A language-independent approach to the extraction of dependencies between source code entities

Miloš Savić*, Gordana Rakić, Zoran Budimac, Mirjana Ivanović

Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad
Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia

Abstract

Context. Software networks are directed graphs of static dependencies between source code entities (functions, classes, modules, etc.). These structures can be used to investigate the complexity and evolution of large-scale software systems and to compute metrics associated with software design. The extraction of software networks is also the first step in reverse engineering activities.
Objective. The aim of this paper is to present SNEIPL, a novel approach to the extraction of software networks that is based on a language-independent, enriched Concrete Syntax Tree representation of the source code.
Method. The applicability of the approach is demonstrated by the extraction of software networks representing real-world, medium to large software systems written in different languages which belong to different programming paradigms. To investigate the completeness and correctness of the approach, class collaboration networks (CCNs) extracted from real-world Java software systems are compared to CCNs obtained by other tools. Namely, we used Dependency Finder which extracts entity-level dependencies from Java bytecode, and Doxygen which realizes language-independent fuzzy parsing approach to dependency extraction. We also compared SNEIPL to fact extractors present in language-independent reverse engineering tools.
Results. Our approach to dependency extraction is validated on six real-world medium to large-scale software systems written in Java, Modula-2, and Delphi. The results of the comparative analysis involving ten Java software systems show that the networks formed by SNEIPL are highly similar to those formed by Dependency Finder and more precise than the comparable networks formed with the help of Doxygen. Regarding the comparison with language-independent reverse engineering tools, SNEIPL provides both language-independent extraction and representation of fact bases.
Conclusion. SNEIPL is a language-independent extractor of software networks and consequently enables language-independent network-based analysis of software systems, computation of design software metrics, and extraction of fact bases for reverse engineering activities.

Keywords: software networks, dependency extraction, enriched concrete syntax tree, software metrics, fact extraction, reverse engineering

1. Introduction

Modern software systems consist of many hundreds or even thousands of interacting entities at different levels of abstraction. For example, complex software systems written in Java consist of packages, packages group related classes and interfaces, while classes and interfaces declare or define related methods and class attributes. Interactions, dependencies, relationships, or collaborations between software entities form various

*Corresponding author; phone: +381-21-458888; fax: +381-21-6350458;
Email addresses: svc@dmi.uns.ac.rs (Miloš Savić), goca@dmi.uns.ac.rs (Gordana Rakić), zjb@dmi.uns.ac.rs (Zoran Budimac), mira@dmi.uns.ac.rs (Mirjana Ivanović)

types of *software networks* that provide different granularity views of corresponding software systems. In the literature software networks are also known as software collaboration graphs [1], software architecture maps [2], and software architecture graphs [3]. Depending on the level of abstraction specific software networks, such as package, class and method collaboration networks [4], can be distinguished. Additionally, different coupling types between entities of the same type determine different software networks [5]. Due to the terminological and type diversity we use generic term "software network" to refer to any architectural (entity-level) graph representation of real-world software systems, and to distinguish them from networks representing other complex natural, social, conceptual or man-made systems.

Software networks can be viewed as the sub-notion of a more general notion of real-world complex network, i.e. network representing a real and evolving system. Complex network theory [6, 7, 8, 9] provides a set of techniques for statistical analysis and modeling of real-world networks. When applied to software systems such techniques are able to identify and explain connectivity patterns and evolutionary trends in dependency structures formed by software entities. Links in software networks denote various relationships between software entities such as coupling, inheritance, and invocation. This means that software networks can be used to compute software metrics related to software design. The first step in reverse engineering, architecture recovery, and software comprehension activities is the identification of software entities and relations among them [10]. Therefore, software networks can be also viewed as fact bases required for the mentioned activities. Graphical representations of software entities and dependencies between them have long been accepted as comprehension aids to support reverse engineering processes [11]. Moreover, the nodes in a software network can be enriched with software metrics information in order to provide visual, polymetric views (such as the system complexity view in [11] or the MettricAttitude view in [12]) of analyzed software systems. Additionally, software networks can be exploited to identify and remove bad smells in a source code [13], to support static concept location in the source code [14] and to support program comprehension during incremental change [15].

This paper addresses the process of extraction of software networks. SNEIPL[1], a tool that is able to extract software networks at different levels of abstraction, will be presented. The main characteristic of SNEIPL is that it uses the enriched Concrete Syntax Tree (eCST) representation [16, 17] of the source code to form software networks. eCST is the language-independent source code representation, and consequently makes SNEIPL independent of programming language. Therefore, the main contribution of SNEIPL is that enables the language-independent analysis of software systems under the framework of complex network theory, language-independent computation of software design metrics, and language-independent extraction and representation of fact bases for reverse engineering activities.

The rest of the paper is structured as follows. The background and motivation for this study are given in Section 2 focusing on the three fields of research and practice. The contributions of the paper are highlighted in Section 3. In the next section software networks that can be extracted using SNEIPL are introduced and defined. The overview of the eCST representation is given in Section 5. The next section covers the architecture of SNEIPL. In the same section important details of the dependency extraction process are discussed. The results of the experimental evaluation that demonstrate the validity of our approach are given in Section 7. The comparative analysis of networks extracted by SNEIPL and networks formed using two other tools is provided in Section 8. The related work is discussed in Section 9. The last section concludes the paper.

2. Background and motivation

In this section we discuss the importance of the extraction of software networks focusing on the three fields of research and practice: analysis of software networks under the framework of complex network theory, computation of software design metrics, and reverse engineering of software systems.

[1]The source code of SNEIPL can be downloaded at http://ssqsa.googlecode.com/svn/trunk/sneipl/

2.1. Analysis of software networks

In the past decade, a large and growing body of research investigated properties of complex real-world networks representing various biological, social, technological, and conceptual systems, including also networks representing software systems [6, 7, 8]. Even though those networks represent totally different types of systems, they share many common properties such as the small-world property [18], the scale-free property reflected by a power-law degree distribution [19], higher degree of local clustering compared to a random graph of the same size [18], the "robust yet fragile" property [20], the absence of the propagation threshold in spreading processes [21], and formation of highly modular or community structures [22, 23]. These studies have served to draw together many disparate fields into an emerging theory of complex networks whose focus is on statistical analysis techniques, organizational principles, evolutionary mechanisms, and mathematical models that can reveal and explain frequently observed macroscopic and topological characteristics of real-world networks [9, 24]. Empirical investigations of concrete software networks under the framework of complex network theory [1, 2, 4, 25, 26, 27, 28] showed that their statistical properties can help us to understand and quantify the complexity and evolution of corresponding software systems.

2.2. Software design metrics

Software engineering practice or even the application of simple software metrics such as LOC, can show us that modern software systems are complex artifacts. An essential complexity of software is a consequence of a high number of software entities defined in the source code and the complex interactions among them [29]. Most of traditional software metrics used for the estimation of software complexity (such as LOC, Cyclomatic complexity, Halstead metrics, etc.) are mainly oriented towards the internal complexity of software entities. They are used to identify algorithmically complex entities that should be re-decomposed into the sets of smaller, less complex, easily maintainable entities that can be reused later as the software system evolves. The main characteristic of the metrics of internal complexity is that they do not take into account existing interactions between software entities. The complexity of interactions among software entities can be quantified by the class of software design metrics that reflect *coupling*, *cohesion*, *inheritance*, and *invocation*, to mention a few. Widely known and used metrics from this category are those introduced in the Chidamber-Kemerer metric suite [30]: CBO (Coupling between objects), DIT (Depth of inheritance tree), NOC (Number of children), LCOM (Lack of cohesion of methods), and RFC (Response for a class). In order to calculate the metrics of software design, source code entities and relations between them have to be identified, which means that network representations of the software system have to be extracted.

2.3. Reverse engineering

The primary goal of a reverse engineering activity is to identify system's components and relationships among them in order to create the representation of the system at a higher level of abstraction [31]. A typical reverse engineering activity starts with the extraction of fact bases [32]. Source code is the most popular, valuable, and trusted source of information for fact extraction because other artifacts (documentation, release notes, information collected from version management, bug tracking systems, etc.) may be missing, outdated, or unsynchronized with the actual implementation. Fact extraction is an automatic process during which the source code is analyzed to identify software entities and their mutual relationships. This process results in an abstract representation (model) of the extracted information [10]. In syntactic fact extraction the exported facts include variable and class references, procedure calls, use of packages, association and inheritance relationships among classes [33]. Software networks are used as a part of input for computing reflection models in software reflexion analysis [34]. Architecture recovery techniques usually perform software network partitioning [35, 36, 37, 38, 39] or cluster software entities according to feature vectors that can be constructed from software networks [40, 41, 42].

3. Contributions

The first prototype of SNEIPL was described in [43], where basic principles of the extraction of software networks based on the eCST representation of the source code are explained. In the same article it was also

shown that the prototype of SNEIPL extracted isomorphic software networks representing two small, but structurally and semantically equivalent software systems written in different programming languages (Java and C#).

This paper extends the work presented in [43] and its contributions can be summarized as follows. Firstly, the applicability of SNEIPL is demonstrated by the extraction of software networks that represent real-world, medium to large-scale software systems written in Java, Modula-2, and Delphi. Mentioned languages are characteristic representatives of three programming paradigms: object-oriented (Java), procedural (Modula-2), and mix of these two (Delphi). Therefore, the demonstration of the applicability on these languages can express applicability of the approach in a broad range of languages. Also, we demonstrate that SNEIPL is able to identify dependencies at different levels of abstraction, thus providing different granularity views of analyzed software systems.

Secondly, in this paper we investigate the correctness and completeness of our dependency extraction approach. Class collaboration networks (CCNs) associated with ten real-world Java software systems are extracted using SNEIPL and then compared to CCNs extracted by Dependency Finder, a language-specific tool which forms CCNs from Java bytecode. In the comparative analysis we also include CCNs formed with the help of Doxygen, a language-independent documentation generator tool. Doxygen is able to form local class collaboration graphs in a language-independent way that is based on the unified fuzzy parsing approach, i.e. there is one unified but light-weight parser providing dependency extraction for several languages. Results of the comparative analysis show that dependency networks extracted by SNEIPL are highly close to those extracted by the language-dependent tool, and that the eCST-based approach to language-independent, entity-level dependency extraction provides far more precise results than the unified fuzzy parsing approach realized by Doxygen. Moreover, we investigated how differences between networks obtained by SNEIPL and Dependency Finder affect the analysis of design complexity of real-world software systems and computation of software metrics.

Finally, we compared our dependency extraction approach to the fact extractors of relevant language-independent reverse engineering tools and frameworks. It is shown that language-independent reverse engineering tools provide language-independent representations of fact bases, but their extraction is mostly done in a language-dependent way. On the other side, SNEIPL provides language-independent representation of fact bases in terms of General Dependency Networks, as well as their language-independent extraction.

4. Software networks

High-level programming languages enable declaration or definition of different types of entities at different levels of abstraction. In general, the following groups of referable software entities can be distinguished:

- function-level entities (functions and variables) that are at the lowest level of abstraction,
- class-level entities (modules in procedural languages; classes and interfaces in OO languages) that group related function-level entities, but can also contain nested class-level entities, and
- package-level entities (packages, namespaces, units) which group related entities from the lower levels of abstraction.

Software networks can be either homogeneous (networks connecting software entities of the same type by links denoting the same kind of relationships) or heterogeneous (entities and/or connections are of different types). Links in software networks that connect entities from the same level of abstraction will be called "horizontal". On the other hand, links in heterogeneous software networks that connect entities appearing at different levels of abstraction will be called "vertical".

4.1. Function-level networks

Most programs written in a procedural programming language consists of procedures (also called subroutines or functions) which collaborate using the call-return mechanism provided by the language. In object-oriented software systems, software entities known as methods collaborate using the same mechanism.

From this point on, we do not make the explicit distinction between functions, procedures and methods - the mentioned constructs will be used interchangeably since they are function-level entities representing the same concept across different programming paradigms. Call-return relationships between procedures define a software network that is often referred to as a *static call graph* (SCG). In this kind of network two nodes A and B, which represent two different procedures A and B defined in a program, are connected by the directed link $A \rightarrow B$ if A explicitly calls B. Static call graphs for object-oriented (OO) software systems are also known as *method collaboration networks* [4]. It is important to observe that function calls through a reflection mechanism, if it is present in a language, do not form static (structural, compile-time) dependencies between functions, but run-time dependencies.

FUGV (Function Uses Global Variable) networks are heterogeneous software networks that describe dependencies between function-level entities. Similarly as for procedures and methods, we do not make the explicit distinction between global variables in procedural style and class member variables (class attributes) in OO style. FUGV networks are bipartite directed graphs. The nodes in a FUGV network represent functions and global variables. Function A is directly connected to global variable B, if B is used (read or written) in the statements that constitute the body of A. FUGV networks can be used to compute metrics measuring lack of cohesion in methods because in those metrics we are interested to know if two different methods access the same class attribute (global variable) [44].

4.2. Class-level networks

Collaborations of classes and interfaces in an OO software system constitute a *class collaboration network* (CCN). By the term class collaboration network will be also assumed the term *module collaboration network* that denotes collaborations of modules in procedural programming languages. Classes and modules represent the concept of grouping related function-level entities in different programming paradigms. Similarly, we do not make the explicit distinction between interfaces and definition modules.

Two nodes A and B contained in a CCN are connected by the directed link $A \rightarrow B$ if the class or interface represented by node A references the class or interface represented by node B. A can reference B in many ways: by extending the functionality of B, defining a member variable whose type is B, realizing a method which calls some method defined in B, etc. Class collaboration networks can be viewed as simplified class diagrams that preserve only the existence of relations between classes, and discard other types of information about nodes (classes) and links (OO relations). By the definition given in [30], the coupling between objects (CBO) metric for a class is the number of other classes that the class is coupled to (the number of unique classes referenced by the class plus the number of classes that refer to the class). In other words, the CBO is the total degree of a node representing the class in appropriate class collaboration network. Additionally, homogeneous software networks that represent different forms of class coupling, such as inheritance trees or aggregation networks, can be isolated from class collaboration networks [5].

4.3. Package-level networks

At the highest level of abstraction, package-level entities form *package collaboration networks* (PCN). Two packages PA and PB are connected by the directed link $PA \rightarrow PB$ if package PA contains a class or interface that references at least one class or interface from package PB. Similarly as for class collaboration networks, PCNs can be used to calculate coupling metrics at the package level. *Afferent coupling* [45] of a package is the number of incoming links attached to the node representing the package in the PCN. The number of outgoing links measures *efferent coupling* of the corresponding package.

4.4. Vertical dependencies

Hierarchy tree is a heterogeneous software network that contains all entities defined in a software system. This type of network captures vertical dependencies between entities. Two entities A and B are connected by the directed link $A \rightarrow B$ if entity A defines or declares entity B. Hierarchy tree can be used when we are interested to know where an entity is defined (the parent of the entity), and which other entities it defines (the children of the entity). It also enables the calculation of software metrics such as NOC for packages (the number of classes and interface contained in a package), NOM/NOA (the number of methods/attributes

defined in a class) and abstractness (the number of abstract classes divided by the total number of classes in a package). Hierarchy trees are also often used together with other software networks. For example, the computation of RFC (response for a class) metric requires information contained in the static call graph and hierarchy tree of the system.

4.5. General Dependency Network

From the eCST representation of the source code [16, 17] SNEIPL forms a heterogeneous software network called *General Dependency Network* (GDN). GDN is a directed and attributed multigraph: the nodes have type and name, while the links have type and weight (the strength of connection) as attributes. Also, a pair of nodes can be connected by parallel links denoting different coupling types. GDN nodes represent package-, class- and function-level entities defined in the corresponding software system. GDN links represent various types of relations: CALLS relations between functions, REFERENCES relations between package-level entities, REFERENCES relations between class-level entities, USES relations between functions and variables, and CONTAINS relations that reflect the hierarchy of entities. There are also seven types of relations that represent different forms of coupling between class-level entities:

- EXTENDS relation $A \to B$ denotes that A extends the functionality of B,
- IMPLEMENTS relation $A \to B$ denotes that A implements the declarations contained in B,
- INSTANTIATES relation $A \to B$ denotes that A instantiates the objects of B,
- AGGREGATES relation $A \to B$ denotes that A contains a global variable whose type is B,
- WEAK_AGGREGATION relation $A \to B$ denotes that A contains at least one function that declares local variable whose type is B,
- PARAMETER_TYPE relation $A \to B$ denotes that A contains at least one function that has parameter whose type is B,
- RETURN_TYPE relation $A \to B$ denotes that A contains at least one function whose return type is B.

It can be observed that GDN is designed to represent a union of collaboration networks at different levels of abstractions with incorporated CONTAINS links that maintain the hierarchy of entities. Thus, all software networks introduced in the previous subsections can be obtained by GDN filtration, i.e. by the selection of nodes of specified types that are connected by links of specified types.

Figure 1 shows the GDN representation of a simple software system written in Java that consists of two classes (A and B) contained in two packages (PA and PB). The selection of nodes representing packages connected by REFERENCES isolates the package collaboration network of the system. Similarly, the class collaboration network is the sub-network of the GDN induced by nodes representing classes connected by REFERENCES links. The static call graph can be obtained by the selection of nodes representing functions connected by CALLS links ($m \to f$). The FUGV network consists of USES links connecting functions to global variables ($m \to b$). The hierarchy tree has the same set of nodes as the GDN, but the set of links is restricted to CONTAINS links. AGGREGATES and INSTANTIATES links appear in the networks showing specific forms of coupling between classes.

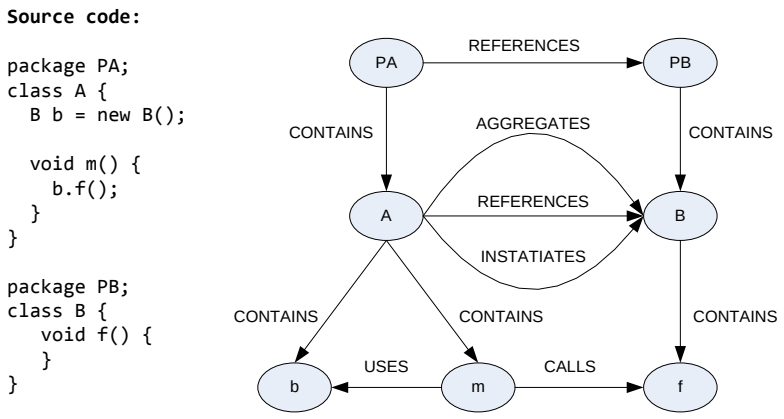

Figure 1: General Dependency Network for a software system consisting of two classes.

5. Enriched Concrete Syntax Tree representation of source code

The main characteristic of SNEIPL is that it uses the enriched Concrete Syntax Tree representation [16] of the source code as the starting point to identify source code entities and dependencies between them. The development of the eCST representation started with SMIILE [46], a language-independent tool for computing software metrics that reflect the internal complexity of software entities (metrics such as LOC, Cyclomatic complexity, Halstead complexity metrics, etc.). In [16] the authors of the eCST representation identified other fields of the research where the eCST representation can be utilized to construct language-independent tools which solve particular language processing problems. This research also lead to the constitution of the SSQSA framework [17, 47], a set of language-independent tools that operate on the eCST representation produced by the SSQSA front-end known as eCST Generator. Besides SMIILE, SSQSA currently contains two other back-ends: SSCA [48] which enables language-independent metric-based analysis of evolutionary changes in the hierarchical structure of software systems, and SNEIPL, the tool that is subject of this paper.

5.1. Fundamentals of eCST representation

As the name of the representation suggests, eCST is a tree representation of the source code. In this subsection of the paper we will explain the principal differences between eCST and two other widely used tree representations of source code: concrete syntax tree (CST) and abstract syntax tree (AST).

The concrete syntax tree (CST) representation shows how a programming language construct is derived according to the context-free grammar of the language. The root node of a CST represents starting non-terminal symbol of the grammar, interior nodes in CST correspond to syntactical categories of the language identified by non-terminal symbols of the grammar, while leaf nodes represent tokens of the construct. Abstract syntax tree (AST) is an alternative and more compact way to represent language constructs. The AST representation retains the hierarchical structure of language constructs, while omitting details that are either visible from the structure of AST or unimportant for a language processing task.

Figure 2 shows the CST, AST and eCST representation of a simple Java source code fragment ("class *A* extends *B* { }"). As it can be observed, all tokens present in the fragment are leaf nodes of the CST and eCST. On the other side, tokens representing keywords ("class" and "extends") appear as the interior nodes in the AST. Separator tokens are not present in AST since grouping parentheses are implicit in the tree structure. All interior nodes in the CST are non-terminal symbols from the Java grammar, while the interior nodes in the eCST are different eCST universal nodes. Clearly, CST is the language-dependent source code representation, since it is closely connected to the grammar of a language. On the contrary, AST abstracts away from the concrete syntax. However, interior nodes of ASTs are keywords and operators of the language, or imaginary tokens introduced to enable tree representation of constructs that can not

be adopted to the "operator-operands" scheme. The usage of lexical elements of the language as interior nodes in the intermediate representation makes the representation language-dependent. The concept of universal nodes introduced in the enriched Concrete Syntax tree (eCST) representation is what makes it substantially different from the AST and CST representations. Universal nodes contained in eCSTs, such as CONCRETE_UNIT_DECL (CUD) in Figure 2, are language-independent markers of semantic concepts expressed by language constructs. One universal node denotes particular semantic concept realized by the syntax construction embedded into the eCST sub-tree rooted at the universal node. For example, CUD universal node in Figure 2 denotes that the sub-tree rooted at CUD contains the definition of a concrete class-level entity. Nodes of eCST can be divided into three categories:

- Universal nodes with predefined, language-independent meanings which denote semantic concepts expressed by language constructs.
- Imaginary nodes with language-dependent meanings which correspond to a subset of non-terminal symbols in the grammar. Those nodes serve only to retain natural hierarchical structure of language constructs in case that there is no universal node that correspond to some non-terminal symbol.
- Tokens that are leaf nodes of eCSTs.

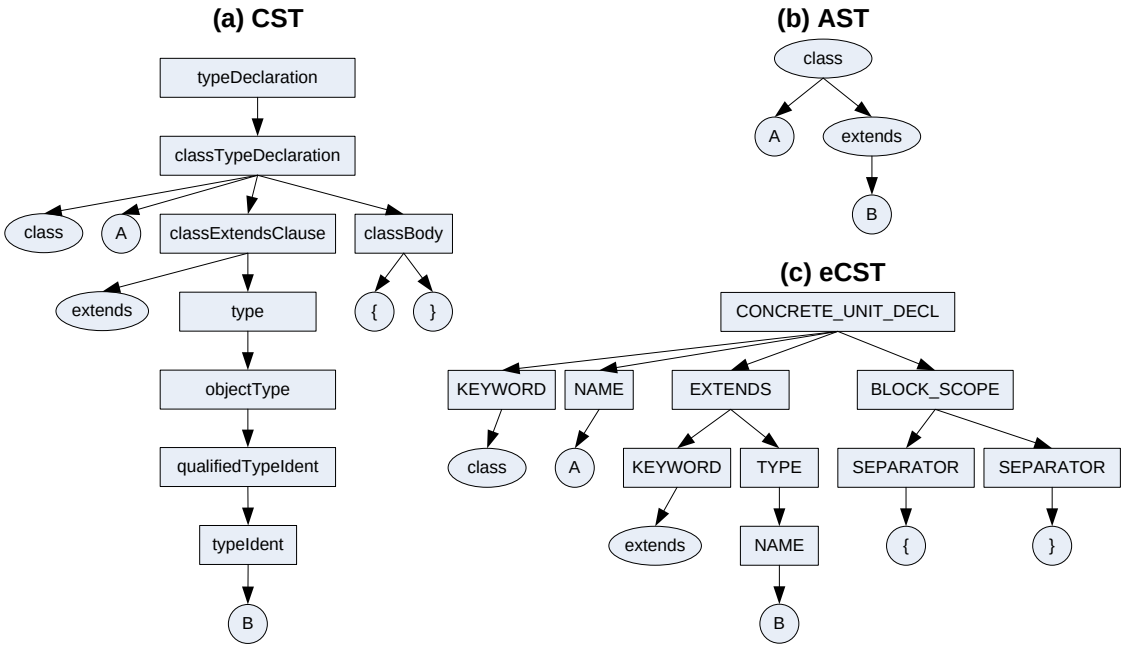

Figure 2: Concrete syntax tree (a), abstract syntax tree (b), and enriched concrete syntax tree (c) representing Java fragment "class A extends B { }".

An eCST is usually more compact than the corresponding CST: one universal or imaginary node can substitute a chain of non-terminal symbols in the CST that is derived through a sequence of unary productions. As it can be observed from Figure 2 the TYPE universal node substituted the chain of three unary productions (type → objectType → qualifiedTypeIdent). On the other hand, the eCST is more voluminous than the corresponding AST, because the eCST includes all tokens present in the source code, while imaginary tokens in the AST are either universal or imaginary nodes in the eCST.

Each eCST universal node expresses some general concept of high-level programming languages. The main design intention is to keep the set of universal nodes as small as possible in order to avoid the redundancy of equivalent concepts that are differently expressed in different programming languages. The set

of universal nodes and the dependency constraints among them are determined by the problems solved by existing SSQSA back-ends, not by the syntactical structures of supported languages. When a new language processing problem is stated, the schema of eCST universal nodes is explored in order to determine if it can support the development of a new SSQSA back-end which solves the problem. This analysis may result in the introduction of new universal nodes in the schema. The support for a new programming language is achieved through the alignment of the schema with the grammar of the language. In this process each eCST universal node is mapped to one or more syntactical categories of the language that are represented by non-terminal symbols of the grammar.

For turning source code into a representation suitable for analysis, comprehension, and transformation in the process of modernization of legacy systems, the OMG group advocates the usage of two metamodels: ASTM (Abstract Syntax Tree Metamodel [49]) and KDM (Knowledge Discovery Metamodel [50]). ASTM is composed of GASTM (Generic Abstract Syntax Tree Metamodel) and SASTM (Specific Abstract Syntax Tree Metamodel). GASTM gives a specification of the common concepts of general purpose programming languages in the form of metatypes. The concept of GASTM metatype is similar to the concept of eCST universal node: the key idea of both concepts is to mark concrete language constructs with generic, conceptual and language-agnostic denotations. However, the similarities between eCST and AST conforming the GASTM are only at the conceptual level. The set of eCST universal nodes is drastically smaller than the set of GASTM metatypes, and evolves together with the development of SSQSA back-ends. Therefore, the SSQSA back-ends which solved concrete language processing problems directly validate the existence, usefulness, and the size of the set of language-independent concepts that are introduced in the eCST representation.

Unlike ASTM, KDM covers not only the source code, but also the operational environment and the domain-specific knowledge integrated in a system. While ASTM is oriented to the specification of ASTs, the program elements layer of KDM establishes a specification for language-independent abstract semantic graphs (ASG). GDNs extracted from the eCST representation show dependencies between software entities, and can be viewed as subgraphs of ASG induced by the nodes representing software entities. Similarly as ASG, GDN provides a higher-level, architectural view of represented code in comparison with eCST/AST conforming ASTM. The difference is that GDN does not contain low-level behavioural details (control and data flows) present in ASG. In other words, the GDN representation is more compact than the ASG representation, since it ignores details which do not reflect design aspects of represented systems.

5.2. Universal nodes used by SNEIPL

Currently the set of eCST universal nodes contains 33 different nodes that can be classified into three groups:

- Lexical-level eCST universal nodes mark individual tokens with appropriate lexical category (keywords, separators, identifiers, etc.).
- Statement-level eCST universal nodes mark individual statements, groups of statements or parts of statements with appropriate concept expressed by them (jump statement, loop statement, branch statement, condition, import statement, block scope, etc.)
- Entity-level eCST universal nodes mark definitions and declarations of package, class and function level entities, and explicitly stated relations between them (such as inheritance, instantiation, implementation, etc.).

SNEIPL naturally relies on the entity-level eCST universal nodes to extract software networks. Table 1 shows the list of all eCST universal nodes used by SNEIPL. All universal nodes listed in the table, except FUNCTION_CALL universal node, where introduced before SNEIPL was designed, implemented, and included in the SSQSA framework, and already used by the two previously created SSQSA back-ends (SMIILE and SSCA).

Figure 3 shows a part of the eCST representation for two structurally equivalent code fragments written in Modula-2 and Java, respectively. The definition of class/implementation module A is marked with the

Table 1: List of eCST universal nodes used to extract software networks.

Universal node	**Abbr.**	**Marks**
COMPILATION_UNIT	CU	Root of each eCST
PACKAGE_DECL	PD	Declaration of packages, namespaces and units
CONCRETE_UNIT_DECL	CUD	Declaration of classes and implementation modules
INTERFACE_UNIT_DECL	IUD	Declaration of interfaces and definition modules
TYPE_DECL	TD	User-defined data types that are not CUDs and IUDs
ATTRIBUTE_DECL	AD	Declaration of class attributes, class fields, global variables
FUNCTION_DECL	FD	Declaration of functions, procedures, methods
FORMAL_PARAM_LIST	FPL	List of parameters in FD definition
PARAMETER_DECL	PAR	One parameter in FPL
VAR_DECL	VD	Declaration of local variables in FD
IMPORT_DECL	ID	Import statements
BLOCK_SCOPE	BS	Block scope within a FD or another BS
FUNCTION_CALL	FC	Function call statements
ARGUMENT_LIST	AL	List of parameters passed to FC
ARGUMENT	ARG	One argument in AL
EXTENDS	EXT	CUD/IUD inheritance
IMPLEMENTS	IMP	IUD implementation
INSTANTIATES	INST	instantiation of objects
TYPE	TYPE	identifiers representing types
NAME	NAME	identifiers

CUD universal node. Entity A contains the definition of global variable/class attribute gv whose type is T, and the definition of procedure/method p. Therefore, the definitions of both mentioned entities are located in the sub-tree rooted at CUD A, and marked with appropriate eCST universal nodes (AD for gv and FD for p, see Table 1). Each identifier is marked with NAME universal node. The parent of NAME universal node determines what the identifier actually represents. If the parent is the TYPE universal node then the identifier represents a type (RT and T).

As already mentioned, SNEIPL is one of the back-ends present in the SSQSA framework. The SSQSA front-end, known as eCST Generator, produces the eCST representation for a given source code [17]. Therefore, the set of programming languages supported by eCST Generator entirely determines the set of programming languages supported by SNEIPL and other SSQSA back-ends. Based on the extension of an input compilation unit, eCST Generator recognizes programming language and instantiates appropriate parser which forms the eCST representation. eCST Generator uses parsers generated by the ANTLR parser generator [51]. The advantage of using ANTLR to describe languages supported by SSQSA is the ANTLR grammar notation itself. This notation enables syntax tree modifications specified by tree-rewrite rules attached to grammar productions. Therefore, when we want to extend SSQSA to support a new language, we have to make the ANTLR grammar for the language and use ANTLR tree-rewrite syntax to specify how existing eCST universal nodes are embedded into produced syntax trees. In other words, the support for a new language is done in a purely declarative way. For example, Listing 1 shows how CONCRETE_UNIT_DECL universal node is incorporated in the grammar productions which describe declarations of Modula-2 implementation modules and Java classes. The extensibility of the SSQSA framework is in details discussed in [52].

Modula-2:

```
IMPLEMENTATION MODULE A;
  VAR
     gv: T;

  PROCEDURE p(fp: T): RT;
  BEGIN
   …
  END p;
END A.
```

Java:

```
class A
{
  T gv;

  RT p(T fp)
  {
    …
  }
}
```

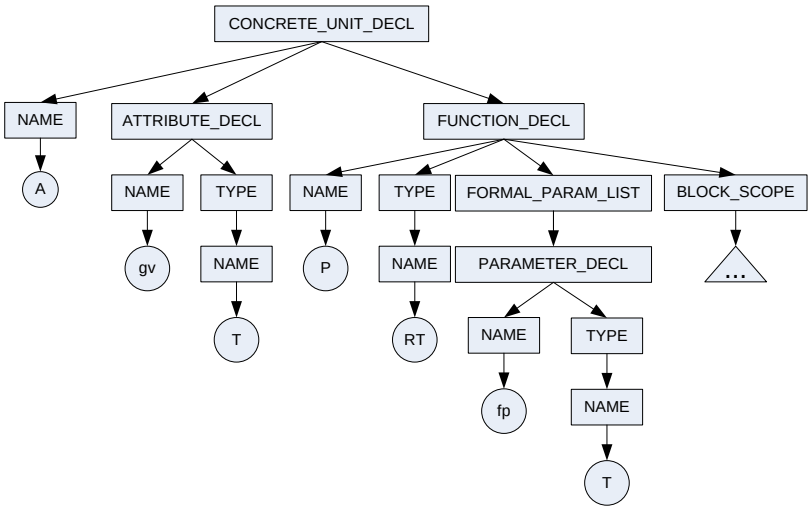

Figure 3: Two code fragments in Modula-2 and Java with the same structure of eCST universal nodes in the eCST representation (only universal nodes and tokens that represent identifiers are shown). Universal nodes are drawn as rectangles while tokens (leaf nodes) are shown as circles. Only eCST universal nodes relevant to dependency extraction are shown.

Listing 1. CONCRETE_UNIT_DECL universal node in ANTLR grammar rules describing the declaration of Modula-2 implementation modules and Java classes, respectively.

```
// excerpt from Modula-2 grammar
moduleDeclaration : 'IMPLEMENTATION'? 'MODULE' ident priority? ';'
                    importList* export* block ident
  -> ^(CONCRETE_UNIT_DECL
           ^(KEYWORD 'IMPLEMENTATION')? ^(KEYWORD 'MODULE') ^(NAME ident)
           priority? ^(SEPARATOR ';') importList* export* block
      );

// excerpt from Java grammar
classDeclaration : 'class' ident genTypes? extClause? impClause? classBody
  -> ^(CONCRETE_UNIT_DECL
         ^(KEYWORD 'class') ^(NAME ident) genTypes? extClause? impClause? classBody
      );
```

6. SNEIPL architecture and the extraction process

SNEIPL consists of two components: GDN Extractor and GDN Filter (see Figure 4). From a set of eCSTs produced by eCST Generator, GDN Extractor constructs the General Dependency Network (GDN)

representation of a software system. GDN Filter, as the name of the component suggests, filters extracted GDN to form the output set of software networks.

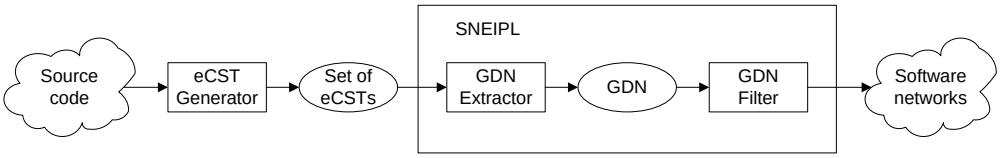

Figure 4: Data flow in software networks extraction process.

GDN is incrementally built in two phases, where both phases analyze each eCST in the input set. Phase 1 recognizes declarations of software entities and creates GDN nodes. To recognize software entities SNEIPL relies on the following subset of universal nodes

$$U_{Phase_1} = \{\text{PD, CUD, IUD, FD, TD, AD}\}.$$

Each universal node from the U_{Phase_1} set has NAME universal node in the sub-tree which contains the name of the software entity, while universal nodes themselves determine the type of newly created GDN nodes (see Table 1).

Vertical dependencies (CONTAINS links) are also created in Phase 1. This means that Phase 1 results in the hierarchy tree representation of analyzed software system. Vertical dependencies are induced from the structure of U_{Phase_1} nodes in eCST. Let a and b denote two software entities declared in the same compilation unit represented by eCST e. Let A and B ($A, B \in U_{Phase_1}$) denote universal nodes that marks the declaration of a and b in e, respectively. Entity a is declared in the body of entity b, and connected by the CONTAINS link $b \rightarrow a$ in GDN, if B is the first universal node from the U_{Phase_1} set found on the backwards path connecting A with the root of e.

Phase 2 creates the rest of GDN links, which means that in this phase SNEIPL identifies horizontal dependencies. The extraction of horizontal dependencies is much harder task than the extraction of vertical, CONTAINS links. In order to deduce horizontal dependencies identifiers have to be matched with their definitions. SNEIPL realizes the name resolution algorithm based on information contained in import statements (marked with the IMPORT_DECL universal node), lexical scoping rules (BLOCK_SCOPE and universal nodes in U_{Phase_1} reflect different lexical scopes), and rapid type analysis that is adopted for the eCST representation. Rapid type analysis [53] is the extension of class hierarchy analysis [54] that takes class instantiation information into account. Class hierarchy analysis is the name resolution algorithm based on the hierarchy-tree representation extended with inheritance relations between classes. EXTENDS, IMPLEMENTS, and INSTANTIATES universal nodes in the eCST representation enable that rapid type analysis can be adopted for the eCST representation. Since the extraction of horizontal dependencies is the most critical and complex part of GDN extraction, the most important details of this procedure are discussed in Subsection 6.1.

Figure 5 ilustrates the extraction of GDN when two simple eCSTs are provided as the input. The first eCST represents a compilation unit which defines the class-level entity A contained in the package-level entity P. A declares the global variable b whose type is the class-level entity B imported from the second eCST. The declaration of software entities P, A, Q, B and b are recognized in Phase 1 of GDN extraction and appropriate GDN nodes are created. From the structure of U_{Phase_1} universal nodes in eCSTs CONTAINS links are induced, and fully qualified names are assigned to each GDN node. The TYPE universal node in the first eCST indicates that there is a potential REFERENCES link between A and some other class-level entity. The name of the TYPE universal node in the first eCST is matched with the CUD universal node in the second eCST (according to the import statement in the first eCST, since the definition of B is not present in the first compilation unit), and the REFERENCES link $P.A \rightarrow Q.B$ is established. The parent of the TYPE universal node (ATTRIBUTE_DECL) gives the context in which B is referenced by A: b is the global variable in A which means that A aggregates B. Since A and B are contained in P and Q respectively,

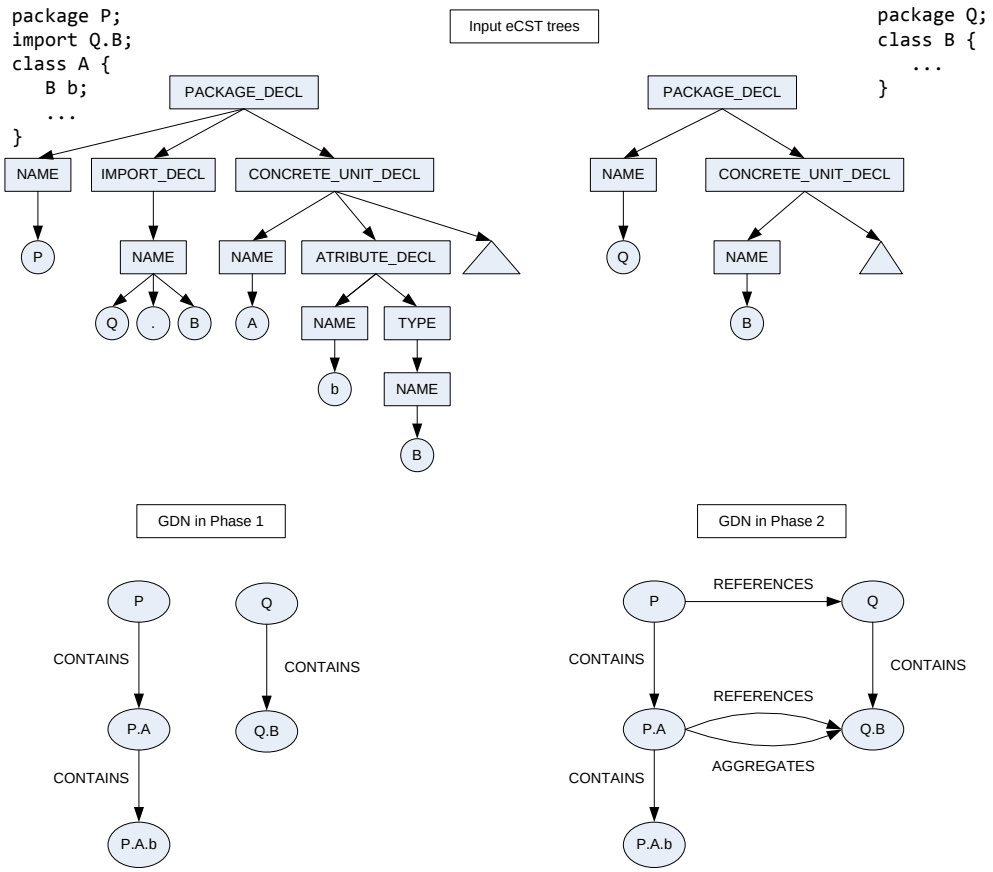

Figure 5: Two phases in GDN extraction: Phase 1 forms hierarchy tree while Phase 2 creates horizontal dependencies.

the REFERENCES link between package-level entities P and Q is induced from the REFERENCES link $P.A \rightarrow Q.B$.

GDN Filter takes extracted GDN and executes a sequence of parameterized "Select NT Connected by LT" queries to isolate software networks. Parameters NT and LT specify node and link types, respectively. For example, the query "select {FD} connected by {CALLS}" forms a static call graph, while the query "select {CUD, IUD} connected by {REFERENCES}" isolates a class/module collaboration network. Table 2 shows the parametrization of queries that are executed by GDN Filter.

6.1. Extraction of horizontal dependencies

The extraction of horizontal dependencies in the second phase of GDN extraction is based on the following principles:

- Horizontal dependencies between class-level entities are determined before horizontal dependencies between other types of software entities. This principle enables rapid type analysis when resolving horizontal dependencies between function-level entities, because EXTENDS and IMPLEMENTS relations among class-level entities are already identified.
- Horizontal dependencies between function-level entities are resolved in the bottom-up manner: function calls that appear as arguments of other function calls are evaluated first. When a FUNCTION_CALL subtree is evaluated it is rewritten by a single node which contains the return type of called function.

Table 2: Software networks extracted by SNEIPL and the parameterization of "select-connected by" queries.

Software network	**Select**	**Connected by**
Package collaboration network	PD	REFERENCES
Class collaboration network	CUD, IUD	REFERENCES
Static call graph	FD	CALLS
FUGV network	FD, AD	USES
Aggregation network	CUD, IUD	AGGREGATES
Weak aggregation network	CUD, IUD	WEAK_AGGREGATION
Inheritance network	CUD	EXTENDS
Bipartite network of implemented interfaces	CUD, IUD	IMPLEMENTS
Instantiate network	CUD	INSTANTIATES
Parameter type network	CUD, IUD	PARAMETER_TYPE
Return type network	CUD, IUD	RETURN_TYPE
Hierarchy network	PD, CUD, IUD, FD, AD, TD	CONTAINS

- Horizontal dependencies between function-level entities can induce additional, "hidden" horizontal dependencies between class-level entities. Those are dependencies induced from function calls and statements in which a global variable from some other compilation unit is accessed. In both cases appropriate entities are referenced by fully qualified names, which means that they are not explicitly imported,

- Horizontal dependencies between package-level entities are directly induced from horizontal dependencies between class-level entities.

In order to match an identifier with its definition SNEIPL internally maintains two data structures: import list and symbol space.

Import list is a list of GDN nodes that represent imported (visible) names declared outside an eCST that is currently processed. There is one import list per eCST (compilation unit). The import list is populated by the analysis of subtrees rooted at IMPORT_DECL universal nodes. Software entities declared in the scope of one PACKAGE_DECL are mutually visible without explicit import statements. Those entities are automatically added to the import list using the hierarchy tree formed in the first phase of GDN extraction.

An identifier marked with TYPE universal node represents some class-level entity. The hierarchy tree extracted in Phase 1 is used to determine if the class-level entity corresponding to the type identifier is declared in the same eCST. If the type identifier is not declared in the currently processing eCST then the type identifier is matched against the import list in order to determine the corresponding GDN node (if exists). The TYPE universal node also indicates that there is a horizontal dependency between the GDN node corresponding to the first enclosing class-level universal node (CUD, IUD) and the GDN node corresponding to the type identifier. Thus, REFERENCES links between class-level entities are created by the analysis of of eCST subtrees rooted at TYPE universal nodes. The parent of TYPE universal node determines the form of coupling between two class-level entities. It is important to notice that class-level REFERENCES links are not established by connecting class-level entities declared in the compilation unit with all entities contained in the import list. This means that SNEIPL discards unused imports. Also, it is possible to have definitions of two or more class-level entities in one eCST (for example, two or more Java classes can be defined in one .java file). In other words, different class-level entities may share the same import list. The analysis of eCST subtrees rooted at TYPE universal nodes ensures that the REFERENCES link between class-level entity A and imported class-level entity B is created if and only if B is referenced in the body of A.

SNEIPL attaches a local symbol table to each BLOCK_SCOPE and FUNCTION_DECL universal nodes in eCST. Local symbol table is the list of tuples (name, type) describing local variables declared in the scopes that are determined by the mentioned eCST universal nodes. They are created by the analysis of subtrees rooted at VAR_DECL and PARAMETER_DECL universal nodes. Variables contained in those subtrees are added to the local symbol table of the first enclosing BLOCK_SCOPE or FUNCTION_DECL universal node.

Each identifier introduced in the source code will be located either in some of local symbol tables or in the hierarchy tree extracted in Phase 1. Thus, local symbol tables together with the set of GDN nodes formed in Phase 1 constitute the symbol space structure of the whole program. Symbol space is searched during the analysis of subtrees rooted at NAME and FUNCTION_CALL universal nodes. NAME subtrees are analyzed to identify USES links between functions and global variables. The analysis of FUNCTION_CALL subtrees yields to the creation of CALLS links between functions. The search of symbol space is used when it is necessary to determine the following:

- if some variable is locally declared when there is the global variable (ATTRIBUTE_DECL) with the same name,
- the class or package-level entity which defines a function,
- the type of a object calling a function, or
- the type of a variable that is passed as the argument to a function call in case that the function call can not match the function definition relying solely on the function name and the number of arguments.

Let v be an arbitrary NAME universal node in some eCST. The search of the symbol space starts with the local symbol table attached to the first enclosing BLOCK_SCOPE universal node with respect to v. If the name marked with v is not found in the current local symbol table, then the symbol table attached to the next enclosing BLOCK_SCOPE is examined. In case that the symbol is not present in local symbol tables of all enclosing BLOCK_SCOPEs, the search is continued in the symbol tables attached to enclosing FUNCTION_DECLs in order to determine whether the symbol is a formal parameter of enclosing functions. This means that SNEIPL relies on basic lexical scoping rules when trying to match identifiers with their definitions. Additionally, the INSTANTIATES subtrees located under the universal node that marks the current scope are analyzed in order to check whether the type of v is contained in instantiate statements. If the identifier is not found in local symbol tables or instantiate statements search is continued using GDN. Starting from the CUD/IUD that declares the last enclosing FD, the search is backwards propagated according to the EXTENDS and IMPLEMENTS GDN links. For each visited GDN node, entities defined in the body of the node (accessible via GDN CONTAINS links) are inspected in order to check if there is a node whose name matches the name of v.

Implicit castings are handled by rapid type analysis (RTA) which assumes that variable y of type Y can be assigned to variable x ($x := y$) of type X, where X is supertype of Y (then Y is subtype of X). Let t denote variable of type T in CUD U on which function f is called, i.e. class-level entity U in one of its functions contains the function call statement "$t.f(...)$". RTA searches for the definition of f in super and subtypes of T which are instantiated in U and all CUDs directly or indirectly coupled to U. Since RTA is flow-insensitive and does not keep per-statement information there can be multiple targets for f after the analysis. In such cases SNEIPL does not create CALLS links in order to prevent Type I errors (creation of non-existent or false positive CALLS links). Supertypes of T are all GDN nodes reachable from GDN node representing T via EXTENDS or IMPLEMENTS GDN links. Consequently, if GDN node representing T is reachable via EXTENDS or IMPLEMENTS links from GDN node S then S is the subtype of T. Another situation relevant to implicit castings occurs after the call to f is matched with the definition of f. Then U references all types present in the list of formal arguments in the definition of f, since arguments in the call of f may be implicitly casted to the types requested by the definition of f.

Every type identifier is marked with TYPE eCST universal node. Therefore, the explicit cast of variable v to type T is represented by an eCST tree rooted at NAME universal node which contains two children: (1) token representing the name of variable v, and (2) the TYPE sub-tree containing the name of type T. The explicit cast statement, due to the existence of the TYPE universal node, causes the creation of the explicit class-level dependency between the class-level entity containing the cast statement and the class-level entity representing type T. Additionally, TYPE information in the NAME sub-tree determines the type of variable v when the explicit cast is the part of a function call statement.

In programming languages that support function overloading it is possible that a function call can not be uniquely matched with the definition of called function using only the name and the number of arguments. In

such cases SNEIPL tries to determine the type of each argument in order to select the appropriate definition from a set of candidates that are obtained by rapid type analysis. However, this process may result in unresolved types for arguments if an argument itself is the call to a function imported from a library, and consequently not present in analyzed eCSTs. In case that the successfully resolved types of arguments do not contain enough information to choose the right candidate, CALLS link can not be created. This means that SNEIPL extracts *optimistic call graphs* where non-existent (false positive) CALLS links are not present, but missing (false negative) CALLS link can occur. The typical example is illustrated by the following Java class:

```
class UnmatchedCallDef {
  void f(int a)    { }
  void f(String b) { }
  void caller() {
    f(Integer.parseInt("15"));
  }
}
```

In the example, we can see that the method `caller` calls the method `f`, but the argument of the call can not be resolved (`Integer.parseInt` is the method from the standard Java library). This means that SNEIPL has two candidates for the destination node of existing CALLS link, but can not determine which of them is the right one.

SNEIPL currently extracts only direct function calls, i.e. those calls where the name of the function is used to reference the function. Indirect function calls via function pointers or variables of procedural data types are not yet supported. Those features, when they exist in a language, are mostly extensively used in system programming, and have to be taken into account when extracting call graphs for the optimization tasks done by compilers. The aim of SNEIPL is to extract architectural graph representations of software systems that can be used for software engineering purposes: in the statistical analysis of design complexity of software systems, computation of design software metrics, and to serve humans which want to get insights into the internal organization of systems under investigation. As pointed out in [55], the requirements placed on tools that compute call graphs for software engineering purposes are typically more relaxed than for compilers, and those tools usually ignore rarely used language features which drastically increase the complexity of static code analysis.

7. Extraction of software networks from real-world software systems

The first extraction of software networks using SNEIPL was described in [43]. Namely, we designed two small programs written in different programming languages (Java and C#) that have the same requirements specification (administration of typical student activities) and the same design, i.e. the same set and structure of software entities. Then we employed SNEIPL to obtain software networks representing software systems under the investigation and compared them. The conclusion derived from the comparison is that SNEIPL extracts the same networks up to isomorphism from structurally and semantically equivalent software systems written in different programming languages. In other words, the experiment in [43] showed that eCST is a suitable representation for the language-independent extraction of software networks.

The aim of experiments conducted in this paper are different: we want to demonstrate that SNEIPL is able to identify dependencies in real-world software systems written in different programming languages which belong to different language paradigms. Therefore, in this paper we present and discuss software networks extracted from the following software projects written in Java, Modula-2, and Delphi (two projects per language):

- Commons-IO[2] (CIO), an open-source Java library of utilities to assist with developing IO functionality,
- Apache Tomcat[3], an open-source web server and Java servlet container written in Java,

[2] http://commons.apache.org/io/
[3] http://tomcat.apache.org/

- Modula-2 Algebra System[4] (MAS), an open-source computer algebra system written in Modula-2,
- Lumos[5], an open-source operating system for a computer called Stride 440 written in Modula-2,
- Model Scene Editor (MSE)[6], an open-source 3D scene editor written in Delphi,
- A proprietary, database-oriented Delphi application which realizes management, accounting and reporting functionalities for a company employing direct sellers organized into a multi-level marketing compensation hierarchy (we will use the term "DelPro" to denote this software). One of the authors of this paper took a part in the development of DelPro. Due to the familiarity with this software, we were in the position to verify that SNEIPL produces meaningful results when it is employed to identify dependencies in a large-scale software system.

It can be seen that our experimental corpus consists of both open-source and proprietary software systems. Three of six products (MAS, Lumos and MSE) can be considered as "orphaned" software since they are not maintained anymore. Software systems written in Java were selected randomly from the list of Apache Software Foundation open-source projects. Additionally, software networks associated with eight more Java software systems from the same list are extracted for the purpose of the comparative analysis presented in the next section of the paper. The Modula-2 projects from the corpus are the largest two open-source Modula-2 programs listed in Free Modula-2 Pages web site[7]. Selected Delphi projects are compatible with Delphi 6 which is the dialect of Delphi currently supported by eCST Generator (SSQSA component which forms the eCST representation). Four projects from the corpus (CIO, Tomcat, MAS, DelPro) are products of a team effort, while the other two (Lumos and MSE) are one-man projects. Table 3 summarizes software systems used in the experiment, and for each system shows the number of lines of code (LOC), the number of eCSTs produced by eCST Generator (this number is equal to the number of compilation units in the source code distribution), and the total number of eCST nodes in produced eCSTs. It can be seen that for each programming language we have one large size (more than 10^5 LOC) and one medium size (more than 10^4 LOC) software system in the corpus.

Table 3: The summary of software systems used in the extraction experiment.

Software system	**CIO**	**Tomcat**	**MAS**	**Lumos**	**DelPro**	**MSE**
Version	2.4	7.0.29	1.01	2	-	0.13
Language	Java	Java	Modula-2	Modula-2	Delphi	Delphi
LOC	25663	329924	100546	37250	104438	41858
#eCSTs	103	1083	329	297	491	113
#eCST nodes	88063	1650355	824043	297095	1151923	466061

Table 4 summarizes the properties of extracted General Dependency Networks for software systems from the corpus. Links representing self references are excluded from the counts. The distribution of GDN nodes per type shows us how many nodes will appear in a particular software network. For example, GDNs extracted from Commons IO and MSE contains 6 and 113 nodes, respectively. Those nodes correspond to different PACKAGE_DECL (PD) universal nodes in appropriate eCST representations. With PD universal nodes are marked declarations of packages in Java and units in Delphi code (there is no Modula-2 entity type that corresponds to PD universal node, see Section 4). Therefore, the package collaboration network associated with Commons IO contains 6 nodes that represent 6 different Java packages, while the unit collaboration network associated with MSE contains 113 nodes that represent 113 different Delphi units.

[4]http://krum.rz.uni-mannheim.de/mas/
[5]http://www.uranus.ru/download/lumos.zip
[6]http://mse.sourceforge.net/
[7]http://freepages.modula2.org/

Table 4: The number and distribution of nodes and links in extracted General Dependency Networks.

Software system	CIO	Tomcat	MAS	Lumos	DelPro	MSE
#nodes	1518	24287	6857	4104	13721	8359
PD - packages/units	6	97	0	0	491	113
CUD - classes/imp. modules	104	1351	163	193	501	156
IUD - interfaces/def. modules	4	143	166	115	0	22
TD - other user-defined types	0	21	66	293	43	1061
AD - class attrs./global vars.	328	7364	1128	1475	9846	4252
FD - methods/functions	1076	15311	5334	2028	2840	2755
#links	3001	71093	31558	12174	18267	11630
vertical dependencies	1517	24270	6528	3807	13230	8246
horizontal dependencies	1484	46823	25030	8367	5037	3384
package-level dependencies	9	493	0	0	895	268
class-level dependencies	341	13962	3024	1559	965	692
method-level dependencies	1134	32368	22006	6808	3177	2424
calls dependencies	611	20831	17456	3738	1522	772
access dependencies	523	11537	4450	3070	1655	1652

7.1. Vertical dependencies

From extracted GDNs SNEIPL form the set of software networks at different levels of abstraction, thus providing different granularity views of the organizational structure of software systems under investigation. Vertical dependencies (CONTAINS links) reflect the hierarchy of software entities, and together with the set of GDN nodes constitute hierarchy tree view of analyzed source code. Characteristics of hierarchy networks for examined systems are shown in Table 5. IN_0 denotes the number of nodes whose in-degree is equal to zero. Those nodes represent software entities which are not contained in other entities. Hierarchy network is a disjoint union of hierarchy trees, and each zero in-degree node is the root of one hierarchy tree. In the case of Java software systems zero in-degree nodes are root packages (packages not contained in other packages), in Modula-2 systems they represent non-local (non-nested) modules, while in Delphi systems each unit is one zero in-degree node (Delphi units can not be nested). On the other side, nodes having out-degree equals to zero (OUT_0) are located on the periphery of the hierarchy network, i.e. those nodes do not define other entities. From the data presented in Table 5, it can be seen that the majority of software entities are zero out-degree nodes (from 92.49% in CIO to 95.22% in MAS). All global variables (class attributes) are zero out-degree nodes, as well as all functions (methods) that do not define nested function- or named class-level entities. The most voluminous package in all examined software systems is `DirectX`, Delphi unit from MSE project. This unit comprises 22 Delphi classes, 118 procedures and 2208 global variables (most of them are constants). At the same time this unit is the largest compilation unit in the corpus: it consists of 11,014 LOC which is 26.31% of the total LOC in MSE. The most voluminous class/module in the corpus is `StandardContext`, Java class from Tomcat which is located in `org.apache.catalina.core` package. The mentioned class defines one inner class and encompasses 306 methods and 127 class attributes. This class is also the largest class in Tomcat having 6,523 lines of code.

7.2. Package-level dependencies

Horizontal dependencies connect software entities appearing at the same level of abstraction. At the highest level of abstraction we have dependencies between package-level entities (packages in Java and units in Delphi). Table 6 shows characteristics of extracted package collaboration networks (PCNs) for Java and Delphi systems from the corpus. The number of isolated nodes (nodes having zero total-degree) in a PCN tells us how many packages/units in the corresponding software system do not reference and are not being referenced by other packages defined in the source code. Table 6 also contains information about the packages with the maximal value of Robert Cecil Martin's afferent and efferent coupling metrics (MaxAC and MaxEC, respectively). For example, `Scene` is the most reused Delphi unit in MSE (referenced by 34 other units), while unit `Main` has the highest degree of aggregation of other units (it references 25 other units).

Table 5: Characteristics of extracted hierarchy networks: #nodes - the number of nodes, #links - the number of links, IN_0 - the number of nodes without in-coming links, OUT_0 - the number of nodes without out-going links, UPP - the average number of units per package, FPU - the average number of functions per unit, and VPU - the average number of global variables per unit.

Software system	**CIO**	**Tomcat**	**MAS**	**Lumos**	**DelPro**	**MSE**
#nodes	1518	24287	6857	4104	13721	8359
#links	1517	24270	6528	3807	13230	8246
IN_0	1	17	329	297	491	113
OUT_0	1404	22694	6529	3808	12756	7978
UPP	17.166	11.525	0	0	1.02	1.575
FPU	9.962	10.24	16.212	6.584	5.522	7.87
VPU	3.037	4.924	3.428	4.788	18.261	10.702

Table 6: Characteristics of extracted package collaboration networks: #nodes - the number of nodes, #links - the number of links, #isol - the number of isolated nodes, MaxAC - the highest value of in-degree (afferent coupling), MaxEC - the highest value of out-degree (efferent coupling).

Software system	**CIO**	**Tomcat**	**DelPro**	**MSE**
#nodes	6	97	491	113
#links	9	493	895	268
#isol	0 (0%)	1 (1.03%)	21 (4.28%)	6 (5.31%)
MaxAC	5	58	169	34
MaxAC name	io	juli.logging	AmcCountrySP	Scene
MaxEC	2	30	144	25
MaxEC name	io.output	catalina.core	MainAmcBS	Main

7.3. Class-level dependencies

At the middle level of abstraction there are dependencies between class-level entities: classes and interfaces in Java and Delphi, and definition and implementation modules in Modula-2. In class (module) collaboration networks all parallel links denoting different coupling types between two classes (modules) are reduced to one link, i.e. different coupling types between two nodes are recorded as attributes of one REFERENCES link. The characteristics of extracted class collaboration networks for software systems from the corpus are given in Table 7. The table also provides the information about the fraction of isolated nodes. It can be observed that for Tomcat, MAS and Lumos the fractions of isolated nodes are very low (less than 2% of the total number of classes/modules), suggesting that in those systems unused ("dead") code is reduced to the minimum. For other systems, isolated nodes do not necessarily point to unused code. In the case of libraries, isolated nodes can denote simple utility classes directly available to programmers. The example of such isolated class is `io.CopyUtils` from CIO. The mentioned class provides the set of static methods for copying files and relies only on JDK classes from `java.io` package. To the contrary, for standalone user applications, such as DelPro and MSE, it is more likely that isolated classes indicate unused or unfinished code. The examples of such classes in MSE are `TSplashScreen`, `TRegisterDialog` and `THelpIndexDialog`. The mentioned classes declare only Delphi visual components as class attributes without corresponding event handler methods, and their names clearly suggest that they represent non-core features planned to be introduced in one of the future releases.

Table 7 also shows classes/modules with the highest values of in-degree (MaxIn) and out-degree (MaxOut) in corresponding class collaboration networks (CCN). The in-degree of a class is the number of other classes referencing the class, and it reflects the degree of internal reuse of the class. To the contrary, the number of out-going dependencies (out-degree) reflects the degree of internal aggregation of other classes in the source code. In case of Delphi programs classes with the largest out-degree (`TFMainAmcBS` from DelPro and `TMainForm` from MSE) are at the same time classes having the largest total-degree (sum of in- and out-degree), i.e. the highest value of Chidamber-Kemerer CBO coupling metric. To the opposite, classes/modules present in Java and Modula-2 projects from the corpus that have the largest CBO are at the same time the most internally reused entities.

Table 7: Characteristics of extracted class/module collaboration networks: #nodes - the number of nodes, #links - the number of links, #isol - the fraction of isolated nodes, MaxIn - class/module having the highest in-degree, MaxOut - class/module having the highest out-degree (the exact values of in- and out- degrees are given in brackets).

Software system	#nodes	#links	#isol (%)	MaxIn	MaxOut
CIO	108	174	15.74	AbstractFileFilter (19)	FileFilterUtils (16)
Tomcat	1494	6839	1.67	Log (293)	StandardContext (73)
MAS	329	2054	0.91	MASStor (277)	RqePRRC (36)
Lumos	308	973	1.94	R2SysCalls (78)	L2SysCalls (25)
DelPro	501	770	5.58	TAmcCountry (57)	TFMainAmcBS (145)
MSE	178	343	6.74	TShape (35)	TMainForm (57)

7.4. Function-level dependencies

At the lowest level of abstraction SNEIPL identifies function-level dependencies: CALLS links between functions and ACCESS links between functions and global variables. Table 8 presents the characteristics of extracted static call graphs for software systems from the corpus, while Table 9 shows the functions having the highest in- and out-degree. The in-degree of function A in the SCG is the number of other functions that call A, while out-degree stands for the number of function that are called by A. For example, method `isDebugEnabled` from class `Log` is called by 429 other methods defined in Tomcat, while method `startInternal` from class `StandardContext` calls 64 other methods.

Table 8: Characteristics of extracted static call graphs/method collaboration networks.

Software system	CIO	Tomcat	MAS	Lumos	DelPro	MSE
#nodes	1076	15311	5334	2028	2840	2755
#links	611	20831	17456	3738	1522	772
#isolated (%)	43.77	30.49	43.4	37.33	59.05	75.97
#calls resolved (%)	88.47	95.46	100	100	100	99.05
#hard to match (%)	30.67	7.48	0	0	0	0.11
hard to match resolved	132	832	-	-	-	0
hard to match unresolved	102	1494	-	-	-	9

Table 9: Functions with the maximal values of in- and out- degree in extracted static call graphs.

Software system	MaxIn		MaxOut	
CIO	Charsets.toCharset	24	DirectoryWalker.walk	7
Tomcat	Log.isDebugEnabled	429	StandardContext.startInternal	64
MAS	MASStor.ADV	1132	MASLoadE.InitExternalsE	128
Lumos	R2SysCalls.WriteString	121	L2SysCalls.InitPr	100
DelPro	AmcCountrySP.UpdateSQLWithSchema	257	TFActivityHandler.DoActivity	24
MSE	Misc.SaveStringToStream	16	TSceneData.MouseDown	12

In case of Modula-2 programs nodes in a SCG represent function declarations in definition modules and function definitions in corresponding implementation modules. Since Modula-2 does not have function overloading and inheritance features (Modula-2 functions can not be overridden, neither Modula-2 function calls are dynamically dispatched), a Modula-2 function call is always matched with the definition of called function in the implementation module. In other words, isolated nodes in extracted Modula-2 SCGs represent either function declarations in definition modules or unused functions in implementation modules. The SCG nodes representing functions from definition modules can be easily pruned from the SCG (they are attached to CONTAINS links emanating from GDN nodes whose type is IUD eCST universal node), thus leaving only unused functions from implementation modules as isolated nodes in the SCG.

For object-oriented languages, due to function overloading and overriding, some function calls may be unresolved by SNEIPL (see Section 6.1). In such cases the rapid type analysis realized by SNEIPL's function

call resolver results in multiple function definitions as destination candidates (targets) for a single function call. All candidates represent functions with the same name and the same number of formal parameters. We call such functions *hard to match*. The number of hard to match functions can be easily determined: a function is hard to match if there is at least one function with the same name and the number of arguments in the class (overloaded functions) or in one of super or subclasses (both overloaded and overridden functions). Table 8 shows the fraction of hard to match functions for systems from the corpus, as well as the fraction of resolved function calls. A function call is resolved when there is exactly one candidate in the candidate list for the destination function definition after rapid type analysis. Naturally, due to the absence of dynamic binding, there are no hard to match functions in Modula-2 systems, and each function call is properly resolved. The number of hard to match functions in DelPro is equal to zero, which means that all functions present in this software are unique up to name and the number of formal parameters. In other systems, due to the existence of hard to match functions, there are unresolved function calls. The upper bound of missing CALLS links in extracted SCGs is equal to the number of unresolved function calls: for CIO unresolved function calls create maximally 102 CALLS links, for Tomcat 1494, and for MSE maximally 9 CALLS links. Table 8 also provides the information about the number of resolved calls to hard to match functions. In such cases, the candidate list for a function call contains multiple targets representing overloaded functions, and SNEIPL was able to reduce the list of targets after the evaluation of argument types.

Missing calls links may cause isolated nodes in a SCG. When this is not the case an isolated node in SCG does not necessarily represent unused function. For example, Delphi methods from the user-interface layer in GUI applications are event handlers (methods activated in response to user actions), and never explicitly called by other methods in the source code. Also, isolated nodes can represent methods that are dynamically invoked through reflection mechanisms of a language such as Java Reflection API (those are run-time dependencies, and not static compile-time dependencies). Another case is that those methods represent call-back methods used by the standard language library or third-party libraries. For example, Java classes usually override `Object.toString` method which is the classical example of a call-back method: `toString` is usually never called by other methods in the source code, but from methods in the Java class library, or when an instance of the class is the argument in a string concatenation operation. Another frequent example is `compareTo` method present in all classes implementing `Comparable` interface: this method is usually called only from classes in the JDK collections framework.

8. Comparative analysis

In order to investigate the correctness and completeness of the dependency extraction procedure realized by SNEIPL, we extracted class collaboration networks representing ten real-world, open-source, and widely used software systems written in Java (two of them are already used as case studies in the previous section), and compared them to the class collaboration networks extracted by a language-dependent tool – Dependency Finder[8] version 1.2.1, and a language-independent tool – Doxygen[9] version 1.8.5. The characteristics of software systems used in the comparative analysis are summarized in Table 10. In the comparative analysis only Java systems are examined: to the our best knowledge SNEIPL is the only currently available dependency extractor able to process Modula-2 source code, while for Delphi other class dependency extractors are commercial, closed-source products. However, the fact that only Java systems are examined in the comparative analysis does not limit the generalizability of its results, because SNEIPL extracts dependencies from the standardized, language-independent representation of the source code. Also, it is important to notice that the comparative analysis covers class-level dependencies (dependencies at the middle level of abstraction) and not package and function-level dependencies. This also does not limit the generalizability of the analysis, since package-level dependencies are completely induced from class-level dependencies, while function-level dependencies completely induce implicit class-level dependencies.

[8] http://depfind.sourceforge.net/
[9] http://www.stack.nl/~dimitri/doxygen/

Table 10: Java software systems used in the comparative analysis.

Software system	Version	LOC	Short description
CommonsIO	2.4	25663	IO library
Forrest	0.9	4683	Web publishing framework
PBeans	2.0.2	8502	Object/relational database mapping framework
Colt	1.2.0	84592	High performance scientific computing library
Lucene	3.6.0	111763	Text search engine library
Log4j	1.2.17	43898	Java logging library
Tomcat	7.0.29	329924	Web server and servlet container
Xerces	2.11.0	216902	XML parser library
Ant	1.9.2	219094	Build tool
JFreeChart	1.0.17	226623	Chart creator

8.1. Characteristics of tools used in the comparative analysis

Dependency Finder is an open-source dependency graph extractor for Java with positive recommendations from professionals coming both from industry and academia.[10] Dependency Finder extracts Java class dependencies from Java bytecode. It is able to read all types of compiled Java: JAR files, zip files, or class files. Class dependencies are gathered by a handmade Java bytecode parser which traverses the structure of a class file and collects explicit class dependencies. Implicit dependencies between classes are formed through the link maximizer algorithm which induces class-level dependencies from method-level dependencies.

Doxygen is a documentation generator tool that supports more than ten programming languages, including support for Java. Doxygen was already used to extract CCNs from real-world software systems written in different programming languages, that were later analyzed under the framework of complex network theory [1, 4]. Doxygen was also used as dependency extractor in empirical investigations of architecture and reusability of open-source software systems [56, 57, 58], as well as in research works dealing with the prediction of vulnerable software components [59] and software validation [60].

Doxygen can be configured to extract local class collaboration graphs and export them in the dot[11] file format. Local class collaboration graphs show direct and indirect inheritance and aggregation dependencies for individual classes, so we wrote a simple dot aggregation tool that incrementally builds CCN from a set of dot files. The extraction of local class collaboration graphs in Doxygen for programs written in C, C++, C#, Objective-C, Java, JavaScript, D, PHP and IDL is based on the unified fuzzy parsing approach. This means that there is one light-weight parser for all mentioned languages realized as a big state machine generated by Flex lexical analyzer generator. The parser converts non-skipped parts of a given source code into a tree of entries which is then analyzed by Doxygen's Data organizer component. Each entry is a blob of loosely structured information and contains the special field called "section" which specifies the kind of information contained in the entry. Data organizer builds dictionaries from extracted entries for the purpose of generating documentation, and during this step dependencies between entries are identified. For other languages supported by Doxygen (Tcl, Python and Fortran), the extraction of local class collaboration graphs is not language-independent. For those languages Doxygen has independent parsers and each of them realizes language specific extraction of local class collaboration graphs.

The similarity between Doxygen's unified fuzzy parsing approach and SNEIPL is that in both approaches there is a language-independent intermediate representation of the source code which is used to identify dependencies between entities defined in the source code. The differences are in the used intermediate representations and in the ways they are formed. Doxygen converts analyzed source code into a tree of entries that is formed by lexical analysis, and achieves language independence using one fuzzy parser for several languages. Since the intermediate representation used by Doxygen is formed by lexical analysis, the unified fuzzy parsing approach to dependency extraction has two major disadvantages:

[10] The recommendations can be found on the web site of the tool.
[11] http://www.graphviz.org/doc/info/lang.html

- Doxygen is unable to form symbol tables that completely reflect nested scopes. As the consequence, Doxygen can not resolve dependencies associated with identifiers that do not have unique names. To prevent problems Doxygen ignores all of the classes with the same name except one[12].
- The extensibility of the unified fuzzy parser approach is restricted to a family of languages where the same concept is expressed by similar language constructs – similar in the sense that they can be reduced to the same lexical rule without loosing relevant information. For example, declaration of variables in Java and C can be reduced to the same lexical rule in which variable names can be distinguished from the associated type that is always the first token in the declaration. On the other side, the same construct in Modula-2 is described by completely different lexical rule, since the type is the last non-separator token in the declaration.

In comparison with Doxygen, SNEIPL uses much richer intermediate representation of the source code which enables the construction of symbol tables that fully reflect nested scopes. Secondly, the eCST representation is not produced by a unified, light-weight parser that restricts the extensibility to a certain family of languages (e.g., Pascal-like or C-like languages).

8.2. Network similarity measure

Nodes in a class collaboration network are identified by fully qualified class names. Therefore it is easy to match two nodes from two different networks representing the same class (different in the sense that they are formed by two different tools). Consequently, a link in a class collaboration link is uniquely identified by the fully qualified names of the source and destination class.

Since a network consists of a set of nodes and a set of links, comparing two networks is equivalent to comparing two sets of nodes and two sets of links. The Jaccard coefficient (also Jaccard index or Jaccard similarity measure) is a commonly used measure of the similarity between two sets. It is defined as the size of the intersection divided by the size of the union of the sets. The Jaccard coefficient J is a value in the range $[0, 1]$: $J = 0$ implies two disjoint sets, $J = 1$ denotes identical sets, and the higher value of J indicates the higher degree of overlap between sets.

In our comparative analysis we use two Jaccard coefficients, one expressing the similarity between two sets of CCNs nodes, and another showing the similarity between two sets of CCNs links. Let $\mathrm{CCN}_A = (N_A, L_A)$ and $\mathrm{CCN}_B = (N_B, L_B)$ denote two class collaboration networks extracted by tools A and B, respectively. With N_x and L_x are denoted sets of nodes and links respectively, in the CCN extracted by tool x ($x = A$ or $x = B$). To formally define Jaccard coefficients for nodes and links we use the following notation (we also use the same notation in the next subsection of the paper):

$\mathrm{MN}_{A,B}$	– the number of mutual nodes, i.e. nodes that appear in both CCN_A and CCN_B,
UN_A	– the number of nodes that are unique to CCN_A and do not appear in CCN_B,
UN_B	– the number of nodes that are unique to CCN_B and do not appear in CCN_A,
$\mathrm{ML}_{A,B}$	– the number of mutual links, i.e. links that appear in both CCN_A and CCN_B,
UL_A	– the number of links unique to CCN_A and do not appear in CCN_B,
UL_B	– the number of links unique to CCN_B and do not appear in CCN_A.

The Jaccard coefficient for nodes is defined as

$$\mathrm{JN}(A, B) := \frac{\text{number of mutual nodes}}{\text{total number of different nodes}} = \frac{|N_A \cap N_B|}{|N_A \cup N_B|} = \frac{\mathrm{MN}_{A,B}}{\mathrm{MN}_{A,B} + \mathrm{UN}_A + \mathrm{UN}_B}.$$

Similarly, the Jaccard coefficient for links is defined as

$$\mathrm{JL}(A, B) := \frac{\text{number of mutual links}}{\text{total number of different links}} = \frac{|L_A \cap L_B|}{|L_A \cup L_B|} = \frac{\mathrm{ML}_{A,B}}{\mathrm{ML}_{A,B} + \mathrm{UL}_A + \mathrm{UL}_B}.$$

[12] http://www.stack.nl/~dimitri/doxygen/manual/trouble.html

From the definition of the Jaccard coefficient for nodes (and the same is for links), it can be seen that this measure is sensitive to missing nodes from the perspective of both A and B – the denominator of the measure counts both nodes unique to CCN_A (nodes missing in CCN_B) and nodes unique to CCN_B (nodes missing in CCN_A). Additionally, if node x contained in CCN_A is missing in CCN_B then all links attached to x are missing in CCN_B and vice versa. Therefore, the Jaccard coefficient for links is sensitive to missing non-isolated nodes from the perspective of both A and B.

Let us assume that CCN_B is the 100% correct class collaboration network, i.e. CCN_B contains all classes and all class dependencies present in the corresponding software system. Then JN and JL quantify both the completeness and correctness of the node and link sets obtained by tool A. Namely, UN_B and UL_B represent the number of existent nodes and links respectively, that are not identified by tool A (missing nodes and links). The higher UN_B and UL_B imply the lower degree of completeness of results obtained by tool A. On the other hand, UN_A and UL_A represent the number of non-existent nodes and links respectively, that are created by tool A. Therefore, the higher UN_A and UL_A imply the lower degree of correctness of results obtained by tool A. In our comparative analysis we examine three different dependency extraction approaches where one is language-independent (Dependency Finder). Therefore, under the assumption that the language-dependent tool produces 100% correct results, JN and JL quantify the completeness and correctness of the two other language-independent approaches.

8.3. Results and discussion

Tables 11 and 12 summarize differences between class collaboration networks extracted using SNEIPL, Dependency Finder, and Doxygen. Both tables show the number of nodes ($|N_x|$) and links ($|L_x|$) in the CCN formed by tool x, the number of mutual nodes (MN) and mutual links (ML), the number of unique nodes (UN_x) and unique links (UL_x) with respect to tool x, and the values of the Jaccard coefficients for nodes (JN) and links (JL). It can be observed that the CCNs formed by SNEIPL are highly similar to those formed by Dependency Finder: for all analyzed systems, except for Tomcat, we have JN = 1.0 (identical sets of nodes), while JL is always higher than 0.9 implying highly overlapping sets of links (class dependencies). That is not the case with Doxygen where the maximal JL is equal to 0.41. As may be noted from the data presented in Table 12, the CCNs extracted by Doxygen are significantly smaller ($|L_B| \ll |L_A|$) proper sub-graphs ($\mathrm{UN}_B = 0 \wedge \mathrm{UL}_B = 0$) or close to proper sub-graphs ($\mathrm{UN}_B \ll \mathrm{UN}_A \wedge \mathrm{UL}_B \ll \mathrm{UL}_A$) of corresponding CCNs formed by Dependency Finder.

In case of Tomcat all classes identified by Dependency Finder are also identified by SNEIPL ($\mathrm{UN}_B = 0$), but SNEIPL identified seven classes more ($\mathrm{UN}_A = 7$). The same seven classes are also present in the CCN extracted by Doxygen (see UN_B value in Table 12). We manually verified that those classes exist in the Tomcat source code distribution. The analysis of the Ant script that is used to build Tomcat revealed that those classes are not the part of the Tomcat binary distribution, but belong to extra components (JMX Remote Lifecycle Listener and JSR 109 web services support).

From the data presented in Table 11, it can be observed that for all examined systems SNEIPL identified a small portion ($\mathrm{UL}_A \ll \mathrm{ML}$) of class dependency links which are not identified by Dependency Finder. For example, the Forrest CCN formed by DependecyFinder is a sub-graph of the Forrest CCN formed by SNEIPL ($\mathrm{UL}_A = 4$, $\mathrm{UL}_B = 0$), i.e. all classes and dependencies identified by Dependency Finder are also identified by SNEIPL, but SNEIPL identified 4 dependencies more. Those dependencies are represented by the following links:

locationmap.lm.AbstractNode	→ locationmap.lm.LocationMap
locationmap.RegexpLocationMapMatcher	→ locationmap.lm.LocationMap
locationmap.WildcardLocationMapMatcher	→ locationmap.lm.LocationMap
locationmap.WildcardLocationMapHintMatcher	→ locationmap.lm.LocationMap

It can be simply checked in the Forrest source code distribution that the above listed links represent existing dependencies. Class `LocationMap` is the destination node of all four links. Among other methods and attributes, `LocationMap` defines four public, static and final String attributes which are accessed by methods in `AbstractNode` and LocationMapMatcher classes. Dependency Finder was unable to identify the mentioned

dependencies simply because they do not exist in the bytecode. Namely, for a final String attribute, or a final attribute of some primitive type, the Java compiler inlines the value of the attribute directly into all client classes, so dependencies to the class which owns the attribute are lost. Another situation observed in our case studies when the translation from source to bytecode can lead to the loss of dependencies occurs when a dependency between two classes is caused solely by the existence of local variables whose type is the dependent class. Type information for local variables is not present in bytecode: the Java compiler validates assignments involving local variables and then discards information about their types.

Table 11: Similarity between class collaboration networks extracted by A = SNEIPL and B = Dependency Finder.

Software system	$\|N_A\|$	$\|N_B\|$	MN	UN_A	UN_B	**JN**	$\|L_A\|$	$\|L_B\|$	ML	UL_A	UL_B	**JL**
CommonsIO	108	108	108	0	0	**1.0**	174	174	173	1	1	**0.99**
Forrest	35	35	35	0	0	**1.0**	56	52	52	4	0	**0.93**
PBeans	58	58	58	0	0	**1.0**	143	144	140	3	4	**0.95**
Colt	299	299	299	0	0	**1.0**	1272	1280	1254	18	26	**0.97**
Lucene	789	789	789	0	0	**1.0**	3544	3606	3439	105	167	**0.93**
Log4j	251	251	251	0	0	**1.0**	883	853	839	44	14	**0.93**
Tomcat	1494	1487	1487	7	0	**0.99**	6839	6832	6512	327	320	**0.91**
Xerces	876	876	876	0	0	**1.0**	4775	4677	4517	258	160	**0.91**
Ant	1175	1175	1175	0	0	**1.0**	5521	5517	5345	176	172	**0.94**
JFreeChart	624	624	624	0	0	**1.0**	3218	3249	3208	10	41	**0.98**

Table 12: Similarity between class collaboration networks extracted by A = Dependency Finder and B = Doxygen.

Software system	$\|N_A\|$	$\|N_B\|$	MN	UN_A	UN_B	**JN**	$\|L_A\|$	$\|L_B\|$	ML	UL_A	UL_B	**JL**
CommonsIO	108	100	100	8	0	**0.93**	174	71	71	103	0	**0.41**
Forrest	35	33	33	2	0	**0.94**	52	21	21	31	0	**0.40**
PBeans	58	36	36	22	0	**0.62**	144	19	19	125	0	**0.13**
Colt	299	228	228	71	0	**0.76**	1280	263	263	1017	0	**0.21**
Lucene	789	637	637	152	0	**0.81**	3606	925	907	2699	18	**0.25**
Log4j	251	230	230	21	0	**0.92**	853	246	245	608	1	**0.29**
Tomcat	1487	1310	1303	184	7	**0.87**	6832	1707	1694	5138	13	**0.25**
Xerces	876	813	813	63	0	**0.93**	4677	1494	1494	3183	0	**0.32**
Ant	1175	1055	1055	120	0	**0.90**	5517	1406	1401	4116	5	**0.25**
JFreeChart	624	597	597	27	0	**0.96**	3249	792	792	2457	0	**0.24**

The translation of Java source to Java bytecode can lead to the loss of class dependencies, but also during the compilation new class dependencies that do not exist in the source code may be created. During the manual inspection of a portion of links that appear in CCNs extracted by Dependency Finder and do not appear in CCNs extracted by SNEIPL, we observed that the majority of them represent dependencies from a non-static inner class to the outer (enclosing) class. Also, we checked that in such cases the inner class does not make any reference to the outer class, i.e. it does not use any field or method defined in the outer class. However, for non-static inner classes the Java compiler always create the synthetic field called `this$0` which represents the reference to the instance of the outer class. In other cases, missing class dependencies are caused by missing CALLS links, i.e. a class dependency is solely caused by method calls, and SNEIPL was unable to resolve them. The precise quantification of missing calls links in networks extracted by SNEIPL for software systems used in the comparative analysis is given in Table 13.

To investigate the practical implications of the observed differences between CCNs extracted by SNEIPL, Dependency Finder, and Doxygen, we consider two perspectives: one associated with researchers interested in empirical investigations of design complexity of large-scale software systems, and another with practitioners interested in software metrics.

Researchers interested in the design complexity of real-world, large-scale software systems examine complementary cumulative degree distributions of software networks in order to determine the type of design

Table 13: Quantification of missing CALLS dependencies in networks extracted by SNEIPL: Calls resolved (%) – the fraction of resolved function calls, HTM – the fraction of hard to match functions in the source code, HTM resolved – the number of resolved calls to hard to match function, and HTM unresolved – the number of unresolved calls to hard to match functions.

Software system	**Calls resolved (%)**	**HTM (%)**	**HTM resolved**	**HTM unresolved**
CommonsIO	88.47	30.67	132	102
Forrest	100	0	-	-
PBeans	89.08	7.54	23	84
Colt	94.23	11.58	586	516
Lucene	97.92	8.19	271	206
Log4j	98.47	13.50	162	51
Tomcat	95.46	7.48	832	1494
Xerces	95.96	4.20	535	831
Ant	88.61	4.46	569	2512
JFreeChart	96.28	7.26	483	604

complexity of studied systems[1, 2, 4, 26]. Complementary cumulative degree distribution $CCD(k)$ is the probability of observing a node with degree greater than or equal to k in a CCN. Therefore, we investigated if there are statistically significant differences between CCDs computed from CCNs formed by SNEIPL, Dependency Finder, and Doxygen. The existence of statistically significant differences between two complementary cumulative distributions can be checked using the two sample Kolmogorov-Smirnov (KS) test [61]. The KS test is a non-parametric statistical procedure based on the D statistics which is the maximal vertical distance between tested distributions. The test checks the null hypothesis that there are no statistically significant differences between tested distributions in terms of their locations, spreads, and shapes. To perform KS tests we used an open-source Java library called JCS (Java Statistical Classes)[13]. The results of KS tests are summarized in Table 14. The null hypothesis is accepted if the obtained value of the significance probability (p) is higher than 0.05. It can be observed that for all examined systems there are no statistically significant differences between CCDs computed from class collaboration networks extracted by SNEIPL and Dependency Finder. In other words, the degree distribution analysis of CCNs obtained by SNEIPL and Dependency Finder would result in the same conclusion about the type of design complexity of corresponding software systems. On the other hand, statistically significant differences between CCDs computed from CCNs formed by Dependency Finder and Doxygen are present for all examined software systems, except for Forrest (the smallest examined system).

Table 14: Results of two-sample Kolmogorov-Smirnov tests: D – Kolmogorov-Smirnov statistics, p – the value of the significance probability. "Accepted" denotes if the null hypothesis (no statistically significant differences between distributions) is accepted or not.

	SNEIPL – DependencyFinder			**DependencyFinder – Doxygen**		
Software system	D	p	**Accepted**	D	p	**Accepted**
CommonsIO	0.009	0.99	**yes**	0.45	< 0.01	**no**
Forrest	0.085	0.99	**yes**	0.33	0.057	**yes**
PBeans	0.034	1.00	**yes**	0.72	< 0.01	**no**
Colt	0.013	1.00	**yes**	0.61	< 0.01	**no**
Lucene	0.022	0.98	**yes**	0.52	< 0.01	**no**
Log4j	0.051	0.89	**yes**	0.53	< 0.01	**no**
Tomcat	0.017	0.97	**yes**	0.50	< 0.01	**no**
Xerces	0.042	0.41	**yes**	0.44	< 0.01	**no**
Ant	0.007	1.00	**yes**	0.54	< 0.01	**no**
JFreeChart	0.008	0.99	**yes**	0.52	< 0.01	**no**

Since the degree of a node in a CCN is at the same the value of Chidamber-Kemerer CBO metric for the corresponding class, the degree distribution of CCN is at the same time the distribution of CBO values

[13]http://www.jsc.nildram.co.uk/

for all classes present in the source code. Therefore, the previous statistical analysis based on KS tests tells us also that there are no statistically significant differences between values of CBO metric when they are computed using CCNs extracted by SNEIPL and Dependency Finder. However, software engineers are usually not interested in the overall statistical properties of metric values, but want to know concrete values of CBO for classes present in a software system. Therefore, we examined the distribution of CBO differences when CBO is computed from CCNs extracted by SNEIPL and Dependency Finder. Results are presented in Table 15. As it can be seen, large CBO differences ($\triangle CBO \geq \pm 4$) occur very rarely (for less than 4% of the total number of classes). On the other hand, for more than 65% of the total number of classes in each examined system, the CBO obtained by SNEIPL has the same value as the CBO obtained by Dependency Finder ($\triangle CBO = 0$). Doxygen is not considered in the analysis of CBO differences, because the degree distributions obtained by Doxygen are significantly different from those obtained by Dependency Finder, which automatically implies large CBO differences. It is important to observe that CBO calculated from Java source code may be different than CBO calculated from Java bytecode. As pointed earlier, during the compilation some class dependencies may be lost due to inline optimizations, and at the same time new dependencies may be introduced. In other words, the CBO differences presented in Table 15 are not caused entirely by two different implementations of CCN extraction, but also by different sources for CCN extraction.

Table 15: The distribution of CBO differences ($\triangle CBO$) when they are calculated using CCNs extracted by SNEIPL and Dependency Finder.

Software system	0 (%)	± 1 (%)	± 2 (%)	± 3 (%)	$\geq \pm 4$ (%)
CommonsIO	96.3	3.7	-	-	-
Forrest	85.71	11.43	-	-	2.86
PBeans	82.76	13.79	3.45	-	-
Colt	81.61	16.05	0.33	0.67	1.34
Lucene	65.78	26.36	4.69	0.76	2.41
Log4j	76.1	19.92	1.59	0.8	1.59
Tomcat	65.37	23.13	5.45	3.43	2.62
Xerces	64.95	25.23	5.14	1.26	3.42
Ant	72	21.11	3.57	1.45	1.87
JFreeChart	90.54	7.21	1.28	0.64	0.32

9. Related work

There is a variety of software networks extractors, but in most cases they are tied to a particular programming language and extract just one type of software network. For example, the review of static call graphs extractors for C programming language can be found in [55]. In this section it will be discussed how software networks are extracted in research works dealing with the analysis of software systems under the framework of complex network theory, and in existing language-independent reverse engineering tools and environments.

9.1. Extraction of software networks for statistical analyses

In research works that deal with the statistical analysis of software systems under the framework of complex network theory, software networks are usually extracted using language-specific tools. For example, in [62, 27] networks are extracted by parsing C++ header files, in [26, 63] by parsing Java source code, in [64] by parsing Java class files, in [5] by using Java Doclet capabilities to inspect the source code structure, and in [65] by parsing JavaDoc HTML pages.

Usage of a language-specific software networks extraction tool naturally restricts the statistical study to software systems written in a particular language. However, the authors of [1] and [4] used basically the same extraction methodology based on Doxygen to form software networks associated with software systems written in different programming languages (C static call graphs and C++ class collaboration networks in [1];

Java package, class and method collaboration networks in [4]). In our comparative analysis we showed that software networks extracted using SNEIPL are far more accurate than those extracted with the help of Doxygen.

9.2. Software networks extraction in reverse engineering tools and environments

SNEIPL forms a General Dependency Network (GDN) from the eCST representation of the source code. GDN is a heterogeneous software network that contains all software entities defined in the source code, as well as various relations among them. This means that GDNs can be viewed as fact bases used in reverse engineering activities. Therefore, in this Section we review how fact bases are formed in widely used language-independent reverse engineering tools, environments, and frameworks. As we will see all reviewed systems provide a language-independent representation of fact bases, but perform language-dependent fact extraction. This makes them fundamentally different from SNEIPL which provides both language-independent fact extraction and language-independent representation of fact bases.

Rigi [32] is a reverse engineering environment that allows the visual exploration of software systems in the form of graphs showing software entities and their relationships. It offers the language-independent exchange format based on a graph-based data model, fact extractors for C, C++, and COBOL, and an interactive graph editor called Rigiedit. Rigi's graph-based data models are capable to represent architectural elements of software systems: program components (functions, global variables, etc.) and their relationships (calls to function, references to variables, etc.). Rigi's architecture decouples fact extractors from the graph editor via the exchange format. Rigi's fact extractors for C and COBOL are parsers built with the help of Yacc parser generator. Those parsers identify software entities and their dependencies in a source code and store extracted information in the textual exchange format known as RSF (Rigi Standard Format). For the reverse engineering of software systems written in other languages, users are expected to produce RSF files. The authors of Rigi advocate usage of lightweight fact extractors based only on lexical analysis (which produce imprecise, but useful fact bases) for analyses of legacy software systems, because those systems are often in the state that the source code can not be compiled (due to missing files) or contains syntax errors. However, there is no support in Rigi to build parser-based or lightweight fact extractors. In other words, Rigi is capable to analyze and visualize software networks representing software systems written in different programming languages but their extraction is not language-independent.

Moose [66] is a tool environment for reverse engineering and re-engineering of object-oriented software systems. It consists of a repository to store language-independent models of software systems, and provides query and navigation facilities. Moose models are instances of the FAMIX meta-model and capture architectural elements of software systems: defined entities (classes, methods, attributes, etc.) and their dependencies (inheritance, invocation, access and reference). In other words, Moose operates on software networks and it is capable to visualize them in various forms. There are two ways to form Moose models. In case of Smalltalk fact extraction is performed via built-in parser. For other languages, Moose provide an import interface for CDIF and XMI files. Over this interface Moose uses external parsers for languages other than Smalltalk. However, Moose, similarly as Rigi, does not support language-independent fact extraction: each parser independently recognizes entities and relationships in order to instantiate the FAMIX meta-model with concrete information about software systems.

Gupro [67] is an integrated workbench to support program understanding of heterogeneous software systems on different levels of granularity. In Gupro software artifacts are stored in a graph repository which reflects relationships between defined software entities, and abstraction is done by graph queries. The extraction of information is done by parsers generated using the PDL parser generator. PDL extends the Yacc parser generator by EBNF syntax and notational support for compiling textual languages into TGraphs. TGraphs are directed graphs whose nodes and edges may be attributed, typed and ordered. Those graphs are used to conceptually represent software systems: software entities are represented by nodes, relationships among entities by edges, a common type is assigned to similar objects and relationships, and ordering of relationships is expressed by edge order. TGraphs are produced by individual PDL parsers and consequently the fact extraction in Gupro is not language-independent.

Bauhaus [68] is a tool suite that supports program understanding and reverse engineering on all layers of abstraction, from the source code to the architecture. It is capable to analyze programs in Ada, C,

C++ and Java. In Bauhaus two separate program representation exist: InterMediate Language (IML) and Resource Flow Graph (RFG) representation. The IML representation is defined by the hierarchy of predefined classes, where each class represents a certain universal programming language construct. IML is generated from the source code by a language-specific front-end. While IML represent the system on a very concrete and detailed level, the architectural aspects of the system are modeled by means of RFG. An RFG is a hierarchical graph that consists of typed nodes and edges. Nodes represent architecturally relevant elements of software systems (routines, types, files, components, etc.). Different aspects of the architecture (call graph, hierarchy of modules, etc.) can be obtained using different granularity views. In other words, RFG is, similarly as GDN, a union of software networks at different levels of abstraction. For C and C++, an RFG is automatically generated from the IML representation, whereas for other languages RFG is generated from other intermediate representations (such as Java class files) or compiler supported interfaces (such as Ada Semantic Interface Specification). Therefore, fact extraction in Bauhaus is not fully language-independent, since RFGs for some languages are not formed directly from IML.

Compared to described language-independent reverse engineering tools and environments, SNEIPL provides language-independent fact extraction, since the extraction of software networks is solely based on the eCST representation. In other words, extraction of software networks in SNEIPL is not tied or incorporated in language-specific front-ends that generate a language-independent representation of the source code, but the language-independent representation (eCST representation) serves as the starting point for fact extraction.

It should be also mentioned that there are language-dependent reverse engineering tools which realize software network extraction procedures that can be generalized to a variety of languages. For example, MetricAttitude [12], a tool for the visualization of Java software, relies on improved class hierarchy analysis [54] to approximate the run-time types of function call receivers. Namely, the static analysis approach proposed in [12] distinguishes between two types of the function call relationship: virtual and abstract delegations. The authors of MetricAttitude observed that such differentiation can be exploited to identify design patterns in the source code. Similarly, the Soot framework for Java byte-code optimization uses variable-type analysis (VTA) and declared-type analysis (DTA) when extracting static call graphs from Java byte code [69]. Both of these analyses can be thought as more refined versions of rapid type analysis (RTA) which is used by SNEIPL when resolving candidates for a function call site. Whereas RTA simply collects instantiated types, DTA and VTA find which types reach each variable (i.e. which allocated objects might be assigned to a variable) using information contained in so called *type propagation graph*. SNEIPL discards CALLS links when RTA results in more than one candidate for a function call site in order to prevent creation of non-existent links. In such cases VTA and DTA may result in exactly one candidate. Therefore, in our future work VTA/DTA will be considered as the substitute for RTA in order to obtain more precise approximations of the run-time types of function call receivers.

10. Conclusion

Real-world software systems are characterized by complex inter-entity interactions. Those interactions can be modeled in terms of software networks which show dependencies between software entities. In order to understand the complexity of dependency structures of software systems, to compute metrics associated with software design, or to recover system architecture from the source code, networks representing software systems have to be extracted. In this paper we have presented SNEIPL, the language-independent software networks extractor based on the enriched Concrete Syntax Tree (eCST) representation of the source code. The eCST representation extends parse trees with so called universal nodes which are predefined semantic markers of syntactical constructions. The set of eCST universal nodes contains nodes that mark definitions of software entities which appear as nodes in software networks, as well as universal nodes such as TYPE, NAME and FUNCTION_CALL that serve as the starting point to recover horizontal dependencies between software entities. From the hierarchy of universal nodes in eCSTs different types of horizontal dependencies can be deduced, which means that SNEIPL is able to extract software networks at different levels of abstraction.

The applicability of SNEIPL was shown by the extraction of software networks associated with real-world, medium to large-scale software systems written in different programming languages (Java, Modula-2, and Delphi). To investigate the correctness and completeness of the extraction algorithm, we compared class collaboration networks extracted from ten Java software systems with networks extracted using Dependency Finder (language-dependent software networks extractor) and Doxygen (language-independent documentation generator tool). Obtained results showed that networks extracted by SNEIPL and Dependency Finder are highly similar, and that the eCST-based approach to language-independent dependency extraction provides far more precise results compared to the unified fuzzy parsing approach realized by Doxygen. Since SNEIPL operates on the language-independent representation of the source code, this result can be generalized to networks representing software systems written in other languages.

We also compared SNEIPL to language-independent reverse engineering tools and frameworks, showing that SNEIPL provides both language-independent fact extraction and language-independent representation of extracted facts. This means that besides language-independent network based analysis of software systems and language-independent computation of software design metrics, SNEIPL can be used to provide language-independent extraction of fact bases for reverse engineering, architecture recovery, and software comprehension activities.

Acknowledgments

The authors gratefully acknowledge the support of this work by the Serbian Ministry of Education, Science and Technological Development through project *Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support*, no. OI174023. The authors would also like to thank the anonymous reviewers for their valuable comments.

References

[1] C. R. Myers, Software systems as complex networks: structure, function, and evolvability of software collaboration graphs, Phys. Rev. E 68 (4) (2003) 046116. doi:10.1103/PhysRevE.68.046116.

[2] S. Valverde, R. F. Cancho, R. V. Solé, Scale-free networks from optimal design, EPL (Europhysics Letters) 60 (4) (2002) 512–517. doi:10.1209/epl/i2002-00248-2.

[3] S. Jenkins, S. R. Kirk, Software architecture graphs as complex networks: a novel partitioning scheme to measure stability and evolution, Information Sciences 177 (2007) 2587–2601. doi:10.1016/j.ins.2007.01.021.

[4] D. Hylland-Wood, D. Carrington, S. Kaplan, Scale-free nature of Java software package, class and method collaboration graphs, Tech. Rep. TR-MS1286, MIND Laboratory, University of Maryland, College Park, USA (2006).

[5] R. Wheeldon, S. Counsell, Power law distributions in class relationships, in: Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003, pp. 45–54. doi:10.1109/SCAM.2003.1238030.

[6] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Rev. Mod. Phys. 74 (1) (2002) 47–97. doi:10.1103/RevModPhys.74.47.

[7] M. E. J. Newman, The structure and function of complex networks, SIAM Rev. 45 (2003) 167–256. doi:10.1137/S003614450342480.

[8] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, D. Hwang, Complex networks: structure and dynamics, Physics Reports 424 (45) (2006) 175–308. doi:10.1016/j.physrep.2005.10.009.

[9] M. Newman, Networks: An Introduction, Oxford University Press, Inc., New York, NY, USA, 2010.

[10] Á. Beszédes, R. Ferenc, T. Gyimóthy, Columbus: a reverse engineering approach, in: Proceedings of the 13th IEEE Workshop on Software Technology and Engineering Practice (STEP 2005), IEEE Computer Society, IEEE Computer Society, 2005, pp. 93–96.

[11] M. Lanza, S. Ducasse, Polymetric views - a lightweight visual approach to reverse engineering, IEEE Transactions on Software Engineering 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.

[12] M. Risi, G. Scanniello, Metricattitude: a visualization tool for the reverse engineering of object oriented software, in: Proceedings of the International Working Conference on Advanced Visual Interfaces, AVI '12, ACM, New York, NY, USA, 2012, pp. 449–456. doi:10.1145/2254556.2254643.

[13] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, A. De Lucia, Identifying method friendships to remove the feature envy bad smell, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, ACM, New York, NY, USA, 2011, pp. 820–823. doi:10.1145/1985793.1985913.

[14] G. Scanniello, A. Marcus, Clustering support for static concept location in source code, in: Proceedings of the 19th International Conference on Program Comprehension (ICPC 2011), 2011, pp. 1–10. doi:10.1109/ICPC.2011.13.

[15] J. Buckner, J. Buchta, M. Petrenko, V. Rajlich, Jripples: a tool for program comprehension during incremental change, in: Proceedings of the 13th International Workshop on Program Comprehension, IWPC '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 149–152. doi:10.1109/WPC.2005.22.

[16] G. Rakić, Z. Budimac, Introducing enriched concrete syntax trees, in: Proceedings of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS), 2011, pp. 211–214.
[17] Z. Budimac, G. Rakić, M. Savić, SSQSA architecture, in: Proceedings of the Fifth Balkan Conference in Informatics, BCI '12, ACM, New York, NY, USA, 2012, pp. 287–290. doi:10.1145/2371316.2371380.
[18] D. J. Watts, S. H. Strogatz, Collective dynamics of "small-world" networks, Nature 393 (1998) 440–442. doi:10.1038/30918.
[19] A.-L. Barabási, R. Albert, Emergence of scaling in random networks, Science 286 (5439) (1999) 509–512. doi:10.1126/science.286.5439.509.
[20] R. Albert, H. Jeong, A. Barabasi, Error and attack tolerance of complex networks, Nature 406 (6794) (2000) 378–382. doi:10.1038/35019019.
[21] X. Kong, Y. Qi, X. Song, G. Shen, Modeling disease spreading on complex networks, Computer Science and Information Systems 8 (4) (2011) 1129–1141. doi:10.2298/CSIS110312061K.
[22] S. Fortunato, Community detection in graphs, Physics Reports 486 (3-5) (2010) 75 – 174. doi:10.1016/j.physrep.2009.11.002.
[23] M. Savić, M. Radovanović, M. Ivanović, Community detection and analysis of community evolution in Apache Ant class collaboration networks, in: Proceedings of the Fifth Balkan Conference in Informatics, BCI '12, ACM, New York, NY, USA, 2012, pp. 229–234. doi:10.1145/2371316.2371361.
[24] B. Bollobás, O. M. Riordan, Mathematical results on scale-free random graphs, in: S. Bornholdt, H. G. Schuster (Eds.), Handbook of Graphs and Networks, Wiley, 2005, pp. 1–34.
[25] A. Potanin, J. Noble, M. Frean, R. Biddle, Scale-free geometry in OO programs, Commun. ACM 48 (2005) 99–103. doi:10.1145/1060710.1060716.
[26] M. Savić, M. Ivanović, M. Radovanović, Characteristics of class collaboration networks in large Java software projects, Information Technology and Control 40 (1) (2011) 45–54. doi:10.5755/j01.itc.40.1.192.
[27] A. P. S. de Moura, Y.-C. Lai, A. E. Motter, Signatures of small-world and scale-free properties in large computer programs, Phys. Rev. E 68 (1) (2003) 017102. doi:10.1103/PhysRevE.68.017102.
[28] N. Labelle, E. Wallingford, Inter-package dependency networks in open-source software, in: Proceedings of the 6th International Conference on Complex Systems (ICCS), paper no. 226, 2006.
[29] J. Brooks, F.P., No silver bullet: essence and accidents of software engineering, Computer 20 (4) (1987) 10–19. doi:10.1109/MC.1987.1663532.
[30] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493. doi:10.1109/32.295895.
[31] E. J. Chikofsky, J. H. Cross II, Reverse engineering and design recovery: a taxonomy, IEEE Software 7 (1) (1990) 13–17. doi:10.1109/52.43044.
[32] H. M. Kienle, H. A. Müller, Rigi - an environment for software reverse engineering, exploration, visualization, and redocumentation, Science of Computer Programming 75 (4) (2010) 247–263. doi:10.1016/j.scico.2009.10.007.
[33] M. Shtern, V. Tzerpos, Clustering methodologies for software engineering, Advances in Software Engineering 2012 (2012) 1:1–1:18. doi:10.1155/2012/792024.
[34] G. C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between source and high-level models, in: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, SIGSOFT '95, ACM, New York, NY, USA, 1995, pp. 18–28. doi:10.1145/222124.222136.
[35] Y. Chiricota, F. Jourdan, G. Melançon, Software components capture using graph clustering, in: Proceedings of the 11th IEEE International Workshop on Program Comprehension, IWPC '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 217–226. doi:10.1109/WPC.2003.1199205.
[36] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, 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, IWPC '98, IEEE Computer Society, Washington, DC, USA, 1998, pp. 45–52. doi:10.1109/WPC.1998.693283.
[37] B. S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208. doi:10.1109/TSE.2006.31.
[38] J. Wu, A. E. Hassan, R. C. Holt, Comparison of clustering algorithms in the context of software evolution, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 525–535. doi:10.1109/ICSM.2005.31.
[39] G. Scanniello, A. D'Amico, C. D'Amico, T. D'Amico, Using the Kleinberg algorithm and vector space model for software system clustering, in: 18th International Conference on Program Comprehension (ICPC 2010), 2010, pp. 180–189. doi:10.1109/ICPC.2010.17.
[40] R. W. Schwanke, An intelligent tool for re-engineering software modularity, in: Proceedings of the 13th international conference on Software engineering, ICSE '91, IEEE Computer Society Press, Los Alamitos, CA, USA, 1991, pp. 83–92. doi:10.1109/ICSE.1991.130626.
[41] N. Anquetil, C. Fourrier, T. C. Lethbridge, Experiments with clustering as a software remodularization method, in: Proceedings of the Sixth Working Conference on Reverse Engineering, WCRE '99, IEEE Computer Society, Washington, DC, USA, 1999, pp. 235–255. doi:10.1109/WCRE.1999.806964.
[42] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Transactions on Software Engineering 33 (11) (2007) 759–780. doi:10.1109/TSE.2007.70732.
[43] M. Savić, G. Rakić, Z. Budimac, M. Ivanović, Extractor of software networks from enriched concrete syntax trees, in: Proceedings Of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2011, 2nd Symposium on Computer Languages, Implementations and Tools (SCLIT), Vol. 1479, 2012, pp. 486–489. doi:10.1063/1.4756172.
[44] L. C. Briand, J. W. Daly, J. Wüst, A unified framework for cohesion measurement in object-oriented systems, Empirical

Software Engineering 3 (1) (1998) 65–117. doi:10.1023/A:1009783721306.
[45] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
[46] G. Rakić, Z. Budimac, SMIILE prototype, in: Proceedings of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Symposium on Computer Languages, Implementations and Tools (SCLIT), 2011, pp. 544–549. doi:10.1063/1.3636867.
[47] G. Rakić, Z. Budimac, M. Savić, Language independent framework for static code analysis, in: Proceedings of the 6th Balkan Conference in Informatics, BCI '13, ACM, New York, NY, USA, 2013, pp. 236–243. doi:10.1145/2490257.2490273.
[48] Č. Gerlec, G. Rakić, Z. Budimac, M. Heričko, A programming language independent framework for metrics-based software evolution and analysis, Computer Science and Information Systems 9 (3) (2012) 1155–1186. doi:10.2298/CSIS120104026G.
[49] OMG, Architecture-driven modernization (ADM): abstract syntax tree metamodel (ASTM), Version 1.0 (January).
[50] OMG, Architecture-driven modernization (ADM): knowledge discovery metamodel (KDM), Version 1.3 (August).
[51] T. J. Parr, R. W. Quong, ANTLR: a predicated-LL(k) parser generator, Software: Practice and Experience 25 (7) (1995) 789–810. doi:10.1002/spe.4380250705.
[52] J. Kolek, G. Rakić, M. Savić, Two-dimensional extensibility of SSQSA framework, in: Proceedings of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), 2013, pp. 35–43.
[53] D. F. Bacon, P. F. Sweeney, Fast static analysis of C++ virtual function calls, in: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '96, ACM, New York, NY, USA, 1996, pp. 324–341. doi:10.1145/236337.236371.
[54] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: M. Tokoro, R. Pareschi (Eds.), Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP95), Vol. 952 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1995, pp. 77–101. doi:10.1007/3-540-49538-X_5.
[55] G. C. Murphy, D. Notkin, W. G. Griswold, E. S. Lan, An empirical study of static call graph extractors, ACM Transactions on Software Engineering and Methodology (TOSEM) 7 (2) (1998) 158–191. doi:10.1145/279310.279314.
[56] A. Capiluppi, T. Knowles, Software engineering in practice: design and architectures of FLOSS systems, in: C. Boldyreff, K. Crowston, B. Lundell, A. Wasserman (Eds.), Open Source Ecosystems: Diverse Communities Interacting, Vol. 299 of IFIP Advances in Information and Communication Technology, Springer Berlin Heidelberg, 2009, pp. 34–46. doi:10.1007/978-3-642-02032-2_5.
[57] A. Capiluppi, C. Boldyreff, K.-J. Stol, Successful reuse of software components: a report from the open source perspective, in: S. Hissam, B. Russo, M. Mendona Neto, F. Kon (Eds.), Open Source Systems: Grounding Research, Vol. 365 of IFIP Advances in Information and Communication Technology, Springer Berlin Heidelberg, 2011, pp. 159–176. doi:10.1007/978-3-642-24418-6_11.
[58] A. Capiluppi, C. Boldyreff, Identifying and improving reusability based on coupling patterns, in: H. Mei (Ed.), High Confidence Software Reuse in Large Systems, Vol. 5030 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 282–293. doi:10.1007/978-3-540-68073-4_31.
[59] V. H. Nguyen, L. M. S. Tran, Predicting vulnerable software components with dependency graphs, in: Proceedings of the 6th International Workshop on Security Measurements and Metrics, MetriSec '10, ACM, New York, NY, USA, 2010, pp. 3:1–3:8. doi:10.1145/1853919.1853923.
[60] D. Berner, H. Patel, D. Mathaikutty, S. Shukla, Automated extraction of structural information from SystemC-based IP for validation, in: Sixth International Workshop on Microprocessor Test and Verification (MTV '05), 2005, pp. 99–104. doi:10.1109/MTV.2005.8.
[61] W. Feller, On the Kolmogorov-Smirnov limit theorems for empirical distributions, The Annals of Mathematical Statistics 19 (2) (1948) 177–189.
[62] S. Valverde, V. Solé, Hierarchical small worlds in software architecure, Dyn. Contin. Discret. Impuls. Syst. Ser. B: Appl. Algorithms 14(S6) (2007) 305–315.
[63] M. Savić, M. Ivanović, M. Radovanović, Connectivity properties of the Apache Ant class collaboration network, in: Proceedings of the 15th International Conference on System Theory, Control, and Computing (ICSTCC), 2011, pp. 544–549.
[64] L. Wen, R. G. Dromey, D. Kirk, Software engineering and scale-free networks, IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 39 (2009) 845–854. doi:10.1109/TSMCB.2009.2020206.
[65] D. Puppin, F. Silvestri, The social network of Java classes, in: Proceedings of the 2006 ACM symposium on Applied computing, SAC '06, ACM, New York, NY, USA, 2006, pp. 1409–1413. doi:10.1145/1141277.1141605.
[66] S. Ducasse, M. Lanza, S. Tichelaar, Moose: an extensible language-independent environment for reengineering object-oriented systems, in: 2nd International Symposium On Constructing Software Engineering Tools (COSET 2000), 2000.
[67] J. Ebert, B. Kullbach, V. Riediger, A. Winter, GUPRO: generic understanding of programs – an overview, in: Electronic Notes In Theorethical Computer Science, Vol. 72, 2002, pp. 47–56. doi:10.1016/S1571-0661(05)80528-6.
[68] A. Raza, G. Vogel, E. Plödereder, Bauhaus: a tool suite for program analysis and reverse engineering, in: Proceedings of the 11th Ada-Europe international conference on Reliable Software Technologies, Ada-Europe'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 71–82. doi:10.1007/11767077_6.
[69] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, C. Godin, Practical virtual method call resolution for Java, in: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '00, ACM, New York, NY, USA, 2000, pp. 264–280. doi:10.1145/353171.353189.