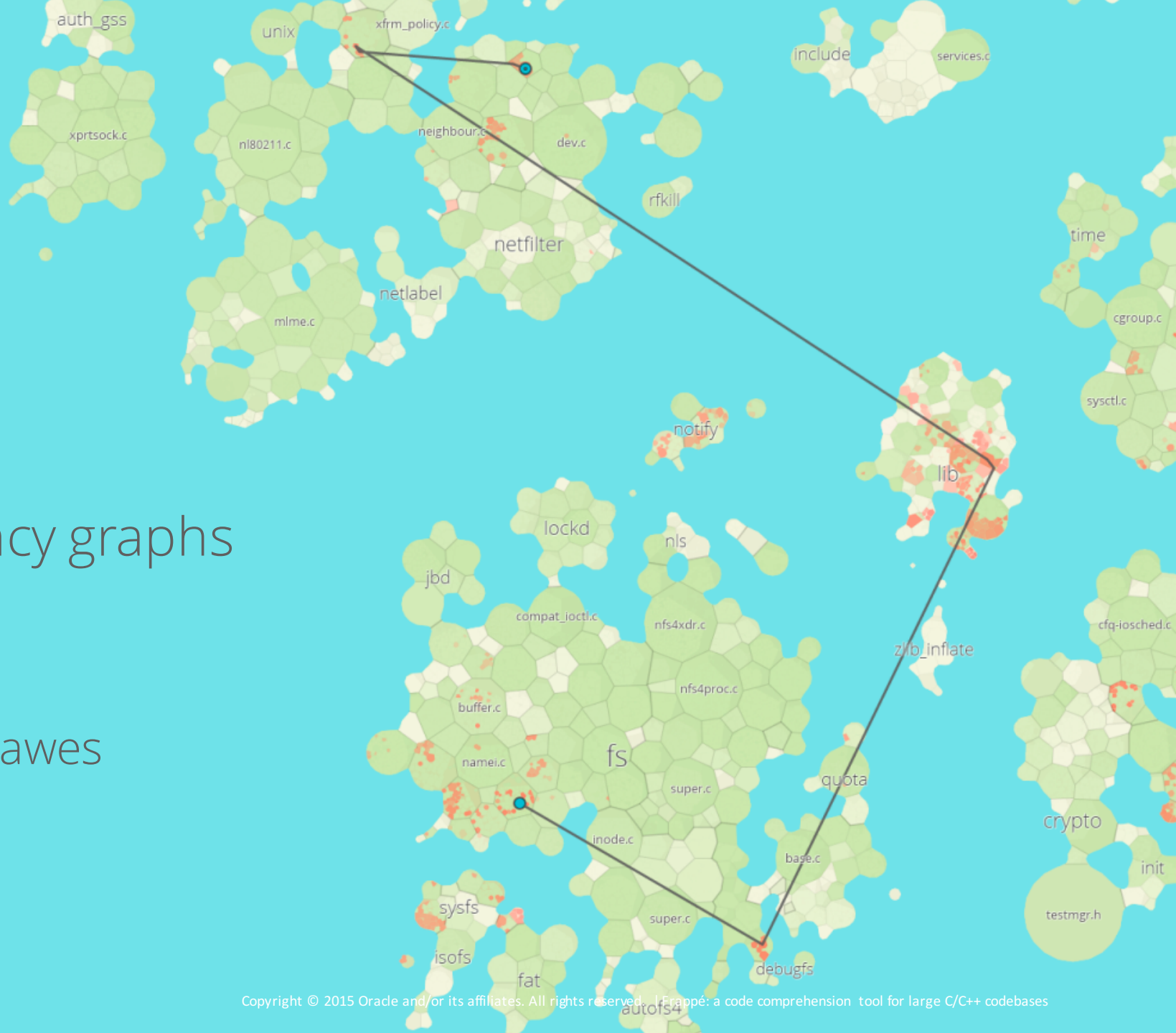



# FRAPPÉ

## Querying and managing evolving code dependency graphs

David Meibusch & Nathan Hawes

Oracle Labs Australia

June 2016

ORACLE

# 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**

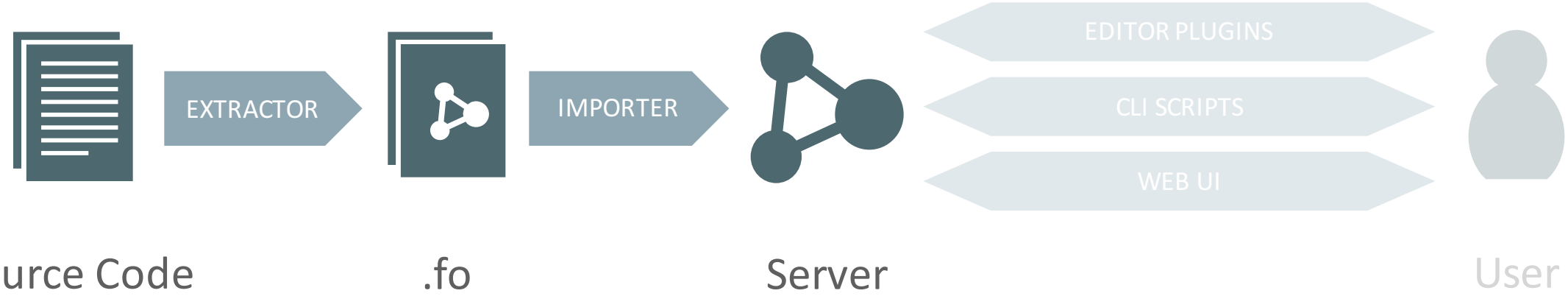
 

# Frappé: code as a property graph

- 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



# Graph model example

foo.h

```
int bar(int);
```

foo.c

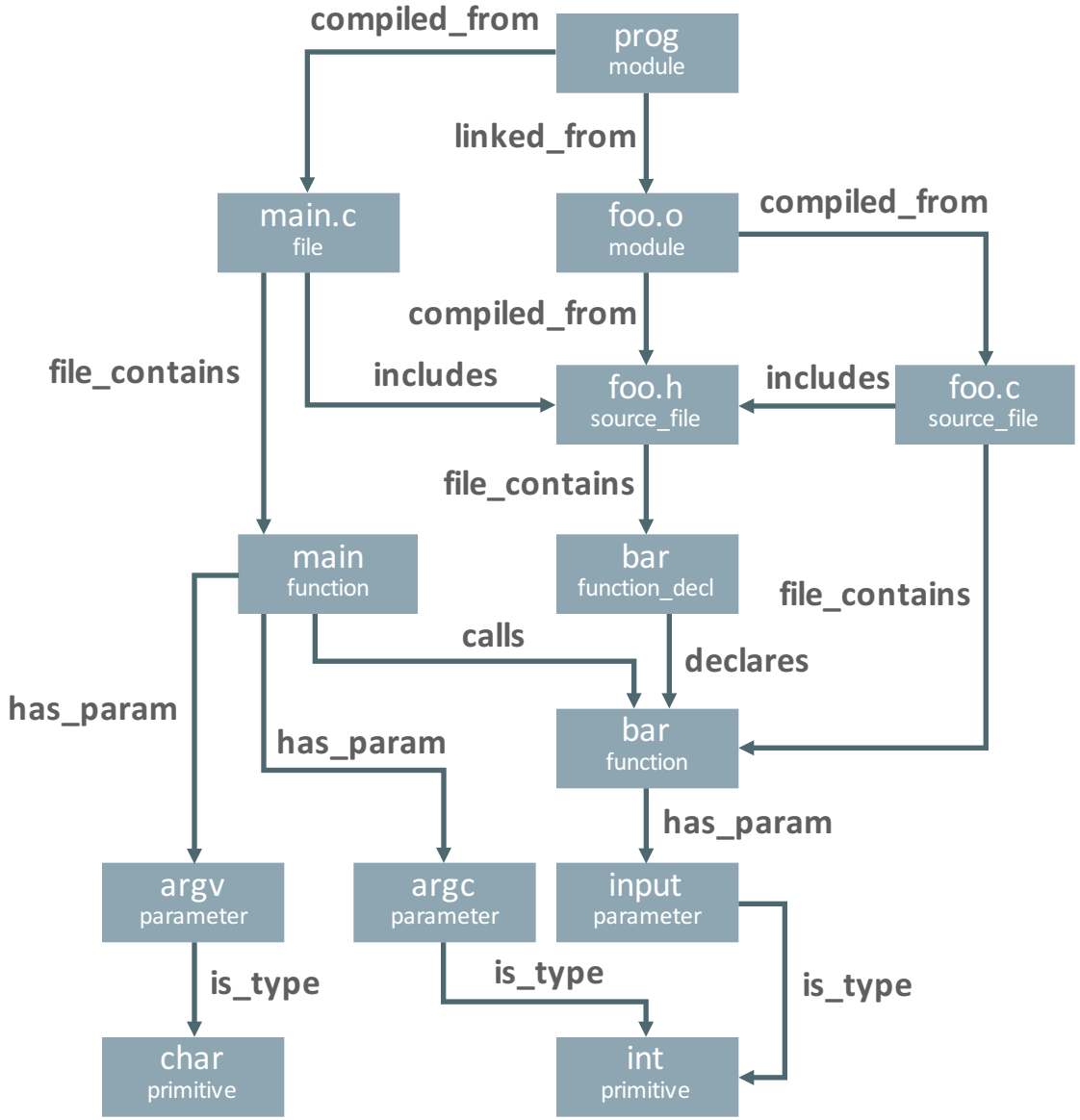
```
#include “foo.h”
int bar(int *input) {
  return *input * 2;
}
```

main.c

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

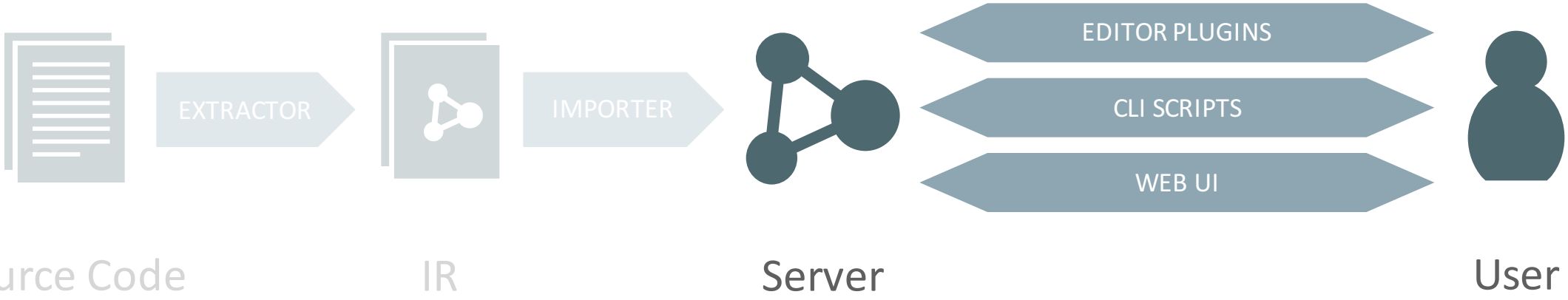
build

```
gcc foo.c –c –o foo.o
gcc main.c foo.o –o prog
```



# How it works



# Use cases

- Code search
- Code navigation
  - Go to definition
  - Find references

- Code comprehension
  - Visualization
  - Transitive closure calls, includes, etc.
  - Shortest path queries

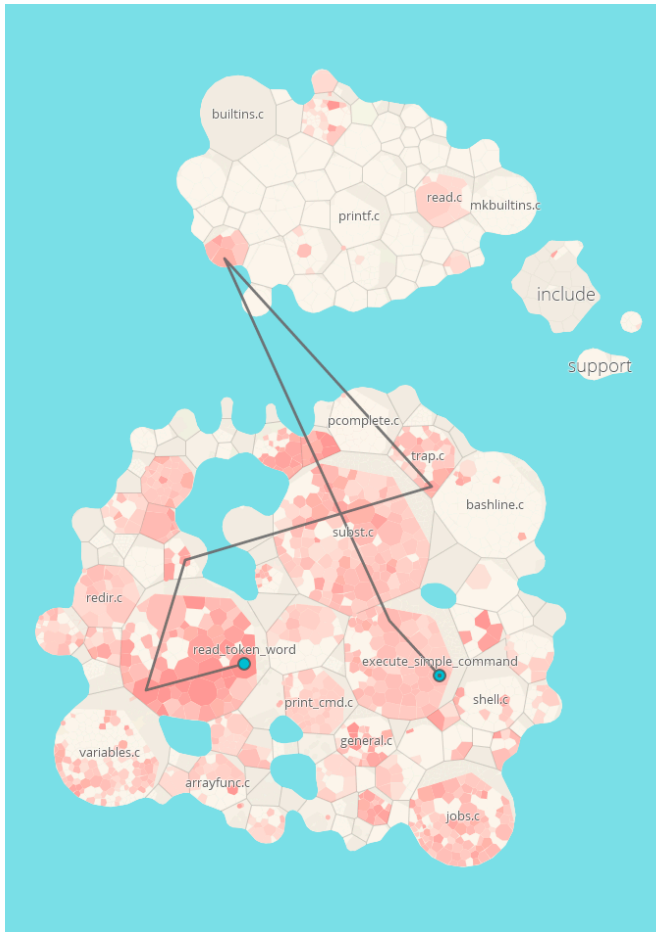
 

# How it looks

```
357
358 /* Return the working directory for the curren
359    job_working_directory, this does not call r
360    of the functions it calls.  This is so that
361    from a signal handler. */
362 static char *
363 current_working_directory ()
364 {
365   char *dir;
366   static char d[PATH_MAX];
367
368   dir = get_string_value ("PWD");
369
370   if (dir == 0 && the_current_working_director
371     dir = the_current_working_directory;
372
373   if (dir == 0)
374     {
375       dir = getcwd (d, sizeof(d));
376       if (dir)
377         dir = d;
378     }
jobs.c
References to "current_working_directory" (node 1
declared by Function… current_working…
called by   Function  print_pipeline   at jobs.c
called by   Function  start_job        at jobs.c
called by   Function  notify_of_job_s… at jobs.c
called by   Function  notify_of_job_s… at jobs.c
~
~
~
```

F **rl_rubout**
lib/readline/text.c

```
1058 /* Rubout the character behind point. */
1059 int
1060 rl_rubout (count, key)
1061      int count, key;
1062 {
1063   if (count < 0)
1064     return (rl_delete (-count, key));
1065
1066   if (!rl_point)
1067     {
1068       rl_ding ();
1069       return -1;
1070     }
1071
1072   if (rl_insert_mode == RL_IM_OVERWRITE)
1073     return (_rl_overwrite_rubout (count, key));
1074
1075   return (_rl_rubout_char (count, key));
1076 }
```



# 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

# Code search


```
(n with name='foo.c')
```



# Code search


```
(n with name='foo.c')
```

```
struct foo {
    int c;
}
```

# Code search

foo.c

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

```
struct foo {
    int c;
}
```



# Code search

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;
```

# Code search

foo.c

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

```
struct bar {
    int c;
}

typedef struct bar foo;
```

# Code search

foo.c

```
(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

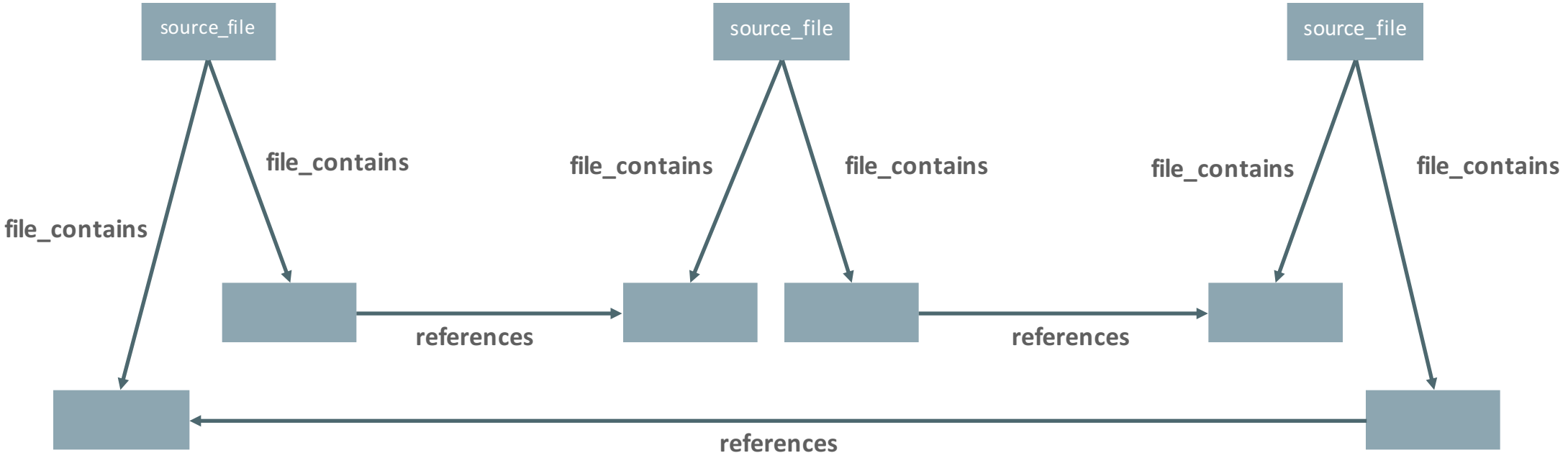
- Code search
- Code navigation
  - Go to definition
  - Find references
- Code comprehension
  - Visualization
  - Transitive closure calls, includes, etc.
  - Reachability queries

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

Queries and Neo4j performance detailed in GRADES'15 paper:

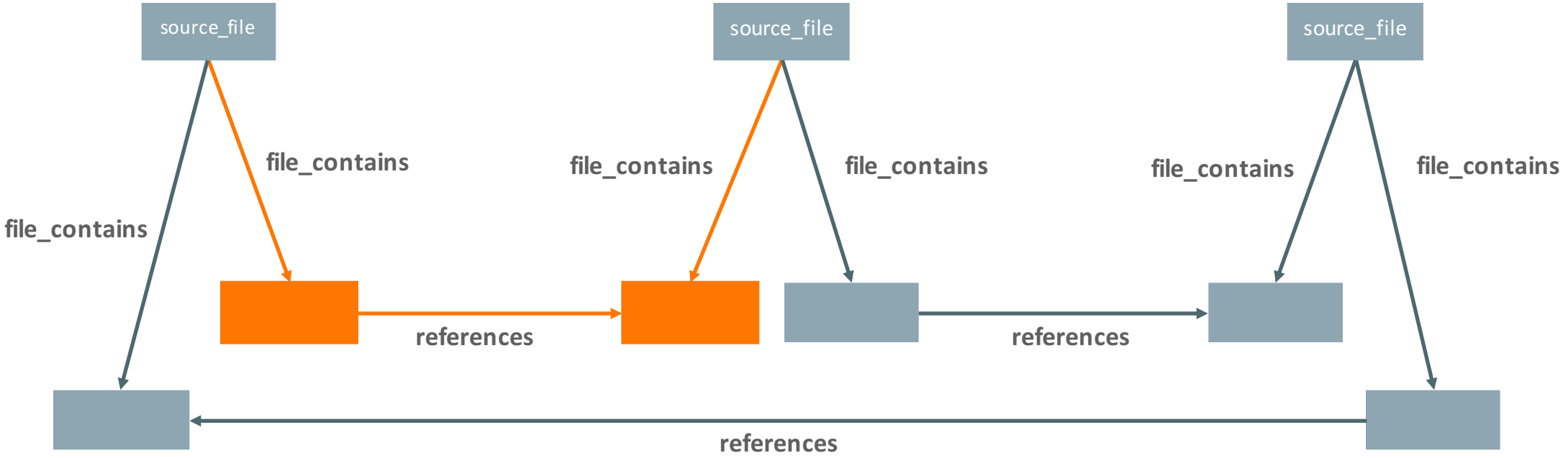
"Frappé: Querying the Linux Kernel dependency graph"

# Dependency cycle



# Dependency cycle



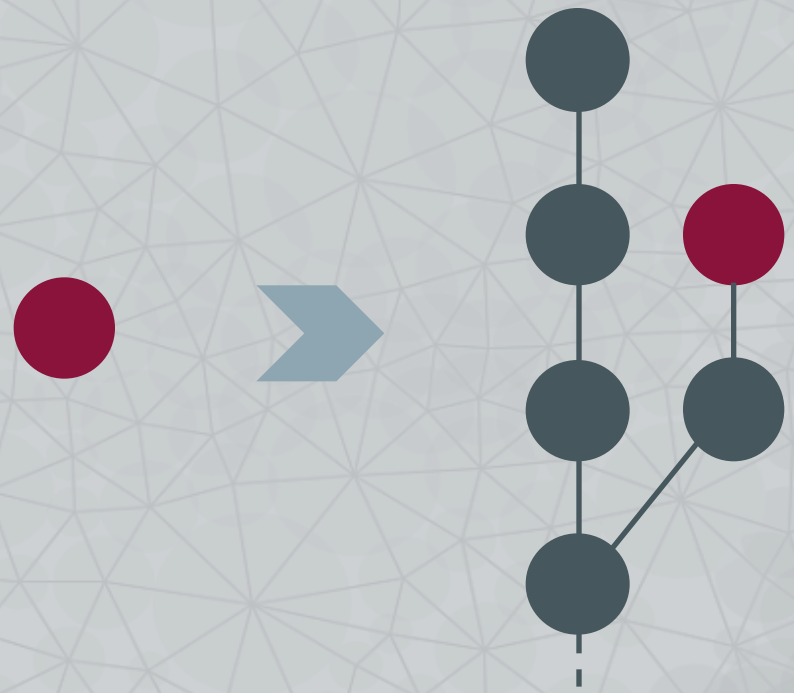
# Dependency cycle


```
                                    ()-[
:includes | uses_namespace | expands | interrogates | undefined | aliases |
  uses_enumerator | 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

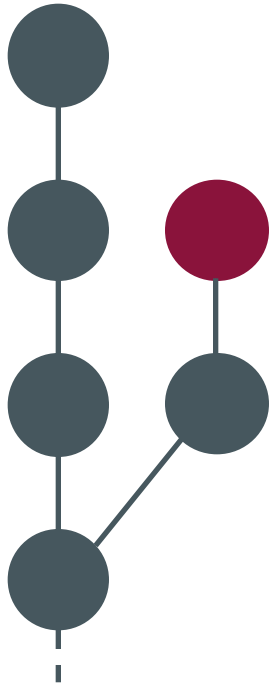
**Developers working off of different versions of the code**



# Target Scenario

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



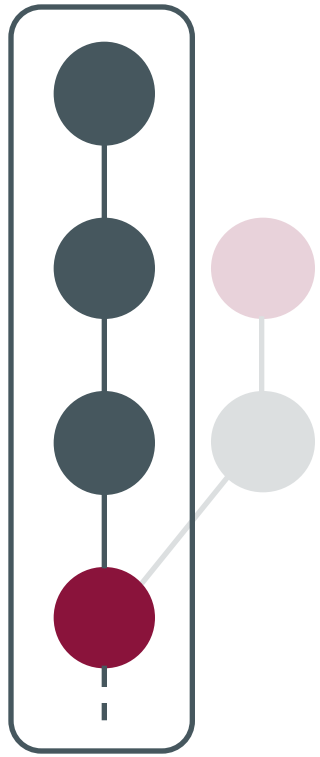


# Current Deployment

- 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



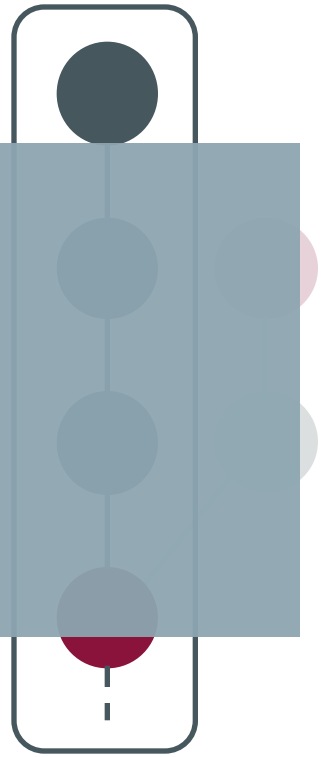


# Current Deployment

- 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

Select 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)
```

VS

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

VS

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

# Open Questions

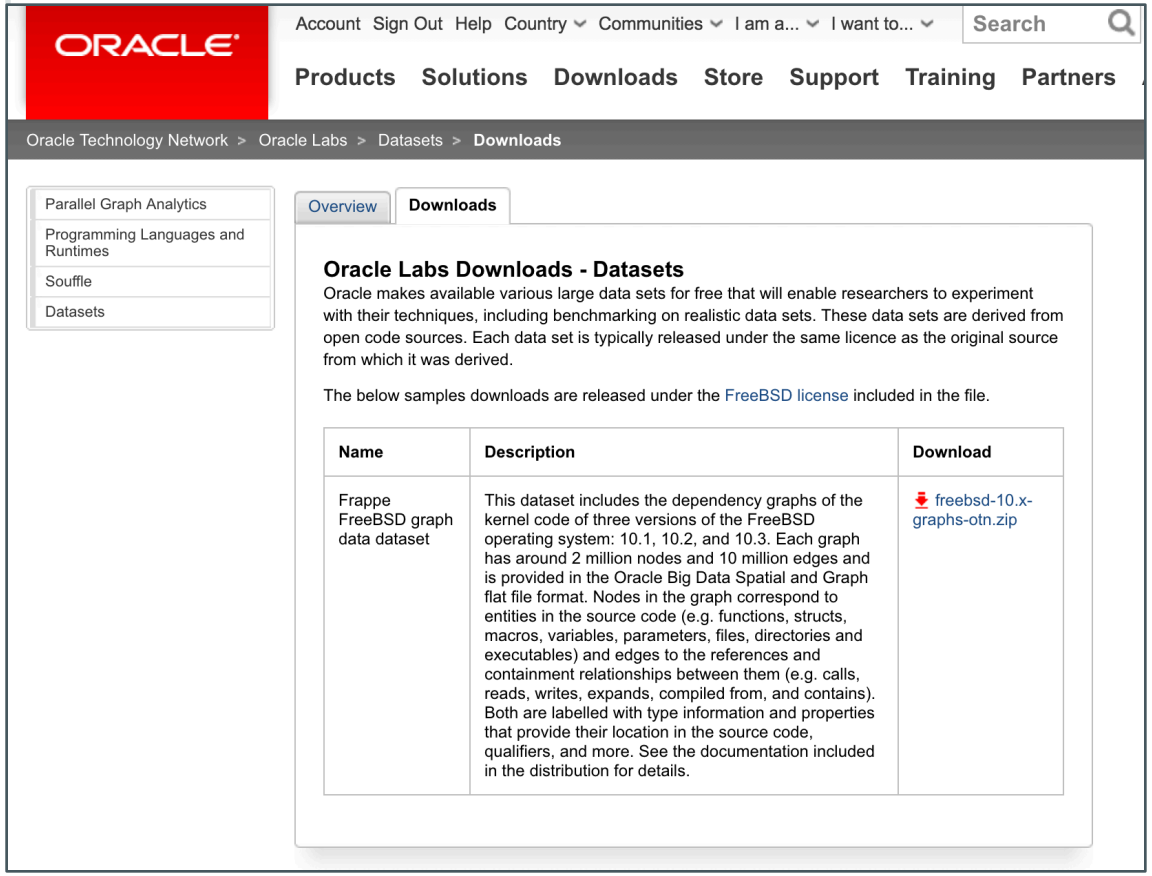
- Node/Edge identity
  - Supplied or derived using graph information
- Query language expressiveness
  - Regular path expressions
  - Multiple edge labels
- Efficient storage and querying for multiple graph versions:
  - List or ideally DAG of versions
  - Union, intersection, difference of results from 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

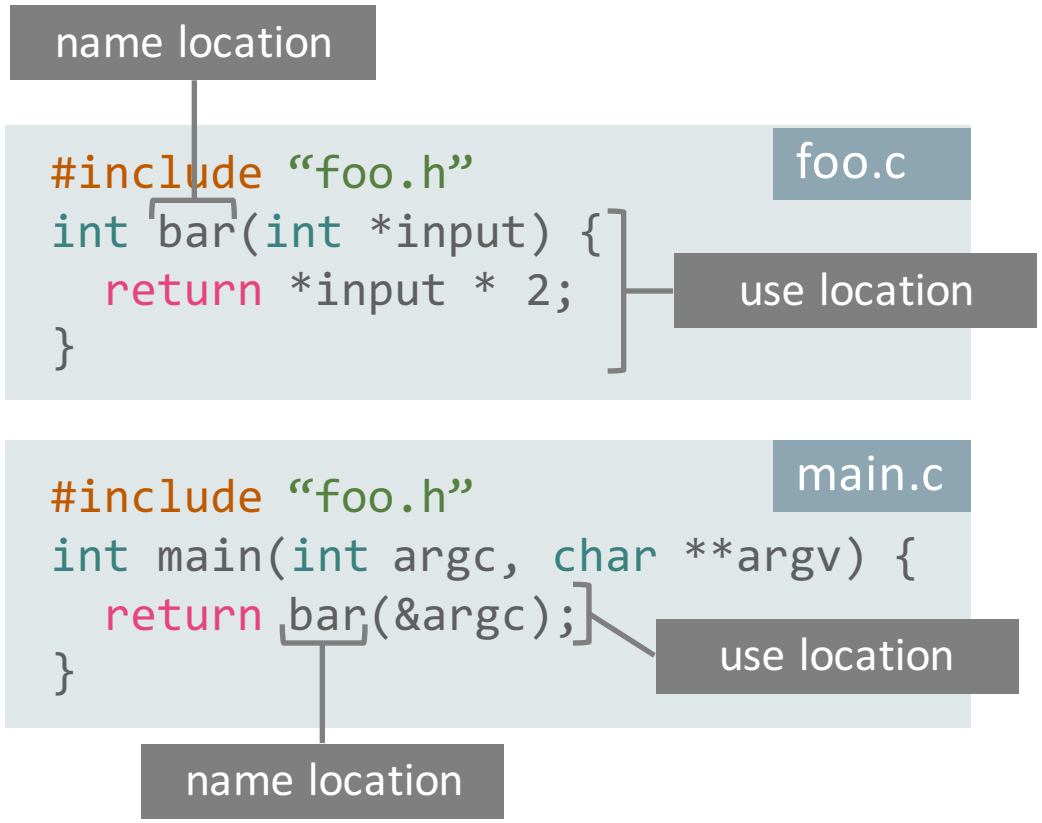
http://www.oracle.com/technetwork/oracle-labs/datasets/downloads/index.html

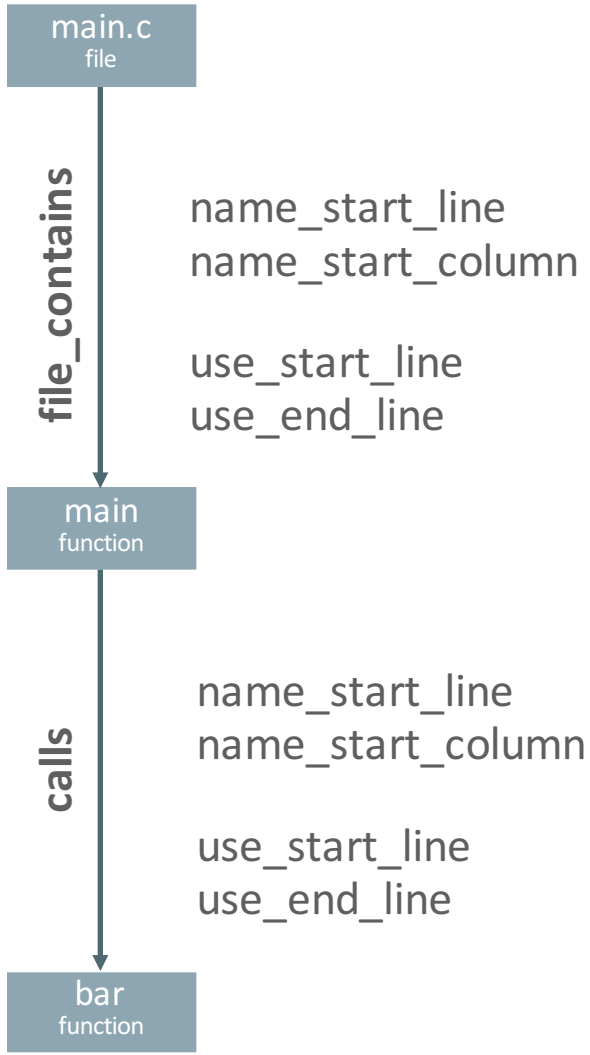



ORACLE®

# Graph model example



# Graph model example

foo.h

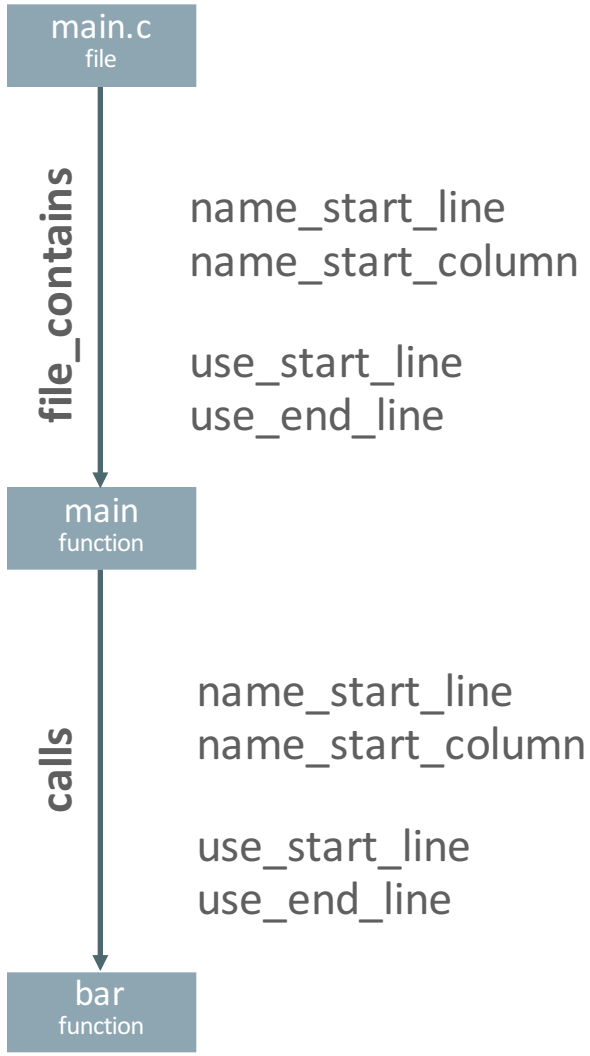
```
#define BAR(A) bar(A)
int bar(int);
```

foo.c

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

main.c

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



# Graph model example

foo.h

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

name location

foo.c

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

main.c

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

use location