Security Level:

# CodeBot

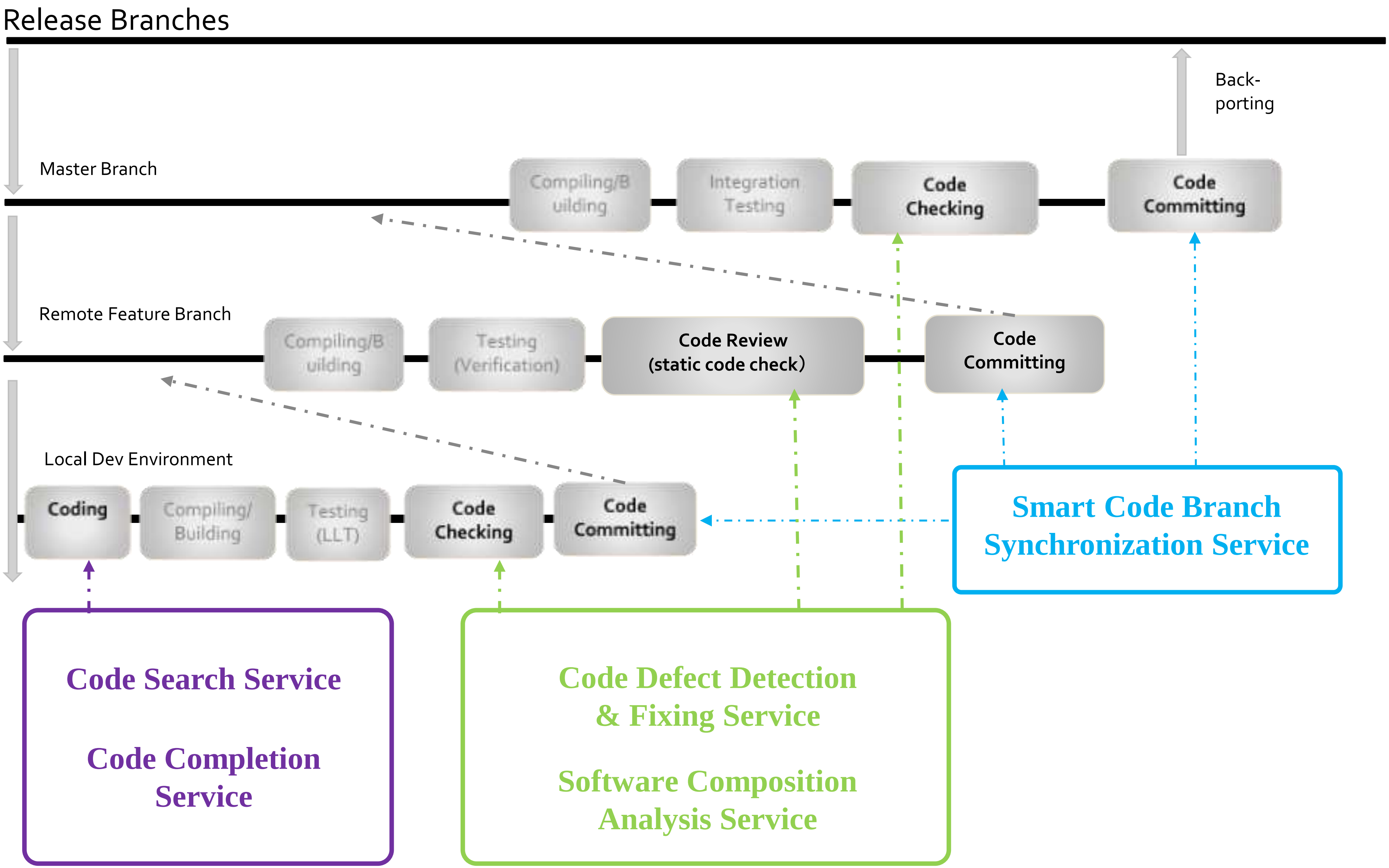
## A SMART WEAPON to rescue developers from ANNOYING CODING PROCESSES

www.huawei.com

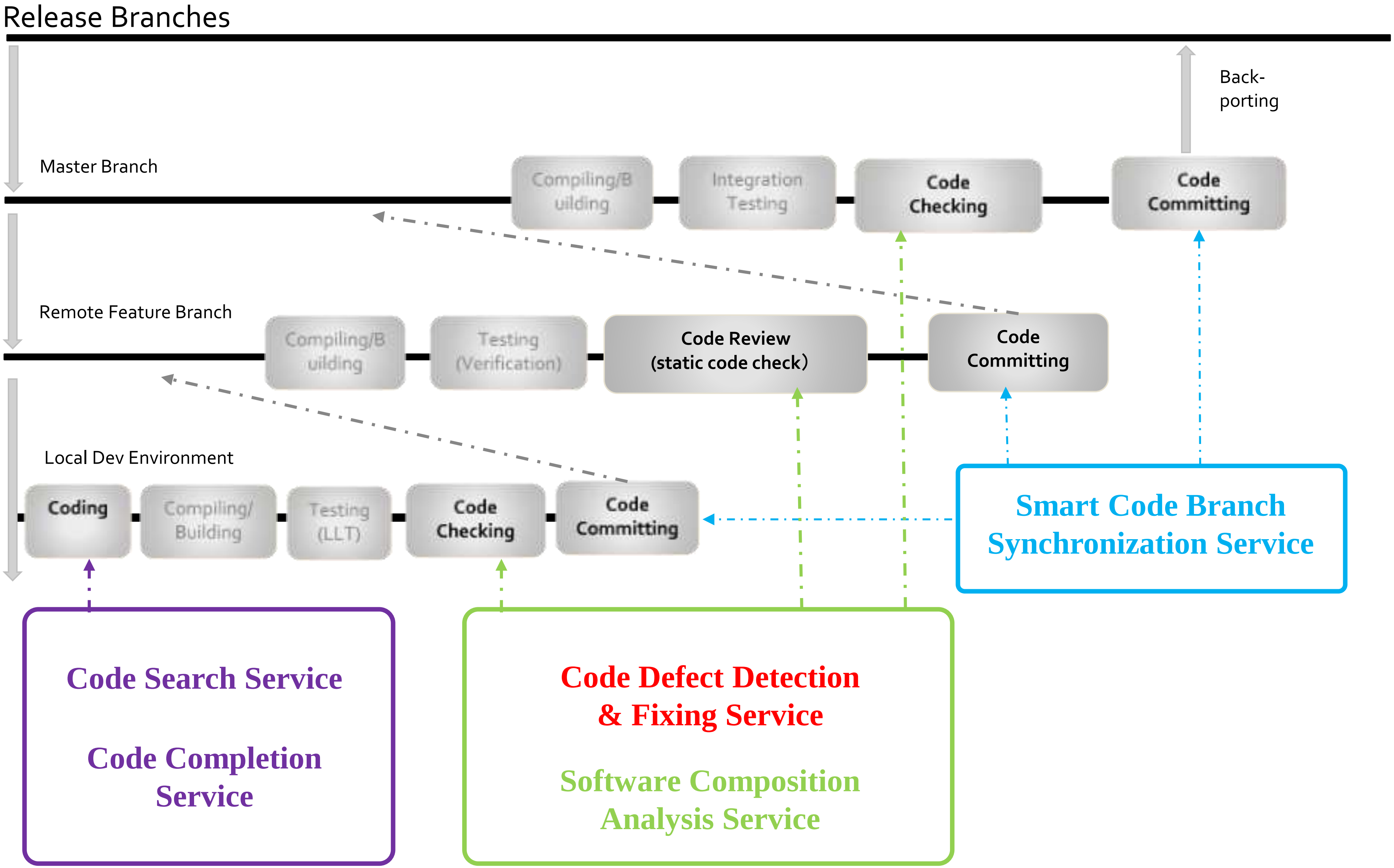
Author/ Email: Guangtai Liang (梁广泰) / liangguangtai@huawei.com
Version: V2.0 (20191130)

HUAWEI TECHNOLOGIES CO., LTD.

# CodeBot Overview



# CodeBot Overview

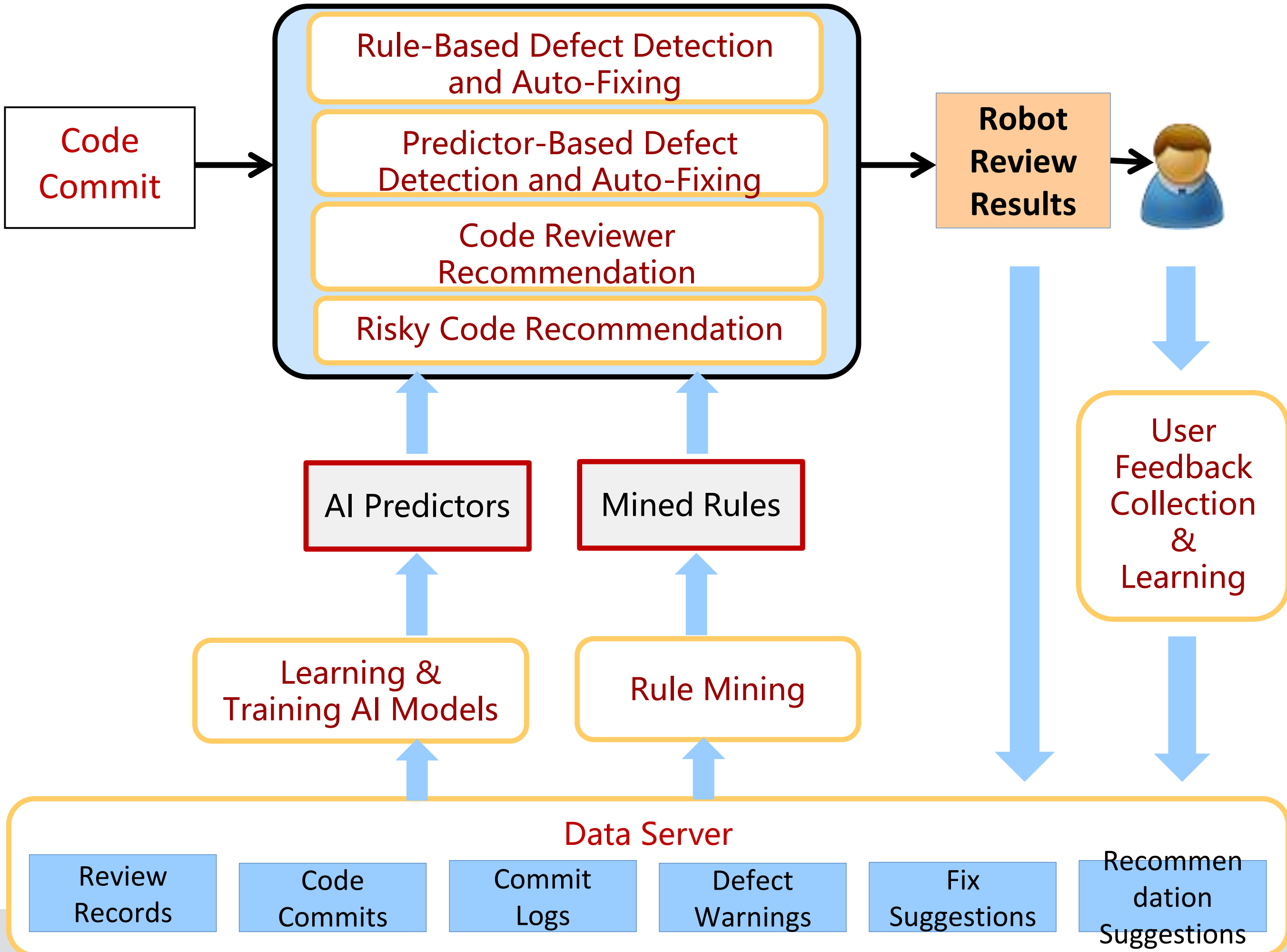


# Smart Code Defect Detection & Fixing Service

**Goals**

Building an ecosystem for detecting various kinds of defects efficiently and effectively

1. Producing effective results (precision > 90%)
2. Scalable for easily integrating third-party code detectors
3. Integrated with existing working flow (coding, code review, code release)
4. Continuously collect and learn from historical code defects




## Key Techs

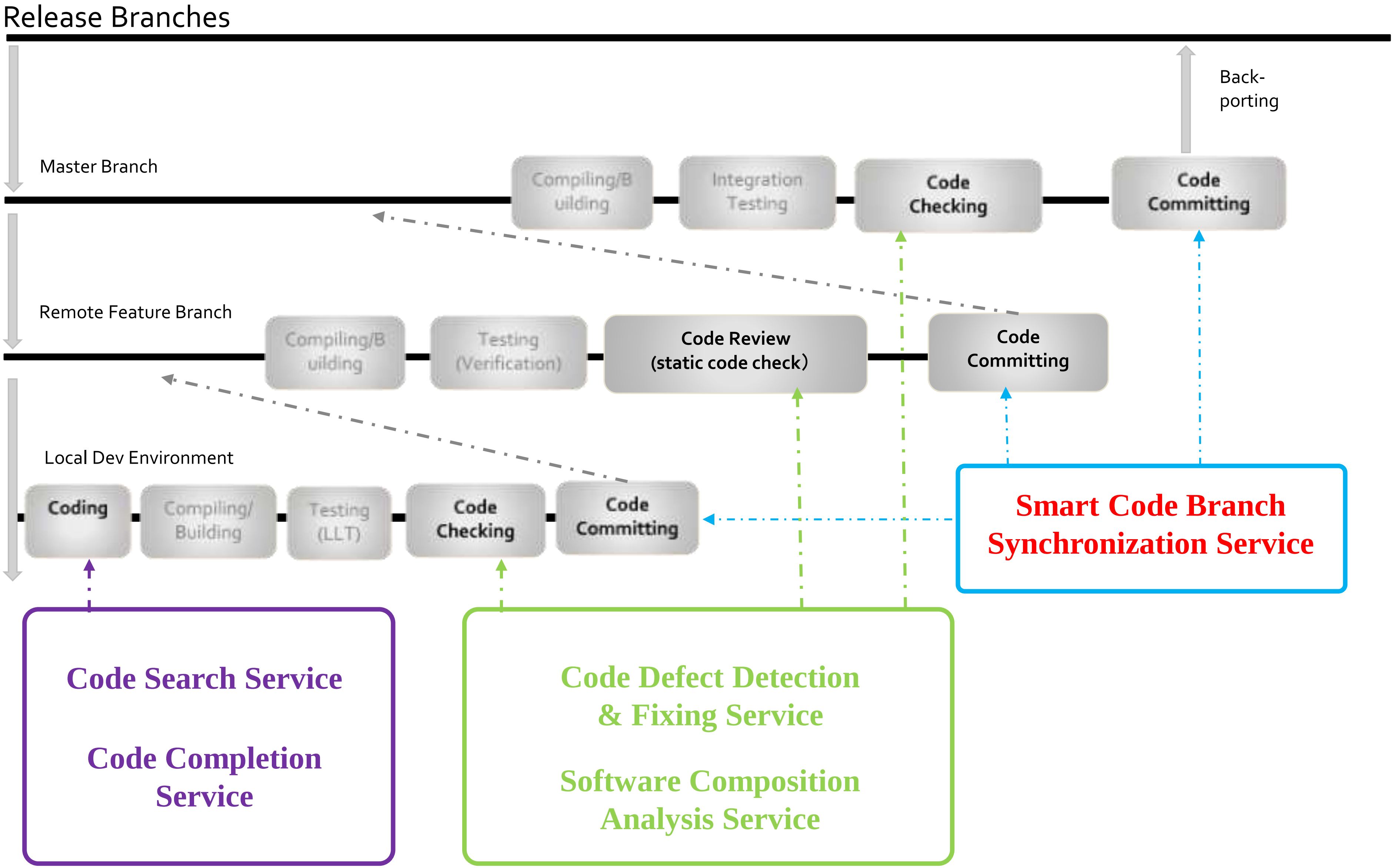
**① Defect Detection**

- Defect pattern mining
- Deep/precise/scalable analysis engine
- Formal approaches: Theorem proving、abstract interpretation, symbolic execution and etc.
- **AI based false positive reducing**

**② Defect Fixing**

- Fix Pattern Mining
- Fix Pattern Auto-Applying
  - Fix example providing
  - Fix code auto-generation
  - Interactive code fixing

# CodeBot Overview



# Smart Code Branch Sync. Service

Linux openstack android hadoop kubernetes → HUAWEI Customized Systems (e.g., EMUI)

**The Code Syncing Processes are**

- **huge conflicts (for android P upgrading, 249342 conflict lines) awaiting manual resolution**
- **labor-intensive and low efficiency (android N/O upgrading costs 800/1200 person-months)**
- **error-prone**
- **false merges**
- **false conflicts**

**Existing Process:** Change Analysis → Code Merge & Conflict Resolving → Compiling & Unit Testing → Screen-Brightening Testing → Apk Usability Testing → Formal Testing → Feature-Gap Development

**Pain Points:**

- Labor-intensive, low efficiency, false negatives
- Huge conflicting lines, tight schedule, lack of guidance, uncontrollable quality
- Unlimited manual testing efforts, lack of guidance, Testing blind spots, inefficient root cause analysis
- Lots of code couplings committed into the git repo

**Key Features:**

**Code Change Analysis**

Changed features, APIs, UI pages, impact scopes, potential conflicts and etc.

Diff Analyzer

**Code Conflict Auto-Resolving**

Code conflict hub construction, Merge rule mining, Rule based merge engine, AI based predictors

Conflict Auto-Resolver

**Implicit-Conflict Analysis & Forewarning**

False merge identification, Incomplete merge identification and etc.

Implicit Conflict Warner

**Conflict Prevention**

Conflict prediction & warning, Clean commits, Feature-based merge

Conflict Preventer

# Smart Code Branch Sync. Service

Linux, openstack, android, hadoop, kubernetes → HUAWEI Customized Systems (e.g., EMUI)

**The Code Syncing Processes are**

- **huge conflicts (for android P upgrading, 249342 conflict lines) awaiting manual resolution**
- **labor-intensive and low efficiency (android N/O upgrading costs 800/1200 person-months)**
- **error-prone**
- **false merges**
- **false conflicts**

**Existing Process:** Change Analysis → Code Merge & Conflict Resolving → Compiling & Unit Testing → Screen-Brightening Testing → Apk Usability Testing → Formal Testing → Feature-Gap Development

**Pain Points:**

- Labor-intensive, low efficiency, false negatives
- Huge conflicting lines, tight schedule, lack of guidance, uncontrollable quality
- Unlimited manual testing efforts, lack of guidance, Testing blind spots, inefficient root cause analysis
- Lots of code couplings committed into the git repo

**Key Features:**

| **Code Change Analysis** | **Code Conflict Auto-Resolving** | **Implicit-Conflict Analysis & Forewarning** | **Conflict Prevention** |
|---|---|---|---|
| Changed features, APIs, UI pages, impact scopes, potential conflicts and etc. | Code conflict hub construction, Merge rule mining, Rule based merge engine, AI based predictors | False merge identification, Incomplete merge identification and etc. | Conflict prediction & warning, Clean commits, Feature-based merge |
| Diff Analyzer | Conflict Auto-Resolver | Implicit Conflict Warner | Conflict Preventer |

# OOPSLA-2019 Work:

# IntelliMerge: A Refactoring-Aware Software Merging Technique

Bo Shen[1], Wei Zhang[1], Haiyan Zhao[1], Guangtai Liang[2], Zhi Jin[1], and Qianxiang Wang[2]

[1]Peking University, China

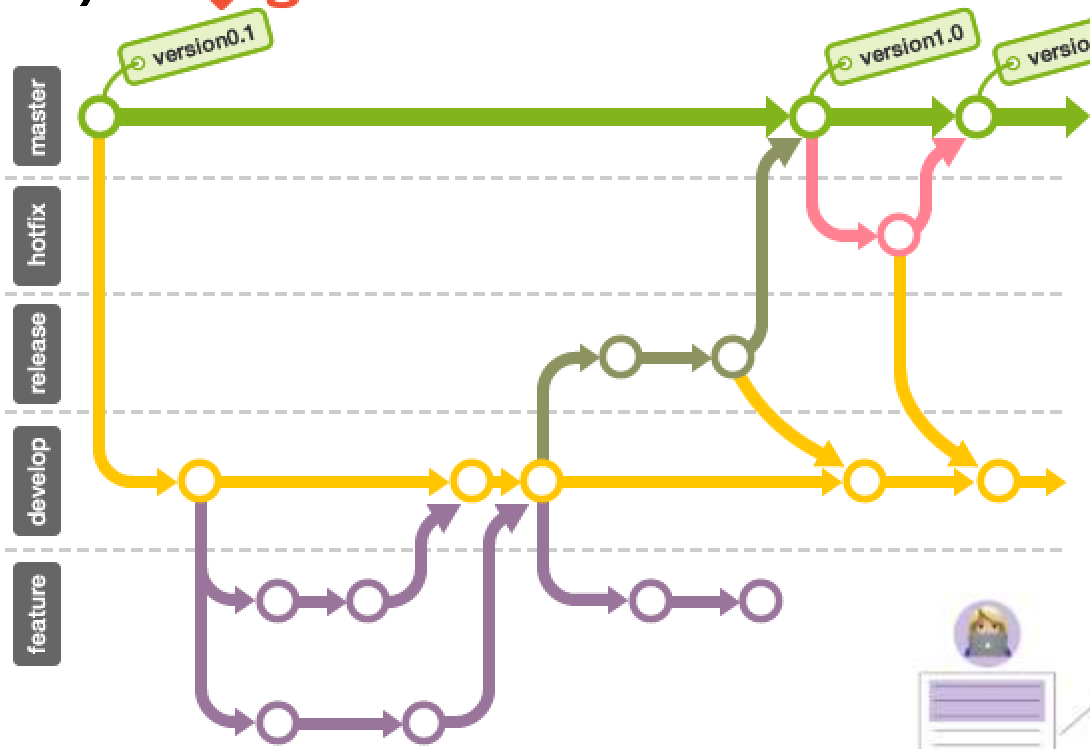
[2]Huawei Technologies Co. Ltd, China

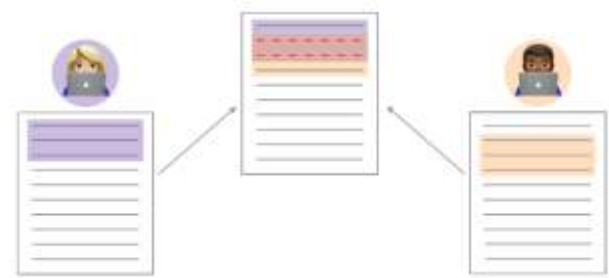
HUAWEI

北京大學 PEKING UNIVERSITY

# Software/Program/Code Merging

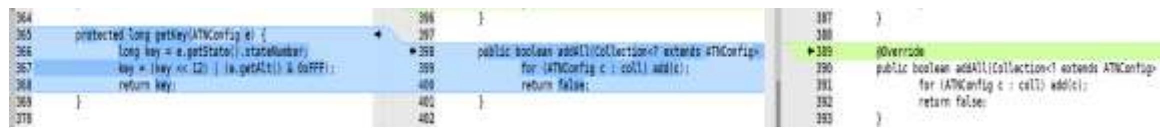
Merging happens frequently in version control systems (like git )and branch-based workflow.

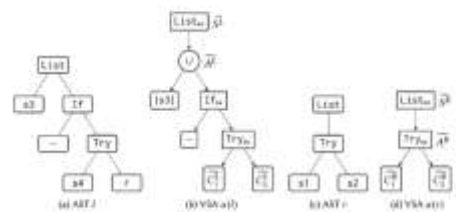


# Merging Techniques

| Merging Technique | Category | Example |
|---|---|---|
| | Unstructured | GitMerge[1] (Text-line based) MOST POPULAR |
| | Semi-structured | jFSTMerge[2] (Tree based) ***OOPSLA2017*** |
| | Structured | AutoMerge[3] (AST based) ***OOPSLA2018*** |






1. https://git-scm.com/docs/git-merge

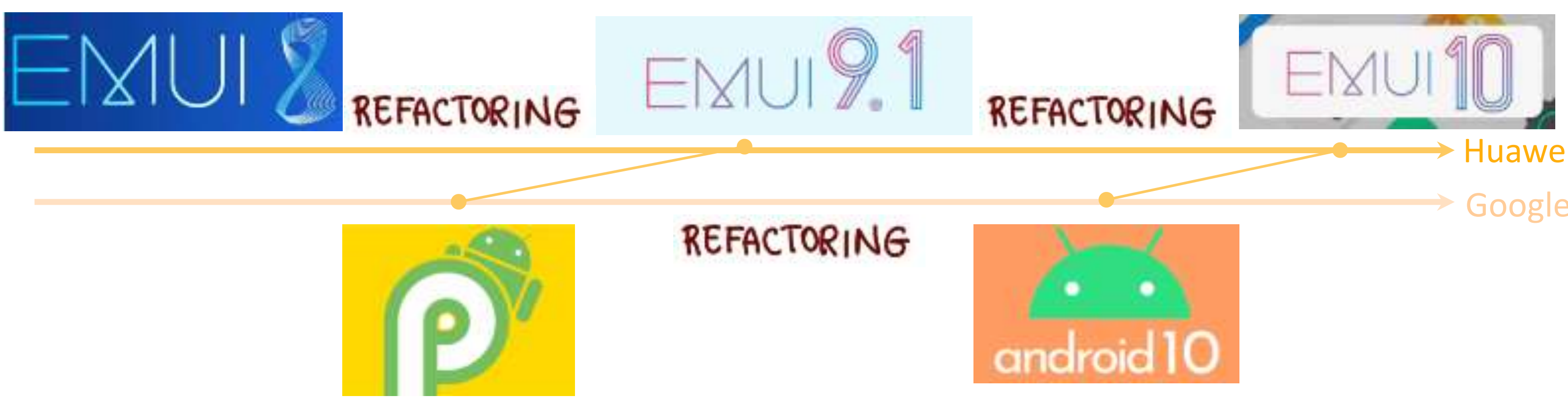
2. Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 59.

3. Fengmin Zhu and Fei He. 2018. Conflict resolution for structured merge via version space algebra. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 166.

# When *Merging* Meets *Refactoring* (1/2)

Refactoring: a transformation to the program (e.g., Rename/Move Field and Extract/Inline Method) that improves its internal design without changing its externally observable behavior [Fowler 2002].

*Refactorings become increasingly common, but they bring trouble to the existing merging approaches, especially to the most widely-used GitMerge.*



# When *Merging* Meets *Refactoring* (2/2)

According to a recent study[1] on about 3,000 Java projects from Github:
(1) **>22%** merge conflicts are related with refactorings;
(2) refactorings-involved conflicts are **more complex and difficult to resolve.**

## Challenges to correctly merge refactorings:

- **Matching: refactoring often leads to mismatching in existing merging approaches.**
- **Consistency: refactoring consists of changes across many places, which should be merged consistently.**
- **Comprehension: refactoring history is often unavailable when merging programs or resolving conflicts.**

1. Mehran Mahmoudi, Sarah Nadi, and Nikolaos Tsantalis. 2019. Are Refactorings to Blame? An Empirical Study of Refactorings in Merge Conflicts. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 151–162

# Refactoring-Aware Merging[1]

Motivation:

*Matching the changed code correctly* is the basis of a better merging algorithm.

Approach:

Match refactored code based on the graph representation of object-oriented programs.

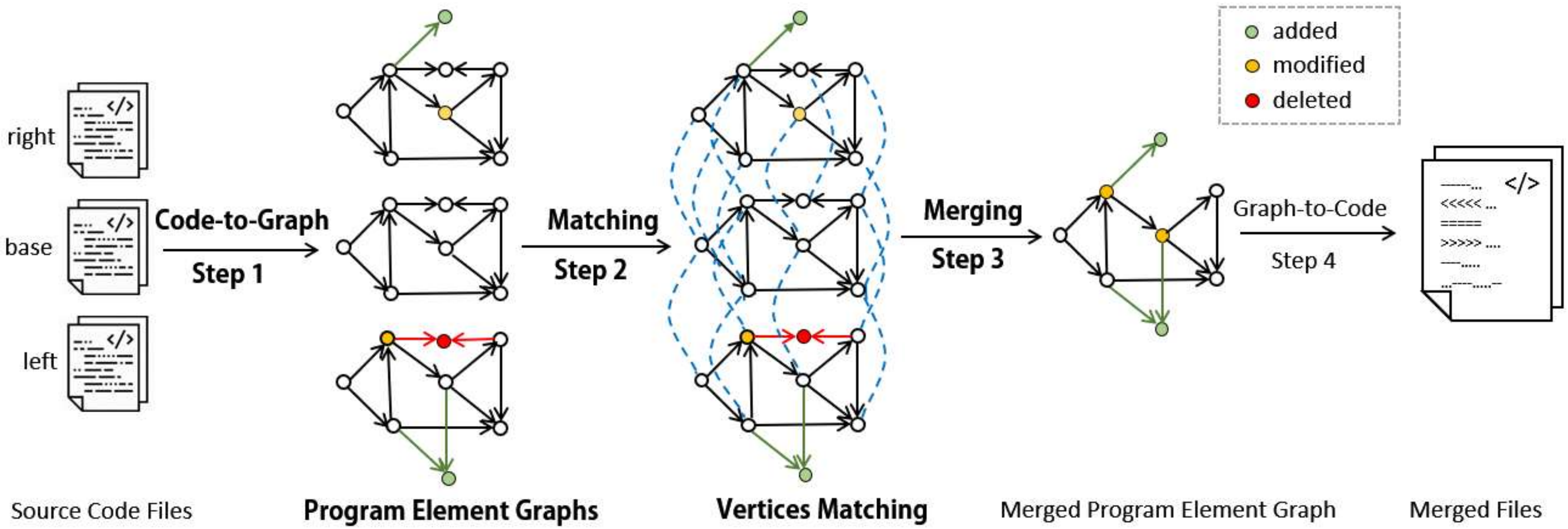
Target:

Better merging results, fewer but more reasonable conflicts.

1. Danny Dig, Tien N Nguyen, Kashif Manzoor, and Ralph Johnson. 2006b. MolhadoRef: a refactoring-aware software configuration management tool. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. ACM, 732–733

# Overview of IntelliMerge[1]

The **graph-based** and **refactoring-aware** semi-structured merging tool for Java.




1 https://github.com/Symbolk/IntelliMerge

# Experiments

We collect 1, 070 merge scenarios that contain refactoring-related conflicts, from the history of 10 popular and active Java open-source projects hosted on Github.

| Project | Stargazers | LOC | Merge Commits with Conflicts | Merge Commits with Refactoring-related Conflicts |
|---|---|---|---|---|
| cassandra | 5038 | 562K | 3923 | 587 (14.96%) |
| elasticsearch | 39635 | 1906K | 568 | 147 (25.88%) |
| antlr4 | 5400 | 92K | 345 | 88 (25.51%) |
| deeplearning4j | 10555 | 884K | 588 | 72 (12.24%) |
| gradle | 8652 | 66K | 710 | 65 (9.15%) |
| realm-java | 10359 | 141K | 579 | 56 (9.67%) |
| storm | 5618 | 398K | 258 | 21 (8.14%) |
| javaparser | 2346 | 215K | 78 | 18(23.08%) |
| junit4 | 7376 | 44K | 47 | 8 (17.02%) |
| error-prone | 4572 | 220K | 24 | 8 (33.33%) |

To evaluate different merging techniques on refactorings, we compare:

- IntelliMerge: the proposed graph-based semi-structured merging tool
- GitMerge: the most widely-used unstructured merging tool
- jFSTMerge: the state-of-the-art tree-based semi-structured merging tool

# Evaluation on Merged Part

Co

m

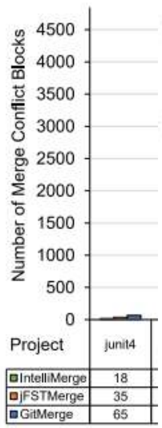
```
Project: realm-java
Commit Id: b6a78c64de381f6c5f111b4dc931dcf3eedc567d
Commit Message: Merge branch 'next-major' into
                merge-c357ac-to-next-major
File Path: realm/realm-library
           /src/objectServer/java/io/realm/SyncConfiguration.java
------------------------------------------------------------------
public SyncConfiguration.Builder readOnly() {
      this.readOnly = true;
      return this;
}


@Deprecated
      return this;
}


public SyncConfiguration.Builder fullSynchronization() {
      this.isPartial = false;
      return this;
}
```

```
Project: deeplearning4j
Commit Id: e34f03bd0c7c805789bdb9da427db7334e61cedc
Commit Message: Merge branch 'master' into
                mp_samediff_conv_consistencies
File Path: nd4j/nd4j-backends/nd4j-api-parent/nd4j-api
           /src/main/java/org/nd4j/autodiff/samediff/SameDiff.java
------------------------------------------------------------------
public SDVariable size(SDVariable in){
      return size(null, in);
}

public SDVariable size(String name, SDVariable in){
      SDVariable ret = f().size(in);
      return updateVariableNameAndReference(ret, name);
}
      return rank(null, in);
}

public SDVariable rank(String name, SDVariable in) {
      SDVariable ret = f().rank(in);
      return updateVariableNameAndReference(ret, name);
}
```

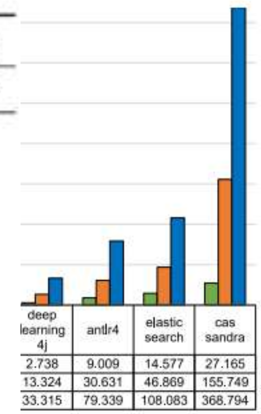
| | | |
|---|---|---|
| realm-java | 99.53% | 82.55% |
| storm | 99.61% | 73.75% |
| javaparser | 99.31% | 81.99% |
| junit4 | 99.24% | 86.81% |
| error-prone | 99.80% | 78.27% |
| Average | 99.46% | 81.28% |

```
Project: cassandra
Commit Id: 02d5ef3e765a34c738bc26796f2761e8cc7b715a
Commit Message: merge from 1.2
File Path: src/java/org/apache/cassandra/tracing/Tracing.java
------------------------------------------------------------------
public void run()
{
      CFMetaData cfMeta = CFMetaData.TraceSessionsCf;
      ColumnFamily cf = ArrayBackedSortedColumns.factory.create(cfMeta);
      addColumn(cf, buildName(cfMeta, bytes("duration")), elapsed);
<<<<<<< HEAD
      RowMutation mutation = new RowMutation(TRACE_KS, sessionIdBytes, cf);
      StorageProxy.mutate(Arrays.asList(mutation), ConsistencyLevel.ANY);
||||||| merged common ancestors
      RowMutation mutation = new RowMutation(TRACE_KS, sessionIdBytes);
      mutation.add(cf);
      StorageProxy.mutate(Arrays.asList(mutation), ConsistencyLevel.ANY);
=======
      mutateWithCatch(new RowMutation(TRACE_KS, sessionIdBytes, cf));
>>>>>>> cassandra-1.2
}
});

sessions.remove(state.sessionId);
this.state.set(null);
}
```

# Evaluation on Conflicting Part




| Project | Number of Conflict Blocks | | Lines of Conflicting Code | |
|---|---|---|---|---|
| | IntelliMerge | jFSTMerge | IntelliMerge | jFSTMerge |
| junit4 | 72.31% | 46.15% | 96.49% | 67.24% |
| error-prone | 62.97% | 29.63% | 94.03% | 47.08% |
| javaparser | 81.89% | 74.29% | 96.18% | 54.81% |
| storm | 65.35% | 59.41% | 92.82% | 82.17% |
| realm-java | 69.27% | 34.04% | 88.63% | 57.39% |
| gradle | 54.72% | 54.99% | 89.79% | 60.42% |
| deeplearning4j | 74.74% | 47.08% | 91.78% | 60.01% |
| antlr4 | 62.90% | 54.79% | 88.64% | 61.39% |
| elasticsearch | 53.73% | 45.24% | 86.51% | 56.64% |
| cassandra | 55.33% | 56.86% | 92.63% | 57.77% |
| Overall | **58.90%** | 53.38% | **90.98%** | 58.27% |




code

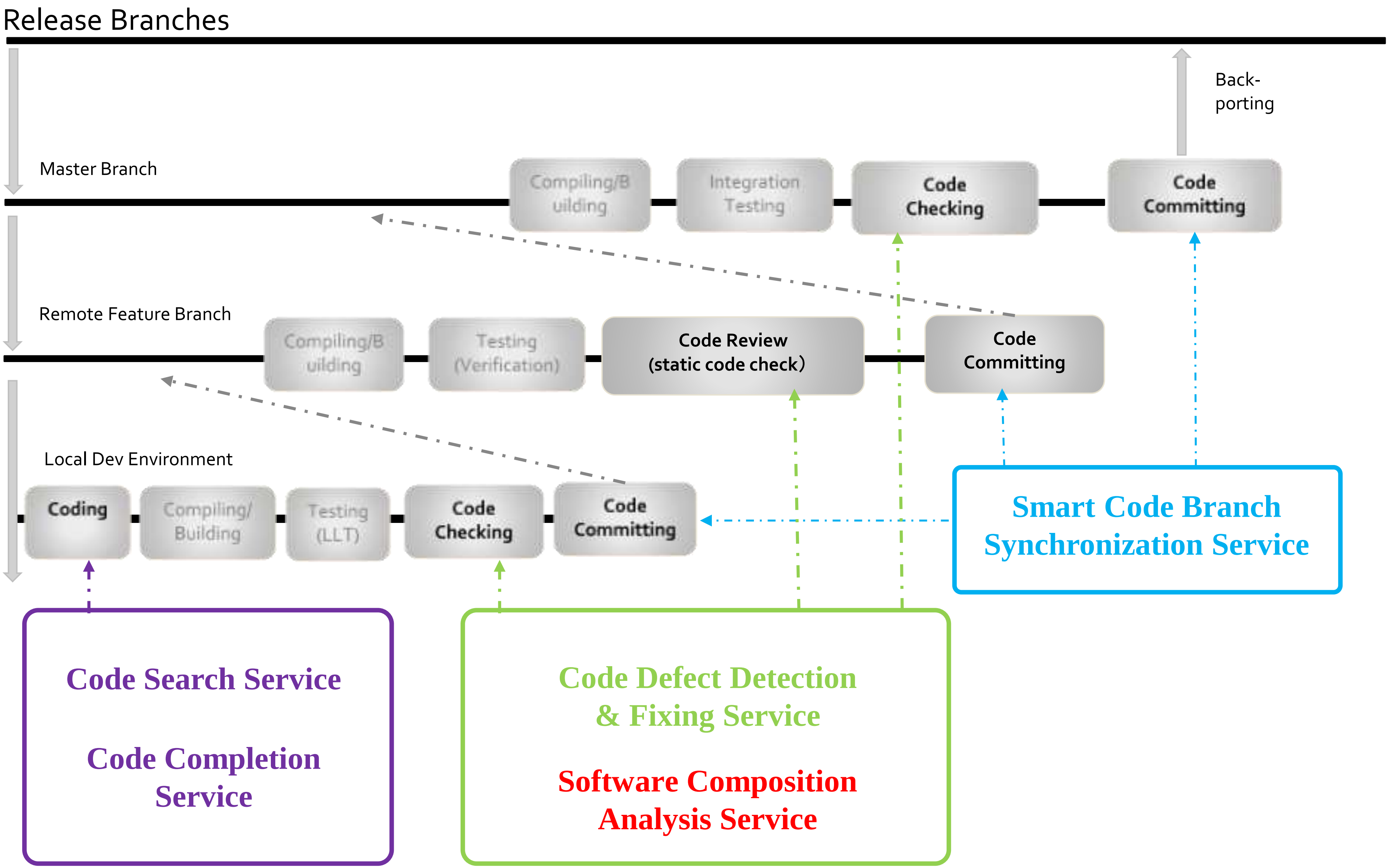
- Both *semi-structured* approaches **significantly reduce** conflicts comparing with unstructured GitMerge.
- Comparing with GitMerge, IntelliMerge reduces the number of conflict blocks by 58.90% and the lines of conflicting code by 90.98%.
- Comparing with jFSTMerge, IntelliMerge further reduces the number of merge conflicts by 11.84% and the lines of conflicting code by 78.38%.

# Conclusion and Future Work

- We propose an algorithm that merges the program in the form of graph to match and merge refactored code.
- We implement IntelliMerge, which is open-source: https://github.com/Symbolk/IntelliMerge

- What we are doing based on the PEG:
  - Exploiting relations and dependencies between conflict blocks to assist developers in manually resolving a series of related conflicts;
  - Automatically checking the syntactic consistency between merged program elements.

# CodeBot Overview



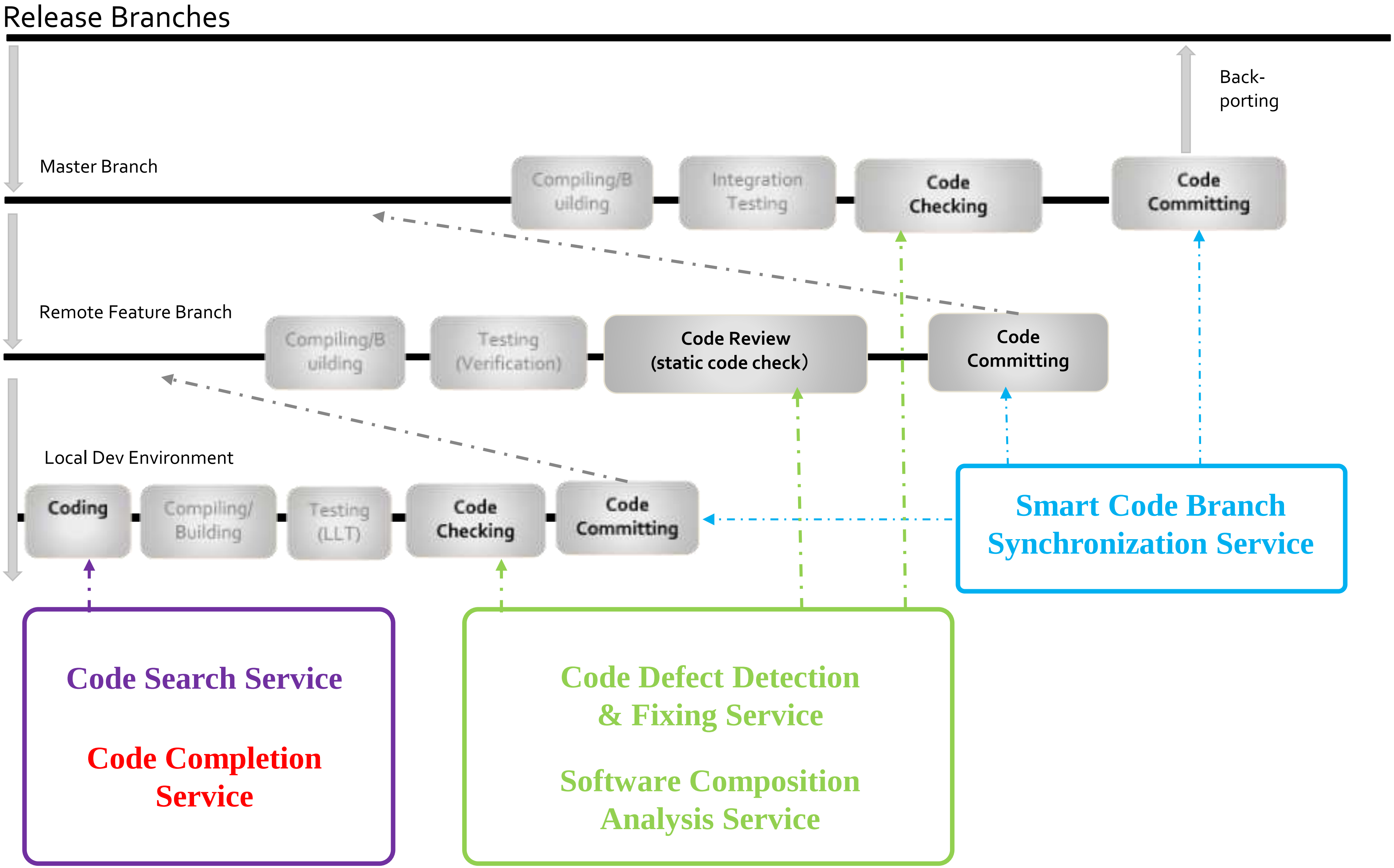
# Software Composition Analysis Service

Software Composition Analysis tool that scans your code for open source licenses and vulnerabilities, and gives you full transparency and control of your software products and services, avoiding the license related violations

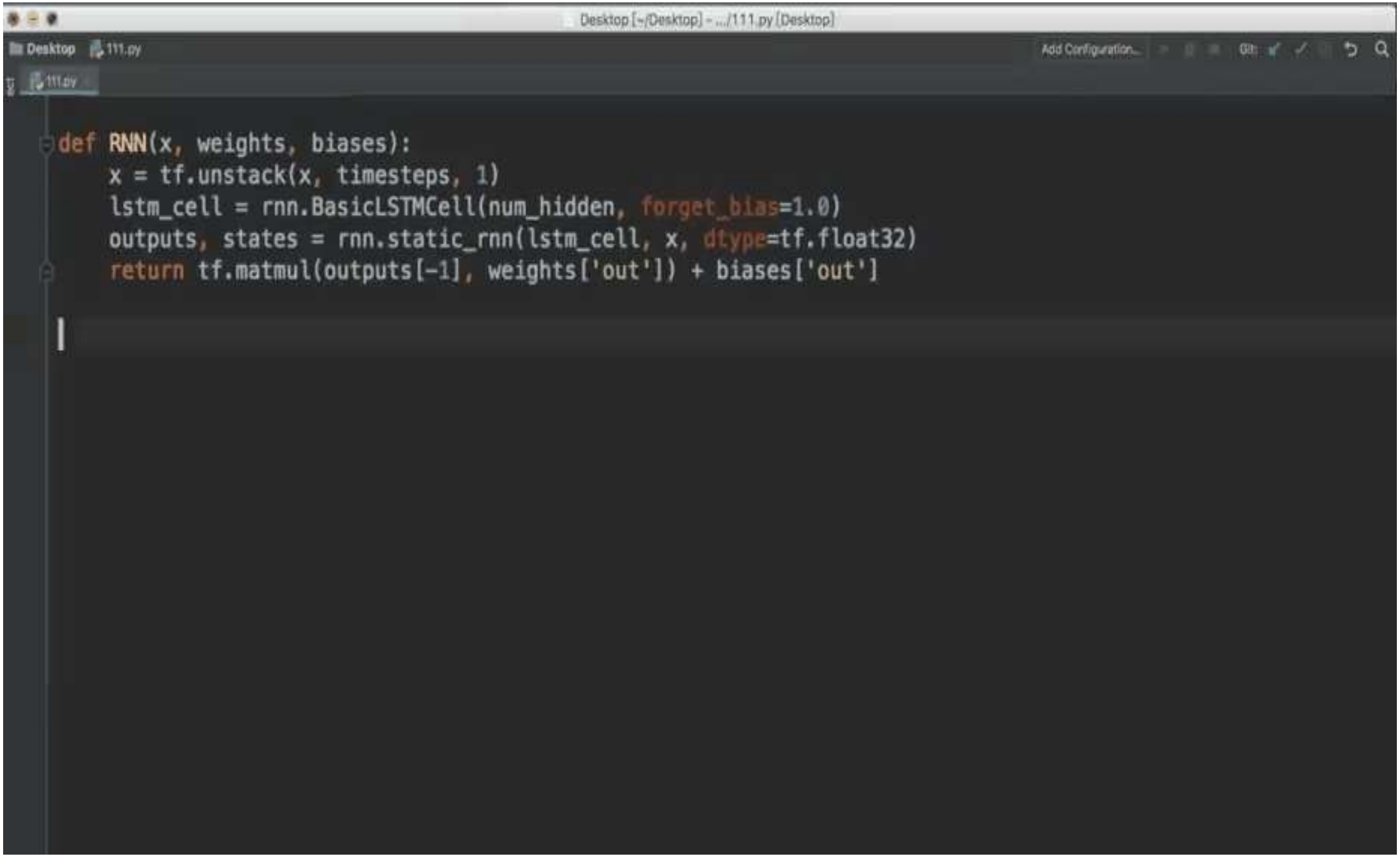

**Key Techs**

- Accurate Origins Analysis: Build the BIG knowledge base contains all open source repositories; Accurate and scalable code clone detection tech;
- Lightning Fast Scans: Apply revolutionary search engine techniques to enable the lightning fast scans (70 files/s)
- Precise Results: Apply AI, data-driven solutions to automatically eliminate false-positives.
- Ease of use: Users can easily scan, audit, generate a variety of reports; support CI integration; flexible deployment

# CodeBot Overview

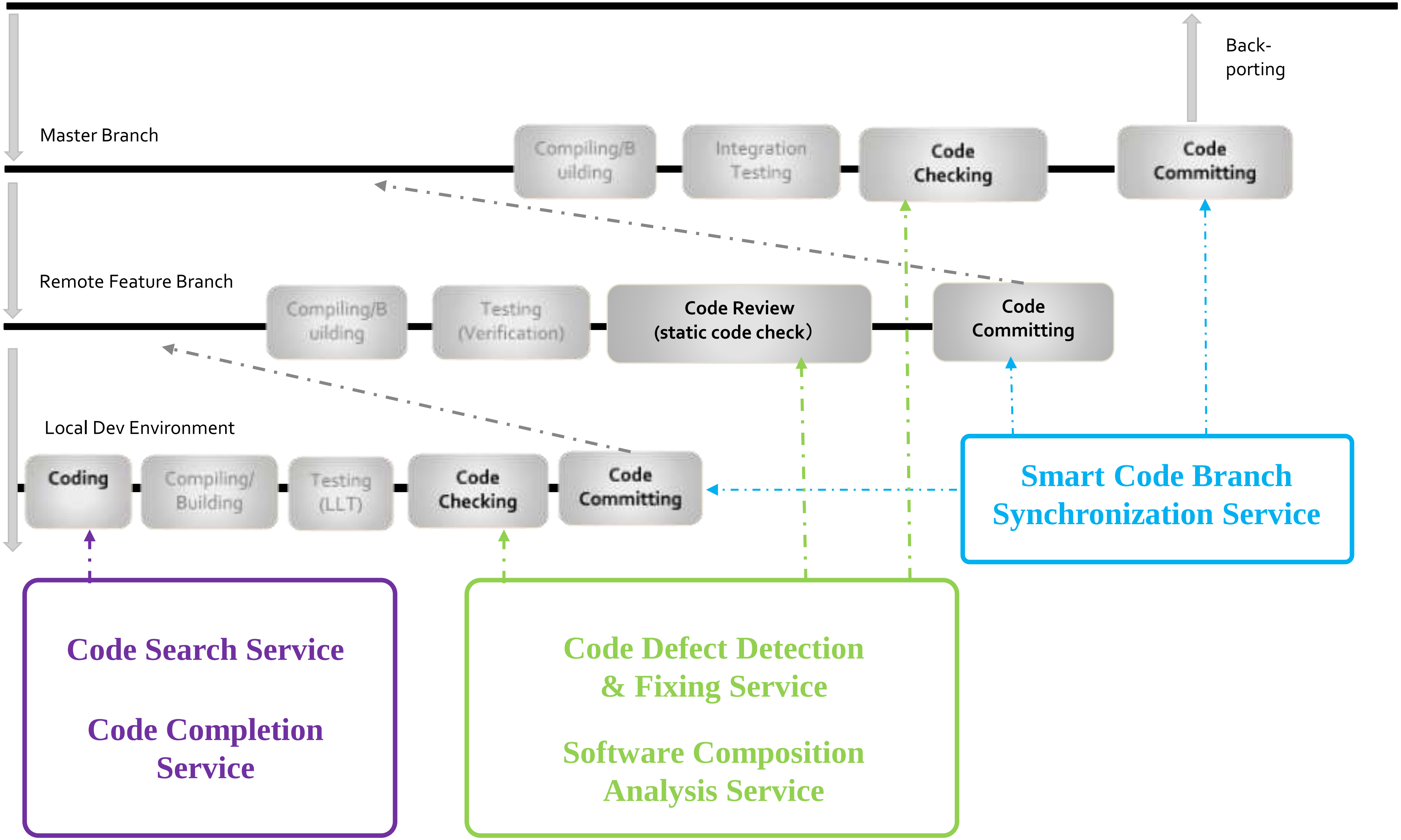


# Smart Code Completion Service



# Questions?

## Release Branches



# Backups

# Program Element Graph (PEG)

[Definition] Program Element Graph: a *labeled*, *weighted*, and *directed* graph $G = (V, E)$ *that* encodes the program structure and data&control flow above the field/method level.

**Vertex Set V**: program elements (e.g., class/method/field declaration), consists of *terminal* and *non-terminal* vertices.

**Edge Set E:** relation and interaction between program elements (e.g., extend, method invocation, field access)

The implementation of PEG is language-specific, in ours for Java 8:

- Supported program elements:

Project, Package, CompilationUnit, Class, Enum, Annotation, Interface, Field, Constructor, Method, EnumConstant, AnnotationMember, InitializerBlock, etc.

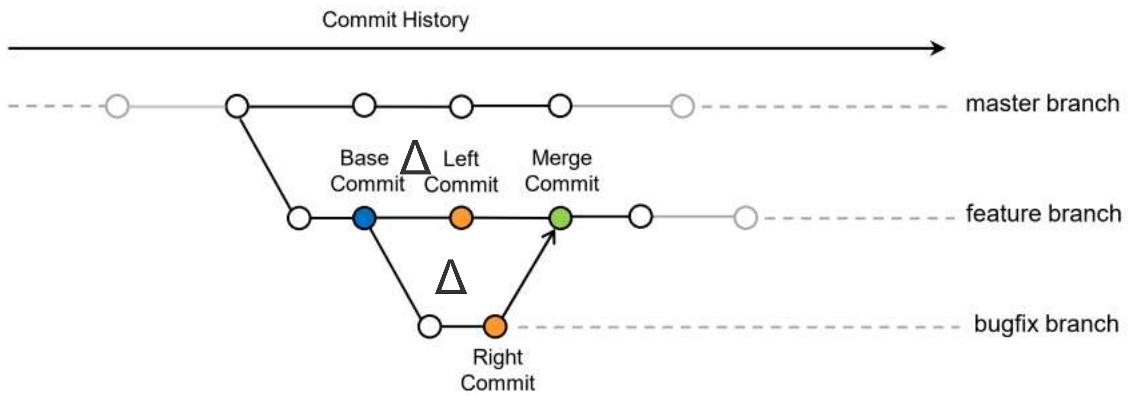
- Supported relation types:

contain, import, extend, implement, define, declare, read, write, call, instantiate, etc.

# Code to Graph

**Input**: the *left* and *right* commit (*HEAD* commits of two branches to be merged)
**Output**: the PEGs for the *left/right/base version*, respectively

1. Find the base: use the nearest common ancestor (NCA) commit as the *base* version;




2. Collect files to analyze: compare the *left/right* version with the *base version* to find *diff* files and *imported* files;
3. Parse the code: parse the code in each source file sets into abstract syntax trees (ASTs);
4. Form the vertices: extract program elements from AST to form vertices;
5. Build the edges: extract hierarchical relations and interactions by analyzing the statements inside bodies of terminal verticess.

# Code to Graph (2)

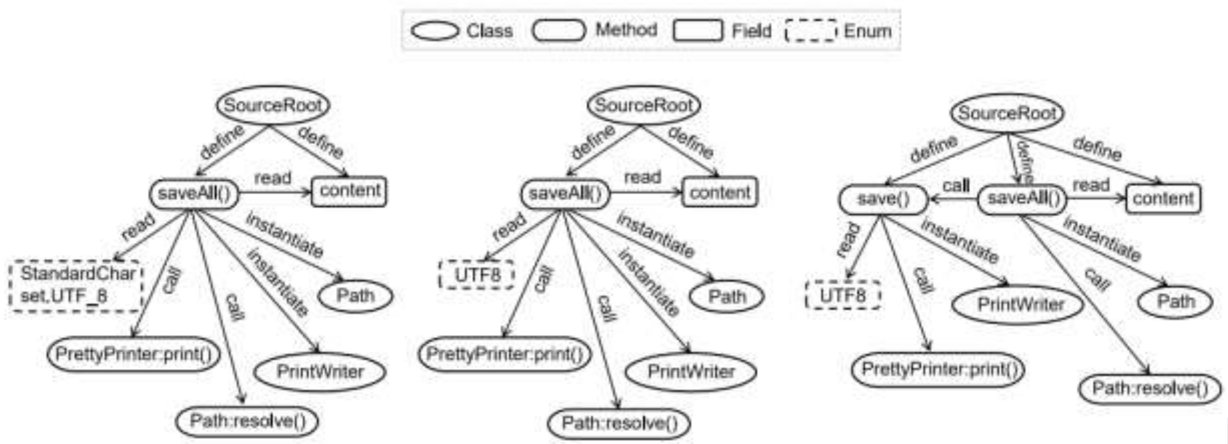
The necessary information are captured for matching:

**Vertex Attributes:**

- type (v) = the type of v, same as the type of the corresponding AST node
- signature (v) = the fully-qualified name of v, e.g. edu.pku.intellimerge.util.SourceRoot
- source (v) = the body of terminal vertices or the original declaration of non-terminal vertices, which will be merged textually

**Edge Attributes:**

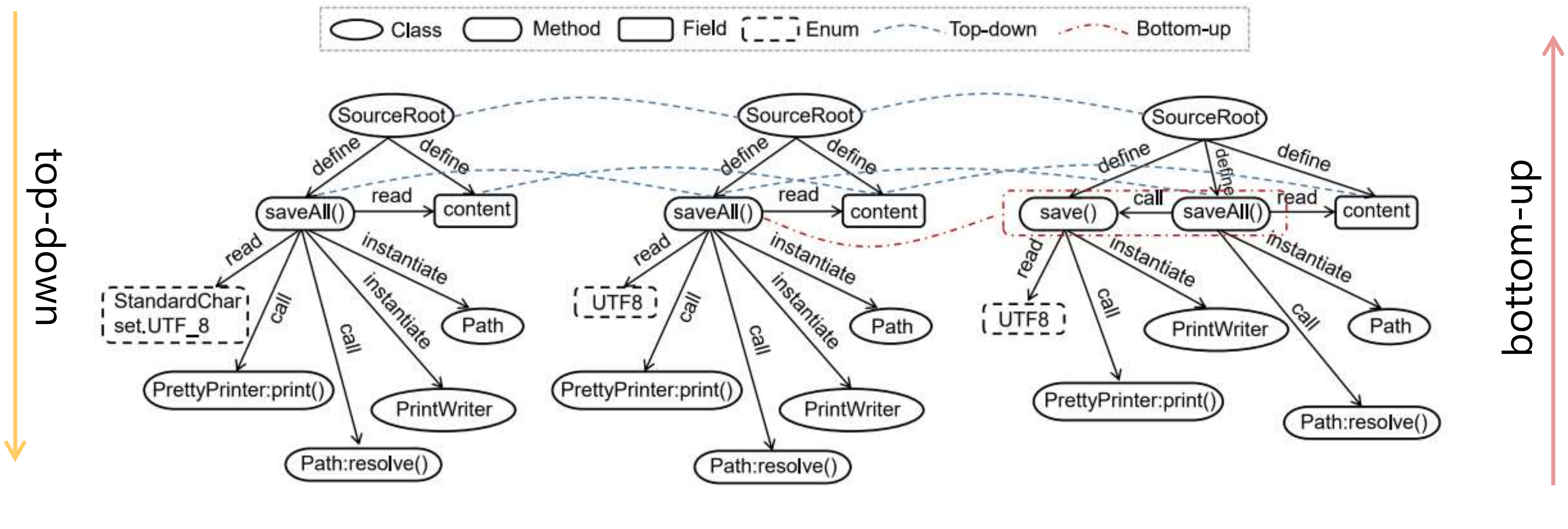
- type (e) = the relation type that e represents
- weight (e) = the times that one type of relation appears between two vertices




(a) *Left PEG* (b) *Base PEG* (c) *Right PEG*

# Matching

***Target:***
to match program elements before and after refactoring (and other) changes




(a) *Left PEG* (b) *Base PEG* (c) *Right PEG*

***Basic insight:*** *A large part of the code between base version and left/right version remain unchanged in most cases.*

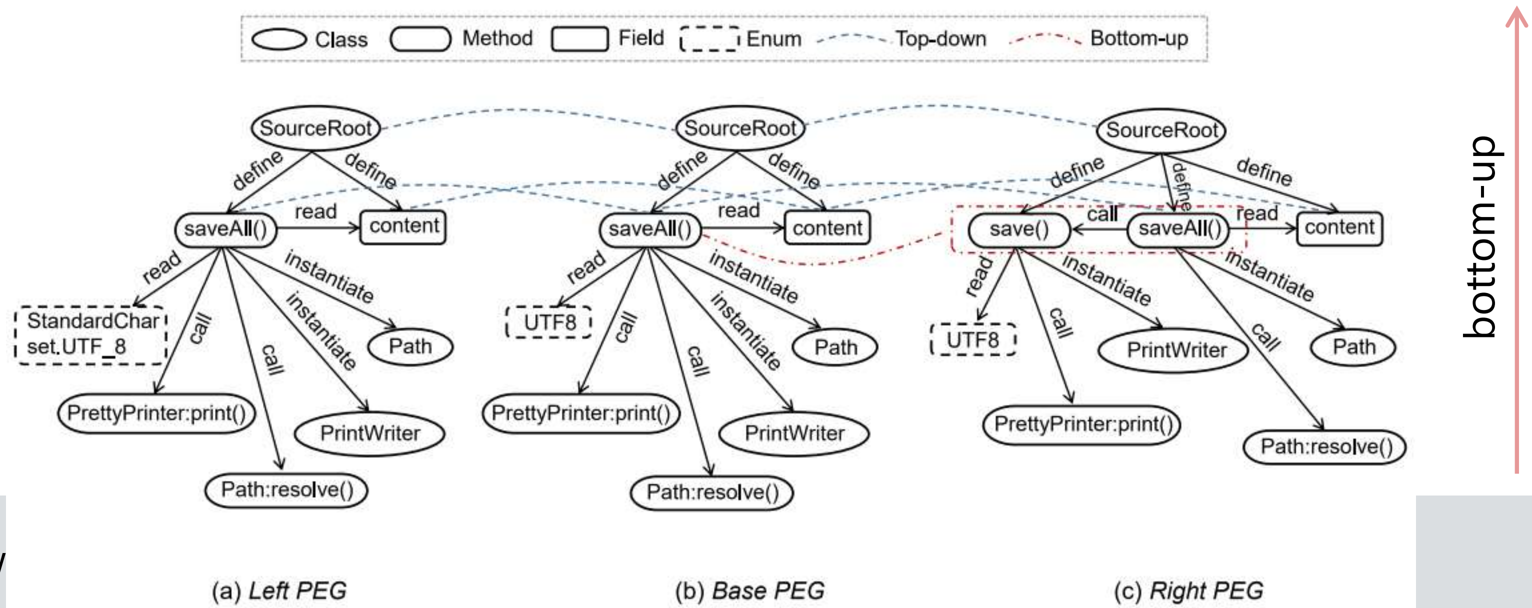
**Top-down:** Following the hierarchical order, match vertices by hashed vertex signature.
**Bottom-up:** From terminal vertices to non-terminal vertices, match vertices according to the *matching degree*

# Matching (2)

***Basic assumption**: Matched program elements must have the same type, and do the similar things in the program.*

Matching-degree estimates the similarity of two vertices:

- For terminal vertices: *weighted_average(signature similarity, body tree similarity, context edges similarity)*
- For non-terminal vertices: *weighted_average(signature similarity, children list similarity, context edges similarity)*



# Matching (3)

***Basic assumption**: Matched program elements must have the same type, and do the similar things in the program.*

Instead of explicitly detecting each type of refactorings, we categorize them into two categories according to their effect:

| Matching Kind | Vertex Type | Refactoring Type | Matching Rule |
|---|---|---|---|
| 1-to-1 | *fld* | Rename, Move<br>Pull Up, Push Down | $\exists(fld_1, fld_2) \mid contextSimilarity(fld_1, fld_2) + nameSimilarity(fld_1, fld_2) > \eta$ |
| | *mtd* | Rename, Move<br>Pull Up, Push Down | $\exists(mtd_1, mtd_2) \mid contextSimilarity(mtd_1, mtd_2) + bodySimilarity(mtd_1, mtd_2) > \eta$ |
| | *cls* | Rename, Move | $\exists(cls_1, cls_2) \mid contextSimilarity(cls_1, cls_2) > \eta$ |
| | *pkg* | Rename | $\exists(pkg_1, pkg_2) \mid contextSimilarity(pkg_1, pkg_2) > \eta$ |
| m-to-n | *mtd* | Extract | $\exists(mtd_1,\{mtd_2,mtd_u\}) \mid \exists(mtd_1, mtd_2) \wedge \text{contextSimilarity}(mtd_1, (mtd_2 + mtd_u)) > \text{contextSimilarity}(mtd_1, mtd_2) > \eta$ |
| | | Inline | $\exists(\{mtd_1,mtd_u\},mtd_2) \mid \exists(mtd_1,mtd_2) \wedge \text{contextSimilarity}((mtd_1 + mtd_u),mtd_2) > \text{contextSimilarity}(mtd_1, mtd_2) > \eta$ |

Divide and conquer for each type of vertices:

1. For 1-to-1 matching: match vertices with biparitie maximum matching;
2. For m-to-n matching: match vertices by joining/splitting the context of multiple vertices.

# Merging

**Input**: the matched vertices triple: <left vertex, base vertex, right vertex> (each of them can be optional but not all of them).
e.g.:

- Added: <a, NULL, NULL>
- Deleted: <NULL, b, b>
- Modified: <d, c, c>

**Output**: the merged code files with possible conflict blocks embedded

1. Locate all vertices of type *cut* (CompilationUnit, which corresponds to the source code file);
2. Traverse hierarchical relation edges (e.g. define/contain) with the *cut* vertex as the source vertex, merge target vertices *recursively;*
3. Merge vertex *components* following the basic rules of three-way merging:
   - <a, NULL, NULL> → a, <a, NULL, a> → a
   - <NULL, b, b> → NULL, <NULL, b, c> → conflict!
   - <d, c, c> → d, <d, c, e> → conflict!