Resolving Linkage Anomalies in Extracted Software System Models

Jingwei Wu and Richard C. Holt
School of Computer Science
University of Waterloo
Waterloo, Canada
{j25wu, holt}@plg.uwaterloo.ca

Abstract

Program model linking has been largely overlooked and not constrained properly in the extraction of software system models. This often results in inaccurate system models at different levels of abstraction even if programs can be extracted correctly. This paper describes two constrained approaches toward accurate linkage resolution. The first approach is purely based on a set of linking heuristics. The second approach leverages the software build process and also utilizes linking heuristics. We compare these two approaches and discuss their benefits and limitations. The empirical results from a case study of the PostgreSQL database system are also presented. Our study shows that inappropriate linkage resolution leads to a relatively large number of dependency anomalies at higher levels of abstraction. These anomalies can be effectively removed using our proposed approaches.

1 Introduction

Modern software systems are extraordinarily large. For example, large database systems have millions of lines of code and are extremely complex. In order to analyze such large systems, it is often necessary to extract system models that have linkage dependencies at various levels of abstraction, such as function level, module level, and subsystem level. A linkage dependency normally refers to a function call, a variable access, or a use dependency between two modules. This kind of system model represents static software structure and supports downstream software analysis, for example, reflexion modelling [8], software clustering [7, 12], and architecture extraction and repair [6, 11].

To ensure the accuracy of extracted system models, a significant amount of effort has been expended in developing high quality program extractors [1, 2, 3, 10]. By contrast, the process of linking separately extracted program models into system models has been largely overlooked. If not properly done, this linking process can either produce unexpected linkage or cause linkage absences. We refer to both phenomena as linkage anomalies. In an extracted system model, linkage anomalies only account for a very small percentage of all the resolved linkage dependencies but they propagate to higher levels of abstraction and cause undesirable impacts on downstream software analysis.

In a case study of the PostgreSQL system [9], we found that a small number of linkage anomalies caused by unconstrained linking produced a relatively large number of dependency anomalies at higher levels of abstraction. We will explain unconstrained linking in Section 2.2. About 1.7% of the resolved cross-references in PostgreSQL were anomalies. The high-level dependency anomalies caused by these anomaly cross-references accounted for 3.5% of dependencies at the module level and 7.2% at the subsystem level. If we do reflexion modelling analysis on the extracted system model, we will end up with many architectural divergences that are not caused by the system implementation but by improper linkage resolution.

This paper discusses what causes linkage anomalies and how to resolve them under the assumption that programs are extracted separately and accurately. The rest of this paper is organized as follows. Section 2 explains program model linking and linkage anomalies. Section 3 discusses a set of heuristics and proposes two approaches to resolving linkage anomalies. Section 4 presents the empirical results of our case study. Section 5 concludes the paper.

2 Program Model Linking

We define program model linking as the process of combining multiple linkable program models (LPM) into one model and replacing symbolic references with direct references. In this definition, a LPM is a semantic representation of an object module. Therefore, we can see that program model linking plays the role of code linking, in which object modules are replaced with LPMs.

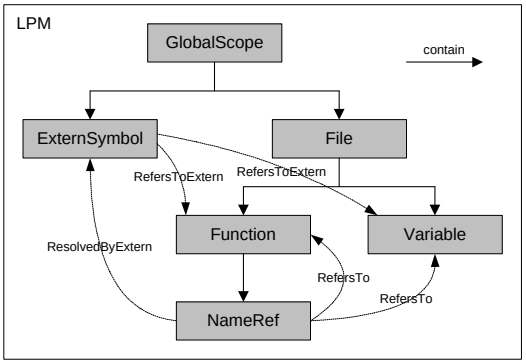


Figure 1. LPM Schema

2.1 LPM Schema

A large number of program extractors exist [1, 2, 3, 10]. They output facts in accordance with different schemas. To facilitate our discussion, we adopt a fact schema shown in Figure 1, which is a reduced version of the CPPX/DATRIX schema [2, 4]. Many abstract syntax tree (AST) nodes such as `Type` and `Statement` have been removed.

The `GlobalScope` represents the universal root of all LPMs. A `File` entity is a source file or an object module. Each `ExternSymbol` entity has a unique identifier that is a symbolic name in the global naming space. A `NameRef` entity is the domain of a symbolic reference.

There are four relations in the schema. The `contain` is a structural relation, which forms a tree in any LPM. The `RefersTo` represents the resolved references within a file. The `ResolvedByExtern` relation represents symbolic references, which need to be resolved by looking for extern definitions. The `RefersToExtern` means that a global definition is exported as a symbolic name.

2.2 LPM Linking

The resolution of symbolic references among a collection of LPMs can be concisely expressed and calculated as a relational composition of two binary relations:

```
LinksTo = ResolvedByExtern o RefersToExtern
```

We use `o` to refer to relational composition. The `LinksTo` relation denotes cross-references from `NameRef` entities to `Function` and `Variable` entities. For example, if X is resolved by Y and Y refers to Z, then X links to Z. By contrast, the `RefersTo` relation denotes resolved references within a LPM. These two relations together denote all direct references within a software system.

Program model linking provides a mechanism to derive software system models. A system model is the combination of program models that are progressively generated as the system is being built, and it has no dangling symbolic references among those generated program models. In this definition, the prohibition of dangling symbolic references mandates that any symbolic references resolvable within the system must be resolved. However, it is optional to resolve those symbolic references pointing to the environment, in which the system is embedded.

An approach to deriving system models is unconstrained if it solely uses the above equation to resolve symbolic references without considering constraints, such as program build dependencies and heuristics, which we will discuss in more detail in Section 3. An unconstrained approach may produce linkage anomalies due to the multi-resolution of symbolic references.

2.3 Linkage Anomalies

A linkage anomaly means an unexpected or a missing linkage dependency in a software system model. In the following, we discuss possible causes of linkage anomalies.

Multi-Resolution

A symbolic name with more than one definition can lead to multiple resolutions of a symbolic reference that refers to the name. This can occur when the software system comprises a collection of executables and dynamic libraries. In this case, it is highly possible that one symbolic name has more than one definition in various parts of the system. If model linking is not properly controlled, the resolution of a symbolic reference may lead to multiple direct references with only one of them representing the true cross-reference dependency. For example, the PostgreSQL system has 18 executable programs and 28 dynamic libraries targeted at the Linux platform. There exist 75 symbolic names with multiple global definitions. One of them is a function called `EncodeDateTime`, which is defined in two different source files, `datetime.c` and `dt_common.c`. The first file is used to generate the executable `postgres`, which is the backend database server, and the second file is used to build the dynamic library `libecpg.so`, which provides support for embedding SQL in C. If we apply unconstrained linking, every symbolic reference to the name `EncodeDateTime` will be resolved to two direct references.

Incomplete Resolution

To overcome the multi-resolution problem, one solution is to leverage the software build process to constrain model linking. We refer to this approach as simulation-based linking, which will be described in Section 3. Unfortunately, the simulation-based approach does not guarantee complete linkage resolution. A software system build often includes

dynamic libraries. When generating a dynamic library, a code linker such as the GNU linker leaves some dangling symbolic references, which will be resolved by the runtime linker as the dynamic library is loaded into memory. However, a system model requires that any dangling symbolic references must be resolved if they refer to definitions within the system. We now describe such a case found in PostgreSQL. The function `tuplestore_begin_heap` defined in the file `tuplestore.c` is called by the function `exec_init_tuple_store` defined in `pl_exec.c`. The file `tuplestore.c` is part of the backend executable `postgres`, and the file `pl_exec.c` is compiled and then linked into a shared library called `libplpgsql.so`, which provides a loadable procedural language. If we only follow the steps of building PostgreSQL to do model linking, we will not be able to resolve any symbolic references to `tuplestore_begin_heap` in the file `pl_exec.c`.

3 Resolving Linkage Anomalies

We now describe two approaches to linkage resolution. In contrast to unconstrained linking, these approaches apply constraints to control the process of model linking. The first approach purely utilizes five linking heuristics. The second approach simulates the software build process to guide model linking and then uses heuristics for postprocessing.

3.1 Linking Heuristics

In the following, we discuss five heuristics, which can be applied to improve the accuracy of program model linking.

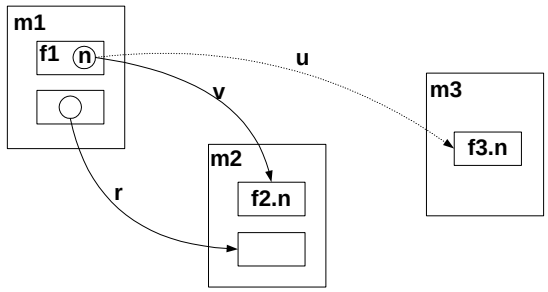
H1 Unique Definition

If a symbolic name refers to exactly one definition, all the symbolic references pointing to that symbolic name can be resolved without ambiguity. This heuristic is the most useful since it resolves the majority of symbolic references.

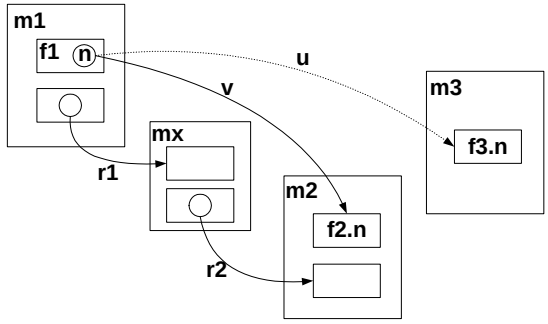
H2 Module-Module

If a cross-reference already exists from module `m1` to module `m2`, we give `m2` higher priority when searching for the definition of any unresolved symbolic references in `m1`.

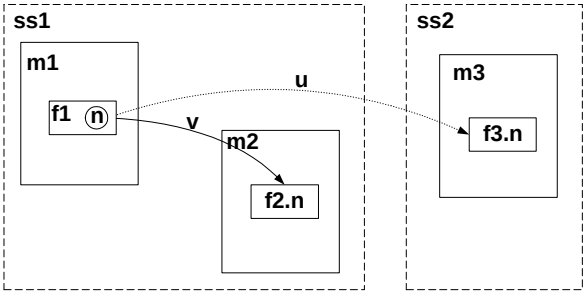
Figure 2(a) provides an illustration of this heuristic. When resolving the symbolic reference in module `m1` to the name `n`, we have two cross-reference candidates, `v` and `u`. We use the notation `x.n` to indicate that `x` is named `n`. Since there already exists a resolved cross-reference `r` between `m1` and `m2`, `v` is determined to be the correct reference in this case.



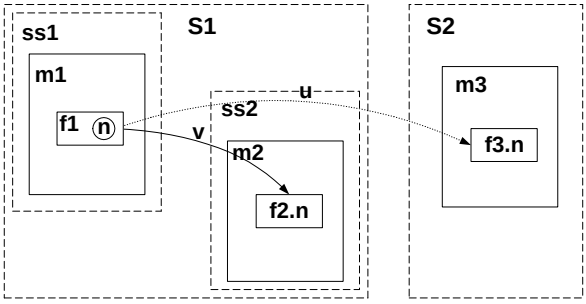
(a) Module-Module



(b) Module-Module Transitive Closure



(c) Same Subsystem



(d) Nearest Super Subsystem

Figure 2. Linking Heuristics (H2-H5)

H3 Module-Module Transitive Closure

This heuristic is an extension of H2. We lift resolved cross-references to dependencies at the module level and then compute the transitive closure of these module dependencies. This closure is then used to guide H2 to resolve linkage anomalies.

Figure 2(b) shows a simple case. There is no direct cross-reference between `m1` and `m2`, but an indirect module dependency between them is formed by `r1` and `r2`. We give `m2` priority over `m3` when searching for the symbolic name `n`. Thus, `u` is determined to be the anomaly to eliminate.

H4 Same Subsystem

In the C programming language, a static definition in a compilation unit has higher priority of being linked to when a symbolic reference is to be resolved in that unit, even if an extern definition with the same name exists. We extend this rule and apply it as a heuristic in the resolution of symbolic names within the scope of the subsystem[1]. We refer to this heuristic as the *Same Subsystem*. Due to the nature of software organization, files in the same subsystem are generally more closely related to each other than those distributed in different subsystems.

Figure 2(c) illustrates this heuristic. The symbolic name `n` used inside function `f1` is ambiguously linked to functions in `m2` and `m3`. Since the two modules `m1` and `m2` are in subsystem `ss1`, `v` is determined to be correct linkage while `u` is an anomaly.

H5 Nearest Super Subsystem

Given a nesting hierarchy of subsystems, a module's super subsystem is the one that indirectly contains that module. In heuristic H5, a nearer super subsystem of a module has a higher priority to be searched than a farther super subsystem when a symbolic reference in that module needs to be resolved without ambiguity.

In Figure 2(d), the nearest super subsystem of `m1` that has the definition of the symbolic name `n` is `S1`. So, `u` is determined to be an anomaly.

3.2 Heuristics-Based Linking

The following describes a heuristics-based algorithm. `R` represents the resolved `LinksTo` linkage relation, and `U` stores the unresolved `ResolvedByExtern` symbolic references. `H` refers to the set of heuristics to apply. The algorithm simply loops through all the heuristics in `H` until no symbolic reference in `U` can be resolved.

[1] In our studies, we adopted the source directory structure as the hierarchy of subsystems.

```
H = Set of Heuristics
R = Empty Resolved References
U = Unresolved Symbolic References

loop
    for h in H
        r = h(U)
        R = R + r
        U = U - (dom r) o U
    end for
while(U is reduced)
```

In the beginning of this algorithm, `R` is initialized to be be an empty relation, and `U` contains all the symbolic references to resolve. The expression `h(U)` applies heuristic `h` to `U` to determine correct linkage. The expression `dom r` returns the domain of the binary relation `r`, and the expression `(dom r) o U` calculates the resolved symbolic references in `U`. This algorithm stops if no heuristic in `H` is able to reduce `U`.

In our implementation of this algorithm, we specified an order of the five heuristics discussed in Section 3.1, from H1 to H5. In addition, we removed the outer loop and applied heuristics H2 and H3 after the execution of the inner for loop to change `R` and `U` accordingly. That is, we do not loop the five heuristics in random but apply them in the order of H1, H2, H3, H4, H5, H2, H3. This order gives an optimal solution and reduces linkage anomalies to a minimum degree. The order is important for the following reasons: (1) H1, H4 and H5 cannot resolve any symbolic references after their first use; (2) H3 is less accurate than H2; (3) H5 is less accurate than H4; (4) it is useful to apply H2 and H3 again if H4 or H5 have resolved at least one symbolic reference; and (4) H2 and H3 are not effective after their second use.

3.3 Simulation-Based Linking

In contrast to pure heuristics-based linking, our second approach simulates the process of building programs and uses heuristics for postprocessing. The basic idea is to instrument the build process to collect program build dependencies, which are then sequentially applied to guide model linking. As described in Section 2.3, dangling symbolic references may appear if dynamic or shared libraries are generated. We apply heuristics H1-H5 to eliminate dangling references.

We instrumented the GNU linker *ld* to dump module dependencies. For example, as the `heap` subsystem in the PostgreSQL is being built, the following fact is dumped:

```
SUBSYS.o : hio.o heapam.o tuptoaster.o
```

It means that three object modules are linked into a relocatable module called `SUBSYS.o`. Our model linker, called *ldm*, is instructed to create the linked model `SUBSYS.ta` based on `hio.ta`, `heapam.ta` and `tuptoaster.ta`,

Approach	Resolved	Unresolved	Anomalies
Unconstrained	40512	0	706
Heuristics	39763	43	2
Simulation	39806	0	0
Symbolic References = 39806 Resolved Cross References = 39806			

Table 1. Linking Results of the PostgreSQL

which are files representing the LPMs of the three object modules. The *ldm* applies the equation given in Section 2.2. An implicit naming convention is used to map modules to LPMs in the form of TA [5]. After processing all the build dependencies, the *ldm* further applies linking heuristics to eliminate dangling symbolic references resolvable within the system.

4 Case Study

We studied the PostgreSQL, which is a large open source database system with 570 KLOC [9]. Its source code was dated January 01, 2004. For brevity, two kinds of trivial symbolic references were ignored: references resolvable within files or modules, and references to system-wide libraries such as *libc.so* and *libcrypt.so*. We were only interested in studying cross-references among files within the system.

As shown in Table 1, the PostgreSQL system model had 39806 symbolic references and the same number of cross-references. The simulation-based approach resolved 39806 references, which we verified to be correct linkage. By contrast, unconstrained linking produced 706 anomalies and heuristic linking left 43 symbolic references unresolved. This study indicates that heuristic linking is better than unconstrained linking but itself does not guarantee a complete linkage resolution.

We now discuss what impacts linkage anomalies have on the derivation of models at higher levels of abstraction. Based on the simulation result, we calculated 3763 dependencies among 462 modules (files) and 657 dependencies among 87 subsystems. In the case of unconstrained linking, the 706 anomalies introduced 138 dependency anomalies at the module level and 51 at the subsystem level. Correspondingly, the error rates were 3.5% and 7.2%. In the case of heuristics-based linking, the 43 unresolved symbolic references caused 20 missing dependencies at the module level and 12 at the subsystem level. The inaccuracy rates were approximately 0.5% and 1.8%.

5 Conclusions

This paper investigated linkage resolution in the extraction of large software system models. We showed that unconstrained linking resulted in linkage anomalies. To solve this problem, we proposed two approaches. One is purely based on a set of heuristics. The other is guided by program build simulation and constrained by heuristics. We conducted a case study to evaluate their effectiveness. The empirical results showed that heuristics-based linking and simulation-based linking reduced linkage anomalies effectively.

References

[1] Acacia. *The C++ Information Abstraction System Online References*. http://www.research.att.com/sw/tools/Acacia, 1996.

[2] CPPX. *The C/C++ Source Extractor Online References*. http://swag.uwaterloo.ca/∼cppx, 2002.

[3] R. Ferenc, A. Beszedes, F. Magyar, and T. Gyimothy. A short introduction to columbus/can. *Technical Report*, 2001.

[4] A. E. Hassan, R. C. Holt, B. Lague, S. Lapierre, and C. Leduc. E/r schema for the datrix c/c++/java exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 284–286, Brisbane, Australia, November 23-25 2000.

[5] R. C. Holt. *An Introduction to TA: the Tuple-Attribute Language*. http://plg.uwaterloo.ca/∼holt/cv/papers.html, 2002.

[6] R. Kazman and S. J. Carrière. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, BC, Canada, June 1998.

[7] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–52, Ischia, Italy, June 24-26 1998.

[8] G. C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, October 1995.

[9] PostgreSQL. *The PostgreSQL Database System Online References*. http://www.postgresql.org, 2003.

[10] TkSee/SN. *The TkSee/SN Source Code Extractor Online References*. http://www.site.uottawa.ca/∼tcl/kbre, 2003.

[11] J. B. Tran, M. W. Godfrey, E. H. S. Lee, and R. C. Holt. Architecture repair of open source software. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 48–59, Limerick, Ireland, June 10-11 2000.

[12] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 258–267, Brisbane, Australia, November 23-25 2000.