Defining Substitutability Criteria for Object Oriented Components

Venera Arnaoudova

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada
September 2008

Your file Votre référence
ISBN: 978-0-494-45512-8
Our file Notre référence
ISBN: 978-0-494-45512-8

Abstract

Defining Substitutability Criteria for Object Oriented Components

Venera Arnaoudova

Component-Based Software Development (CBSD) promotes software reusability by allowing new functionalities to be added and existing functionalities to be removed or replaced easily. However, high reusability comes with its own cost, namely the difficulty in selecting suitable candidates for adaptation tasks. Even though research has been conducted toward identification of such components, current methods rely on an existing system specification, which is more often either not available or inconsistent with other artifacts such as implementation. In this dissertation, we complement current works by proposing a novel approach to compare software components at source code level independent on the existence, or otherwise, of the specification. We consider Open-Source Software (OSS) components written in Java at three levels of granularity, namely methods, types and packages. Consequently, we define substitutability criteria at three levels of abstraction and provide metrics indicating the degree of matching of two components. We provide automation and tool support through an Eclipse plug-in and we demonstrate our method through a case study. We expect our approach to be beneficial to maintainers during the selection of suitable candidate components.

Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Constantinos Constantinides, whose knowledge and guidance contributed to my graduate work and experience.

A sincere thank to Laleh Eshkevari for her contributions and encouragements during those two years. I would also like to thank Hamoun Ghanbari for his advices and suggestions during this research.

I would like to acknowledge the debt I owe to my fiancé Julien Pireaud for being always beside me. Without him, this work would not be possible.

I would also like to thank my family for supporting me during my studies.

This project has been partially funded by Defence Research and Development Canada - Valcartier.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In software engineering, the main goal of developing new software or maintaining an existing one is to provide functionalities to satisfy stakeholders' needs. In order to do this, sometimes new functionalities are added, existing functionalities are replaced, or unused functionalities are removed. More often during the maintenance phase of the software life cycle, some part of the software that provides a specific feature needs to be replaced with a new one for different reasons (costly maintainability, upgrade to new technology, etc.). These changes, together with the time consumed during their implementation and their associated costs, constitute the motivation behind Component-Based Software Development (CBSD) [6]. CBSD focuses on building software by plugging Commercial Off-The-Shelf (COTS) components or Open-Source Software (OSS) components in the system rather than building functionalities from scratch. CBSD is widely used despite its associated risks and challenges [64]. Its advantages are well-known and have been discussed in the literature [59].

During maintenance, a request for change is followed by a comprehension task, after which system dependencies are identified and the impact of the potential change is measured before any actual change takes place. Should a change be implemented, the process is completed by a verification step [2]. In order to perform comprehension and change impact analysis, one should:

1. understand both, what and where, functionalities should be added/ replaced.
2. select a component (or a group of components) that provides these functionalities.
3. evaluate the effort and changes needed to integrate the selected component(s).

The second activity indicated above is accomplished by comparing the required information with that provided by available components. Existing approaches on component matching in the literature perform this comparison based on quality characteristics, metrics, and attributes of COTS. The difficulty of applying such approaches comes from i) the lack of standard type of information for COTS components [9], ii) incomplete information [3], and iii) inapplicability of these approaches to OSS components. Other approaches which are applicable to the comparison of OSS components are code clone detection approaches. The limitation of these approaches for the purpose of identifying substitutable components however is that they only identify identical code.

1.1 Objectives

The objective of this research is to complement existing approaches by addressing those cases where the lack of specification and the unavailability of documentation increase the difficulty of comparing two components. We provide support for users to select appropriate OSS/COTS object-oriented components by extracting the necessary information from their source code.

1.2 Organization

The remainder of this thesis is organized as follows: in Chapter 2 we provide the necessary background for this research. In Chapter 3 we discuss the problem and motivation behind this research. We discuss our proposal in Chapter 4. In Chapter 5 we discuss how to implement our proposal and in Chapter 6 we demonstrate our methodology through a case study. In Chapter 7 we describe the automation and tool support. In Chapter 8 we discuss related work. We list conclusions and recommendations for further work in Chapter 9.

Chapter 2

Background

In this chapter we discuss the necessary background to this research.

2.1 Software maintenance

ISO/IEC and IEEE define maintenance as the modification of a software product after delivery in order to correct faults, improve performance (or other attributes) or adapt the product to a modified environment [43]. Four different types of maintenance are identified: corrective, preventive, adaptive and perfective. Corrective maintenance includes all changes made to a system after deployment to correct problems. Preventive maintenance includes all changes made to a system after deployment to prevent faults[1] to become failures[2]. Adaptive maintenance includes all changes made to a system after deployment to support operability in a different (software or hardware) environment. Perfective maintenance includes all changes made to a system after

[1]Software errors which can cause improper functioning of the system [43].

[2]Activated software faults [43].

deployment to address new requirements.

Bennett and Rajlich [2] define a model whereby a software system undergoes distinctive stages during its life: initial development, evolution, servicing, phase-out, and closedown. Initial development would produce a deployable system (the first operating version). After deployment, evolution would extend the capabilities of the system, possibly in major ways. Once evolution is no longer viable, the software would enter the servicing stage (often referred to as maturity, or most commonly legacy stage). As the term suggests, only small changes are possible during this stage. Finally, once servicing is no longer viable, the system enters a phase-out stage where deficiencies are known but not addressed. At closedown, the system is withdrawn from the market. In an alternative model (versioned staged model), during evolution, a version is publicly released and subsequently enters the servicing stage. Meanwhile, the system continues to evolve in order to produce the next version.

To ease the maintenance phase, we feel that one can follow two paths that are mutually supportive: In the first path, one looks into development and focuses on the provision of quality attributes which can affect maintainability (predeployment approach). In the second path, one improves current methods that have been utilized to perform the various maintenance activities (postdeployment approach) [12].

Currently, there are a number of methods and tool support to ease these activities, focusing on comprehension, which tends to consume a large proportion of time during maintenance. The analysis of artifacts can be categorized as static and dynamic. Static analysis is performed by examining design or implementation artifacts and reasoning over possible behaviors that might arise during execution. Dynamic analysis is performed by executing a program and observing the executions.

One method for performing either kind of analysis is program slicing [66]. A slice includes all program statements affecting variables at a certain position in the program. A forward slice consists of all program points that are affected by a given point in the program. A backward slice consists of all program points that affect a given point in the program.

2.2 Software components

Component-Based Software Development (CBSD) is based on the idea that there are many software components already built, and that building new systems by selecting the appropriate existing components and assembling them following a well defined software architecture can be more rapid and cost effective approach than building a system from scratch [49]. CBSD has a potential for:

1. reducing the cost and time of the software development and maintenance, thus increasing productivity.

2. improving the system maintainability and flexibility by allowing replacement of old components.

3. enhancing the system quality by allowing components to be built by experts in the domain.

CBSD consists of the following activities:

1. Requirement analysis.

2. Architecture selection and creation.

3. Component selection.

4. Component integration.

5. System testing.

Of interest to this thesis is component selection which requires the identification of functional and quality requirements, identification of the trade-offs, evaluation of the impact of those trade-offs on the system, and identification of the components to be adapted to this integration [50].

In the literature there is no standardized definition for the term component. Szyperski defines a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is subject to composition by third parties" [59][3]. Brown and Wallnau [4] discuss four established definitions in order to determine the characteristics of a component. Cai et al. [6] summarize those characteristics to be as follows: A component

1. is an independent and replaceable part of the system that fulfills a clear function.

2. works in the context of a well defined architecture.

3. communicates with other components via their interfaces.

Components are grouped into two main categories: Commercial Off-The-Shelf (COTS), usually closed source, and Open-Source Software (OSS), usually non-commercial.

[3]The definition initially appeared as an outcome of the ECOOP Workshop on Component-Oriented Programming in 1996 [60].

Even though both COTS and OSS components have similar main advantages from a software maintenance point of view, each has its own advantages and disadvantages. Li et al. [39] investigate the decision making while choosing components. COTS components are usually delivered as binaries rather than as source code in order to protect the rights of developers. OSS components are provided as source code, which is more often the only available documentation. Some of the disadvantages of OSS components discussed in the literature are the lack of a face-to-face development process, code quality [20], and nonmaintainability [55].

Popular sources for components include ComponentSource [11], SourceForge [57], JARS [30], FreshMeat [22] etc.

In the next chapter, we discuss the problem that has motivated this research.

Chapter 3

Problem and motivation

In this chapter we discuss the problem and the motivation behind the research that constitutes the scope of this dissertation.

In the domain of CBSD, a critical question that maintainers should answer is *Buy or build?* In order to decide on a viable option, one should go through the following process:

1. Specify the desired functionalities.

2. Identify potential component candidates.

3. Compare the specifications provided by the component candidates with the one requested by maintainers.

4. Filter out non-suitable component candidates.

5. Analyze the changes necessary for integrating the components.

6. Decide whether it is more viable to deploy one of the selected components or build a new one from scratch.

While going through these steps and searching for an answer to the main question, other questions surface, such as: *How to identify potential components? How to specify a component? How to compare two specifications? How to ensure that the most suitable component has been chosen?*

Some of the existing approaches which deal with software classification and identification problems deploy description logic to develop ontologies or propose taxonomies for classification (Cechich et al. provide a summary of these approaches [9]). Some research has been done to measure the semantic distance between required and provided components [31], while other researchers propose early measurement of the functionality suitability of COTS [8].

Research also has been done in the domain of component matching, where a clear distinction is made between component interface and component behavior. An existing approach involves behavior matching and defines different types of matching (from exact to more relaxed matching). It uses a formal language (Larch) to specify and compare the components in terms of pre-conditions/post-conditions [69]. There exist other approaches which deploy different formal specification languages for the same purpose [18, 19, 29, 46].

The common assumption of these approaches is the existence of a component specification. Research toward such a standard specification has been done [9, 51, 63], but the current lack of such a specification [3] makes those approaches difficult to adopt in practice. Moreover, even if such a standard existed, it would probably be adopted for COTS components, since to formally specify a component requires expertise which is not generally part of the development process of OSS components.

The motivation behind this research is to provide support for maintainers during the component substitution task, when nothing but source code is available. Without such support, maintainers should analyze and filter the irrelevant information manually by going throughout the entire source code. Therefore, we believe that there is a need for an automated environment which would allow automatic filtering of that part of the information which is either not relevant or is too detailed, as well as automatic analysis of the extracted information. Automatic filtering should allow maintainers to focus and further analyze only part of the source code. Automatic analysis should provide an indication of the degree of substitutability of two components, based on predefined metrics which take into account the essential characteristics (in this case object-oriented concepts) of the components.

We discuss our proposal to the above problem in the next chapter.

Chapter 4

Proposal

In this chapter, we discuss our research proposal.

Different aspects of a component are taken into consideration by current approaches. Most of them rely on an existing specification, which often is either missing or incomplete. What is irrefutable, however, is that for some COTS components and for all OSS components the source code is available.

We believe that components can be characterized by their definition (signature), state and behavior, which can be extracted from their source code. Thus, for calculating the degree of substitutability of two components, their definition, state, and behavior should be compared.

To this end, we propose to implement the following:

1. Identify criteria for component substitutability in an object-oriented context at various levels of granularity.

2. Extract the necessary information from the source code (at an appropriate level)

in order to be able to apply the criteria defined in (1).

3. Provide automation and tool support and integrate them into a popular IDE.

4. Demonstrate a proof of concept through a case study.

In order to reach these goals, we plan to define important characteristics for an object-oriented component by taking Java as an example environment. However, our approach is based on fundamental object-oriented principles, thus keeping the concepts general enough in order to be applicable to other languages in the same paradigm. Next, we will extract the information needed for these characteristics through static analysis. We will then define metrics for measuring the degree of matching of two components at different levels of granularity: package, class, and method. Finally, we will deploy our automation over a case study in order to demonstrate our approach in a practical situation.

4.1 Expected contributions and benefits

The expected contributions of our proposal are to provide support during the maintenance phase of the software life cycle while evaluating the degree of substitutability of two components when the source code is available. We will provide an automated environment for extracting information from the source code following the identified important characteristics of the analyzed components. We will then define measurement procedures, which will allow us to explore the extracted information, thus providing an indicator of substitutability.

In the next chapter we specify what a component is and we define guidelines for the substitutability of two components at three levels of granularity.

Chapter 5

Methodology

In this chapter, we discuss the methodology of our research proposal. In Section 5.1, we define what is a component in the scope of this research. Next, in Section 5.2, we define component substitutability criteria by discussing its definition, state, and behavior. Based on those criteria, we calculate the percentage of the degree of matching of two components which represents a quantitative matching. We then define thresholds based on the percentage of matching from which three categories of qualitative matching are formed: low, medium, and high. The methodology is illustrated on a case study in Chapter 6.

5.1 What is a component?

Before discussing the possibility of comparing two components, we will define what we consider to be a component.

We restrict our discussion to Java source code, which allows us to distinguish

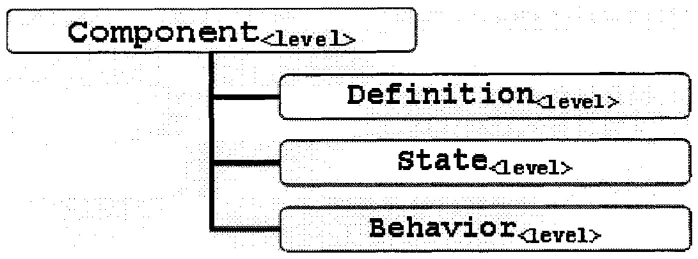


Figure 1: Component specification.

between three levels of granularity of a component: method level, class level, and package level. All three levels will be considered.

At all levels, a component is specified by its definition, state, and behavior (see Figure 1).

At method level (ml), the definition of a component corresponds to the signature of the method. Thus, the important factors for a component here are its formal parameters (name[1], type and position are considered), the type of its output, its name, its visibility, its modifiers, and its declared exceptions (only type is important). The structure of the definition of a component at method level is shown in Figure 2.

The state of a component at method level is characterized by its state variables and the flow of events which affect these variables (see Figure 3). A state is represented by the set of important variables which are possibly modified after an event occurs. Thus, a new state is created every time an event involving one or more state variables

[1]In this dissertation, names are considered to be important as, when assigned properly, they reflect the concepts of the variables, events, types etc.

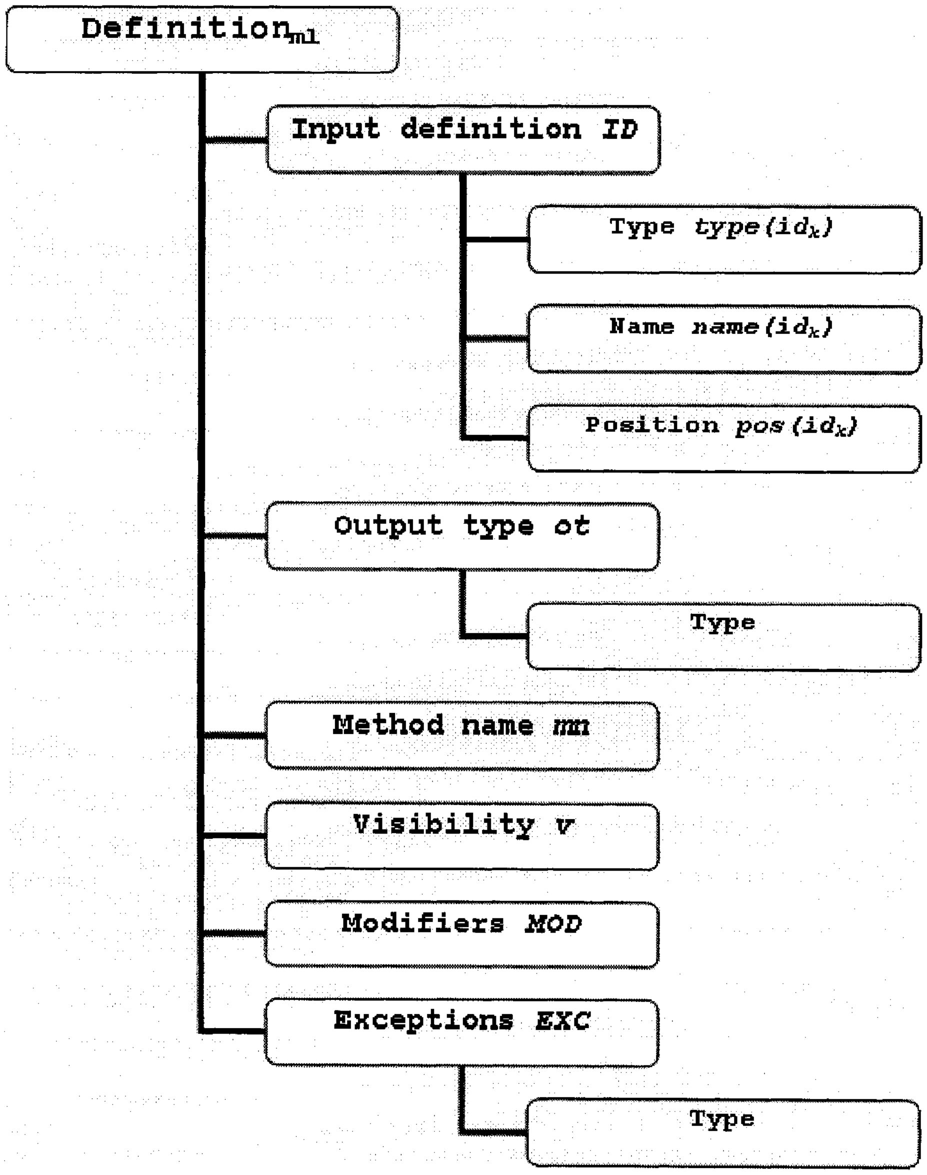


Figure 2: Definition at method level (ml).

occurs.

The behavior of a component at method level is represented by the set of its required types (see Figure 4). A required type is characterized by the name of the package in which it is defined, by its name, its attributes (visibility, type, and name), and its methods (only the definition of the methods are considered as shown in Figure 2).

The definition of a component at class level (*cl*) is characterized by its visibility, modifiers, type, parent modules, name, defined attributes, and defined methods (see Figure 5).

The state of a component at class level is represented by the states of its methods (see Figure 6).

The behavior of a component at class level is represented by the required types of its methods (see Figure 7) and the types of its defined attributes.

At package level (*pl*), a component is defined on the basis of its name and its defined types (see Figure 8).

The state of a component at package level is defined based on the state of its defined types (see Figure 9).

The behavior of a component at package level is represented by the behavior of its defined types (see Figure 10).

Equivalence for all criteria is defined formally in the following section.

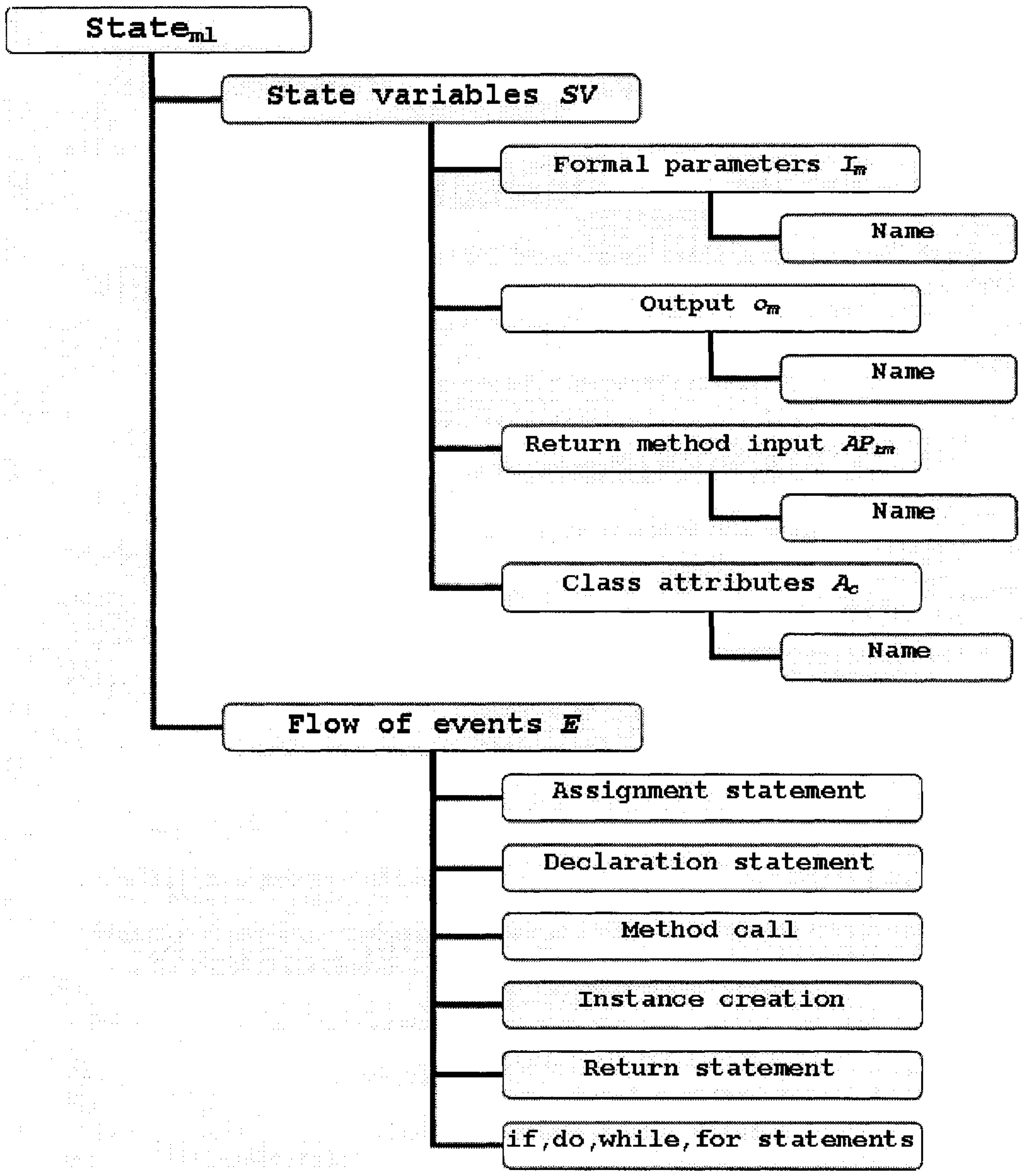


Figure 3: State at method level (ml).

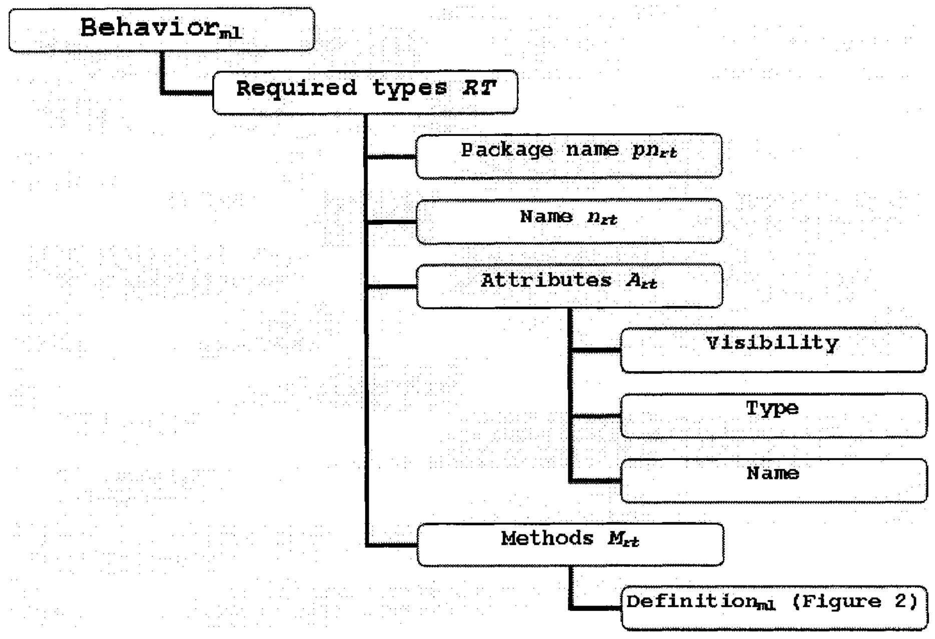


Figure 4: Behavior at method level (ml).

5.2 Component substitutability criteria

We consider all three levels of granularity by defining substitutability criteria for each level.

When comparing two components, intuitively, one tends to compare their definitions: name, kind. Having the same definition, however, often does not imply substitutability. For example, the same component often is designed and implemented differently by different users. Each implementation is adapted to a specific need or environment, thereby providing different functionality. Consequently, comparing component definitions is necessary but not sufficient.

If we see a component as a mathematical function, we may say that for C being the component to be substituted and C' being a candidate component to substitute

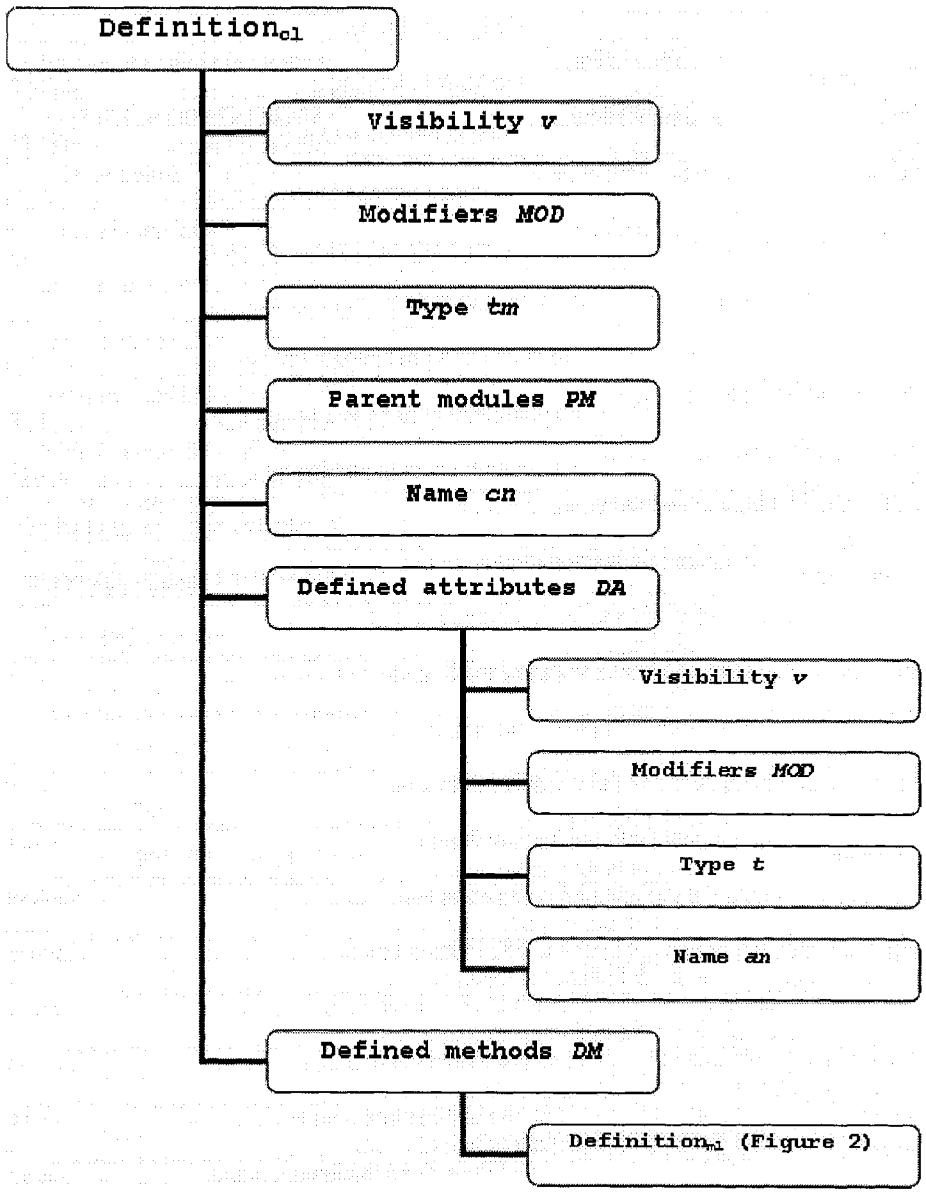


Figure 5: Definition at class level (cl).

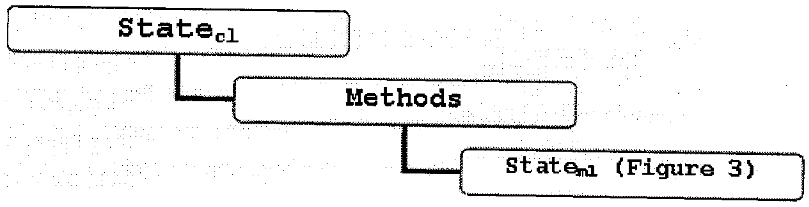


Figure 6: State at class level (cl).

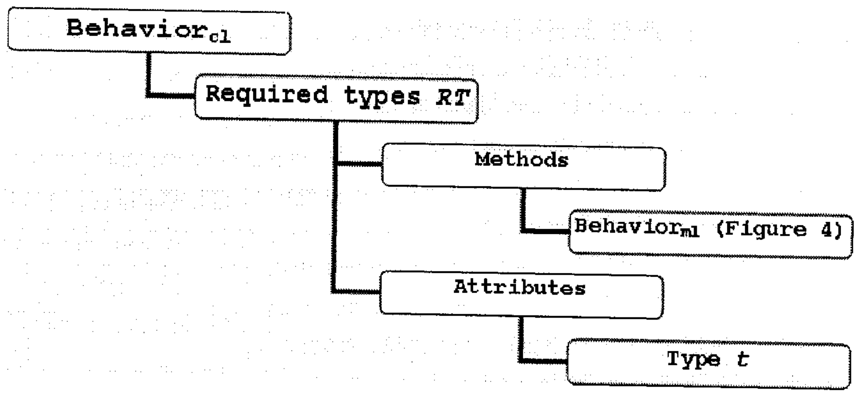


Figure 7: Behavior at class level (cl).

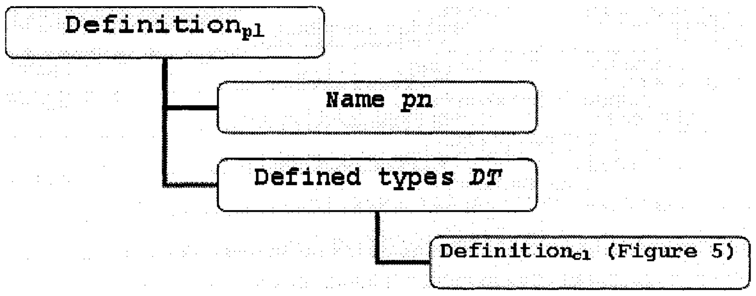


Figure 8: Definition at package level (*pl*).

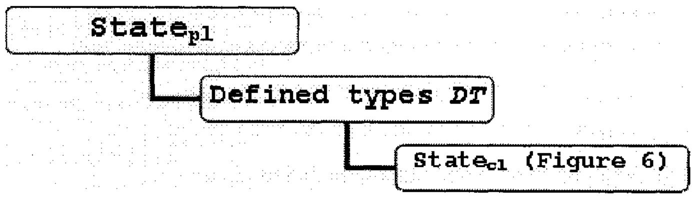


Figure 9: State at package level (*pl*).

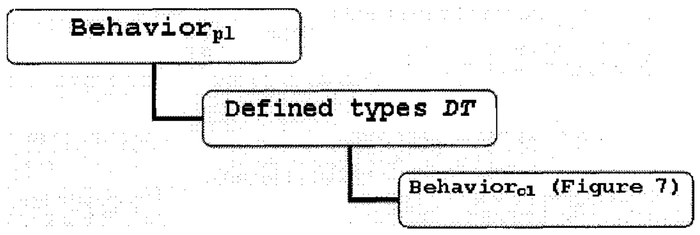


Figure 10: Behavior at package level (pl).

C, then the two components are equivalent and thus entirely substitutable if they have the same domain (see equation 1), the same codomain (see equation 2), and the same association of values of the codomain with values of the domain [26] (see equation 3).

$$Domain\ C \equiv Domain\ C' \tag{1}$$

$$Codomain\ C \equiv Codomain\ C' \tag{2}$$

$$\forall x \in Dom\ C\ \exists x' \in Dom\ C' \bullet\ x \equiv x' \wedge\ C(x) \equiv C'(x') \tag{3}$$

where $C(x) \in Codomain\ C$ and $C'(x') \in Codomain\ C'$.

Components, as viewed in the context of object-oriented programming, are characterized by their state and their behavior. Thus, we can say that two components are equivalent if they satisfy equations 4, 5, 6, 7, and 8 defined in the following paragraphs.

Let E and E' be the sets of events of C and C' respectively upon which the components change from one state to another. Let S and S' represent the sets of states of C and C' respectively, and let MAP and MAP' be the sets of elements associating an event with the state where it occurs and the state in which it transits.

C and C' have the same state if for every state of C, an equivalent state in C' exists, and for every event e_i in C, changing the state from s_n to s_m, an equivalent event e'_i in C', triggering the change of state from s'_n to s'_m, exists ($\forall n, m \in [1..|S|]$).

$$\forall e_i \in E \ \exists e'_i \in E' \bullet \ e_i \equiv e'_i \tag{4}$$

where $|E| = |E'|$.

$$\forall s_j \in S \ \exists s'_j \in S' \bullet \ s_j \equiv s'_j \tag{5}$$

where $|S| = |S'|$.

$$\forall map \in MAP \ \exists map' \in MAP' \bullet map \equiv map' \tag{6}$$

where $|MAP| = |MAP'|$, $map = \ < s_{start}, \ e, \ s_{end} >$, $e \in E$, and $s_{start}, s_{end} \in S$.

Let RT and RT' be the sets of types being in interaction with C and C' respectively, that is, types to which the components have visibility, and let M_{rt_i} and $M'_{rt'_i}$ be the sets of events sent to instance rt_i, where $rt_i \in RT$. Equivalent behavior with respect to other components can be then defined based on equations 7 and 8:

$$\forall rt_i \in RT \ \exists rt'_i \in RT' \bullet \ rt_i \equiv rt'_i \tag{7}$$

$$\forall m_{rt_i} \in M_{rt_i} \ \exists m'_{rt'_i} \in M'_{rt'_i} \bullet \ m_{rt_i} \equiv m'_{rt'_i} \tag{8}$$

which implies that C and C' interact with the same set of objects in the same manner.

The above criteria define 100% substitutability for two components. However, strict matching of C and C' may not be always needed. In that case, the definition of C' can be more general than the definition of C. Also, it may be sufficient that E is a subset of E', and S is a subset of S'. Regarding the behavior with respect to other components, it may be sufficient that $RT \subseteq RT'$ and $M_{rt_i} \subseteq M_{rt'_i}$.

These criteria are general and are refined at the different levels of granularity in the subsequent subsections, where each subsection compares C and C' in terms of:

- their definitions.
- their states at a specific moment after an event has taken place.
- their required types and the way these types are used.

We restrict the discussion to activities related to static analysis. Thus, comparing the runtime behavior of the components is beyond the scope of this research.

5.2.1 Method level criteria

Two cases may occur at this level. The first is when one knows both the method which needs to be substituted and the potential method for this substitution. The second case occurs when one knows the method which needs to be substituted but does not know which method exactly from the candidate component is the most suitable one. The second case may be reduced to the first one by comparing the method to be substituted with each method of the candidate component.

In the following subsections, we refine the criteria defined above for the cases where components are represented by methods.

Method definition

When components correspond to methods, by definition we imply method signature (input and name) and some additional criteria (output, visibility, modifiers, and exceptions). Thus, two methods are said to have the same method definition if the following six conditions hold:

1. They take equivalent ordered sets as input, where an element of this set is represented by its type and its name. The input of the method is important because this is the data that the method will operate on. The name and type of the formal parameters correspond to the concerns they represent. The number of formal parameters and their position affect the easiness of the actual substitution. For example if the component to be substituted operates on two parameters while the candidate component only takes one parameter, more modifications will be performed in order to adapt the candidate component to the system. Let $ID = \textit{Domain } C$ and $ID' = \textit{Domain } C'$ be the ordered sets containing the input definitions of C and C' respectively. Every element in

these sets is represented as the pair $< type, name >$. Assume $k \in [1..|ID|]$:

$$\forall id_k \in ID \ \exists id'_k \in ID' \bullet id_k \equiv id'_k \tag{9}$$

For strict method matching the types of id_k and id'_k should be the same as well as their names. For more lax matching however, $|ID| \leq |ID'|$; the type of id_k is of type or subtype[2] of the type of id'_k ("contravariance of arguments" [40]); the name of id_k is a substring of the name of id'_k or vice versa.

2. They return a value of the same type. The type of the output is important because it reflects the concern representing the data returned by the method. Assume ot and ot' be the defined returned types of C and C' respectively, then:

$$ot \equiv ot' \tag{10}$$

For less strict method matching the equivalence is transformed into subtype relation, that is $ot' \leq ot$ ("covariance of result" [40]).

3. They have the same name. The name of a method when assigned properly is essential because it reflects the functionality provided by this method. Assume mn and mn' be the names of C and C' respectively, then

$$mn \equiv mn' \tag{11}$$

[2]Liskov and Wing [40] introduced $<$ for denoting a subtype relation. In this dissertation we follow this notation and we use $\leq$ for "type of, or subtype of".

For more lax method matching mn may be a substring of mn' or vice versa.

4. They have the same visibility modifiers. Visibility modifiers are important and should be compared because they define who will be able to use the components. Assume v and v' be the visibility modifiers of C and C' respectively and $V = \{public, protected, \text{default}, private\}$ [38] and $V' = \{public, protected, \text{default}, private\}$ ordered from the least to the most restrictive visibility (default visibility is applied when no explicit visibility is specified, in which case it is equivalent to *protected* visibility). Equation 12 should hold:

$$\forall v \in V \; \forall v' \in V' \bullet v \equiv v' \tag{12}$$

For less strict method matching the visibility of the new method may be less restrictive, compared to the visibility of the substituted method.

5. They have the same modifiers. Comparing modifiers is vital because it affects the easiness of the substitution. For example if the candidate component is defined as *final* it will not be possible to extend it, which may be a problem if maintainers were planning to do so. Let MOD and MOD' be the sets of modifiers of C and C' respectively, where $MOD = \{abstract, final, native, static, synchronized\}$ [38] and $MOD' = \{abstract, final, native, static, synchronized\}$. Assume $k \in [1..|MOD|]$, equation 13 should hold:

$$\forall mod_k \in MOD \; \exists mod'_k \in MOD' \bullet mod_k \equiv mod'_k \tag{13}$$

6. They throw the same exceptions. The types of the thrown exceptions reflect how this method handle abnormal situations. They also affect the level of adaptation of the system to the candidate component in cases where it throws more or different exceptions than the one thrown by the component to be substituted. Let EXC and EXC' be the sets of exceptions thrown by C and C' respectively, where $|EXC| = |EXC'|$. Equation 14 should hold:

$$\forall exc \in EXC\ \exists exc' \in EXC' \bullet exc \equiv exc' \tag{14}$$

For non strict matching *exc'* may be a subtype of *exc*, or EXC' be a subset of EXC.

Example Consider C and C' as defined in Listings 1 and 2 respectively. Following is the comparison for the level of substitutability of the two components considering lax matching. Regarding equation 9, the two components take the same number of parameters, equal to two. The first parameter of C, which is $Vector$ is a subtype of the first parameter of C', which is *AbstractCollection*, but their names are different. The second parameters of the two components are of the same type, which is *int* and their names are equivalent. Equation 10 is satisfied since the return type of C' ($String$) is a subtype of the return type of C ($Object$). Equation 11 is satisfied since the names of C and C' are equivalent ($access$). Equation 12 is satisfied since the visibility modifier of C (*protected*) is more restrictive than the visibility modifier of C' (*public*). Equation 13 is not satisfied because C is declared *static* and not C'.

Equation 14 is satisfied since $EXC = \{Exception\}$ and $EXC' = \{\}$, and thus we can conclude that $EXC' \subseteq EXC$.

```
protected static Object access(Vector v, int index) throws Exception{
   // method body
}
```

Listing 1: Method definition for the component to be substituted.

```
public String access( AbstractCollection ac, int index){
   // method body
}
```

Listing 2: Method definition for the candidate component.

Overall definition matching The overall definition matching at method level for two components C and C', namely $odm_{ml}(C, C')$ is calculated based on the degree of matching of their input definitions, their return types, their names, their visibilities, their modifiers, and their exceptions. The importance of all these criteria is defined by their respective weights (see formula 15).

$$\begin{aligned} odm_{ml}(C, C') = \; & w_{ID} * g_{ID}(C, C') + w_{ot} * g_{ot}(C, C') + \\ & w_{mn} * g_{mn}(C, C') + w_v * g_v(C, C') + \\ & w_{MOD} * g_{MOD}(C, C') + w_{EXC} * g_{EXC}(C, C') \end{aligned} \quad (15)$$

where w_{ID}, w_{ot}, w_{mn}, w_v, w_{MOD}, and w_{EXC} are percentages indicating the importance of the associated grade, w_{ID}, w_{ot}, w_{mn}, w_v, w_{MOD}, $w_{EXC} \in [0..100]$, and $w_{ID} + w_{ot} + w_{mn} + w_v + w_{MOD} + w_{EXC} = 100$;

$g_{ID}(C, C')$, $g_{ot}(C, C')$, $g_{mn}(C, C')$, $g_v(C, C')$, $g_{MOD}(C, C')$, and $g_{EXC}(C, C')$ are percentages defining the degree of matching of the associated element, $g_{ID}(C, C')$, $g_{ot}(C, C')$, $g_{mn}(C, C')$, $g_v(C, C')$, $g_{MOD}(C, C')$, $g_{EXC}(C, C') \in [0..100]$.

The grade $g_{ID}(C, C')$ is calculated based on the grades obtained during the comparison of all input definitions, id_k *vs* id'_k[3], as shown in formula 16:

$$g_{ID}(C, C') = \sum_{k=1}^{|ID|} g_{id_k}(C, C') * w_{id_k} \tag{16}$$

where $id_k \in ID$. Considering the example on Listings 1 and 2, $k \in [1..2]$.

The weight of each element in ID, namely w_{id_k} is calculated based on the number of arguments in C, as shown in equation 17 where $k \in [1..|ID|]$, $id_k \in ID$:

$$w_{id_k} = \frac{1}{|ID|} \tag{17}$$

In the previous example $w_{id_1} = w_{id_2} = 50\%$.

The grade of id_k *vs* id'_k, namely $g_{id_k}(C, C')$ is calculated with regards to its type, name, and position (see equation 18).

$$g_{id_k}(C, C') \quad = \quad w_{type} * g_{type}(id_k, id'_k) +$$

[3]We use this notation in order to show that the grade of each element represents the result of the comparison of this element in both components, here id_k is compared versus id'_k.

$$w_{name} * g_{name}(id_k, id'_k) + w_{pos} * g_{pos}(id_k, id'_k) \tag{18}$$

where w_{type}, w_{pos}, and w_{name} are percentages indicating the weight of the associated grade, $w_{type}, w_{pos}, w_{name} \in [0..100]$, and $w_{type} + w_{pos} + w_{name} = 100$.

The percentages of type, position and name are given by maintainers and may vary based on their importance.

The grade $g_{type}(id_k, id'_k)$ returns 1 if id_k is of type or subtype of id'_k; 0 otherwise (see equation 19).

$$g_{type}(id_k, id'_k) = \begin{cases} 1 & \text{if } type(id'_k) \geq type(id_k) \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

Thus, in the example on Listings 1 and 2, $g_{type}(id_1, id'_1) = g_{type}(id_2, id'_2) = 1$.

The grade $g_{name}(id_k, id'_k)$ returns 1 if the name of id_k is equivalent to the name of id'_k or if id_k is substring of id'_k or vice versa; 0 otherwise (see equation 20).

$$g_{name}(id_k, id'_k) = \begin{cases} 1 & \text{if } name(id_k) \equiv name(id'_k) \\ & \vee substr(name(id_k),\ name(id'_k)) \\ & \vee substr(name(id'_k),\ name(id_k)) \\ 0 & \text{otherwise} \end{cases} \tag{20}$$

where $substr(a, b)$ returns $true$ if a is a substring of b. Thus, continuing on the example on Listings 1 and 2, we obtain $g_{name}(id_1, id'_1) = 0$ whereas $g_{name}(id_2, id'_2) = 1$.

The grade $g_{pos}(id_k, id'_k)$ returns 1 if the position of id_k is equal to the position of

id'_k; 0 otherwise (see equation 21).

$$g_{pos}(id_k, id'_k) = \begin{cases} 1 & \text{if } pos(id'_k) \equiv pos(id_k) \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

We can thus obtain $g_{pos}(id_1, id'_1) = g_{pos}(id_2, id'_2) = 1$ corresponding to the previous example.

The grade of *ot vs ot'*, namely $g_{ot}(C, C')$ returns 1 if ot' is of type or subtype of ot; 0 otherwise (see equation 22).

$$g_{ot}(C, C') = \begin{cases} 1 & \text{if } ot \geq ot' \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

Based on equation 22, we obtain $g_{ot}(C, C') = 1$ in the example in Listings 1 and 2.

The grade of *mn vs mn'*, namely $g_{mn}(C, C')$ returns 1 if mn is equivalent to mn' or if mn is substring of mn' or vice versa; 0 otherwise (see equation 23).

$$g_{mn}(C, C') = \begin{cases} 1 & \text{if } mn \equiv mn' \\ & \vee substr(mn,\ mn') \\ & \vee substr(mn',\ mn) \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

Considering the example on Listings 1 and 2, we obtain $g_{mn}(C, C') = 1$.

The grade of *v vs v'*, namely $g_v(C, C')$ returns 1 if v is equivalent to v' or if v' is less restrictive than v; 0 otherwise (see equation 24).

$$g_v(C, C') = \begin{cases} 1 & \text{if } v \leq v' \\ 0 & \text{otherwise} \end{cases} \tag{24}$$

Equation 24 results in $g_v(C, C') = 1$ in the previous example.

The grade of MOD *vs* MOD', namely $g_{MOD}(C, C')$ is calculated as shown in formula 25:

$$g_{MOD}(C, C') = \sum_{k=1}^{|MOD \cup MOD'|} g_{mod_k}(C, C') * w_{mod_k} \tag{25}$$

$\forall mod_k \in MOD \cup MOD'$, and where $g_{mod_k}(C, C')$ and w_{mod_k} are defined in equations 26 and 28 respectively. In the previous example $k = 1$.

$$g_{mod_k}(C, C') = in_C^{mod_k} * in_{C'}^{mod_k} \tag{26}$$

$\forall mod_k \in MOD \cup MOD'$, and where $in_C^{mod_k}$ is defined as shown in equation 27:

$$in_C^{mod_k} = \begin{cases} 1 & \text{if } mod_k \in MOD \\ 0 & \text{otherwise} \end{cases} \tag{27}$$

In the example on Listings 1 and 2, $in_C^{mod_1} = 1$, $in_{C'}^{mod_1} = 0$, which results in $g_{mod_1}(C, C') = 0$ and $g_{MOD}(C, C') = 0$.

The weight of each element in MOD, namely w_{mod_k} is calculated as shown in equation 28.

$$w_{mod_k} = \frac{1}{|MOD \cup MOD'|} \tag{28}$$

$\forall k \in \mathbb{N}, k = 1..\,|MOD \cup MOD'|, \forall mod_k \in MOD \cup MOD'$. In the previous example, $w_{mod_k} = 100\%$

The grade of EXC *vs* EXC', namely $g_{EXC}(C, C')$ is calculated as shown in formula 29.

$$g_{EXC}(C, C') = \sum_{k=1}^{|EXC'|} g_{exc_k}(C, C') * w_{exc_k} \tag{29}$$

where $exc_k \in EXC'$. $g_{EXC}(C, C') = 100\%$ if $EXC' = \{\}$. In the previous example, $g_{EXC}(C, C') = 100\%$.

The weight w_{exc_k} is calculated based on the number of exceptions thrown by C' as shown in equation 30, where $k \in [1..\,|EXC'|]$, $exc_k \in EXC'$.

$$w_{exc_k} = \frac{1}{|EXC'|} \tag{30}$$

The grade $g_{exc_k}(C, C')$ returns 1 if $\forall exc_k \in EXC'$ $\exists exc_j \in EXC$ such that exc_k is of type or subtype of exc_j; 0 otherwise (see equation 31).

$$g_{exc_k}(C, C') = \begin{cases} 1 & \text{if } \forall exc_k \in EXC' \; \exists exc_j \in EXC \bullet exc_j \geq exc_k \\ 0 & \text{otherwise} \end{cases} \tag{31}$$

where $k \in [1..|EXC'|]$, and $j \in [1..|EXC|]$.

Method state

The variables which are important for a method are 1) attributes of the class to which the method belongs to, because they reflect the state of the main concern (the class) in which the method is defined, 2) input of the method (its parameters), because they are needed by the method in order to provide the expected functionality, and 3) output of the method (what the method returns), because it represents the result of the performed calculations. In case the output corresponds to the result of a method call, the actual parameters of this call are taken as important variables, because this method call is the final functionality which needs to be performed before the method ends. Therefore, we consider the state of a component being represented by its important variables at each moment. We refer to them "state variables" (SV) and we define them as follows:

$$SV = I_m \cup o_m \cup AP_{rm} \cup A_c. \tag{32}$$

where I_m denotes the ordered set of the method formal parameters, o_m denotes the output of the method, AP_{rm} denotes the actual parameters passed to the method call in the return statements, and A_c denotes the set of attributes of the class to which the method belongs to. The elements in the set of state variables are represented by their names in order to be able to compare the concerns which they reflect.

By applying backward slicing [66] on each element in SV, we can obtain all statements and variables which affect the state variables. These statements are called *complete flow of events.* At this level of granularity, E and E' represent the set of statements of C and C', respectively, that modify the state variables and the variables affecting them. Every element in S and S' is represented by the set of variables which are modified after an event occurs. The two components are said to have the same states if equations 4, 5, and 6 hold.

If we filter the complete flow of events by considering only the statements which directly affect the elements of SV, we can obtain the *basic flow of events.* For less strict method matching, E and E' are represented by the basic flows of events of C and C' respectively. Here, $\forall i \in [1..|S|]$, $s_i \in S$, $s'_i \in S'$, $s_i \subseteq SV$, and $s'_i \subseteq SV'$.

We define a statement as directly affecting a variable *var* if at least one of the following holds:

1. This statement is an assignment statement, where the left hand side is *var*.
2. This statement is a variable declaration statement, where the left hand side is *var*.
3. This statement is a method call, where *var* is the callee or is part of the actual parameters of the method, in case the type of *var* is not a primitive type.
4. If the type of *var* is not a primitive type and this statement is an instance creation (constructor call), where *var* is part of the actual parameters of the constructor.
5. This statement is a return statement returning *var* or 1, 3 or 4.

6. This statement is a conditional statement or an *if* statement, where 1, 2, 3 or 4 hold for at least one statement in *then* and/or *else* part.

7. This statement is a *do* statement, a *while* statement, or a *for* statement, where 1, 2, 3 or 4 hold for at least one statement in the body of the statement.

Example Consider C in Listing 3. We can extract the set of state variables as follows: $I_m = \{firstArgument,\ secondArgument\}$, $o_m = a$, $AP_{rm} = \{\}$, and $A_c = \{a\}$, which result in the following set of state variables for C: $SV = \{firstArgument,$ $secondArgument,\ a\}$.

```
public class C {
    private Object a;
    public Object aMethod(Object firstArgument, int secondArgument){
        System.out.println("Beginning of aMethod.");
        secondArgument=secondArgument+1;
        if(firstArgument==null)
            firstArgument = new Object();
        a = firstArgument;
        System.out.println("End of aMethod.");
        return a;
    }
    // other methods ...
}
```

Listing 3: Example for calculating the state of method *aMethod*.

Extracting the basic flow of events of method *aMethod* results in the set of statements shown in Listing 4.

```
secondArgument=secondArgument+1;
if(firstArgument==null){
  firstArgument = new Object();
}
a = firstArgument;
return a;
```

Listing 4: Basic flow of events of method *aMethod*.

Overall state matching We define a "minimal word" as a meaningful substring which is extracted from the basic/complete flow of events. For defining a meaningful substring, we follow the Java convention for naming variables and methods, where every word (except the first one) should start with an upper case (e.g. the variable *methodName*, results in two minimal words, namely *method* and *name*).

We define $occ_C^{w_i}$ as the number of occurrences of the minimal word w_i in C.

The overall state matching at method level of two components C and C', namely $ostm_{ml}(C, C')$, is calculated based on formula 33. This formula calculates the state matching of the two components based on the degree of matching of all minimal words extracted from C and those extracted from C'. The importance of each minimal word is reflected through the number of occurrences of this word in C. Thus the weighted grade of each minimal word is the product of the word's existence in C' and its number of occurrences in C. The degree of matching of two states is equal to the sum of the weighted grades for all minimal words extracted from C, divided by the total number of considered minimal words (which is the sum of all occurrences of all minimal words extracted from C).

$$ostm_{ml}(C, C') = \frac{\sum_{i=1}^{n} occ_C^{w_i} * in_{C'}^{w_i}}{\sum_{i=1}^{n} occ_C^{w_i}} \tag{33}$$

where $n \in \mathbb{N}$, and corresponds to the number of minimal words extracted from C, w_i is a minimal word extracted from C, $occ_C^{w_i}$ is the number of occurrences of w_i in C, $1 \leq occ_C^{w_i}$. The value returned by $in_{C'}^{w_i}$ is 1 if w_i occurs at least once in C'; 0 otherwise.

Example The minimal words and their number of occurrences resulting from the basic flow of events of Listing 4 are shown in Table 1. If we assume that the set

Minimal word	**Occurrence**
a	2
null	1
object	1
second	2
1	1
argument	5
first	3

Table 1: Minimal words of C and their occurrence.

of minimal words of the candidate component C' is equivalent to the one shown in Table 1, then we obtain $ostm_{ml}(C, C') = 1$.

Required types

Objects interact through message passing. This interaction requires a visibility from the caller type to the callee type. Visibility is categorized as: attribute visibility, parameter visibility, local visibility, and global visibility [37]. If a component has visibility to a given type and if this component is calling a method or accessing an attribute of this type, then the type and the specific method/attribute are required for the proper functioning of the component.

A required type rt_i is characterized by the name of the package in which it is defined (pn_{rt_i}) because it represents the main concern the type of which it is part of, its name (n_{rt_i}) because it indicates the concern the type represents, the set of attributes accessed or modified by the requiring type (A_{rt_i}) because it indicates the required state of this type, and the set of methods called by the requiring type (M_{rt_i}) because it represents the required behavior of the type, i.e.

$$rt_i =< pn_{rt_i}, n_{rt_i}, A_{rt_i}, M_{rt_i} > \tag{34}$$

The sets of required types RT and RT' (of C and C' respectively) are considered to be equivalent if equation 35 holds:

$$\forall rt_i \in RT \; \exists rt'_i \in RT' \bullet rt_i \equiv rt'_i \tag{35}$$

where $i \in [1..|RT|]$, and $|RT| = |RT'|$.

Equation 35 holds if and only if rt_i and rt'_i have the same name (see equation 36), the name of the package in which they are defined is the same (see equation 37), and the required attributes and methods of rt_i are equivalent to the required attributes and methods of rt'_i (equations 38 and 39).

$$n_{rt_i} \equiv n_{rt'_i} \tag{36}$$

$$pn_{rt_i} \equiv pn_{rt'_i} \tag{37}$$

$$A_{rt_i} \equiv A_{rt'_i} \tag{38}$$

$$M_{rt_i} \equiv M_{rt'_i} \tag{39}$$

Each element in A_{rt} is represented by the triplet $< visibility, type, name >$. The equivalence should then hold for each of these elements.

At method level, however, one may be more interested in the methods and attributes of a required type rather than its name only. Thus, at this level, more detailed information will be compared and the possibility of abstracting it to class level (the name of the required types) and package level (the required packages) will be provided.

For less strict component matching, equivalence between required types is defined as shown in equation 40, where the order in which the required types are used is not important.

$$\forall rt \in RT\ \exists rt' \in RT' \bullet rt \equiv rt' \tag{40}$$

For less strict component matching, similarly to the comparison of method names in Section 5.2.1, the equivalence relation in equations 36 and 37 is replaced with symmetric substring. In addition, in equation 38 A_{rt_i} is a subset of $A_{rt'_i}$. Assume $a_{rt_i} \in A_{rt_i}$, $type_{a_{rt'_i}}$ is a subtype of $type_{a_{rt_i}}$; $visibility_{a_{rt_i}}$ is equal or more restrictive to $visibility_{a_{rt'_i}}$; $name_{a_{rt_i}}$ is equal to or is substring of $name_{a_{rt'_i}}$ or vice versa.

Example Consider the example shown in Listing 3. The extracted required packages, types, and features of method *aMethod* are shown in Figure 11.

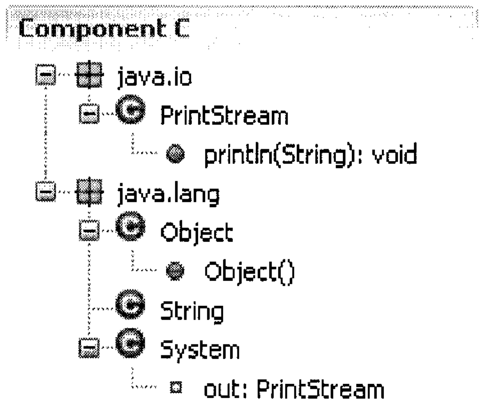

```
Component C
java.io
  PrintStream
    println(String): void
java.lang
  Object
    Object()
  String
  System
    out: PrintStream
```

Figure 11: Required packages, types, and features of method *aMethod*.

Overall behavior matching The overall behavior matching at method level of two components C and C', $obm_{ml}(C, C')$, is given by equation 41, which calculates the summation of all grades given to the existence of the required features of C in C'; the result is divided by the total number of required features in C.

$$obm_{ml}(C, C') = \frac{\sum_{i=1}^{n} in_{C'}^{rf_i}}{n} \tag{41}$$

where rf_i is a required feature by C, $in_{C'}^{rf_i}$ as the existence of feature rf_i in C', and where n corresponds to the total number of the features required by C.

The same formula is defined at type level in order to zoom out as shown in formula 42.

$$obm_{ml}(C,C') = \frac{\sum_{i=1}^{n} in_{C'}^{rt_i}}{n} \tag{42}$$

where rt_i is a required type by C, $in_{C'}^{rt_i}$ as the existence of rt_i in C', and where n corresponds to the total number of the types required by C.

Formula 43 allows yet another level of abstraction where the required packages are analyzed.

$$obm_{ml}(C,C') = \frac{\sum_{i=1}^{n} in_{C'}^{rp_i}}{n} \tag{43}$$

where rp_i is a required package by C, $in_{C'}^{rp_i}$ as the existence of rp_i in C', and where n corresponds to the total number of the packages required by C.

Overall component matching The overall component matching, $ocm_{ml}(C,C')$, is calculated based on the overall definition matching, overall state matching, and overall behavior matching, where the importance of all these factors is defined by their respective weights (see equation 44):

$$\begin{aligned} ocm_{ml}(C,C') &= w_{odm} * odm_{ml}(C,C') + \\ &\quad w_{ostm} * ostm_{ml}(C,C') + w_{obm} * obm_{ml}(C,C') \end{aligned} \tag{44}$$

where w_{odm}, w_{ostm}, w_{obm} are percentages indicating the importance of the associated grade, w_{odm}, w_{ostm}, $w_{obm} \in [0..100]$, $w_{odm} + w_{ostm} + w_{obm} = 100$, and where $obm_{ml}(C, C')$ is calculated based on formula 41, 42 or 43.

5.2.2 Class level criteria

The Liskov principle of substitutability [40] states that if $t2$ is a subtype of $t1$, then an instance of type $t2$ can be used every time an instance of type $t1$ is expected. Consequently, a type can be substituted with any of its subtypes. The situation where one would like to substitute a class with one of its subclasses, however, does not represent a challenge. More often, one would substitute a component C with a component C' where the two components are completely different and thus do not hold the is-a relationship.

We define substitutability criteria for components which are not related in the class hierarchy.

Classes definition

Two classes are said to have the same definition if the following six conditions hold:

1. They have the same visibility modifiers. The discussion on why visibility modifiers are important at method level also applies here. At this level $V = \{public$, default, $private\}$ [38]. For less strict matching the visibility of the new class may be less restrictive, compared to the visibility of the substituted class.

2. They have the same modifiers. At this level of granularity $\forall mod \in MOD$,

$mod = abstract \;|final$ ($final$ being applicable only for classes [38]).

3. They belong to the same type of module. It is important to compare the type of the modules because different types have different purposes. For example, a class provides state and behavior whereas an interface imposes to its subtypes to provide this behavior. Assume $tm \in TM$ and $tm' \in TM'$ be the module types of C and C' respectively and $TM = \{class, interface\}$, $TM' = \{class, interface\}$, then:

$$tm \equiv tm' \tag{45}$$

4. They have equivalent upper inheritance chain. Upper inheritance chain should be compared because it reflects the state and behavior that this class implicitly provides. Assume PM and PM' be the ordered sets containing the parent modules for C and C' respectively. Equation 46 should hold:

$$\forall pm_i \in PM \; \exists pm'_i \in PM' \bullet pm_i \equiv pm'_i \tag{46}$$

where $i \in [1..|PM|]$ and $|PM| = |PM'|$. For non strict class matching $PM \subseteq PM'$ or vice versa, and pm'_i can be a subtype of pm_i.

5. They have the same name. Comparing the names of two components is equivalent to comparing the concerns these components implement. Similarly to method level specification, the strict class specification matching requires equivalence whereas more lax matching allows that the name of one of the components

is a substring of the name of the other.

6. They declare equivalent interfaces, that is, equivalent definitions for attributes and methods. It is essential to compare attributes and methods because they represent the state and functionality the class provides. Only definitions will be compared at this stage whereas the state and behavior of the defined methods will be compared in the state and behavior of the class. Let DA and DA' be the defined attributes and DM and DM' be the defined methods in C and C' respectively. The two components are said to have equivalent interfaces if equations 47 and 48 hold.

$$\forall da \in DA \; \exists da' \in DA' \bullet da \equiv da' \tag{47}$$

which means that the set of defined attributes in C is equivalent to the set of defined attributes in C'. Every element in DA and DA' is defined similarly to the elements in A_{rt} in Section 5.2.1 and thus the discussion regarding lax attribute matching is also applicable here.

$$\forall dm \in DM \; \exists dm' \in DM' \bullet dm \equiv dm' \tag{48}$$

which means that the set of defined methods in C is equivalent to the set of defined methods in C'. It is important to note that only the method definitions are compared here. To decide whether two methods have equivalent definitions, their definition should be compared (see Section 5.2.1).

More often, we do not need one to one correspondence (bijection). That is, we require that for every defined attribute/method in C there exists an equivalent attribute/method defined in C'. Consequently, it is sufficient that DA and DM are subsets of DA' and DM' respectively (injection). However, maintainers should be aware that introducing non-required attributes/methods may result in non-intended behavior of the system caused by attribute hiding, method overriding [38] etc.

Overall definition matching The overall definition matching at class level for two components C and C', namely $odm_{cl}(C, C')$ is calculated based on the level of matching of their visibility, modifiers, type, parent types, name, defined attributes, and defined methods (see equation 49). The importance of these factors is defined by their respective weights.

$$\begin{aligned} odm_{cl}(C, C') = \; & w_v * g_v(C, C') + w_{MOD} * g_{MOD}(C, C') + \\ & w_{tm} * g_{tm}(C, C') + w_{PM} * g_{PM}(C, C') + \\ & w_{cn} * g_{cn}(C, C') + w_{DA} * g_{DA}(C, C') + \\ & w_{DM} * g_{DM}(C, C') \end{aligned} \quad (49)$$

where w_v, w_{MOD}, w_{tm}, w_{PM}, w_{cn}, w_{DA}, and w_{DM} are percentages corresponding to the weight of the corresponding grade; $w_v,\ w_{MOD},\ w_{tm},\ w_{PM},\ w_{cn},\ w_{DA},\ w_{DM} \in [0..100]$, and $w_v + w_{MOD} + w_{tm} + w_{PM} + w_{cn} + w_{DA} + w_{DM} = 100$,

$g_v(C,C')$, $g_{MOD}(C,C')$, $g_{tm}(C,C')$, $g_{PM}(C,C')$, $g_{cn}(C,C')$, $g_{DA}(C,C')$, and $g_{DM}(C,C')$ are percentages indicating the degree of matching of the associated element, and $g_v(C,C')$, $g_{MOD}(C,C')$, $g_{tm}(C,C')$, $g_{PM}(C,C')$, $g_{cn}(C,C')$, $g_{DA}(C,C')$, $g_{DM}(C,C') \in [0..100]$.

The grade of v *vs* v', $g_v(C,C')$ and the grade of MOD *vs* MOD', $g_{MOD}(C,C')$ are calculated as defined at method level (see Section 5.2.1).

The grade of tm *vs* tm', $g_{tm}(C,C')$ returns 1 if tm is equivalent to tm'; 0 otherwise (see equation 50).

$$g_{tm}(C,C') = \begin{cases} 1 & \text{if } tm \equiv tm' \\ 0 & \text{otherwise} \end{cases} \tag{50}$$

The grade of PM *vs* PM', $g_{PM}(C,C')$ is calculated as shown in formula 51.

$$g_{PM}(C,C') = \sum_{k=1}^{|PM|} g_{pm_k}(C,C') * w_{pm_k} \tag{51}$$

where $pm_k \in PM$.

The weight w_{pm_k}, is calculated based on the number of parents of C as shown in equation 52, where $k \in [1..|PM|]$, and $pm_k \in PM$.

$$w_{pm_k} = \frac{1}{|PM|} \tag{52}$$

The grade $g_{pm_k}(C,C')$ returns 1 if $\forall pm_k \in PM$, $\exists pm'_j \in PM'$ such that pm'_j is of type or subtype of pm_k; 0 otherwise (see equation 53).

$$g_{pm_k}(C, C') = \begin{cases} 1 & \text{if } \forall pm_k \in PM \; \exists pm'_j \in PM' \bullet pm_k \geq pm'_j \\ 0 & \text{otherwise} \end{cases} \tag{53}$$

where $k \in [1..|PM|]$, and $j \in [1..|PM'|]$.

The grade of cn *vs* cn', $g_{cn}(C, C')$ returns 1 if cn is equivalent to cn' or if cn is substring of cn' or vice versa; 0 otherwise (see equation 54):

$$g_{cn}(C, C') = \begin{cases} 1 & \text{if } cn \equiv cn' \\ & \vee substr(cn, cn') \\ & \vee substr(cn', cn) \\ 0 & \text{otherwise} \end{cases} \tag{54}$$

The grade of DA *vs* DA', that is $g_{DA}(C, C')$ is calculated based on the average of the grades $g_{da_k}(C, C')$ as shown in formula 55, where $da_k \in DA$:

$$g_{DA}(C, C') = \sum_{k=1}^{|DA|} g_{da_k}(C, C') * w_{da_k} \tag{55}$$

The weight of each element in DA, namely w_{da_k} is calculated based on the number of defined attributes in C (see equation 56):

$$w_{da_k} = \frac{1}{|DA|} \tag{56}$$

where $k \in [1..|DA|]$, and $da_k \in DA$.

The grade $g_{da_k}(C, C')$ is calculated with regards to its visibility, modifiers, type, and name and their corresponding weights (see equation 57):

$$\begin{aligned} g_{da_k}(C, C') &= w_v * g_v(da_k) + w_{MOD} * g_{MOD}(da_k) + \\ & \quad w_{type} * g_{type}(da_k) + w_{name} * g_{name}(da_k) \end{aligned} \tag{57}$$

where w_v, w_{MOD}, w_{type}, and w_{name} are percentages corresponding to the weights of their corresponding grades, w_v, w_{MOD}, w_{type}, $w_{name} \in [0..100]$, and $w_v + w_{MOD} + w_{type} + w_{name} = 100$.

The grade $g_v(da_k)$ returns 1 if the visibility of da_k, namely v is equivalent to the visibility of da'_j ($da'_j \in DA'$), namely v' or if v' is less restrictive than v; 0 otherwise (see equation 58).

$$g_v(da_k) = \begin{cases} 1 & \text{if } v \leq v' \\ 0 & \text{otherwise} \end{cases} \tag{58}$$

The grade $g_{MOD}(da_k)$, is calculated as defined in Section 5.2.1 (see equation 25).

The grade of $g_{type}(da_k)$, returns 1 if the type t' of da'_j is of type or subtype of the type t of da_k; 0 otherwise (see equation 59).

$$g_{type}(da_k) = \begin{cases} 1 & \text{if } t \geq t' \\ 0 & \text{otherwise} \end{cases} \tag{59}$$

The grade of $g_{name}(da_k)$ returns 1 if the name of da_k, namely an is equivalent to the name of da'_j, namely an' or if an is substring of an' or vice versa; 0 otherwise (see equation 60).

$$g_{name}(da_k) = \begin{cases} 1 & \text{if } an \equiv an' \\ & \vee substr(an,\ an') \\ & \vee substr(an,\ an) \\ 0 & \text{otherwise} \end{cases} \tag{60}$$

The grade of DM *vs* DM', that is $g_{DM}(C, C')$ is calculated based on the average of the grades $g_{dm_k}(C, C')$ as shown in formula 61, where $dm_k \in DM$:

$$g_{DM}(C, C') = \sum_{k=1}^{|DM|} g_{dm_k}(C, C') * w_{dm_k} \tag{61}$$

The weight of each element in DM, namely w_{dm_k}, is calculated based on the number of defined methods in C as shown in equation 62, where $k \in [1..|DA|]$, and $da_k \in DM$.

$$w_{dm_k} = \frac{1}{|DM|} \tag{62}$$

The grade $g_{dm_k}(C, C')$ is calculated based on formula 15.

Class state

The state of a component C is defined by the state of the set of its methods M. In order to compare the states of C and C', one should compare the states of every method $m_i \in M$ and $m'_j \in M'$ defined in the components (see Section 5.2.1). The attributes of a class also express its state but we do not address them explicitly. They are implicitly taken into consideration through the state of the methods because they are part of the set of state variables for each method.

For less strict state matching, one may be interested only in having part of the provided functionality of C. This may be the case when some functionality is never used. Thus it would be sufficient to compare the states only of a subset of the functionalities provided by C.

Overall state matching The overall state matching of two components C and C' at class level, namely $ostm_{cl}(C, C')$ is defined as shown in formula 63, where $m_i \in M$, and $m'_j \in M'$:

$$ostm_{cl}(C, C') = \sum_{i=1}^{|M|} ostm_{ml}(m_i, m'_j) * w_{m_i} \tag{63}$$

The weight of the grade of each overall state matching at method level, w_{m_i}, is calculated based on the number methods in C as shown in equation 64, where $i \in [1..|M|]$, and $m_i \in M$:

$$w_{m_i} = \frac{1}{|M|} \tag{64}$$

The overall state matching at method level, $ostm_{ml}(m_i, m'_j)$, is calculated as shown in formula 33.

Required types

At this level, the required types of C represent the union of the required types of the methods of C because they constitute the set of functionalities that C provides and the declared types of its attributes because these types are necessary in order to express the state of C. Assume t being the type of a defined attribute da, $da \in DA$, and rt being a required type of method of C, $rt \in RT$. Equation 65 should hold:

$$RT = \{x | x = t \vee x = rt\} \tag{65}$$

A matching based on a one-to-one correspondence between RT and RT' may not be required. It may be sufficient that RT is a subset of RT'. However, should RT be a subset of RT', the effort for substituting the components may increase in some cases since new types may be introduced and/or conflicts may occur between existing types in the system and the newly introduced required types.

At this level of granularity, one may be interested in omitting detailed information. Thus, it would be interesting here to take into consideration the required types as black boxes with the possibility to zoom in (and have the methods and attributes of the required types) or to zoom out (take into consideration only the packages) when necessary.

Overall behavior matching The overall behavior matching of two components C and C' at class level, namely $obm_{cl}(C, C')$ is defined as shown in formula 66.

$$obm_{cl}(C, C') = w_{b_h} * (\sum_{i=1}^{|M|} obm_{ml}(m_i, m'_j) + \sum_{k=1}^{|DA|} g_{da_k}(C, C')) \tag{66}$$

where $h = |M| + |DA|$, $m_i \in M$, $m'_j \in M'$, and $da_k \in DA$.

The weight w_{b_h} is calculated based on the number methods and attributes in C (see equation 67).

$$w_{b_h} = \frac{1}{|M| + |DA|} \tag{67}$$

$obm_{ml}(m_i, m'_j)$ is calculated as shown in formula 41, 42 or 43.

Overall component matching The overall component matching, $ocm_{cl}(C, C')$, is calculated as shown in equation 68.

$$\begin{aligned} ocm_{cl}(C, C') &= w_{odm} * odm_{cl}(C, C') + \\ &\quad w_{ostm} * ostm_{cl}(C, C') + w_{obm} * obm_{cl}(C, C') \end{aligned} \tag{68}$$

where w_{odm}, w_{ostm}, and w_{obm} are percentages defining the importance of the corresponding grade, $w_{odm}, w_{ostm}, w_{obm} \in [0..100]$, and $w_{odm} + w_{ostm} + w_{obm} = 100$.

5.2.3 Package level criteria

Similar to the abstraction made in 5.2.2, the criteria for substituting packages are defined by reusing the criteria defined in the level below. The definition, state, and behavior of a package are defined mainly by the definition, state, and behavior of the modules defined in it.

Package definition

Two packages are said to have the same definition if the following two conditions hold:

1. They have the same name and hierarchy *pn*. The complete package name is important because it contains the high level concern the package represents.
2. They define the same modules. Modules are vital to compare because they fulfill the functionality which the package is expected to provide. Let DT and DT' be the sets of defined types of C and C' respectively:

$$\forall dt \in DT \; \exists dt' \in DT' \bullet dt \equiv dt' \tag{69}$$

Overall definition matching The overall definition matching at package level for two components C and C', namely $odm_{pl}(C, C')$ is calculated based on the level of matching of their names and the types they define compared at class level (see formula

70):

$$odm_{pl}(C, C') = w_{pn} * g_{pn}(C, C') + \sum_{i=1}^{|DT|} w^d_{dt_i} * odm_{cl}(dt_i, dt'_j) \tag{70}$$

where $odm_{cl}(C, C')$ is defined in formula 49, $i \in [1..|DT|]$, $j \in [1..|DT'|]$, $dt_i \in DT$, $dt'_j \in DT'$, and where $w_{pn} + \sum_{i=1}^{|DT|} w^d_{dt_i} = 100$.

The grade $g_{pn}(C, C')$ returns 1 if pn is equivalent to pn' or if pn is a substring of pn' or vice versa; 0 otherwise (see equation 71).

$$g_{pn}(C, C') = \begin{cases} 1 & \text{if } pn \equiv pn' \vee substr(pn, pn') \vee substr(pn', pn) \\ 0 & \text{otherwise} \end{cases} \tag{71}$$

Package state

The set of defined modules in a package constitute the package itself. Thus, the states of C and C' are said to be equivalent if they define modules with equivalent states (see Section 5.2.2).

Overall state matching The overall state matching of two components C and C' at package level, namely $ostm_{pl}(C, C')$ is defined as shown in formula 72.

$$ostm_{pl}(C, C') = \sum_{i=1}^{|DT|} ostm_{cl}(dt_i, dt'_j) * w^s_{dt_i} \tag{72}$$

where $dt_i \in DT$, $dt'_j \in DT'$.

The weight of each element $w^s_{dt_i}$ is calculated based on the number types in C as shown in equation 73, where $i \in [1..|DT|]$, $t_i \in DT$:

$$w_{t_i} = \frac{1}{|DT|} \tag{73}$$

The overall state matching at class level $ostm_{cl}(dt_i, dt'_j)$ is calculated as shown in formula 63.

Required types

As for the state at package level, the behavior of a package consists of the behavior of its defined modules. Thus, the required types of C represent the union of the required types of the modules defined in C. Similarly to the class level criteria, it may not be necessary to perform exact matching. In either case, the consequences should be analyzed.

Overall behavior matching The overall behavior matching of two components C and C' at package level, namely $obm_{pl}(C, C')$ is defined based on the overall behavior matching at class level calculated for all types in C as shown in formula 74, where $j \in [1..|DT'|]$, $dt_i \in DT$, and $dt'_j \in DT'$:

$$obm_{pl}(C, C') = \sum_{i=1}^{|DT|} obm_{cl}(dt_i, dt'_j) * w^b_{dt_i} \tag{74}$$

The weight of each element $w^b_{dt_i}$ is calculated based on the number of types in C

as shown in equation 75, where $i \in [1..|DT|]$, and $dt_i \in DT$:

$$w^b_{dt_i} = \frac{1}{|DT|} \tag{75}$$

The overall behavior matching at class level $obm_{cl}(dt_i, dt'_j)$ is calculated as shown in formula 66.

Overall component matching The overall component matching, $ocm_{pl}(C, C')$, is calculated as shown in equation 76.

$$\begin{aligned} ocm_{pl}(C, C') &= w_{odm} * odm_{pl}(C, C') + \\ &\quad w_{ostm} * ostm_{pl}(C, C') + w_{obm} * obm_{pl}(C, C') \end{aligned} \tag{76}$$

where w_{odm}, w_{ostm}, and w_{obm} are percentages indicating the importance of the associated grade, $w_{odm}, w_{ostm}, w_{obm} \in [0..100]$, and $w_{odm} + w_{ostm} + w_{obm} = 100$.

In the next chapter, we apply our methodology on different case studies in order to show cases where the compared components have high or low level of substitutability.

Chapter 6

Case study

In this chapter we illustrate our approach on different case studies. We first describe cases where the compared components have high level of matching (see Section 6.1) followed by examples where the components are not substitutable (see Section 6.2).

6.1 High level matching examples

We illustrate our approach on a modified version of Debrief - an open source Java application used for viewing maritime vessel tracks in two and three dimensions [14]. Debrief utilizes XML as an exchange format to retrieve and store plot data to analyze and create tracks. XML itself is the most popular technology for structuring data and thus XML-based encryption is the natural way to handle complex requirements for security in data interchange applications.

For the purpose of this research, we modified the original Debrief application by adding an encrypt/decrypt component [27]. The encryption in Debrief is performed using method *encrypt* in class *Debrief.Tools.Operations.Encrypt_Test.* Its implementation is as shown in Appendix A, Listing 1 and we will refer to this method as C (which represents the component to be substituted).

Our approach is applied should one want to substitute the encryption component in Debrief. Let this be the case and let the candidate secure component be the XML encryption component by Ray Djajadinata [16]. We deploy our approach to decide on the substitutability level on these components.

The choice of the candidate method in class *XMLEncryption* is intuitive based on the name of the method. Since there exist two methods called *encrypt* (see Appendix A, Listings 2 and 3), we decided to analyze both components starting with the method in Listing 3. We refer to it as C' (which represents the candidate component for the substitution).

In the rest of this chapter we apply the substitutability criteria for non strict matching at method level defined in Section 5.2.1 on C and C'.

6.1.1 Definition

Extracting the necessary information at method level from the definitions of C and C' result in the following:

$ID = \{String\ output,\ String\ xmloutput\}$

$ot = void$

$mn = encrypt$

$v = public$

$MOD = \{static\}$

$EXC = \{Exception\}$

$ID' = \{String\ select,\ KeyInfoResolver\ kiResolver,\ EncryptedType\ encType,\ String\ type,\ EncryptionMethod\ method,\ KeyInfo\ keyInfo\}$

$ot' = Node$

$mn' = encrypt$

$v' = public$

$MOD' = \{\}$

$EXC' = \{XMLEncryptionException\}$

The weight of each element w_{id_k} is calculated based on the number of arguments in C (see equation 17). In this case $w_{id_k} = 50\%$ where $k \in [1..2]$. $g_{id_k}(C, C')$ is calculated based on the average of the grades obtained during the comparison of each input definition id_k *vs* id'_k (see Table 2). The grade of id_k *vs* id'_k is calculated with regards to its type (here 40%), position (here 40%) and name (here 20%). Thus, for example since id_1 and id'_1 are declared of the same type (40/40), are at the same position (40/40) but have different name (0/20) the grade after the comparison is 80%. In a similar manner we obtain 40% of matching between id_2 and id'_2. After the normalization we obtain $g_{ID}(C, C') = 60\%$.

The result of matching of $g_{ot}(C, C')$ is 0% because *Node* is not of type *void*.

However, $g_{mn}(C, C')$, together with $g_v(C, C')$, result in 100% matching because C and C' have the same name and visibility.

Input definition	**Grade**	**Weight**	**Weighted grade**
$g_{id_1}(C, C')$	80%	50%	40%
$g_{id_2}(C, C')$	40%	50%	20%
		$g_{ID}(C, C')$:	60%

Table 2: Input definition matching $g_{ID}(C, C')$ for method *encrypt* (Appendix A, Listing 3).

$g_{MOD}(C, C')$ results in 0% because MOD and MOD' do not match.

$g_{EXC}(C, C')$ results in 100% match because EXC and EXC' contain the same number of elements and each element in EXC is a super type of the element in EXC'.

We calculate the overall definition matching $odm_{ml}(C, C')$ while giving more weight on the input, output, and name of a component (see Table 3).

Compared elements	**Grade**	**Weight**	**Weighted grade**
$g_{ID}(C, C')$	60%	25%	15%
$g_{ot}(C, C')$	0%	25%	0%
$g_{mn}(C, C')$	100%	20%	20%
$g_v(C, C')$	100%	10%	10%
$g_{MOD}(C, C')$	0%	10%	0%
$g_{EXC}(C, C')$	100%	10%	10%
		$odm_{ml}(C, C')$:	55%

Table 3: Overall definition matching $odm_{ml}(C, C')$ for method *encrypt* (Appendix A, Listing 3).

The weight of each element in Table 3 is given by maintainers and may vary based on its importance.

6.1.2 State

For extracting the state of the two components we first construct the set of important variables as follows:

$I_m = \{output,\ xmloutput\}$

$o_m = null$

$AP_{rm} = \{document,\ xmloutput\}$

$A_c = \{\}$

Note that since there is no return statement in C, the last statement will be considered as the result of this component. Thus the elements of AP_{rm} are the actual parameters of the last statement. The set of state variables for C is the union of I_m, o_m, AP_{rm} and A_c:

$SV = \{output,\ xmloutput,\ document\}$

Similarly, the important variables for C' are as follows:

$I'_m = \{select,\ kiResolver,\ encType,\ type,\ method,\ keyInfo\}$

$o'_m = null$

$AP'_{rm} = \{select,\ kiResolver,\ encTypeElement,\ type,\ methodElement,$
$\quad keyInfoElement\}$

$A'_c = \{\_node\}$

Resulting in the following set of state variables for C':

$SV' = \{\_node,\ select,\ kiResolver,\ encType,\ type,\ method,\ keyInfo,$
$\quad encTypeElement,\ methodElement,\ keyInfoElement\}$

By applying backward slicing on every element in SV and SV', we extract the

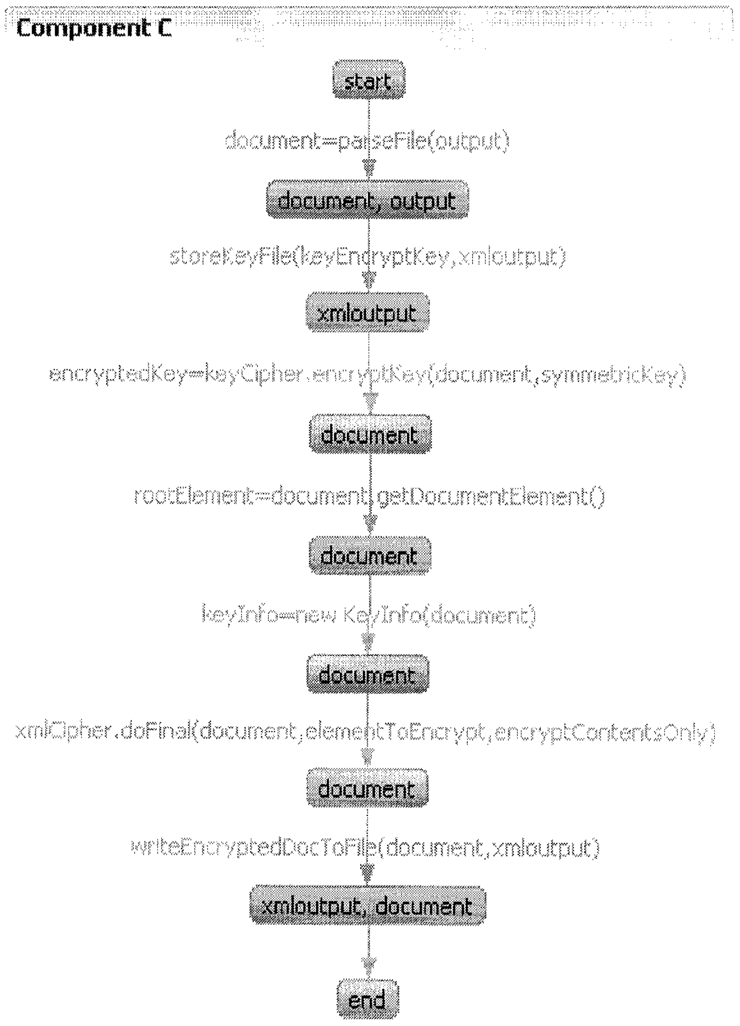

Component C
start
document=parseFile(output)
document, output
storeKeyFile(keyEncryptKey,xmloutput)
xmloutput
encryptedKey=keyCipher.encryptKey(document,symmetricKey)
document
rootElement=document.getDocumentElement()
document
keyInfo=new KeyInfo(document)
document
xmlCipher.doFinal(document,elementToEncrypt,encryptContentsOnly)
document
writeEncryptedDocToFile(document,xmloutput)
xmloutput, document
end

Figure 12: State diagram of C.

basic flow of events for the two components (see Figures 12 and 13).

The minimal words of C and C' are next extracted from their basic flow of events (see Tables 4 and 5 respectively).

Applying formula 33 to C and C' results in 39% lax matching of the states of the two components (see Table 6).

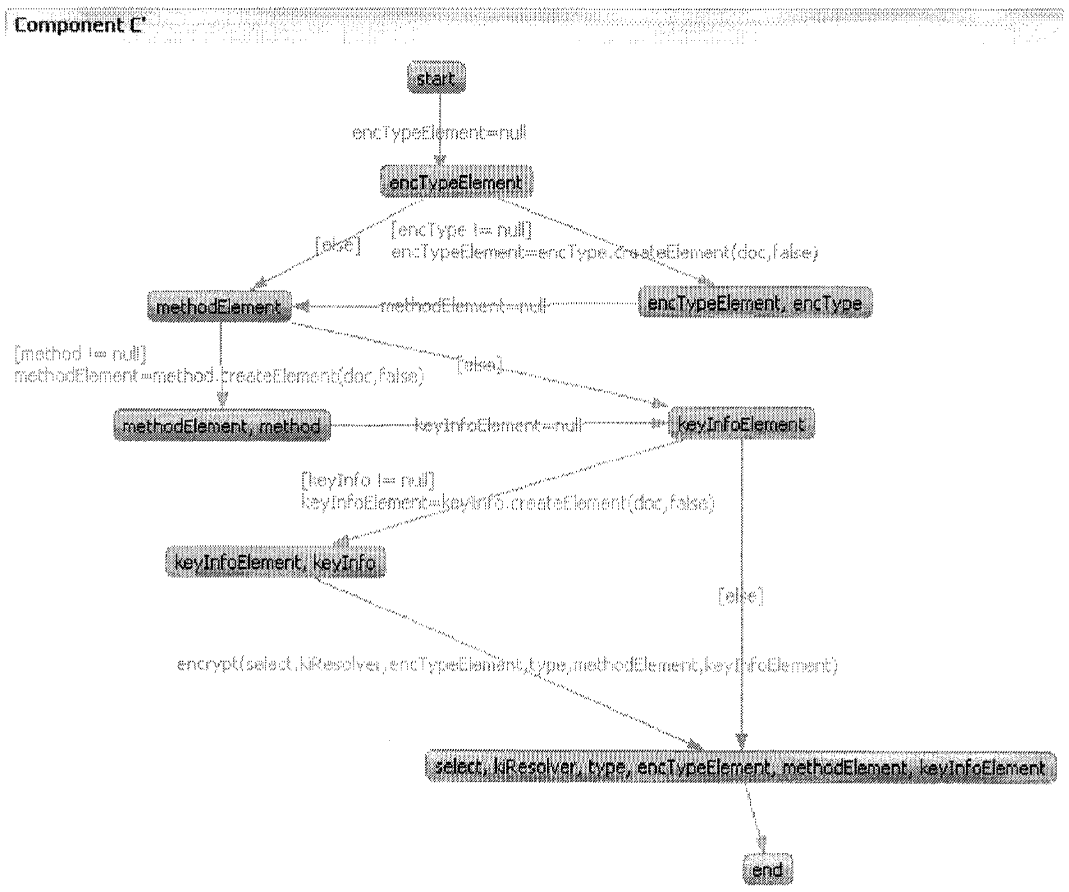

Component C'
start
encTypeElement=null
encTypeElement
[else]
[encType != null]
encTypeElement=encType.createElement(doc,false)
methodElement
methodElement=null
encTypeElement, encType
[method != null]
methodElement=method.createElement(doc,false)
[else]
methodElement, method
keyInfoElement=null
keyInfoElement
[keyInfo != null]
keyInfoElement=keyInfo.createElement(doc,false)
keyInfoElement, keyInfo
[else]
encrypt(select,kiResolver,encTypeElement,type,methodElement,keyInfoElement)
select, kiResolver, type, encTypeElement, methodElement, keyInfoElement
end

Figure 13: State diagram of method *encrypt* (Appendix A, Listing 3).

Minimal word	Occurrence
cipher	2
xmloutput	2
element	3
to	2
final	1
xml	1
store	1
symmetric	1
output	1
only	1
encrypt	4
parse	1
document	7
do	1
root	1
contents	1
write	1
info	2
encrypted	2
file	3
get	1
doc	1
key	9

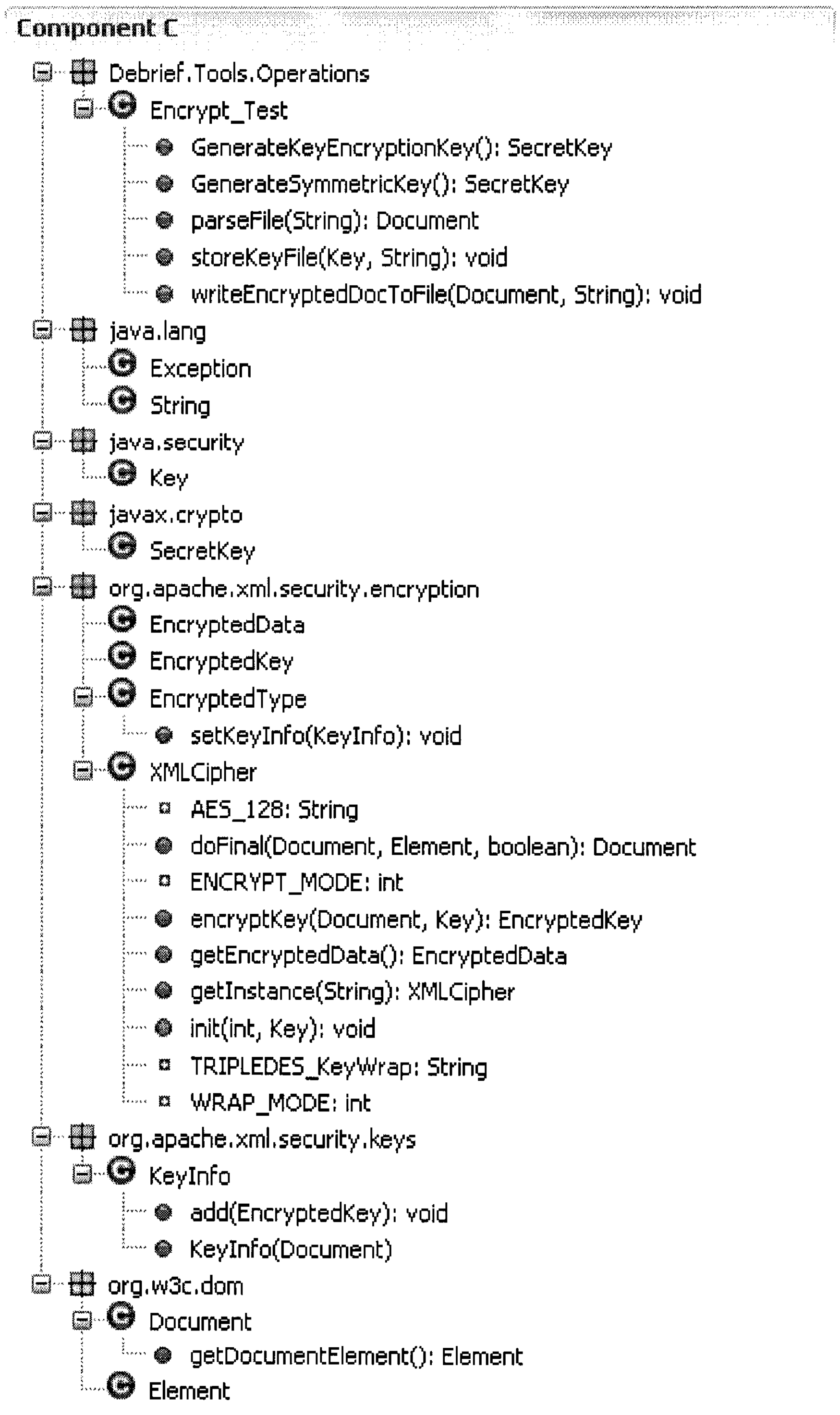
Table 4: Minimal words of C and their occurrence.

6.1.3 Required types

The required types of the two components are summarized in Figures 14 and 15. A type with no features may correspond to parameter passing, a return type or a local variable assignment/declaration.

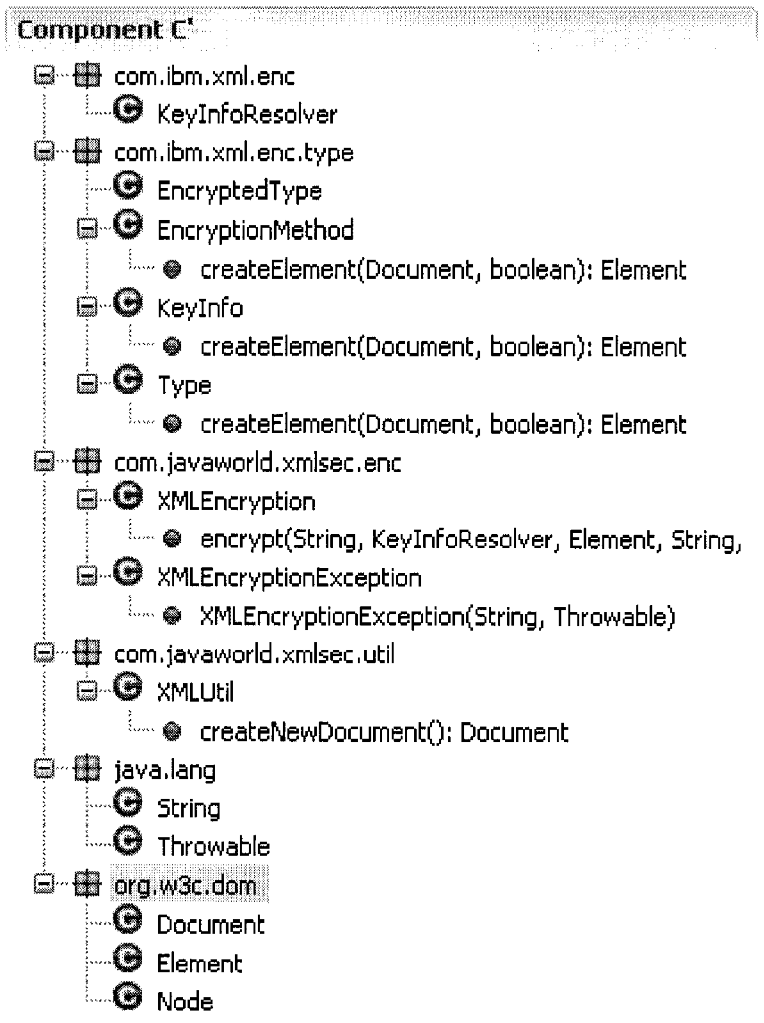
As defined in Section 5.2.1, it may be interesting to omit detailed information by zooming out on the required types and taking into consideration only the names of the packages in which they are defined, rather than the names of the required types and features.

The comparison of the required types is calculated at package level and the details



```
Component C
Debrief.Tools.Operations
    Encrypt_Test
        GenerateKeyEncryptionKey(): SecretKey
        GenerateSymmetricKey(): SecretKey
        parseFile(String): Document
        storeKeyFile(Key, String): void
        writeEncryptedDocToFile(Document, String): void
java.lang
    Exception
    String
java.security
    Key
javax.crypto
    SecretKey
org.apache.xml.security.encryption
    EncryptedData
    EncryptedKey
    EncryptedType
        setKeyInfo(KeyInfo): void
    XMLCipher
        AES_128: String
        doFinal(Document, Element, boolean): Document
        ENCRYPT_MODE: int
        encryptKey(Document, Key): EncryptedKey
        getEncryptedData(): EncryptedData
        getInstance(String): XMLCipher
        init(int, Key): void
        TRIPLEDES_KeyWrap: String
        WRAP_MODE: int
org.apache.xml.security.keys
    KeyInfo
        add(EncryptedKey): void
        KeyInfo(Document)
org.w3c.dom
    Document
        getDocumentElement(): Element
    Element
```

Figure 14: Required packages, types and features for C.



```
Component C'
com.ibm.xml.enc
    KeyInfoResolver
com.ibm.xml.enc.type
    EncryptedType
    EncryptionMethod
        createElement(Document, boolean): Element
    KeyInfo
        createElement(Document, boolean): Element
    Type
        createElement(Document, boolean): Element
com.javaworld.xmlsec.enc
    XMLEncryption
        encrypt(String, KeyInfoResolver, Element, String,
    XMLEncryptionException
        XMLEncryptionException(String, Throwable)
com.javaworld.xmlsec.util
    XMLUtil
        createNewDocument(): Document
java.lang
    String
    Throwable
org.w3c.dom
    Document
    Element
    Node
```

Figure 15: Required packages, types and features for method *encrypt* (Appendix A, Listing 3).

Minimal word	Occurrence
method	5
ki	1
info	5
resolver	1
enc	5
element	12
create	3
null	6
encrypt	1
type	6
select	1
doc	3
key	5
false	3

Table 5: Minimal words of C' and their occurrence for method *encrypt* (Appendix A, Listing 3).

are shown in Table 7.

6.1.4 Overall matching

Finally, the result of the overall matching is summarized in Table 8, where w_{odm}, w_{ostm}, and w_{obm} has been assigned 40%, 30% and 30% respectively.

Similarly, we analyzed the other method *encrypt* in class *XMLEncryption* as a candidate component for the substitution (see Appendix A, Listing 2). The results of this analysis are summarized in Table 9 regarding the definition matching, in Figure 16 and in Tables 10 and 11 regarding the state matching, and in Figure 17 regarding the behavior matching. Table 12 presents the overall matching results.

Once this analysis completed, one may want to proceed with the more expensive analysis, which is to consider complete flow of events. The results of this analysis are summarized in Table 13 for method *encrypt* (Appendix A, Listing 1), in Tables 14

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
cipher	2	0	0
xmloutput	2	0	0
element	3	1	3
to	2	0	0
final	1	0	0
xml	1	0	0
store	1	0	0
symmetric	1	0	0
output	1	0	0
only	1	0	0
encrypt	4	1	4
parse	1	0	0
document	7	0	0
do	1	0	0
root	1	0	0
contents	1	0	0
write	1	0	0
info	2	1	2
encrypted	2	0	0
file	3	0	0
get	1	0	0
doc	1	1	1
key	9	1	9
		$ostm_{ml}(C, C')$:	39%

Table 6: Overall state matching ($ostm_{ml}(C, C')$) for method *encrypt* (Appendix A, Listing 3).

p_i	$in_{C'}^{p_i}$
javax.crypto	0
Debrief.Tools.Operations	0
org.apache.xml.security.keys	0
org.w3c.dom	1
org.apache.xml.security.encryption	0
java.lang	1
java.security	0
$obm_{ml}(C, C')$:	29%

Table 7: Overall behavior matching for method *encrypt* (Appendix A, Listing 3).

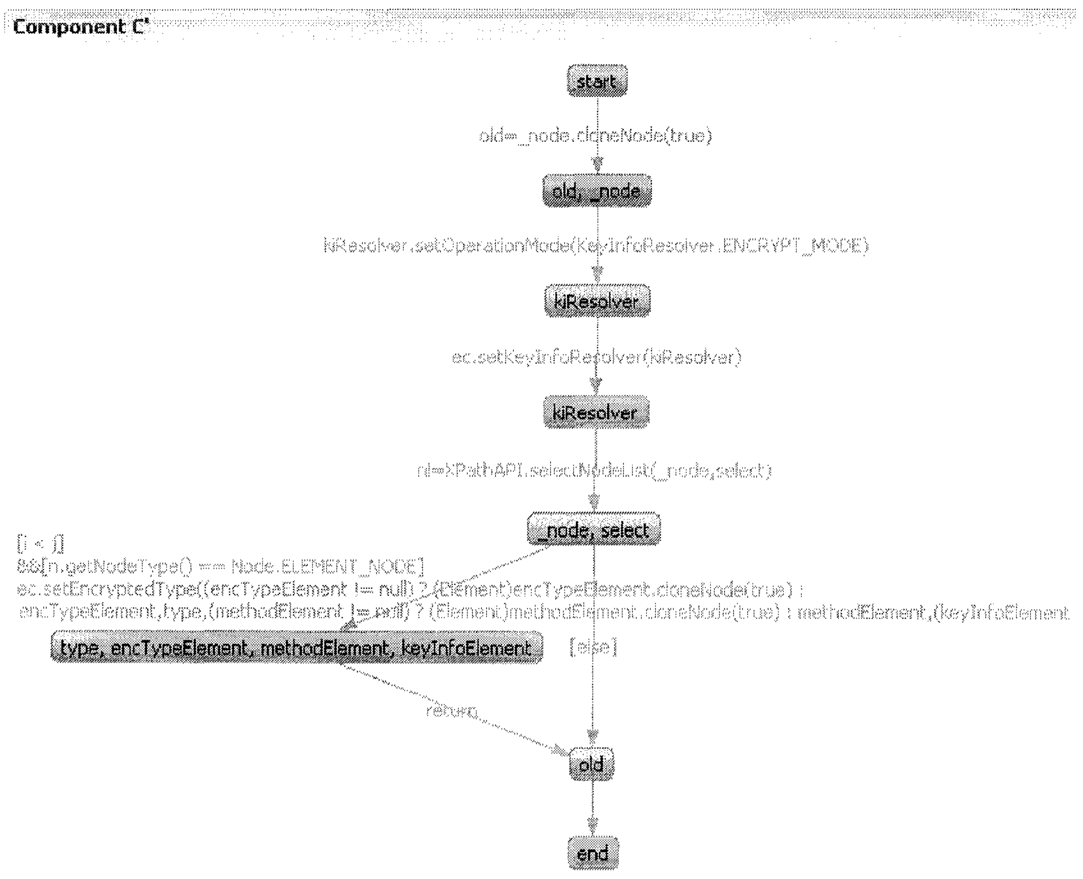
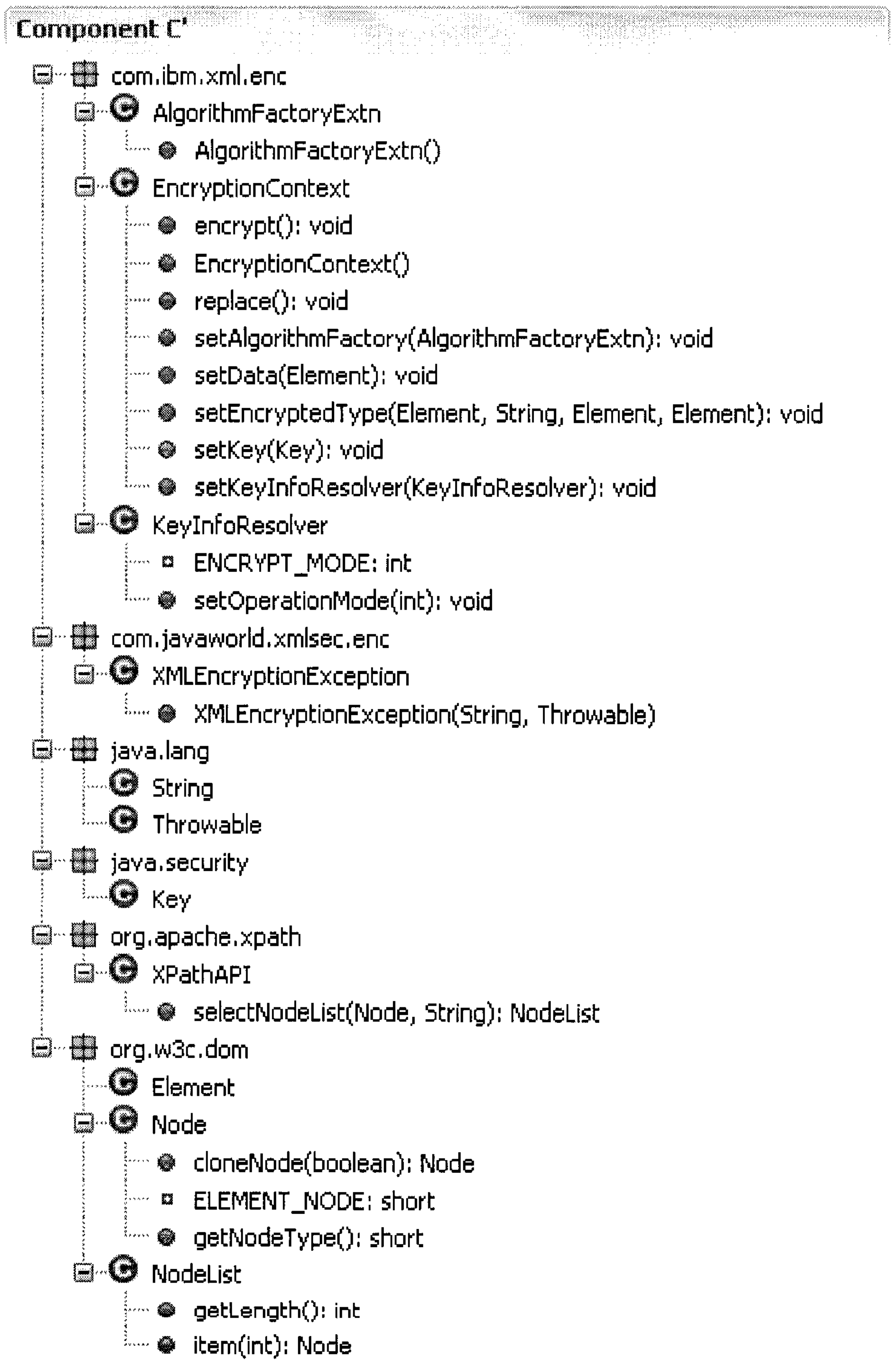

Component C'
start
old=_node.cloneNode(true)
old, _node
kiResolver.setOperationMode(KeyInfoResolver.ENCRYPT_MODE)
kiResolver
ec.setKeyInfoResolver(kiResolver)
kiResolver
nl=XPathAPI.selectNodeList(_node,select)
_node, select
[i < j]
&&[n.getNodeType() == Node.ELEMENT_NODE]
ec.setEncryptedType((encTypeElement != null) ? (Element)encTypeElement.cloneNode(true) :
encTypeElement,type,(methodElement != null) ? (Element)methodElement.cloneNode(true) : methodElement,(keyInfoElement
type, encTypeElement, methodElement, keyInfoElement
[else]
return
old
end

Figure 16: State diagram of method *encrypt* (Appendix A, Listing 2).



```
Component C'
com.ibm.xml.enc
    AlgorithmFactoryExtn
        AlgorithmFactoryExtn()
    EncryptionContext
        encrypt(): void
        EncryptionContext()
        replace(): void
        setAlgorithmFactory(AlgorithmFactoryExtn): void
        setData(Element): void
        setEncryptedType(Element, String, Element, Element): void
        setKey(Key): void
        setKeyInfoResolver(KeyInfoResolver): void
    KeyInfoResolver
        ENCRYPT_MODE: int
        setOperationMode(int): void
com.javaworld.xmlsec.enc
    XMLEncryptionException
        XMLEncryptionException(String, Throwable)
java.lang
    String
    Throwable
java.security
    Key
org.apache.xpath
    XPathAPI
        selectNodeList(Node, String): NodeList
org.w3c.dom
    Element
    Node
        cloneNode(boolean): Node
        ELEMENT_NODE: short
        getNodeType(): short
    NodeList
        getLength(): int
        item(int): Node
```

Figure 17: Required types for method *encrypt* (Appendix A, Listing 2).

Criterion	Grade	Weight	Weighted grade
$odm_{ml}(C,C')$	55%	40%	22%
$ostm_{ml}(C,C')$	39%	30%	12%
$obm_{ml}(C,C')$	29%	30%	9%
		$ocm_{ml}(C,C')$:	43%

Table 8: Overall matching for method *encrypt* (Appendix A, Listing 3).

Compared elements	Grade	Weight	Weighted grade
$g_{ID}(C,C')$	60%	25%	15%
$g_{ot}(C,C')$	0%	25%	0%
$g_{mn}(C,C')$	100%	20%	20%
$g_{v}(C,C')$	100%	10%	10%
$g_{MOD}(C,C')$	0%	10%	0%
$g_{EXC}(C,C')$	100%	10%	10%
		$odm_{ml}(C,C')$:	55%

Table 9: Overall definition matching (odm) for method *encrypt* (Appendix A, Listing 2).

and 15 for method *encrypt* (Appendix A, Listing 3), and in Tables 16 and 17 for method *encrypt* (Appendix A, Listing 2).

Thus, the comparison considering the complete flow of events for method *encrypt* (Appendix A, Listing 3) results in $ocm_{ml}(C,C') = 46\%$ overall matching, whereas for method *encrypt* (Appendix A, Listing 2) $ocm_{ml}(C,C') = 50\%$.

We recognize the fact that thresholds may be defined subjectively and in this dissertation we apply our own. We believe that for *ocm* below 20% the component does not worth substituting (low level of matching). Between 20% and 40% the substitution needs further analysis in order to reach any conclusion (medium level of matching). Above 40% the substitution is promising and should be considered (high level of matching). However, we believe that maintainers should be given an indication of initial thresholds and weights based on statistical data. To this end, we

Minimal word	**Occurrence**
true	4
old	2
element	13
method	3
nl	1
type	6
ki	2
enc	3
ec	2
resolver	4
x	1
operation	1
set	3
mode	2
api	1
null	3
path	1
encrypt	1
select	2
n	1
node	10
j	1
i	1
list	1
info	5
clone	4
encrypted	1
get	1
key	5

Table 10: Minimal words of C' and their occurrence for method *encrypt* (Appendix A, Listing 2).

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
cipher	2	0	0
xmloutput	2	0	0
element	3	1	3
to	2	0	0
final	1	0	0
xml	1	0	0
store	1	0	0
symmetric	1	0	0
output	1	0	0
only	1	0	0
encrypt	4	1	4
parse	1	0	0
document	7	0	0
do	1	0	0
root	1	0	0
contents	1	0	0
write	1	0	0
info	2	1	2
encrypted	2	1	2
file	3	0	0
get	1	1	1
doc	1	0	0
key	9	1	9
		$ostm_{ml}(C, C')$:	43%

Table 11: Overall state matching ($ostm_{ml}(C, C')$) for method *encrypt* (Appendix A, Listing 2).

Criterion	**Grade**	**Weight**	**Weighted grade**
$odm_{ml}(C, C')$	55%	40%	22%
$ostm_{ml}(C, C')$	43%	30%	13%
$obm_{ml}(C, C')$	43%	30%	13%
		$ocm_{ml}(C, C')$:	48%

Table 12: Overall matching for method *encrypt* (Appendix A, Listing 2).

Minimal word	**Occurrence**
store	3
generate	4
add	2
document	23
encrypted	18
cipher	32
doc	3
write	3
true	2
do	3
mode	4
element	21
only	5
key	67
parse	3
xml	25
file	9
128	2
xmloutput	6
symmetric	9
root	5
set	2
output	3
tripledes	2
aes	2
final	3
contents	5
encryption	2
instance	4
wrap	4
init	4
get	9
info	14
data	8
encrypt	22
to	8

Table 13: Minimal words of C and their occurrence, considering the complete flow for method *encrypt* (Appendix A, Listing 1).

Minimal word	Occurrence
util	1
element	27
false	6
method	10
type	12
xml	1
ki	2
enc	10
resolver	2
create	7
null	12
encrypt	2
select	2
document	2
new	1
info	10
key	10
doc	7

Table 14: Minimal words of C' and their occurrence, considering the complete flow for method *encrypt* (Appendix A, Listing 3).

plan to conduct more investigation in our future research in order to verify and refine the thresholds and weights defined in this work.

The results of matching for each component are above 40% and thus promising. Meanwhile there is no much difference between their $ocm_{ml}(C, C')$, therefore both components are to be considered.

In the rest of this chapter we illustrate with two case studies situations where the components are not expected to match.

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
store	3	0	0
generate	4	0	0
add	2	0	0
document	23	1	23
encrypted	18	0	0
cipher	32	0	0
doc	3	1	3
write	3	0	0
true	2	0	0
do	3	0	0
mode	4	0	0
element	21	1	21
only	5	0	0
key	67	1	67
parse	3	0	0
xml	25	1	25
file	9	0	0
128	2	0	0
xmloutput	6	0	0
symmetric	9	0	0
root	5	0	0
set	2	0	0
output	3	0	0
tripledes	2	0	0
aes	2	0	0
final	3	0	0
contents	5	0	0
encryption	2	0	0
instance	4	0	0
wrap	4	0	0
init	4	0	0
get	9	0	0
info	14	1	14
data	8	0	0
encrypt	22	1	22
to	8	0	0
		$ostm_{ml}(C, C')$:	51%

Table 15: State matching considering the complete flow for method *encrypt* (Appendix A, Listing 3).

Minimal word	Occurrence
api	2
encrypted	2
true	8
mode	4
null	7
af	2
element	27
enc	6
length	1
key	11
clone	8
list	3
replace	1
0	1
set	9
factory	3
extn	2
item	1
node	23
resolver	8
operation	2
algorithm	3
x	2
encryption	2
path	2
select	4
n	4
old	4
j	3
get	3
i	5
info	10
method	6
data	1
encrypt	3
nl	4
ki	4
context	2
ec	10
type	12

Table 16: Minimal words for the complete flow of method *encrypt* (Appendix A, Listing 2).

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
store	3	0	0
generate	4	0	0
add	2	0	0
document	23	0	0
encrypted	18	1	18
cipher	32	0	0
doc	3	0	0
write	3	0	0
true	2	1	2
do	3	0	0
mode	4	1	4
element	21	1	21
only	5	0	0
key	67	1	67
parse	3	0	0
xml	25	0	0
file	9	0	0
128	2	0	0
xmloutput	6	0	0
symmetric	9	0	0
root	5	0	0
set	2	1	2
output	3	0	0
tripledes	2	0	0
aes	2	0	0
final	3	0	0
contents	5	0	0
encryption	2	1	2
instance	4	0	0
wrap	4	0	0
init	4	0	0
get	9	1	9
info	14	1	14
data	8	1	8
encrypt	22	1	22
to	8	0	0
		$ostm_{ml}(C, C')$:	50%

Table 17: State matching considering the complete flow for method *encrypt* (Appendix A, Listing 2).

6.2 Low level matching examples

In this section, we illustrate two examples where the compared components do not match. In the first example, the compared components are in the same domain (security), whereas in the second case they are performing completely different functionalities.

6.2.1 HDIV case study

HDIV [28] is a Java Web Application Security Framework which can be obtained from SourceForge. HDIV extends web application frameworks behavior (Struts 1.x, Struts 2.x, Spring MVC) in order to avoid most common web application security vulnerabilities.

In this case study, we estimated the level of substitutability for our component C and method *validate* in class *org.hdiv.dataValidator.DataValidator* (see Appendix A, Listing 4). We obtained 40% matching for the overall definition matching (see Table 18), 0% for the overall state matching (see Table 19), and 14% for the overall behavior matching (see Table 20). Thus, the overall component matching results in 20% (see Table 21). The result is at the limit of low and medium matching. The distribution of this matching however comes predominantly from the overall definition matching. Maintainers can then conclude that the matching is definitely low rather than medium.

The result can be verified by retrieving the documentation for method *validate* which reads as follows: *Checks if the value* `data` *sent by the user to the server in*

Compared elements	Grade	Weight	Weighted grade
$g_{ID}(C, C')$	80%	25%	20%
$g_{ot}(C, C')$	0%	25%	0%
$g_{mn}(C, C')$	0%	20%	0%
$g_v(C, C')$	100%	10%	10%
$g_{MOD}(C, C')$	0%	10%	0%
$g_{EXC}(C, C')$	100%	10%	10%
		$odm_{ml}(C, C')$:	40%

Table 18: Overall definition matching $odm_{ml}(C, C')$ for method *validate* (Appendix A, Listing 4).

the parameter **`parameter`** *is correct or not. The received value is checked with the one stored in the state to decide if it is correct.* We can conclude that although the domain of HDIV, which is security, is the same as the domain for method *encrypt* the functionality provided by method *validate* can not substitute the functionality provided by method *encrypt.*

6.2.2 OPSIS case study

Opsis [44] is a Java applet designed to teach binary search tree algorithms. It uses visual programming in an abstract way. Opsis combines elements of programming, proof, and animation to enhance the learning experience.

We estimated the level of substitutability of component C with method *duplicate _tree* defined in class *Fragment* (see Appendix A, Listings 5 and 6). It is clear from the description of this system and from the name of the candidate method for the substitution that the provided functionality by the candidate component will most probably not match. The results of the comparison are 15% for the overall definition matching (see Table 22), 18% for the overall state matching (see Table 23), and

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
cipher	2	0	0
xmloutput	2	0	0
element	3	0	0
to	2	0	0
final	1	0	0
xml	1	0	0
store	1	0	0
symmetric	1	0	0
output	1	0	0
only	1	0	0
encrypt	4	0	0
parse	1	0	0
document	7	0	0
do	1	0	0
root	1	0	0
contents	1	0	0
write	1	0	0
info	2	0	0
encrypted	2	0	0
file	3	0	0
get	1	0	0
doc	1	0	0
key	9	0	0
		$ostm_{ml}(C, C')$:	0%

Table 19: State matching considering the basic flow for method *validate* (Appendix A, Listing 4).

p_i	$in_{C'}^{p_i}$
javax.crypto	0
Debrief.Tools.Operations	0
org.apache.xml.security.keys	0
org.w3c.dom	0
org.apache.xml.security.encryption	0
java.lang	1
java.security	0
$obm_{ml}(C, C')$:	14%

Table 20: Behavior matching for method *validate* (Appendix A, Listing 4).

Criterion	**Grade**	**Weight**	**Weighted grade**
$odm_{ml}(C, C')$	40%	40%	16%
$ostm_{ml}(C, C')$	0%	30%	0%
$obm_{ml}(C, C')$	14%	30%	4%
		$ocm_{ml}(C, C')$:	20%

Table 21: Overall matching for method *validate* (Appendix A, Listing 4).

Compared elements	**Grade**	**Weight**	**Weighted grade**
$g_{ID}(C, C')$	20%	25%	5%
$g_{ot}(C, C')$	0%	25%	0%
$g_{mn}(C, C')$	0%	20%	0%
$g_{v}(C, C')$	0%	10%	0%
$g_{MOD}(C, C')$	0%	10%	0%
$g_{EXC}(C, C')$	100%	10%	10%
		$odm_{ml}(C, C')$:	15%

Table 22: Overall definition matching $odm_{ml}(C, C')$ for method *duplicate_tree* (Appendix A, Listings 5 and 6).

14% for the overall behavior matching (see Table 24). We thus obtain 15% overall component matching which confirms the low level of substitutability.

w_i	$occ_C^{w_i}$	$in_{C'}^{w_i}$	**Weighted grade**
cipher	2	0	0
xmloutput	2	0	0
element	3	0	0
to	2	0	0
final	1	0	0
xml	1	0	0
store	1	0	0
symmetric	1	0	0
output	1	0	0
only	1	0	0
encrypt	4	0	0
parse	1	0	0
document	7	0	0
do	1	0	0
root	1	0	0
contents	1	0	0
write	1	0	0
info	2	0	0
encrypted	2	0	0
file	3	0	0
get	1	0	0
doc	1	0	0
key	9	1	9
		$ostm_{ml}(C, C')$:	18%

Table 23: State matching considering the basic flow for method *duplicate_tree* (Appendix A, Listings 5 and 6).

p_i	$in_{C'}^{p_i}$
javax.crypto	0
Debrief.Tools.Operations	0
org.apache.xml.security.keys	0
org.w3c.dom	0
org.apache.xml.security.encryption	0
java.lang	1
java.security	0
$obm_{ml}(C, C')$:	14%

Table 24: Behavior matching for method *duplicate_tree* (Appendix A, Listings 5 and 6).

Criterion	**Grade**	**Weight**	**Weighted grade**
$odm_{ml}(C, C')$	15%	40%	6%
$ostm_{ml}(C, C')$	18%	30%	5%
$obm_{ml}(C, C')$	14%	30%	4%
		$ocm_{ml}(C, C')$:	15%

Table 25: Overall matching for method *duplicate_tree* (Appendix A, Listings 5 and 6).

Chapter 7

Automation and tool support

The automation of our approach is provided through an Eclipse [17] plug-in.

It is composed of three main parts: code processor, the metrics implementation, and the plug-in interface.

1. The code processor uses the Eclipse AST in order to extract the needed information. The AST is traversed using the Visitor pattern in order to collect the component definition, state[1], and behavior.

2. The metrics implementation performs the analysis of the extracted information. This part is independent of the language environment and can be reused for other object-oriented languages than Java.

3. The plug-in interface is developed using the Standard Widget Toolkit (SWT), JFace [54] and ZEST (Zoomable Eclipse SHriMP Tool [5]) libraries.

[1]We initially planned to use Bandera, a tool for automatic extraction of finite-state models from Java source code by Corbett et al. [13]. However, we find that the process of writing the required XML file which specifies the settings for the tool is unnecessarily tedious.

In order to select the components to be analyzed, maintainers are given a tree representation of the current workspace. Maintainers can navigate through the tree and select the two components to be analyzed by a right click, choosing *Set as first/second component* (see Figure 18). In order to increase the performance, the analysis is performed following the lazy load approach, which allows an element of the tree and its direct children to be analyzed only if the maintainer clicks on it.

Maintainers are given default values for the measurement metrics which can be adjusted with regard to their needs (see Figure 19).

The results of the analysis are shown in three different views: Definition View (see Figure 20), State View (see Figure 21), and Behavior View (see Figure 22), while the results of the measurements are summarized in Results View (see Figure 23).

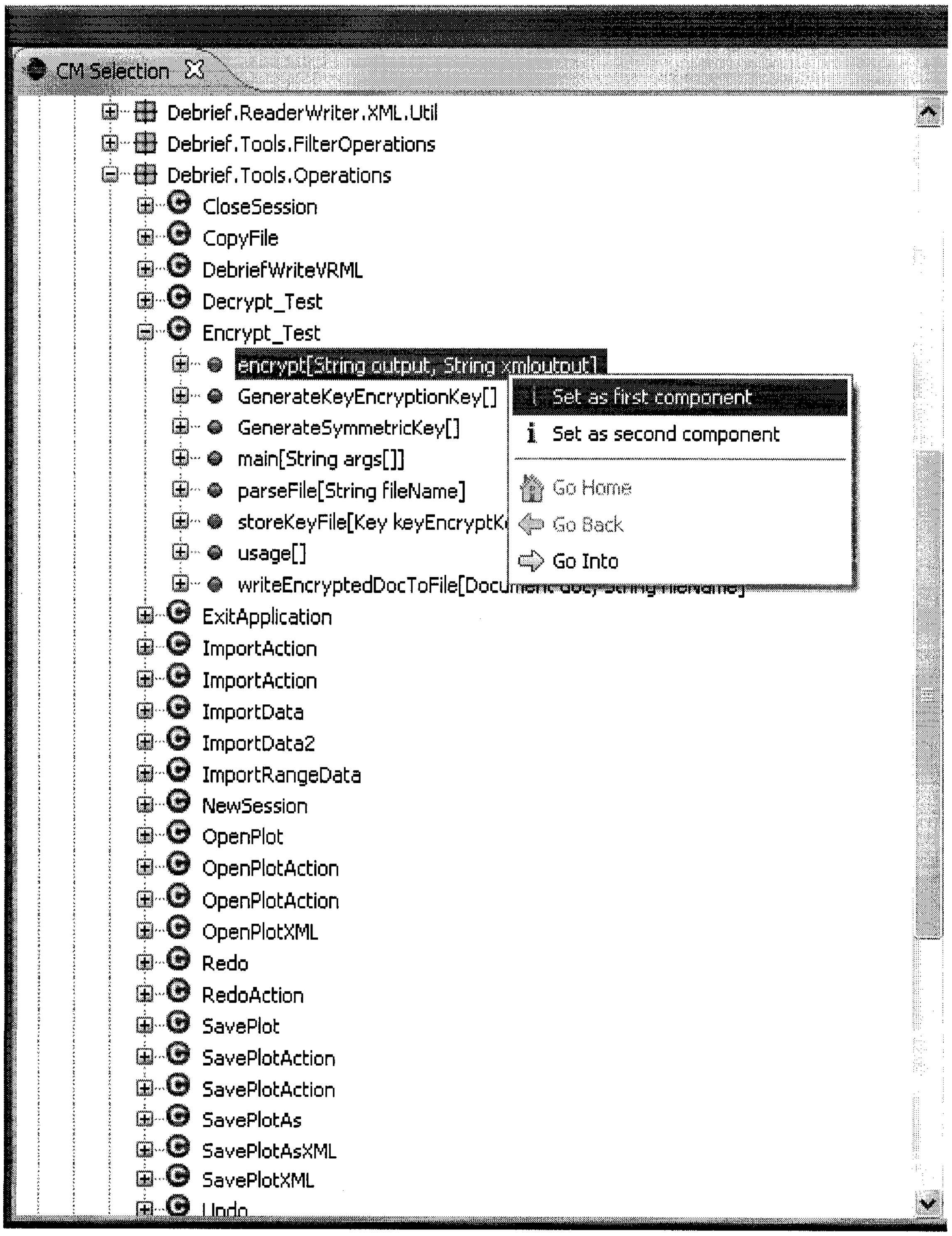
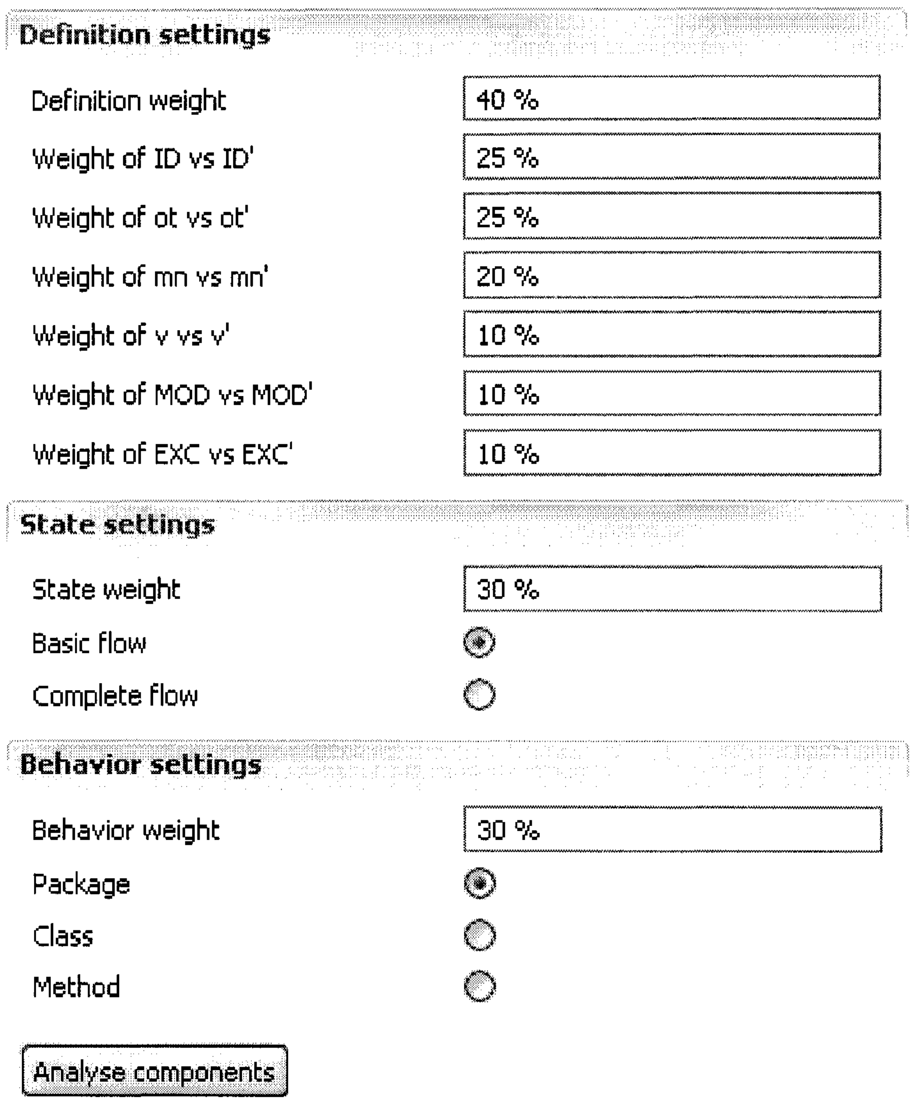


Figure 18: CM Selection View - workspace.



Definition settings

Definition weight	40 %
Weight of ID vs ID'	25 %
Weight of ot vs ot'	25 %
Weight of mn vs mn'	20 %
Weight of v vs v'	10 %
Weight of MOD vs MOD'	10 %
Weight of EXC vs EXC'	10 %

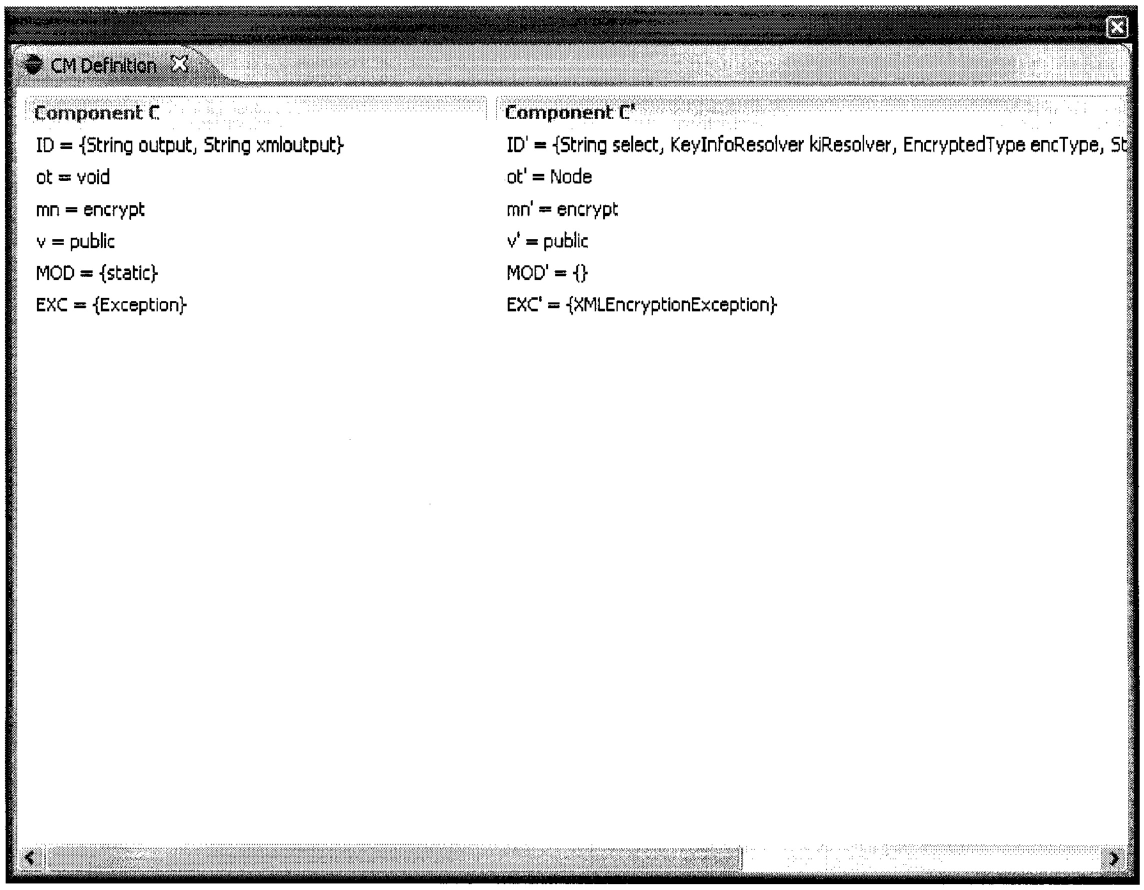
State settings

State weight	30 %
Basic flow	●
Complete flow	○

Behavior settings

Behavior weight	30 %
Package	●
Class	○
Method	○

Figure 19: CM Selection View - settings.



Component C	Component C'
ID = {String output, String xmloutput}	ID' = {String select, KeyInfoResolver kiResolver, EncryptedType encType, St
ot = void	ot' = Node
mn = encrypt	mn' = encrypt
v = public	v' = public
MOD = {static}	MOD' = {}
EXC = {Exception}	EXC' = {XMLEncryptionException}

Figure 20: CM Definition View.

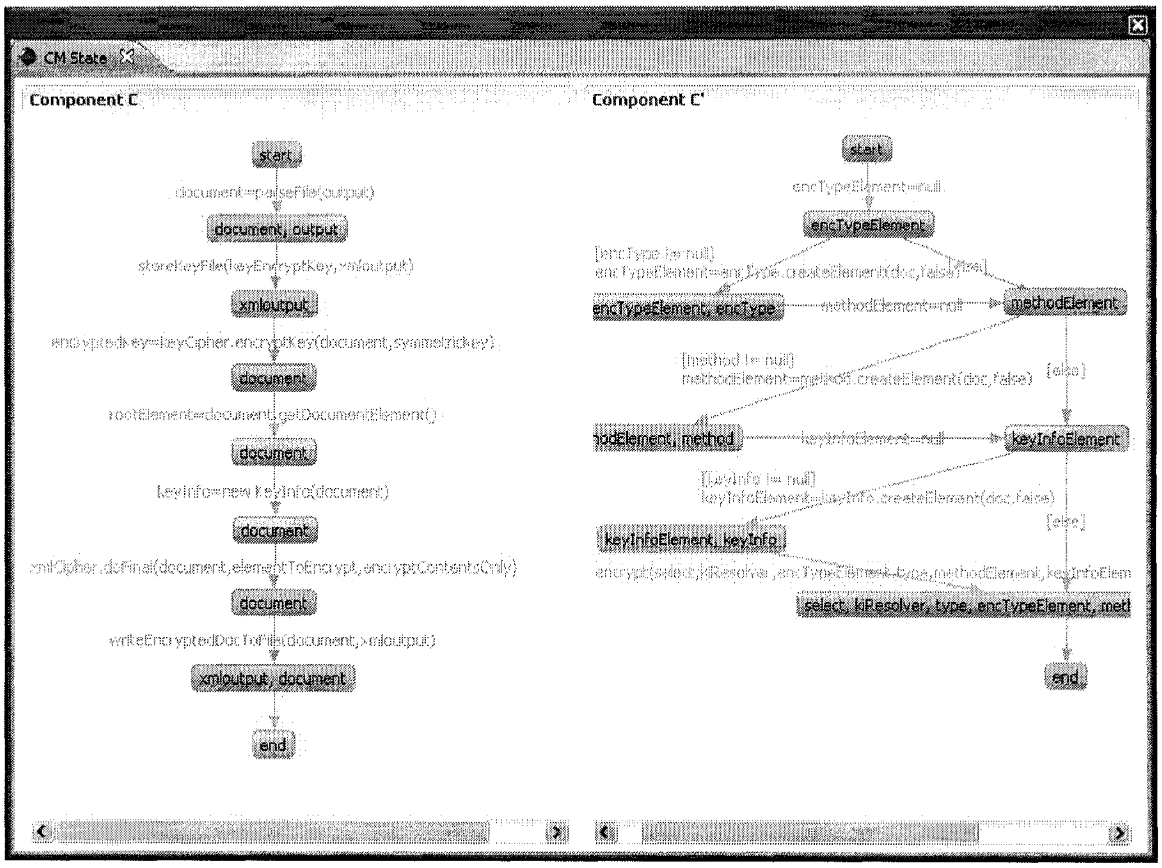
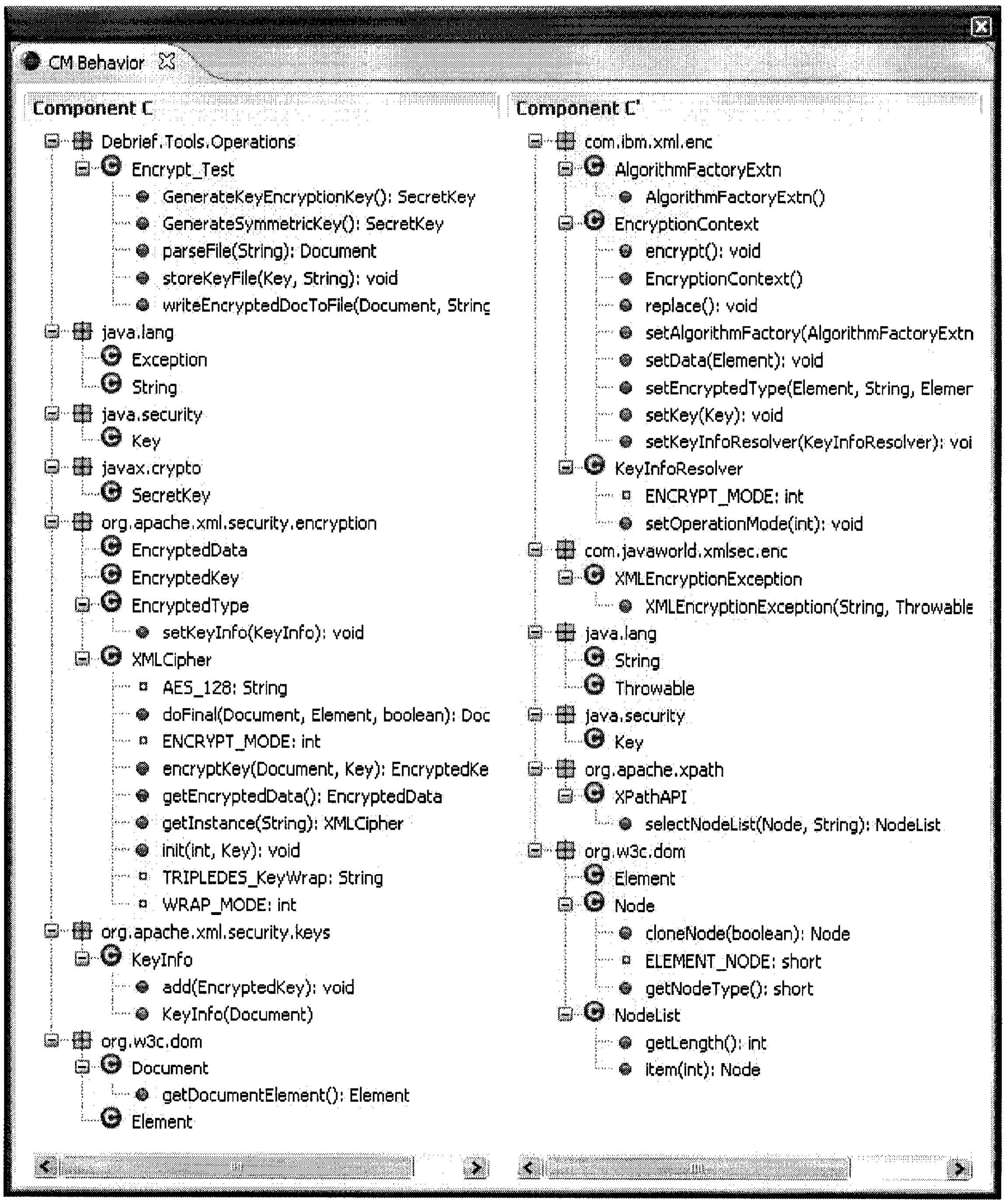

CM State
Component C
start
document=parseFile(output)
document, output
storeKeyFile(keyEncryptKey,xmloutput)
xmloutput
encryptedKey=keyCipher.encryptKey(document,symmetricKey)
document
rootElement=document.getDocumentElement()
document
KeyInfo=new KeyInfo(document)
document
xmlCipher.doFinal(document,elementToEncrypt,encryptContentsOnly)
document
writeEncryptedDocToFile(document,xmloutput)
xmloutput, document
end
Component C'
start
encTypeElement=null
encTypeElement
[encType != null]
encTypeElement=encType.createElement(doc,false)
[else]
encTypeElement, encType
methodElement=null
methodElement
[method != null]
methodElement=method.createElement(doc,false)
[else]
hodElement, method
keyInfoElement=null
keyInfoElement
[keyInfo != null]
keyInfoElement=keyInfo.createElement(doc,false)
[else]
keyInfoElement, keyInfo
encrypt(select,kiResolver,encTypeElement,type,methodElement,keyInfoElem
select, kiResolver, type, encTypeElement, met
end

Figure 21: CM State View.



CM Behavior

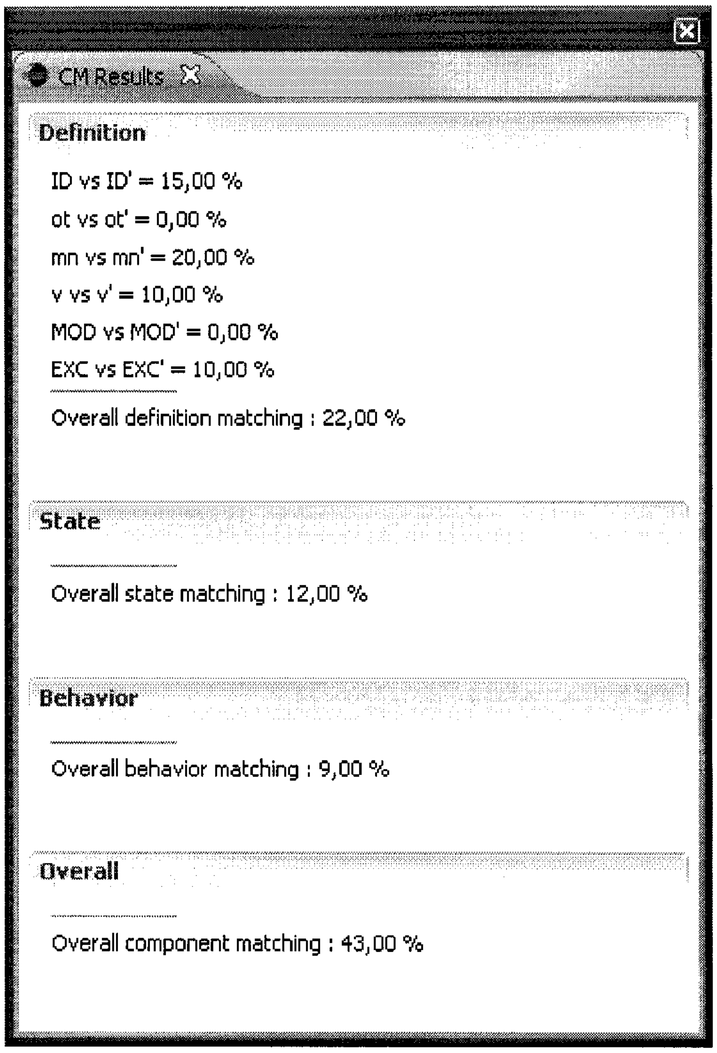
Component C

```
Debrief.Tools.Operations
    Encrypt_Test
        GenerateKeyEncryptionKey(): SecretKey
        GenerateSymmetricKey(): SecretKey
        parseFile(String): Document
        storeKeyFile(Key, String): void
        writeEncryptedDocToFile(Document, Strinc
java.lang
    Exception
    String
java.security
    Key
javax.crypto
    SecretKey
org.apache.xml.security.encryption
    EncryptedData
    EncryptedKey
    EncryptedType
        setKeyInfo(KeyInfo): void
    XMLCipher
        AES_128: String
        doFinal(Document, Element, boolean): Doc
        ENCRYPT_MODE: int
        encryptKey(Document, Key): EncryptedKe
        getEncryptedData(): EncryptedData
        getInstance(String): XMLCipher
        init(int, Key): void
        TRIPLEDES_KeyWrap: String
        WRAP_MODE: int
org.apache.xml.security.keys
    KeyInfo
        add(EncryptedKey): void
        KeyInfo(Document)
org.w3c.dom
    Document
        getDocumentElement(): Element
    Element
```

Component C'

```
com.ibm.xml.enc
    AlgorithmFactoryExtn
        AlgorithmFactoryExtn()
    EncryptionContext
        encrypt(): void
        EncryptionContext()
        replace(): void
        setAlgorithmFactory(AlgorithmFactoryExtn
        setData(Element): void
        setEncryptedType(Element, String, Elemer
        setKey(Key): void
        setKeyInfoResolver(KeyInfoResolver): voi
    KeyInfoResolver
        ENCRYPT_MODE: int
        setOperationMode(int): void
com.javaworld.xmlsec.enc
    XMLEncryptionException
        XMLEncryptionException(String, Throwable
java.lang
    String
    Throwable
java.security
    Key
org.apache.xpath
    XPathAPI
        selectNodeList(Node, String): NodeList
org.w3c.dom
    Element
    Node
        cloneNode(boolean): Node
        ELEMENT_NODE: short
        getNodeType(): short
    NodeList
        getLength(): int
        item(int): Node
```

Figure 22: CM Behavior View.


CM Results

Definition

ID vs ID' = 15,00 %

ot vs ot' = 0,00 %

mn vs mn' = 20,00 %

v vs v' = 10,00 %

MOD vs MOD' = 0,00 %

EXC vs EXC' = 10,00 %

Overall definition matching : 22,00 %

State

Overall state matching : 12,00 %

Behavior

Overall behavior matching : 9,00 %

Overall

Overall component matching : 43,00 %

Figure 23: CM Results View.

Chapter 8

Related work

The approaches that are relevant to our work can be categorized into two main groups. The first group focuses on component retrieval and component matching, whereas the second group explores the detection of code clones. These two groups are presented in Sections 8.1 and 8.2. Next, we compare our approach with the existing work in Section 8.3.

8.1 Component retrieval and matching

Khemakhem, Drira, and Jmaiel [33] present an approach for discovering software components in a repository, thereby helping the developer to integrate the selected component by developing two ontologies: discovery ontology and integration ontology. A query is automatically generated from the specification of the component, after which the components that correspond to the result of this query in a specific repository are indexed. The specifications of the components are described in the Unified Problem-Solving Method Language (UPML) by the component suppliers. The authors recognize this limitation and plan to address it in the future by deducting the description from the source code of the components. When identifying potential components from the repository, the authors distinguish four degrees of similarities: exact (when the two components are equivalent), plugIn (when the requested component is a sub-concept of the component in the repository), subsume (when the requested component is a super-concept of the component of the repository), and disjoint (when there is no element of the repository component description corresponding to the requested description).

Andreou, Vogiatzis, and Papadopoulos [1] propose a method for intelligent classification and retrieval of software components. The method is based on a predefined set of characteristics (such as general functionality, implementation language, platform, memory utilization, price, etc.) from which users will choose those of interest. Once users have chosen the desired threshold, the preferences are encoded using binary strings in order to return the components corresponding to the closest classifier. The classification/retrieval procedures are based on a dedicated genetic algorithm that processes the set of predefined characteristics.

Vitharana, Zahedi, and Jain [65] describe a facet-based approach, relying on XML to design a knowledge-based repository for business components. They propose identifiers (such as name and industry type) for representing the structural information of components and descriptor facets (such as synonym, role, etc.) for the unstructured information. A prototype version of a component knowledge-based repository was developed and the experiments conducted showed that classifying components according to their hierarchical structure at different levels of abstraction and on the basis of their structured and unstructured information can be promising in the domain of component retrieval.

Nakkrasae and Sophatsathit [42] propose the use of computational intelligence, using clustering algorithms for component classification where components are grouped on the basis of Rival Penalized Competitive Learning algorithm. Components are formally specified in the Z language based on three aspects: structural, functional, and behavioral [41].

Systä [58] presents a tool called SCED for modeling both the static and dynamic behavior of object-oriented software systems from source code. The dynamic behavior of the software is extracted, based on behavioral patterns detection from event traces. The total behavior of objects is represented by a synthesized state diagram.

Yu et al. [67] discuss a methodology to extract user goal models from legacy code. Their approach consists of refactoring the source code by applying the extract method strategy based on comments. If the refactored code is not structured, statecharts are constructed to achieve this. An abstract syntax tree (AST) is subsequently built from the structured program, based on which goal models are extracted. In their future work, the authors intend to compare the reverse engineered goal models with those derived from requirements elicitation. The process is not fully automated.

Zaremski and Wing [69] describe an approach which involves the comparison of the behavior of two software components. The authors specifically examine retrieval for reuse, substitution for subtyping, and interoperability. They define components as a function or a module (a set of functions) and use formal specifications of components in terms of pre- and post-condition predicates. Thus, the authors rely on theorem proving to determine match and mismatch and define a lattice of potential satisfaction relationships over axiomatic specifications (they distinguish four kinds of pre/post matches starting from the strongest match, which is an exact match, followed by progressively weaker matches). The theory is illustrated with examples of the implementation of specification matching using the Larch Prover (LP). The signature [68] is used as a filter in order to eliminate the obvious non-matches before trying the more expensive specification matching.

A similar approach is described by Fischer, Kievernagel, and Struckmann [19] where the authors use signature matching to filter promising candidates out of a component library. They build proof obligations from VDM specifications of key and components and feed them into a theorem prover. The validated obligations denote matching components. The components are implemented in imperative languages (VDM and Modula-2) and annotated with implicit VDM specifications. In the first phase, the authors filter one part of the components by signature matching. They consider that a candidate component matches a given search key if it simultaneously has weaker pre-conditions and stronger post-conditions than the key. In other words, if a component requires less than specified but grants more, it can be plugged-in. In the next phase the authors take the remaining components in order to test them for specification matching using OTTER, a general purpose theorem prover based on resolution principle. Overall, the provided automation is able to locate software components via the matching of implicit VDM specifications. The computational effort, however, is high.

Hemer [29] discusses how existing specification matching techniques can be extended to handle matching state based components. The author restricted his approach to three kinds of unit in a module: state schema, initialization schema, and operation schema. A modular extension of the Z specification language is deployed, called Sum specification language.

Feiks and Hemer [18] extend previous approaches in order to reason about class matching in object-oriented programming, with particular attention to information hiding, inheritance, and overriding (redefining). Class matching extends the notation of function matching while taking inheritance into consideration. If the features (attributes and methods) of two classes match, then the two classes are considered to match. For method matching the authors extend the function matching discussed by Zaremski and Wing [69], and for attribute matching they consider the state schema matching routines used for matching state based modules [29].

Penix and Alexander [46, 47] propose the retrieval and reuse of components by using semantic information provided by the formal specifications. Component retrieval is made efficient by using a feature-based classification scheme. Features are assigned to components based on specific necessary conditions that are implied by the component specifications. The formally described features are used as retrieval keys to prune bad solutions before evaluating the satisfaction condition.

Rosa et al. [52] also propose a formal approach. In this work however, non-functional requirements (formally specified in the Z language) are used to filter components.

Gannnod and Cheng [24] derive formal specifications from imperative source code.

These specifications are considered to be at the "as-built" level of abstraction, which indeed facilitates the traceability between specifications and code but which may be difficult to use for higher level reasoning. Gannod and Cheng [25] describe a formal approach for deriving abstract functional specifications from "as-built" specifications while Gannod, Chen and Cheng [23] show the applicability of abstract specifications to support the population of component libraries.

For component retrieval, Podgurski and Pierce [48] propose a method based on the executability of the components, namely, behavior sampling. In this approach, users specify a sample input and the corresponding output. Any component whose output is compatible to the sample output specified by users is retrieved. The authors show that the method is precise for small sample input.

Cechich and Piattini [8] introduce early measurements for identifying suitable COTS components. The authors argue that a first filtering, based on the functionality of the components, should be conducted before proceeding to a comparison of other properties. They focus on semantic mapping and define two groups of measures: component-level measures and solution-level measures. This approach requires that the functionality of the components is described through the functional user requirements (FUR) documents either by scenarios or by using an architectural description language (ADL) [10, 45] which, when not available, should be derived from other software engineering artifacts. The approach is part of a wider iterative process based on Six-Sigma precepts [7].

Approaches such as the one proposed by Andreou, Vogiatzis and Papadopoulos [1] are based on attributes that assume values. Even though they are considered to be more flexible than controlled vocabulary [21], they are hardly applicable for OSS components. Approaches based on facet classification [65] are similar to attribute-values approaches and require manual classification which, however, may be time consuming. Similarly, the specification of the components proposed by Khemakhem, Drira and Jmaiel [33] could be expensive.

8.2 Clone detection

Sager et al. [53] detect similar Java classes using tree similarity algorithms. The comparison is performed on the FAMIX [15] model level which is generated from the AST representation of the compared classes. The end objective of this approach, however, is to detect similar code fragments more often in different versions of the same code.

Kakimoto et al. [32] propose an approach to identify similar Java classes using Java birthmarks [62]. The authors extract four types of birthmarks: constant values in field variables, sequence of method calls, inheritance structure, and used classes. This approach is applicable for finding similar classes that are often constructed by copy-and-paste.

Similarly, Schuler, Dallmeier, and Lindig [56] instrument bytecode and identify similar birthmarks in order to protect a program's copyright. The authors use dynamic birthmarks and observe sequences of method calls per objects, instead of the global traces proposed by Tamada, Nakamura, and Monden [61].

Krinke [36] identifies similar code based on finding maximal similar subgraphs in fine-grained program dependence graphs (PDG). As possible problems which can be solved by this approach, the author suggests errors that should be fixed or modifications to be applied in both original and duplicated code. Another approach which is based on PDG is proposed by Komondoor and Horwitz [34], this time for the purpose of refactoring. The authors propose the identification of duplicated code and the subsequent extraction and replacement of the repeated code with method calls.

Kontogiannis [35] presents an approach for clone detection based on five data and control flow related metrics which are calculated using the AST of the compared programs.

8.3 Discussion

Approaches for component matching generally assume the existence of specifications. The component to be substituted as well as the candidate components should be formally specified. Formal specification languages are utilized in order to specify components rigorously. When specification is available, those methods [1, 33] should be applied to filter irrelevant components in the repository. However, formally specifying a component requires expertise which is not generally part of the development process of OSS components. Our approach does not require any information other than source code and thus can address those cases where specification is not available. At the same time, our approach may be used to complement already existing approaches when the formal specification is part of the documentation.

On the other hand, clone detection approaches rely on source code only. They can thus be applicable to OSS components. However, these approaches are suitable in order to identify different versions of the same source code or to identify code constructed through copy and paste techniques. They are hardly applicable in cases where one tries to compare two different components.

One limitation of our approach is its dependency on the quality of the source code when comparing the state of the components. We assume that developers follow some minimum coding standards thus giving meaningful names to types, attributes, methods, and local variables while programming.

Chapter 9

Conclusion and recommendations for future work

In this dissertation, we proposed an approach for comparing object-oriented components based on source code analysis. We defined substitutability criteria for comparing the extracted information at three levels of granularity, namely i) method, ii) class, and iii) package. The addressed criteria use static information representing the definition, state and behavior of the components. Our method provides maintainers with an indication for the level of substitutability of two components. This work complements current work on component retrieval which is to be applied prior to our approach in order to select candidate components for analysis.

For future developments, we believe that more investigation is needed in order to verify and refine the initial values for the thresholds and weight of each criterion in this comparison. Moreover, we plan to complement this work with dynamic analysis. Dynamic analysis will allow us to address some non-functional requirements of the components, such as performance. In addition, dynamic analysis will allow us to compare the components seen as black boxes (in terms of input/output) and to extract and compare the values of the important variables for the state of a component. It would be also interesting to apply similar analysis on COTS components where source code is not available. Another direction for future work is to adapt the in-memory storage in order to increase the performance and analysis of large-scale components.

Glossary

AP_{rm}	The actual parameters passed to the method call in the return statement of C at method level, part of the set of state variables SV, 37
A_c	The set of attributes of the class to which, at method level, C belongs to, part of the set of state variables SV, 37
A_{rt_i}	The set of attributes of the required type rt_i, 42
C	Component, 20
DA	The set of defined attributes of C, 48
DM	The set of defined methods of C, 48
DT	Set of defined types of C at package level, 57
E	Set of events upon which C changes from one state to another, 24
EXC	Exceptions of C at method level, 29
ID	Input definitions of C, 27
I_m	The set of formal parameters of C at method level, part of the set of state variables SV, 37
M	The set of methods' states of C, class level, 53
MAP	Set of elements associating an event of C with its originating and destination states, 24
MOD	Modifiers of C, 29
M_{rt_i}	Methods of the required type rt_i of C, 26
PM	Set of parent modules of C, 47
RT	Required types of C, 25
S	Set of states of C, 24
SV	The set of state variables of C, at method level, 37
V	The set of visibility modifiers that v may take, 29
a_{rt_i}	Attribute of the required type rt_i of C, 43
da	Defined attribute of C at class level, element of DA, 48
dm	Defined method of C at class level, element of DM, 48

dt	Defined type, element of DT, package level, 57
e_i	Event, element of E, 25
exc	Exception, element of EXC, 30
$g_{DA}(C, C')$	Grade of matching for the set of defined attributes DA of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{DM}(C, C')$	Grade of matching for the set of defined methods DM of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{EXC}(C, C')$	Grade of the set of exceptions EXC of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{ID}(C, C')$	Grade of the set of input definitions ID of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{MOD}(C, C')$	Grade of matching for the set of modifiers MOD of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{MOD}(C, C')$	Grade of the set of modifiers MOD of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{PM}(C, C')$	Grade of matching for the set of parent modules PM of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{cn}(C, C')$	Grade of matching for the name cn of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{da_k}(C, C')$	Grade of matching for a defined attribute da_k of C, used for calculating the grade of matching for the defined attributes $g_{DA}(C, C')$ in the overall definition matching at class level $odm_{cl}(C, C')$, 51
$g_{dm_k}(C, C')$	Grade of matching for a defined method dm_k, used for calculating the grade of matching of the defined methods $g_{DM}(C, C')$, class level, 53
$g_{exc_k}(C, C')$	The grade of each exception exc_k in $g_{EXC}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 36

$g_{id_k}(C, C')$	The grade of each input definition id_k in $g_{ID}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{mn}(C, C')$	Grade of the method name mn of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{mod_k}(C, C')$	The grade of matching for each modifier mod_k in $g_{MOD}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 35
$g_{name}(id_k, id'_k)$	The grade of name matching for each input definition id_k in $g_{id_k}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 33
$g_{ot}(C, C')$	Grade of the output type ot of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
$g_{pm_k}(C, C')$	Grade of matching for parent module pm_k of C, used for calculating the grade of matching for the parent modules $g_{PM}(C, C')$ in the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{pn}(C, C')$	Grade of name matching pn at package level, 58
$g_{pos}(id_k, id'_k)$	The grade of position matching for each input definition id_k in $g_{id_k}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 33
$g_{tm}(C, C')$	Grade of matching for the type tm of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_{type}(id_k, id'_k)$	The grade of type matching for each input definition id_k in $g_{id_k}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 33
$g_v(C, C')$	Grade of the visibility v of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 50
$g_v(C, C')$	Grade of the visibility v of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 32
id_k	Input definition, element of ID, 27

$in_C^{mod_k}$	Returns 1 if mod_k is a modifier of C; 0 otherwise. Used in $g_{mod_k}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 35
$in_{C'}^{rf_i}$	Existence of the required feature rf_i by C, 44
$in_{C'}^{rp_i}$	Existence of the required package rp_i by C in C', 45
$in_{C'}^{w_i}$	Returns 1 if w_i occurs in C'; 0 otherwise, used in the overall state matching at method level $ostm_{ml}(C, C')$, 41
m_i	Method state of C at class level, element of M, 53
map	Association, element of MAP, 25
mn	Name of C at method level, 28
mod_k	Modifier, element of MOD, 29
n_{rt_i}	The name of the required type rt_i, 42
$name_{a_{rt_i}}$	The name of attribute a_{rt_i} of the required type rt_i of C, 43
o_m	The output of C at method level, part of the set of state variables SV, 37
$obm_{cl}(C, C')$	Overall behavior matching at class level, 55
$obm_{ml}(C, C')$	The overall behavior matching at method level, 44
$obm_{pl}(C, C')$	Overall behavior matching at package level, 59
$occ_C^{w_i}$	The number of occurrences of the minimal word w_i in the extracted flow of C, used in the overall state matching at method level $ostm_{ml}(C, C')$, 40
$ocm_{cl}(C, C')$	Overall component matching at class level, 56
$ocm_{ml}(C, C')$	Overall component matching at method level, 45
$ocm_{pl}(C, C')$	Overall component matching at package level, 60
$odm_{cl}(C, C')$	Overall definition matching of C at class level, 49
$odm_{ml}(C, C')$	Overall definition matching at method level for two components C and C', 31
$odm_{pl}(C, C')$	Overall definition matching at package level, 57
$ostm_{cl}(C, C')$	Overall state matching at class level, 54
$ostm_{ml}(C, C')$	The overall state matching at method level, 40
$ostm_{pl}(C, C')$	Overall state matching at package level, 58
ot	Output type of C at method level, 28

pm_i	Parent module of C at class level, element of PM, 47
pn	Name of C, package level, 57
pn_{rt_i}	The name of the package in which the required type rt_i is defined, 42
rf_i	Required feature by C in C', 44
rp_i	Required package by C, 45
rt_i	Required type, element of RT, 25
s_n	State, element of S, 25
t	Type of a defined attribute da used in the definition of the set of required types RT at class level, 54
tm	Type of C at class level, 47
$type_{a_{rt_i}}$	The type of attribute a_{rt_i} of the required type rt_i of C, 43
v	Visibility modifier of C, 29
$visibility_{a_{rt_i}}$	The visibility of attribute a_{rt_i} of the required type rt_i of C, 43
$w^b_{dt_i}$	Weight of the grade of matching of a defined type dt_i of C, used for the overall behavior matching at package level $obm_{pl}(C, C')$, 59
$w^d_{dt_i}$	Weight of the grade of matching of a defined type dt_i of C, used for the overall definition matching at package level $odm_{pl}(C, C')$, 58
$w^s_{dt_i}$	Weight of the grade of matching of a defined type dt_i of C, used for the overall state matching at package level $ostm_{pl}(C, C')$, 58
w_{DA}	Weight of the set of defined attributes DA of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{DM}	Weight of the set of defined methods DM of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{EXC}	Weight of the set of exceptions EXC of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31
w_{ID}	Weight of the set of input definitions ID of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31
w_{MOD}	Weight of the grade of matching for the modifiers of a defined attribute $g_{MOD}(da_k)$, used for calculating the grade of matching $g_{da}(C, C')$, class level, 52

w_{MOD}	Weight of the set of modifiers MOD of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{MOD}	Weight of the set of modifiers MOD of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31
w_{PM}	Weight of the set of parent modules PM of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{b_h}	Weight of each element in the overall behavior matching at class level $obm_{cl}(C, C')$, 56
w_{cn}	Weight of the name cn of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{da_k}	Weight of the grade of matching $g_{da_k}(C, C')$ used for calculating the grade of matching for the defined attributes $g_{DA}(C, C')$ in the overall definition matching at class level $odm_{cl}(C, C')$, 51
w_{dm_k}	Weight of the grade of matching for a defined method $g_{dm_k}(C, C')$, used for calculating the grade of matching of the defined methods $g_{DM}(C, C')$, class level, 53
w_{exc_k}	The weight of each exception exc_k in $g_{EXC}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 36
w_{id_k}	The weight of each input definition id_k in $g_{ID}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 32
w_{m_i}	Weight of the grade for state matching for a method $ostm_{ml}(m_i, m'_j)$ used for the overall state matching at class level $ostm_{cl}(C, C')$, 54
w_{mn}	Weight of the method name mn of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31
w_{mod_k}	The weight of each modifier mod_k in $g_{MOD}(C, C')$ in the overall definition matching at method level $odm_{ml}(C, C')$, 36
w_{name}	Weight of the grade of name matching for an input definition $g_{name}(id_k, id'_k)$ in $g_{id_k}(C, C')$., 33

w_{name}	Weight of the grade of name matching for the defined attribute $g_{name}(da_k)$, used for calculating the grade of matching $g_{da}(C, C')$, class level, 52
w_{obm}	Weight of the overall behavior matching in the overall component matching at class level $ocm_{cl}(C, C')$, 56
w_{obm}	Weight of the overall behavior matching in the overall component matching at method level $ocm_{ml}(C, C')$, 45
w_{obm}	Weight of the overall behavior matching in the overall component matching at package level $ocm_{pl}(C, C')$, 60
w_{odm}	Weight of the overall definition matching in the overall component matching at class level $ocm_{cl}(C, C')$, 56
w_{odm}	Weight of the overall definition matching in the overall component matching at method level $ocm_{ml}(C, C')$, 45
w_{odm}	Weight of the overall definition matching in the overall component matching at package level $ocm_{pl}(C, C')$, 60
w_{ostm}	Weight of the overall state matching in the overall component matching at class level $ocm_{cl}(C, C')$, 56
w_{ostm}	Weight of the overall state matching in the overall component matching at method level $ocm_{ml}(C, C')$, 45
w_{ostm}	Weight of the overall state matching in the overall component matching at package level $ocm_{pl}(C, C')$, 60
w_{ot}	Weight of the output type *ot* of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31
w_{pm_k}	Weight of the grade of matching for a parent module $g_{pm_k}(C, C')$ in $g_{PM}(C, C')$, 50
w_{pn}	Weight of the grade of name matching $g_{pn}(C, C')$, used in the overall definition matching at package level $odm_{pl}(C, C')$, 58
w_{pos}	Weight of the grade of position matching for an input definition $g_{pos}(id_k, id'_k)$ in $g_{id_k}(C, C')$., 33

w_{tm}	Weight of the type tm of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_{type}	Weight of the grade of type matching for a defined attribute $g_{type}(da_k)$, used for calculating the grade of matching $g_{da}(C, C')$, class level, 52
w_{type}	Weight of the grade of type matching for an input definition $g_{type}(id_k, id'_k)$ in $g_{id_k}(C, C')$., 33
w_v	Weight of the grade of visibility matching for a defined attribute da_k of C, used for calculating the grade of matching $g_{da}(C, C')$, class level, 52
w_v	Weight of the visibility v of C, used for calculating the overall definition matching at class level $odm_{cl}(C, C')$, 49
w_v	Weight of the visibility modifier v of C, used for calculating the overall definition matching at method level $odm_{ml}(C, C')$, 31

Appendix A

Source code fragments

```
public static void encrypt(String output, String xmloutput)
throws Exception
{
        // parse file into document
        Document document = parseFile(output);
        // generate symmetric key
        Key symmetricKey = GenerateSymmetricKey();
        // Get a key to be used for encrypting the symmetric key
        Key keyEncryptKey = GenerateKeyEncryptionKey();
        // Write the key to a file
        storeKeyFile(keyEncryptKey,xmloutput);
        // initialize cipher
        XMLCipher keyCipher =
                XMLCipher.getInstance(XMLCipher.TRIPLEDES_KeyWrap);
        keyCipher.init(XMLCipher.WRAP_MODE, keyEncryptKey);
        // encrypt symmetric key
        EncryptedKey encryptedKey = keyCipher.encryptKey(document,
                        symmetricKey);
        // specify the element to encrypt
        Element rootElement = document.getDocumentElement();
        Element elementToEncrypt = rootElement;
        // initialize cipher
        XMLCipher xmlCipher =
                XMLCipher.getInstance(XMLCipher.AES_128);
        xmlCipher.init(XMLCipher.ENCRYPT_MODE, symmetricKey);
        // add key info to encrypted data element
        EncryptedData encryptedDataElement =
                xmlCipher.getEncryptedData();
        KeyInfo keyInfo = new KeyInfo(document);
        keyInfo.add(encryptedKey);
        encryptedDataElement.setKeyInfo(keyInfo);
        // do the actual encryption
        boolean encryptContentsOnly = true;
        xmlCipher.doFinal(document,
                        elementToEncrypt,
                        encryptContentsOnly);
        // write the results to a file
        writeEncryptedDocToFile(document, xmloutput);
}
```

Listing 1: Implementation of method *encrypt* in class *Encrypt_Test*.

```
public Node encrypt(
        String select,
        KeyInfoResolver kiResolver,
        Element encTypeElement,
        String type,
        Element methodElement,
        Element keyInfoElement)
    throws
        XMLEncryptionException {
    try {
        Node old = _node.cloneNode(true);
            // Setting up encryption context
        AlgorithmFactoryExtn af = new AlgorithmFactoryExtn();
            EncryptionContext ec = new EncryptionContext();
        ec.setAlgorithmFactory(af);
            // now set the key resolver
        kiResolver.setOperationMode(KeyInfoResolver.ENCRYPT_MODE);
        ec.setKeyInfoResolver(kiResolver);
            NodeList nl = XPathAPI.selectNodeList(_node, select);
            for (int i = 0, j = nl.getLength(); i < j; i++) {
            Node n = nl.item(i);
            if (n.getNodeType() == Node.ELEMENT_NODE) {
                ec.setData((Element)n);
                ec.setEncryptedType(
                    (encTypeElement != null) ?
                        (Element)encTypeElement.cloneNode(true) : encTypeElement,
                    type,
                    (methodElement != null) ?
                        (Element)methodElement.cloneNode(true) : methodElement,
                    (keyInfoElement != null) ?
                        (Element)keyInfoElement.cloneNode(true) : keyInfoElement);
                ec.setKey(null);
                ec.encrypt();
                ec.replace();
            }
        }
        return old;
    } catch(Exception ex) {
        throw new XMLEncryptionException("Exception raised during encryption!", ex);
    }
}
```

Listing 2: Implementation of first method *encrypt* in class *XMLEncryption*.

```
public Node encrypt(
        String select,
        KeyInfoResolver kiResolver,
        EncryptedType encType,
        String type,
        EncryptionMethod method,
        KeyInfo keyInfo)
    throws
        XMLEncryptionException {
    try {
        Document doc = XMLUtil.createNewDocument();
        Element encTypeElement = null;
        if(encType != null) {
            encTypeElement = encType.createElement(doc, false);
        }
        Element methodElement = null;
        if(method != null) {
            methodElement = method.createElement(doc, false);
        }
        Element keyInfoElement = null;
        if(keyInfo != null) {
            keyInfoElement = keyInfo.createElement(doc, false);
        }
        return encrypt(select, kiResolver, encTypeElement, type, methodElement, keyInfoElement);
    } catch(Exception ex) {
        throw new XMLEncryptionException("Exception raised during encryption!", ex);
    }
}
```

Listing 3: Implementation of second method *encrypt* in class *XMLEncryption*.

```
public IValidationResult validate(String value, String target, String parameter) {
    if (Boolean.TRUE.equals(this.confidentiality) && (!this.isInt(value))) {
        validationResult.setLegal(false);
        return validationResult;
    }
    IParameter stateParameter = this.state.getParameter(parameter);
    if (Boolean.FALSE.equals(this.confidentiality)) {
        if (stateParameter.existValue(value)) {
            validationResult.setResult(value);
            validationResult.setLegal(true);
        } else {
            validationResult.setLegal(false);
        }
        return validationResult;
    } else {
        // confidentiality assures that data is int value
        int position = new Integer(value).intValue();
        if (stateParameter.existPosition(position)) {
            validationResult.setLegal(true);
            // update position value with the original value
            validationResult.setResult(stateParameter.getValuePosition(position));
            return validationResult;
        } else {
            validationResult.setLegal(false);
            return validationResult;
        }
    }
}
```

Listing 4: Implementation of method *validate* in class *DataValidator*.

```
Fragment duplicate_tree(boolean with_selected) {
    // create another copy of structure and return pointer to it
    Fragment new_fragment;
    if (this instanceof DummyNode) {
        new_fragment = new DummyNode(max_no_children);
    }
    else if (this instanceof ValueNode) {
        ValueNode q = (ValueNode) this;
        new_fragment = new ValueNode(q.Value, max_no_children);
        ValueNode p = (ValueNode) new_fragment;
        p.Value = q.Value;
    }
    else if ( this instanceof Node ) {
        new_fragment = new Node(max_no_children);
    }
    else if ( this instanceof EmptySubtree ) {
        new_fragment = new EmptySubtree();
    } else if ( this instanceof Subtree ) {
        new_fragment = new Subtree();
    } else { // assert failure here?
        System.out.println("PROBLEM HERE");
        new_fragment = new Subtree();
    }
    new_fragment.example_node = example_node;
    if (this instanceof Subtree) {
        Subtree p = (Subtree)new_fragment;
        Subtree q = (Subtree)this;
        p.descend_kind = q.descend_kind;
    }
    if (this instanceof Node) {
        Node p = (Node)new_fragment;
        Node q = (Node)this;
        p.text = q.text;
        p.updated = q.updated;
        p.piece_width = q.piece_width;
        p.piece_height = q.piece_height;
    }
```

Listing 5: Implementation of method *duplicate_tree* in class *Fragment* - part 1.

```
  new_fragment.box_debug = box_debug;
  new_fragment.contains_key = contains_key;
  new_fragment.deleted_key = deleted_key;
  new_fragment.inserted_key = inserted_key;
  new_fragment.no_key = no_key;
  new_fragment.greater_than_key = greater_than_key;
  new_fragment.less_than_key = less_than_key;
  new_fragment.definitely_inorder_pred = definitely_inorder_pred;
  new_fragment.definitely_inorder_succ = definitely_inorder_succ;
  // don't copy selected, this simplifies code where we keep selected
  // region on where op activated so this way the successor doesn't have
  // a selected region also
  if ( with_selected ) {
    new_fragment.selected = selected;
  }
  new_fragment.explicit_in_abstract_state = explicit_in_abstract_state;
  for ( int i = 0; i < max_no_children; i++ ) {
    if ( child[i] != null ) {
      new_fragment.child[i] = child[i].duplicate_tree(with_selected);
      new_fragment.child[i].parent = new_fragment;
    }
    else {
      new_fragment.child[i] = null;
    }
  }
  if ( parent == this ) {
    // we are at root
    new_fragment.parent = new_fragment;
  }
  return new_fragment;
}
```

Listing 6: Implementation of method *duplicate_tree* in class *Fragment* - part 2.

Appendix B

Tool installation guide

This tool is provided as plug-in for the Eclipse IDE. In order install it the .jar file should be placed in the plugin folder of Eclipse.

To start using the plug-in the following steps should be performed:

1. Window $->$ Show View $->$ Other (which is equivalent to Alt+Shift+Q,Q).

2. Choose the views under Component Matching category.

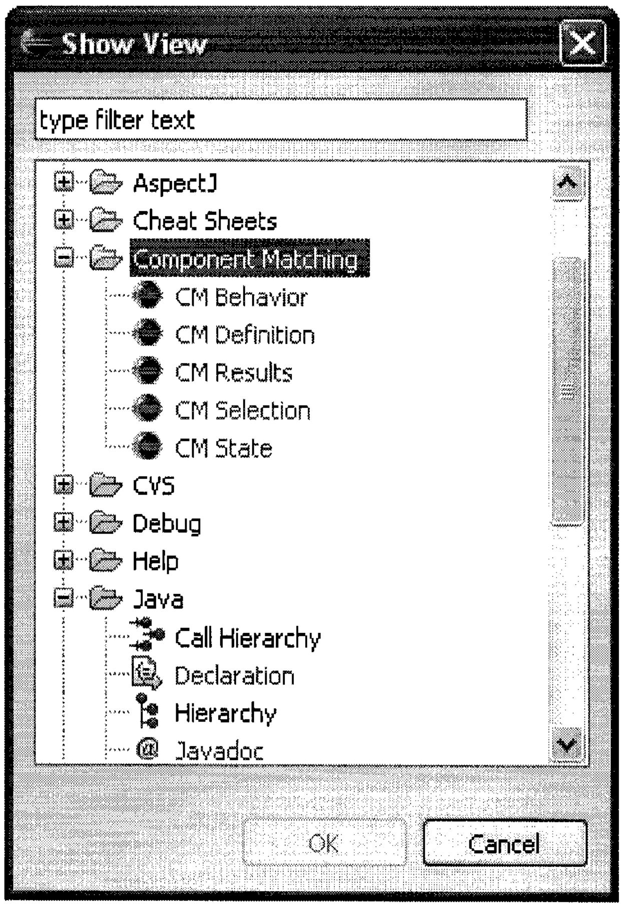


Figure 24: Views of the Component Matching category.

Bibliography

[1] Andreas S. Andreou, Dimitrios G. Vogiatzis, and George A. Papadopoulos. Intelligent classification and retrieval of software components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 2, pages 37 – 40, 2006.

[2] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00) track on The Future of Software Engineering*, pages 73 – 87, New York, NY, USA, 2000. ACM Press.

[3] Manuel F. Bertoa, José M. Troya, and Antonio Vallecillo. A survey on the quality information provided by software component vendors. In *Proceedings of the 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'03)*, pages 25 – 30, 2003.

[4] Alan W. Brown and Kurt C. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37 – 46, September/October 1998.

[5] R. Ian Bull, Casey Best, and Margaret-Anne Storey. Advanced widgets for Eclipse. In *Proceedings of the 2nd OOPSLA Workshop on Eclipse Technology Exchange*, pages 6 – 11, New York, NY, USA, 2004. ACM Press.

[6] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: Technologies, development frameworks, and quality assurance schemes. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference (APSEC'00)*, pages 372 – 379, Washington, DC, USA, 2000. IEEE Computer Society.

[7] Alejandra Cechich and Mario Piattini. Managing COTS components using a six sigma-based process. In *Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES'04)*, pages 553 – 567. Springer-Verlag Berlin Heidelberg, 2004.

[8] Alejandra Cechich and Mario Piattini. Early detection of COTS component functional suitability. *Information and Software Technology*, 49(2):108 – 121, 2007.

[9] Alejandra Cechich, Annya Réquilé-Romanczuk, Javier Aguirre, and Juan M. Luzuriaga. Trends on COTS Component Identification. In *Proceedings of the 5th International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'06)*, page 90, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design (IWSSD'96)*, pages 16 – 25, Washington, DC, USA, 1996. IEEE Computer Society.

[11] ComponentSource website.
`http://www.componentsource.com/`

[12] Constantinos Constantinides and Venera Arnaoudova. Prolonging the aging of software systems. In M. Khosrow-Pour, editor, *Encyclopedia of Computer Science and Information Technology Management*, Hershey, PA, USA, 2008. IGI Global.

[13] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439 – 448, New York, NY, USA, 2000. ACM Press.

[14] Debrief website.
`http://www.debrief.info/index.php`

[15] Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 - The FAMOOS Information Exchange Model. Technical report, University of Berne, August 1999.

[16] Ray Djajadinata. Yes, you can secure your web services documents, Part 1: XML encryption keeps your XML documents safe and secure. *JavaWorld*, August 2002.

[17] Eclipse website. `http://www.eclipse.org/`

[18] Frank Feiks and David Hemer. Specification matching of object-oriented components. In *Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 182 – 190, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[19] Bernd Fischer, Matthias Kievernagel, and Werner Struckmann. VCR: A VDM-based software component retrieval tool. Technical Report 94-08, Technical University of Braunschweig, Germany, November 1994.

[20] Brian Fitzgerald. A critical look at open source. *IEEE Computer*, 37(7):92 – 94, July 2004.

[21] William B. Frakes and Thomas P. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617 – 630, 1994.

[22] FreshMeat website.
`http://freshmeat.net/`

[23] Gerald C. Gannod, Yonghao Chen, and Betty H. C. Cheng. An automated approach for supporting software reuse via reverse engineering. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE'98)*, pages 94 – 104, Washington, DC, USA, 1998. IEEE Computer Society.

[24] Gerald C. Gannod and Betty H. C. Cheng. Strongest postcondition semantics as the formal basis for reverse engineering. *The Journal of Automated Software Engineering*, 3(1,2), 1996.

[25] Gerald C. Gannod and Betty H. C. Cheng. A specification matching based approach to reverse engineering. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 389 – 398, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[26] Judith L. Gersting. *Mathematical Structures for Computer Science.* W. H. Freeman and Company, New York, NY, USA, 1998.

[27] Jeff Hanson. Managing XML encryption with Java. *DevX*, July 2005.

[28] HDIV website.
`http://sourceforge.net/projects/hdiv/`

[29] David Hemer. Specification matching of state-based modular components. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 446 – 455, Washington, DC, USA, 2003. IEEE Computer Society.

[30] JARS website.
`http://www.jars.com/`

[31] Lamia Labed Jilani, Jules Desharnais, and Ali Mili. Defining and applying measures of distance between specifications. *IEEE Transactions on Software Engineering*, 27(8):673–703, 2001.

[32] Takeshi Kakimoto, Akito Monden, Yasutaka Kamei, Haruaki Tamada, Masateru Tsunoda, and Ken ichi Matsumoto. Using software birthmarks to identify similar classes and major functionalities. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR'06)*, pages 171 – 172, New York, NY, USA, 2006. ACM Press.

[33] Sofien Khemakhem, Khalil Drira, and Mohamed Jmaiel. SEC: A search engine for component based software development. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC'06)*, pages 1745 – 1750, New York, NY, USA, 2006. ACM Press.

[34] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*, pages 40 – 56, London, UK, 2001. Springer-Verlag.

[35] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97)*, pages 44 – 54, Washington, DC, USA, 1997. IEEE Computer Society.

[36] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, pages 301 – 309, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[37] Craig Larman. *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development (3rd Edition)*. Pearson Education,

Inc., Upper Saddle River, NJ, USA, 2004.

[38] John Lewis, Peter J. DePasquale, and Joe Chase. *Java Foundations: Introduction to Program Design and Data Structures.* Addison-Wesley, Boston, MA, USA, 2008.

[39] Jingyue Li, Reidar Conradi, Odd Petter N. Slyngstad, Christian Bunse, Marco Torchiano, and Maurizio Morisio. An empirical study on decision making in off-the-shelf component-based development. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 897 – 900, New York, NY, USA, 2006. ACM Press.

[40] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811 – 1841, 1994.

[41] Sathit Nakkrasae and Peraphon Sophatsathit. A formal approach for specification and classification of software components. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 773 – 780, New York, NY, USA, 2002. ACM Press.

[42] Sathit Nakkrasae and Peraphon Sophatsathit. An RPCL-based indexing approach for software components classification. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 14(5):497 – 518, October 2004.

[43] International Standards Organization/ International Electrotechnical Commission; Institute of Electrical and Electronics Engineers. ISO/IEC 14764:2006(E); IEEE Std 14764-2006: International standard: Software engineering - Software life cycle processes - Maintenance (2nd ed.), 2006.

[44] Opsis website.
`http://sourceforge.net/projects/opsis/`

[45] Flavio Oquendo. Formally modelling software architectures with the UML 2.0 profile for π-ADL. *ACM SIGSOFT Software Engineering Notes*, 31(1):1 – 13, January 2006.

[46] John Penix and Perry Alexander. Using formal specification for component retrieval and reuse. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS'98)*, volume 3, pages 356 – 365, Washington, DC, USA, 1998. IEEE Computer Society.

[47] John Penix and Perry Alexander. Efficient specification-based component retrieval. *Automated Software Engineering*, 6(2):139 – 170, April 1999.

[48] Andy Podgurski and Lynn Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In *Proceedings of the 14th International Conference on Software Engineering (ICSE'92)*, pages 349 – 361, New York, NY, USA, 1992. ACM Press.

[49] Gilda Pour. Component-based software development approach: New opportunities and challenges. In *Proceedings of the 26th International Conference and*

Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS USA'98), pages 376 – 383, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[50] Gilda Pour. Moving toward component-based software development approach. In *In Proceedings of the 27th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia'98)*, pages 296 – 300, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[51] Annya Réquilé-Romanczuk, Alejandra Cechich, Anne Dourgnon-Hanoune, and Jean-Christophe Mielnik. Towards a knowledge-based framework for COTS component identification. *ACM SIGSOFT Software Engineering Notes*, 30(4):1 – 4, 2005.

[52] Nelson S. Rosa, Paulo R. F. Cunha, George R. R. Justo, Jaelson F. B. Castro, and Carina F. Alves. Using non-functional requirements to select components: A formal approach. In *Proceedings of the 4th Ibero-American Workshop on Software Engineering and Software Environment (IDEAS'01)*, 2001.

[53] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar Java classes using tree algorithms. In *Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR'06)*, pages 65 – 71, New York, NY, USA, 2006. ACM Press.

[54] Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in Action: GUI Design with Eclipse 3.0.* Manning Publications

Co., Greenwich, CT, USA, 2005.

[55] Stephen R. Schach and A. Jefferson Offutt. On the nonmaintainability of open-source software. In *Proceedings of the 2nd ICSE Workshop on Open Source Software Engineering: Meeting Challenges and Surviving Success*, pages 52 – 54. ACM Press, 2002.

[56] David Schuler, Valentin Dallmeier, and Christian Lindig. A dynamic birthmark for Java. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 274 – 283, New York, NY, USA, 2007. ACM Press.

[57] SourceForge website.
`http://sourceforge.net/`

[58] Tarja Systä. Dynamic modeling in forward and reverse engineering of object-oriented software systems. In *Proceedings of Doctoral Symposium of the 13th IEEE International Conference of Automated Software Engineering (ASE'98)*, 1998.

[59] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, second edition, 2002.

[60] Clemens Szyperski and Cuno Pfister. Workshop on Component-Oriented Programming, Summary. In M. Muehlhaeuser, editor, *Special Issues in Object-*

Oriented Programming - ECOOP'96 Workshop Reader, Heidelberg, Germany, 1997. dpunkt Verlag.

[61] Haruaki Tamada, Masahide Nakamura, and Akito Monden. Design and evaluation of birthmarks for detecting theft of Java programs. In *Proceedings of the IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics*, pages 569 – 575. IASTED/ACTA Press, 2004.

[62] Haruaki Tamada, Masahide Nakamura, Akito Monden, and Ken-Ichi Matsumoto. Java birthmarks - Detecting the software theft. *IEICE Transactions on Information and Systems*, E88-D(9):2148 – 2158, 2005.

[63] Marco Torchiano, Letizia Jaccheri, Carl-Fredrik Sørensen, and Alf Inge Wang. COTS products characterization. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 335 – 338, New York, NY, USA, 2002. ACM Press.

[64] Padmal Vitharana. Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67 – 72, 2003.

[65] Padmal Vitharana, Fatemeh Mariam Zahedi, and Hemant Jain. Knowledge-based repository scheme for storing and retrieving business components: A theoretical design and an empirical analysis. *IEEE Transactions on Software Engineering*, 29(7):649 – 664, July 2003.

[66] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81)*, pages 439 – 449, Piscataway, NJ, USA, 1981. IEEE Press.

[67] Yijun Yu, Yiqiao Wang, John Mylopoulos, Sotirios Liaskos, Alexei Lapouchnian, and Julio Cesar Sampaio do Prado Leite. Reverse engineering goal models from legacy code. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 363 – 372, Washington, DC, USA, 2005. IEEE Computer Society.

[68] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146 – 170, April 1995.

[69] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching of software components. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'95)*, pages 6 – 17, New York, NY, USA, 1995. ACM Press.