FRAPPÉ
Querying and managing evolving code dependency graphs

David Meibusch & Nathan Hawes
Oracle Labs Australia
June 2016
Safe Harbour

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract.

It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle.

Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
The Truth is in the Source!

But the source is often complicated, multi-language and really really big
Frappé: code as a property graph

• Graph natural in this domain
  – Call graph, directory hierarchy, type hierarchy, data/control flow graphs

• Overlay data from different spaces
  – File system, build, preprocessor, AST, cross-language

• Lets users specify queries in terms of this graph
How it works

Source Code → .fo → Server

EDITOR PLUGINS
CLI SCRIPTS
WEB UI

User
Graph model example

```c
#include "foo.h"
int bar(int *input) {
    return *input * 2;
}

#include "foo.h"
int main(int argc, char **argv) {
    return bar(&argc);
}
```

```bash
gcc foo.c -c -o foo.o
gcc main.c foo.o -o prog
```

```
compiled_from

main.c
file

foo.module
module

main.function

foo.h

includes

main.function_decl

bar.function

calls

includes

int

primitive

is_type

argc

parameter

is_type

char

primitive

argv

parameter

is_type

int

primitive

input

parameter

has_param

bar

function

file

contains

foo.c

source_file

includes

foo.h

source_file

compiled_from

compiled_from

linked_from

build
```
How it works

Source Code → EXTRACTOR → IMPORTER → IR → Server

Server → EDITOR PLUGINS → CLI SCRIPTS → WEB UI → User
Use cases

• Code search
• Code navigation
  – Go to definition
  – Find references
• Code comprehension
  – Visualization
  – Transitive closure calls, includes, etc.
  – Shortest path queries
How it looks

```c
/* Return the working directory for the current job working directory, this does not call any of the functions it calls. This is so that the current working directory is not changed from a signal handler. */

static char *
current_working_directory ()
{
    char *dirl;
    static char d[PATH_MAX];
    dirl = get_string_value("PWD");
    if (dirl == 0 &amp; the_current_working_directory)
        dirl = the_current_working_directory();
    if (dirl == 0)
    {
        dirl = getcwd (d, sizeof(d));
        if (dirl)
            dirl = d;
    }
}

int
rl_rubout (count, key)
{
    int count, key;
    if (count < 0)
        return (rl_delete (-count, key));
    if (rl_point)
    {
        rl_ding ();
        return -1;
    }
    if (rl_insert_node == RL_INSERT_OVERWRITE)
        return (rl_overwrite_rubout (count, key))
    return (rl_rubout_char (count, key));
}
Use cases

• Code search

• Code navigation
  – Go to definition
  – Find references

• Code comprehension
  – Visualization
  – Transitive closure calls, includes, etc.
  – Reachability queries

Queries and Neo4j performance detailed in GRADES’15 paper:

“Frappé: Querying the Linux Kernel dependency graph”
Code search

foo.c
Code search

query: foo.c

(n with name='foo.c')
Code search

foo.c

(struct with name='foo.c')

```c
struct foo {
    int c;
}
```
(c with name='foo')
 -[:contains]->
 (n with name='c')
   UNION
 (n with name='foo.c')

struct foo {
  int c;
}

foo.c
(c with name='foo')
-[:contains]->
(n with name='c')

UNION

(n with name='foo.c')

struct bar {
    int c;
}

typedef struct bar foo;
(c with name ‘foo’)
-[:aliases*]->()[:contains]->
  (n with name=‘c’)
  UNION
  (n with name=‘foo.c’)

struct bar {
  int c;
}

typedef struct bar foo;
(c with name ‘foo’)
-[:aliases*]->()[:contains]->
    (n with name='c')
    UNION
    (n with name='foo.c')

struct bar {
    int c;
}

typedef struct bar foo;
Use cases

• Code search

• Code navigation
  – Go to definition
  – Find references

• Code comprehension
  – Visualization
  – Transitive closure calls, includes, etc.
  – Reachability queries

Queries and Neo4j performance detailed in GRADES’15 paper:

“Frappé: Querying the Linux Kernel dependency graph”
Use cases

• Code search
• Code navigation
  – Go to definition
  – Find references
• Code comprehension
  – Visualization
  – Transitive closure calls, includes, etc.
  – Reachability queries

• Custom user queries
  – (Anti) pattern detection
  – *Expose query language*

Queries and Neo4j performance detailed in GRADES’15 paper:

“Frappé: Querying the Linux Kernel dependency graph”
Dependency cycle
Dependency cycle

file_contains

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references

source_file

file_contains

references
Dependency cycle

() - [includes | uses_namespace | expands | interrogates | undefined | aliases | uses_enum | has_friend | isa_type | extends | uses_type | throws | has_ret_type | has_param_type | calls | may_call | overridden_by | declares | reads | writes | dereferences | address_of | type_of | size_of | align_of | casts_to ] -> ()
BUT

Developers working off of different versions of the code
Target Scenario

- 1000s of developers working from main branch
- Changes merged regularly
- Most developers working off of versions from the past 30 days
Current Deployment

• Separate Neo4j instance for the most recent 5 versions of the code
• New graph generated in nightly regression
• Script on the client to determine which server to connect to

• Deployment effort and complexity
• Inefficient use of resources
  – Redundant data
  – Memory requirements
Current Deployment

• Separate Neo4j instance for the most recent 5 versions of the code
• New graph generated in nightly regression
• Script on the client to determine which server to connect to
• Deployment effort and complexity
• Inefficient use of resources
  – Redundant data
  – Memory requirements

Single logical server with multiple versions and efficient storage
Use cases for multiple versions

• Single version use cases per version

• Code review: 2 versions
  – Are there any architectural constraints being newly violated?
  – Are there any new usages of deprecated methods?
  – Are any methods now unused?
Version selection

PATH vcalls := () -[:call WITH 1013 between fromV and toV]-> ()
SELECT path
FROM freebsd
WHERE path =
    (:function WITH name='source', 1013 between fromV and toV)
    -/:vcalls*/->
    (:function WITH name='sink', 1013 between fromV and toV)

SELECT path
FROM freebsd@1013
WHERE path =
    (:function WITH name='source') -/:calls*/-> (:function WITH name='sink')
Use cases for multiple versions

• Single version use cases per version

• Code review: 2 versions
  – Are there any architectural constraints being newly violated?
  – Are there any new usages of deprecated methods?
  – Are any methods now unused?

Match in each version
+ Compare results
Compare results

```
SELECT path
FROM freebsd@1013
WHERE path =
  (:function WITH name='source') -/:calls*/-> (:function WITH name='sink')

Ddifference

SELECT path
FROM freebsd@1014
WHERE path =
  (:function WITH name='source') -/:calls*/-> (:function WITH name='sink')
```

```
SELECT difference path
FROM freebsd@1013, freebsd@1014
WHERE path =
  (:function WITH name='source') -/:calls*/-> (:function WITH name='sink')
```
Open Questions

• Node/Edge identity
  – Supplied or derived using graph information

• Query language expressiveness
  – Regular path expressions
  – Multiple edge labels

• Efficient storage and querying for multiple graph versions:
  – List or ideally DAG of versions
  – Union, intersection, difference of results from different versions
  – Query language that abstracts away versioning
FreeBSD dataset available on OTN

• Includes graphs of 10.1, 10.2, 10.3 kernel + documentation

• Each graph
  – Extracted from 10M LOC
  – 2 million vertices
  – 10 million edges

• Try it out and get in touch:
  – nathan.hawes@oracle.com
david.meibusch@oracle.com
  – ben.barham@oracle.com

Graph model example

```c
#include "foo.h"
int bar(int *input) {
    return *input * 2;
}
```

```c
#include "foo.h"
int main(int argc, char **argv) {
    return bar(&argc);
}
```
Graph model example

```
#include "foo.h"
int bar(int *input) {
    return *input * 2;
}

#include "foo.h"
int bar(int *input) {
    return *input * 2;
}

#include "foo.h"
int bar(int A) {
    return A;
}

int main(int argc, char **argv) {
    return bar(&argc);
}
```

Frappé: querying the Linux kernel dependency graph
Graph model example

```c
#include "foo.h"
int bar(int *input) {
    return *input * 2;
}

#define BAR(A) bar(A)
int bar(int);

main.c file

main.c
int main(int argc, char **argv) {
    return BAR(&argc);
}

main function

calls
name_file_id
name_start_line
name_start_column
use_file_id
use_start_line
use_end_line

file_contains
name_file_id
name_start_line
name_start_column
use_file_id
use_start_line
use_end_line

bar function

foo.h

#define BAR(A) bar(A)
int bar(int);

foo.c

int bar(int *input) {
    return *input * 2;
}

main function

file_contains
name_file_id
name_start_line
name_start_column
use_file_id
use_start_line
use_end_line

bar function

foo.h

int bar(int);

#include "foo.h"
int bar(int *input) {
    return *input * 2;
}

main function

file_contains
name_file_id
name_start_line
name_start_column
use_file_id
use_start_line
use_end_line

bar function

foo.h

#include "foo.h"
int bar(int);

#include "foo.h"
int bar(int *

```