pauloborba.cin.ufpe.br/publication/2024lightweight_semantic_conflict_detection_with_static_analysis/

Building and comparing 3 SDGS

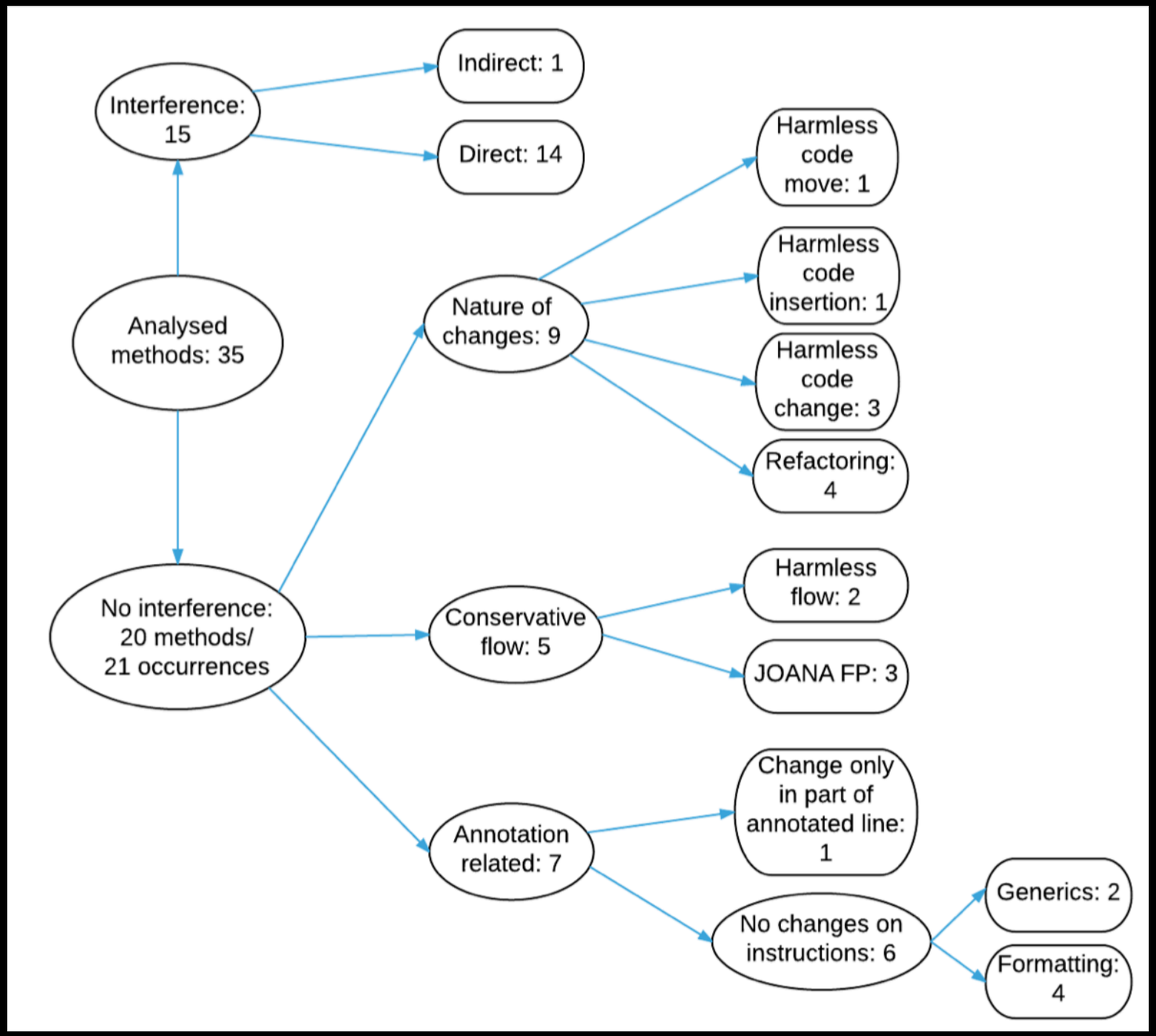
D. Binkley, S. Horwitz, T. Reps. Program Integration for Languages with Procedural Calls. TOSEM 1995.

```
procedure Main
  sum = 0
  i = 1
  while i < 11 do
      call A(sum, i)
  od
  output(sum)
end

procedure A(x, y)
  call Add(x, y)
  call Increment(y)
return

procedure Add(a, b)
  a = a + b
return

procedure Increment(z)
  call Add(z, 1)
return
```



assumes statements (and SDG nodes) are uniquely identified

Information flow (between developers changes) analysis implementation, for a single SDG, is highly resource demanding



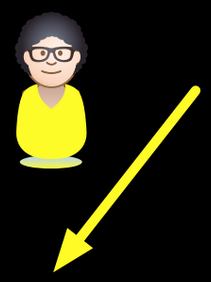
R. Barros Filho, P. Borba. Semantic conflict detection with information flow analysis. 2020.

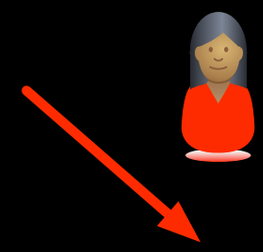
Empirical study 1

- JOANA (information flow analysis) based implementation
- Analysis of 72 merge scenarios with developers changes to a common method
  - using semi-structured merge to integrate changes
  - 35 with information flow between contributions

Is there a reasonably accurate and lightweight approximation?

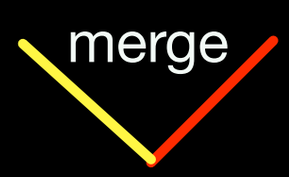
```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```





```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```




```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

```
class Text {
   public String text;
   …
   void cleanText() {
      normalizeWhitespace();
      removeComments();
      removeDuplicateWords();
   }
}
```

~~resulting text has no duplicate whitespace~~

resulting text has no duplicate words

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace()); FAILS!
```

Analyze only the merged program version, annotated with the origin of the changes

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

Detecting data flow between developers changes

```
class Text {
  String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

rd/wr

rd/wr

a path from a yellow to a red command, or vice-versa, indicates interference risk

Detecting data flow confluence from developers changes

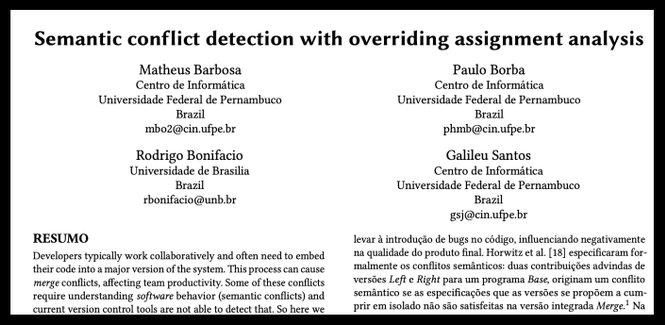
```
class Text {
  String text;
  int words;
  int spaces;
  …
  int countFixes() {
    countDupWhitespace();
    countComments();
    countDupWords();
    return spaces + words;
  }
}
```

paths from yellow and red commands to a common target indicates interference risk

Detecting overriding assignments involving developers changes

```
class Text {
  String text;
  int fixes;
  …
  int countFixes() {
    countDupWhitespace();
    countComments();
    countDupWords();
    return fixes;
  }
}
```

write paths, without intermediate assignments, to a common target indicates interference risk



Semantic conflict detection with overriding assignment analysis

Matheus Barbosa
Centro de Informática
Universidade Federal de Pernambuco
Brazil
mbo2@cin.ufpe.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brazil
phmb@cin.ufpe.br

Rodrigo Bonifacio
Universidade de Brasilia
Brazil
rbonifacio@unb.br

Galileu Santos
Centro de Informática
Universidade Federal de Pernambuco
Brazil
gsj@cin.ufpe.br

RESUMO

Developers typically work collaboratively and often need to embed their code into a major version of the system. This process can cause *merge* conflicts, affecting team productivity. Some of these conflicts require understanding *software* behavior (semantic conflicts) and current version control tools are not able to detect that. So here we

levar à introdução de bugs no código, influenciando negativamente na qualidade do produto final. Horwitz et al. [18] especificaram formalmente os conflitos semânticos: duas contribuições advindas de versões *Left* e *Right* para um programa *Base*, originam um conflito semântico se as especificações que as versões se propõem a cumprir em isolado não são satisfeitas na versão integrada *Merge*.[1] Na

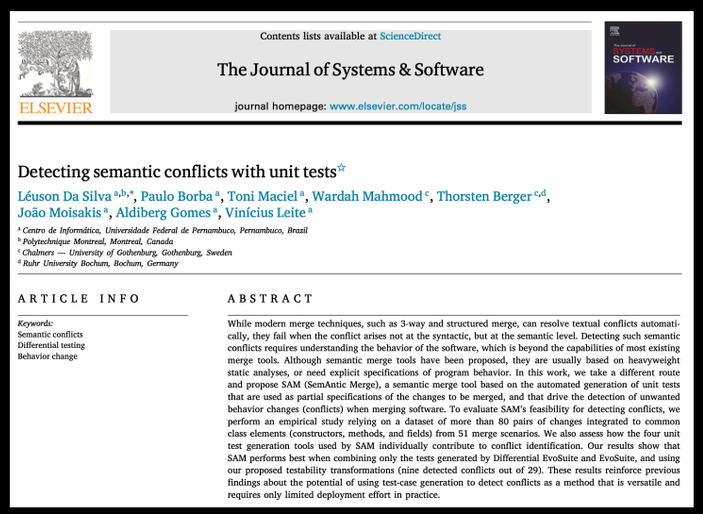
https://pauloborba.cin.ufpe.br/publication/2022semantic_conflict_detection_with_overriding_assignment_analysis/

Detecting control dependences involving developers changes

```
class Text {
  String text;
  …
  void cleanText() {
    if (text != null &&
        hasWhitespace()) {
      normalizeWhitespace();
      removeDuplicateWords();
    }
  }
}
```

Detecting interference with testing

https://pauloborba.cin.ufpe.br/publication/2024detecting_semantic_conflicts_with_unit_tests/



Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Detecting semantic conflicts with unit tests☆

Léuson Da Silva [a,b,*], Paulo Borba [a], Toni Maciel [a], Wardah Mahmood [c], Thorsten Berger [c,d], João Moisakis [a], Aldiberg Gomes [a], Vinícius Leite [a]

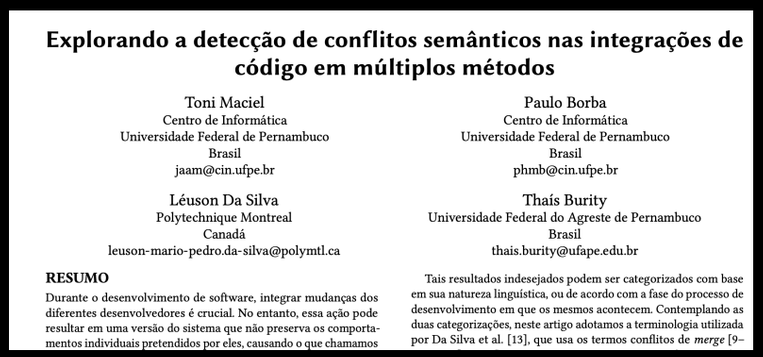
[a] Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brasil
[b] Polytechnique Montreal, Montreal, Canada
[c] Chalmers — University of Gothenburg, Gothenburg, Sweden
[d] Ruhr University Bochum, Bochum, Germany

ARTICLE INFO

Keywords:
Semantic conflicts
Differential testing
Behavior change

ABSTRACT

While modern merge techniques, such as 3-way and structured merge, can resolve textual conflicts automatically, they fail when the conflict arises not at the syntactic, but at the semantic level. Detecting such semantic conflicts requires understanding the behavior of the software, which is beyond the capabilities of most existing merge tools. Although semantic merge tools have been proposed, they are usually based on heavyweight static analyses, or need explicit specifications of program behavior. In this work, we take a different route and propose SAM (SemAntic Merge), a semantic merge tool based on the automated generation of unit tests that are used as partial specifications of the changes to be merged, and that drive the detection of unwanted behavior changes (conflicts) when merging software. To evaluate SAM's feasibility for detecting conflicts, we perform an empirical study relying on a dataset of more than 80 pairs of changes integrated to common class elements (constructors, methods, and fields) from 51 merge scenarios. We also assess how the four unit test generation tools used by SAM individually contribute to conflict identification. Our results show that SAM performs best when combining only the tests generated by Differential EvoSuite and EvoSuite, and using our proposed testability transformations (nine detected conflicts out of 29). These results reinforce previous findings about the potential of using test-case generation to detect conflicts as a method that is versatile and requires only limited deployment effort in practice.



Explorando a detecção de conflitos semânticos nas integrações de código em múltiplos métodos

Toni Maciel
Centro de Informática
Universidade Federal de Pernambuco
Brasil
jaam@cin.ufpe.br

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco
Brasil
phmb@cin.ufpe.br

Léuson Da Silva
Polytechnique Montreal
Canadá
leuson-mario-pedro.da-silva@polymtl.ca

Thaís Burity
Universidade Federal do Agreste de Pernambuco
Brasil
thais.burity@ufape.edu.br

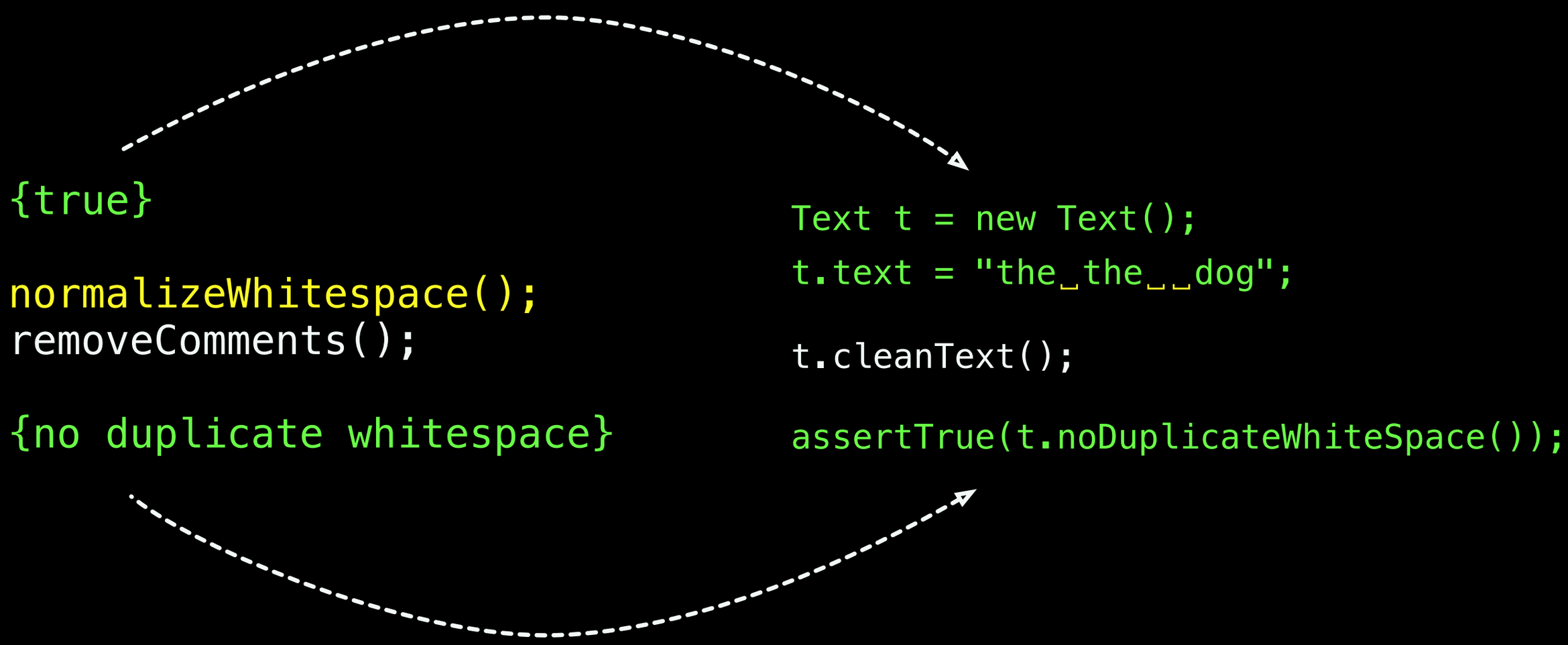
RESUMO

Durante o desenvolvimento de software, integrar mudanças dos diferentes desenvolvedores é crucial. No entanto, essa ação pode resultar em uma versão do sistema que não preserva os comportamentos individuais pretendidos por eles, causando o que chamamos

Tais resultados indesejados podem ser categorizados com base em sua natureza linguística, ou de acordo com a fase do processo de desenvolvimento em que os mesmos acontecem. Contemplando as duas categorizações, neste artigo adotamos a terminologia utilizada por Da Silva et al. [13], que usa os termos conflitos de *merge* [9–

https://pauloborba.cin.ufpe.br/publication/2024explorando_a_deteccao_de_conflitos_semanticos_nas_integracoes_conflitos-com-multiplos-metodos.pdf

Tests as partial specifications of behavior changes



L. Da Silva, P. Borba, W. Mahmood, T. Berger, J. Moisakis. Detecting Semantic Conflicts via Automated Behavior Change Detection. ICSME 2020.

Interference expressed as behavior change specifications

{preL}

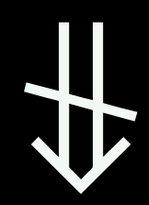
L

{postL}

{preR}

R

{postR}

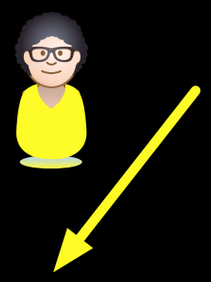


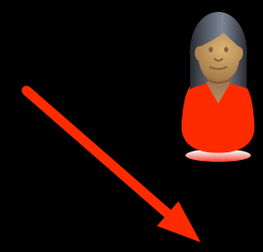
{preL && preR}

merge(L,R,Base)

{postL && postR}

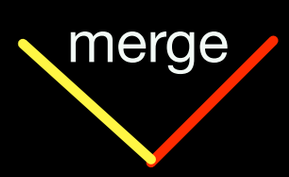
```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```





```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```




```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

Criteria

Changed behavior is not preserved

✘

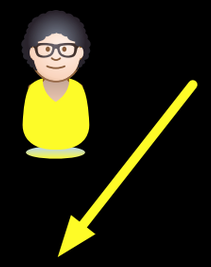
```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

✔

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

✘

```
Text t = new Text();
t.text = "the␣the␣␣dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```



```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

✔

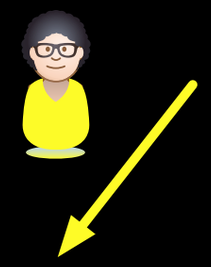
```
Text t = new Text();
t.text =...;
t.cleanText();
assertTrue(...);
```

✘

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```



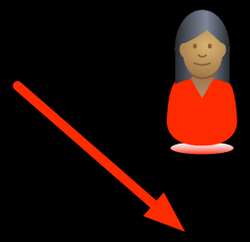
```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

Symmetric criteria for the red change

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```



❌

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```

✔️

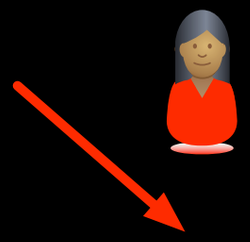
```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

❌

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```



✔

```
Text t = new Text();
t.text =...;
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```

✘

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

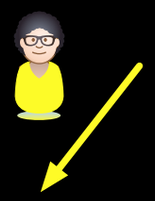
✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

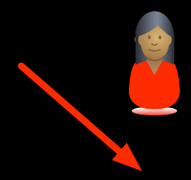
Unchanged behavior is not preserved

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```



```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```



✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    removeDuplicateWords();
  }
}
```

✔

```
Text t = new Text();
t.text = "...";
t.cleanText();
assertTrue(...);
```

merge

✘

```
Text t = new Text();
t.text = ...;
t.cleanText();
assertTrue(...);
```

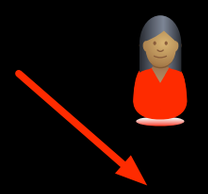
```
class Text {
  public String text;
  …
  void cleanText() {
    normalizeWhitespace();
    removeComments();
    removeDuplicateWords();
  }
}
```

No interference example: criteria do not apply

✘

```
Text t = new Text();
t.text = "the_the__dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
  }
}
```



✔

```
Text t = new Text();
t.text = "the_the__dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeComments();
    normalizeWhitespace();
  }
}
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeDuplicateWords();
    removeComments();
  }
}
```

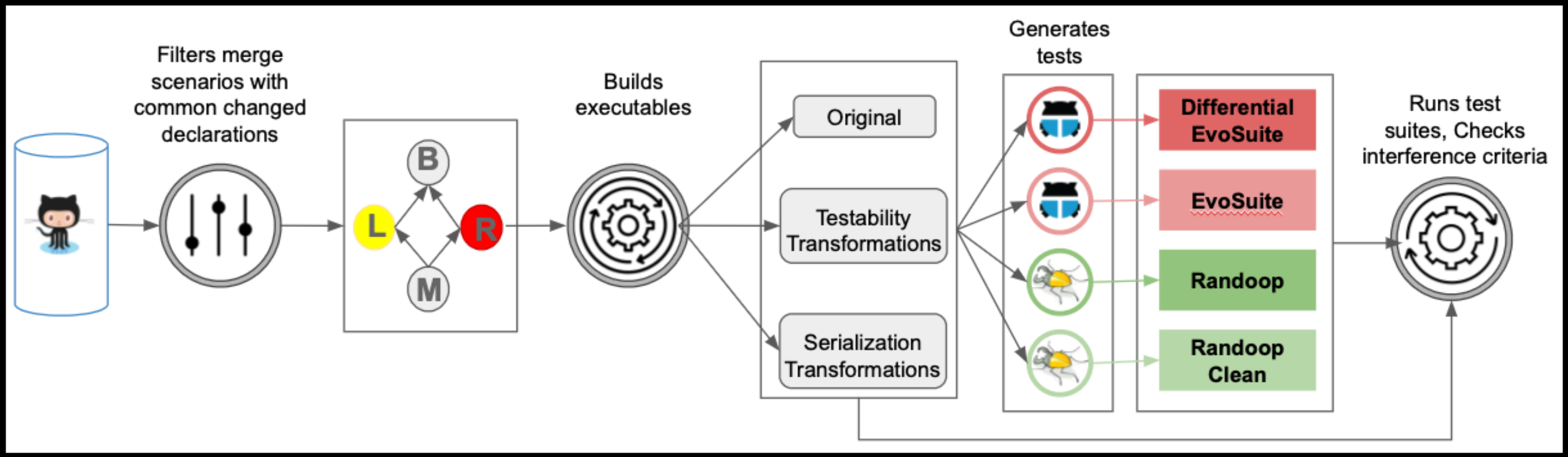
merge

✔

```
Text t = new Text();
t.text = "the_the__dog";
t.cleanText();
assertTrue(t.noDuplicateWhiteSpace());
```

```
class Text {
  public String text;
  …
  void cleanText() {
    removeDuplicateWords();
    removeComments();
    normalizeWhitespace();
  }
}
```

Evaluation overview



Merge and Code Review

Paulo Borba
Informatics Center

pauloborba.cin.ufpe.br

Semantic merge