Component Types Qualification in Java Legacy Code Driven by Communication Integrity Rules

Hugo Arboleda
Universidad Icesi, DRISO Group
Calle 18, No. 122-135
Cali, Colombia
hfarboleda@icesi.edu.co

Jean-Claude Royer
ASCOLA Group, Mines de Nantes - INRIA
4 Rue A. Kastler
44307 Nantes, France
Jean-Claude.Royer@mines-nantes.fr

ABSTRACT

Component Based Software Engineering is a way to improve software modularization and to embed architectural concerns in the source code. Making explicit the architectural concerns in code helps to mitigate the problem of architectural erosion. The restructuring of legacy code with components in mind requires the use of tools to assess compliance with component programming principles. The property of communication integrity is one of the major principles for implementing software architectures. However, there is a paucity of tools for assessing the quality of code components. To cope with this issue, we define a component model in Java and a tool for identifying component types, which relies on a set of rules to statically check potential violations of the communication integrity property in Java source code. We illustrate its application with a case study and report the results of our experiments with it.

Categories and Subject Descriptors

D.2.11 [**Software Architecture**]: Languages; D.3.3 [**Languages Constructs and Features**]: Data types and structures

General Terms

Languages, Experimentation

Keywords

Architecture, communication integrity property, component based programming, component type, data type, assessing quality.

1. INTRODUCTION

Software applications must be maintained under the dynamic conditions of the real world, to improve security, performance or other quality attributes, and to integrate new functionalities. A side effect of this evolution is architectural erosion, namely, the erosion which occurs when a system architecture gradually degrades as developers make changes to the system that violate the original architectural intent. Software architecture plays a crucial role in several aspects of software development [20]: understanding and documenting, construction, evolution, analysis and verification, reuse and management. Developers unintentionally degrade the architecture because they are either not aware of the architectural intent of the system under development, or they do not completely understand the system architecture. A typical consequence is that the system becomes gradually more difficult to maintain and evolve while more communication channels (*e.g.* method calls) are established among all parts of the system. New languages and development methods have been proposed for the benefit of software practitioners, to mitigate architectural erosion and to support software maintenance and evolution. They make explicit the architectural decisions both in architectural models and source code, and promote software modularization. Furthermore, they allow checking of architectural constraints and the application of strict programming principles like the *communication integrity* property [27, 6]. Component Based Software Engineering (CBSE) [29] is a branch of software engineering which is mainly concerned with software modularization, separation of concerns, and architecture.

Here, we consider the problem of re-engineering legacy code with component based programming and how to asses the quality of the resulting application. Such a process cannot be managed without tool support [22] and assessing the CBSE resulting quality is essential to guiding the restructuring process. Metrics based tools are possible, however, they are far from perfect [28, 21] and they are not sufficient to assess semantic properties like type-checking rules or the communication integrity property. One alternative could be to use tools for components recovery and architecture reconstruction from source code. A number of recent proposals provide tools for extracting components or architecture that are suitable to modern component languages with component hierarchies or composites [12, 13, 11, 8]. However, extracting tools are not adequate to assess the component quality of a program since they are devoted to identifying parts in code which can become components after restructuring. Checking tools, such as PMD [2], findbugs [1], or architectural compliance checking tools, such as the one presented in [24], could be used. However, to our current knowledge, there is no tool to assess the quality of CBSE applications with first-class composites and subtyping. None of the existing tools is based on programming rules that explicitly ensure the communication integrity property. In this paper, we focus on the definition of a method and its corresponding tool to inform developers about the current quality of the components types. We provide information about data types, component types, and the provided and required services, among others. To qualify component types and to separate them from data types we rely on the the so-called communication integrity (CI for short) property. This states that the real communication links have to be compliant with the static system architecture. The CI property emphasizes the need to avoid hidden communication channels. In other words, two components can communicate only if a communication channel has been formally defined between them. Our specific contributions in this paper are: *i)* to make explicit the requirements of our method and analyzer tool and to demonstrate that current extracting tool or metrics-based tools are not adequate for the task that we carry out, *ii)* to sketch a Java component model which allows first-class components, component types, subtyping, and a straight implementation in Java, and *iii)* to propose a set of rules to prevent CI violations. The rules set reuses rules previously presented in ArchJava [6, 5], but also adds new ones. Finally, we present a first evaluation of our tool: we measure the component types rate of several applications, restructure some applications, and compare the tool result with the SoMoX component recovery tool.

The remainder of this paper is organized as follows. Section 2 gives the context of this study and describes introductory examples to motivate our study. We present related work in Section 3. In Section 4, we describe the principles of our component model and give details of the rules we use in our tool to prevent communication integrity violations. Section 5 discusses a restructuring example and presents elements to evaluate our tool. Section 6 contains a summary of our contribution and sketches future work.

2. BACKGROUND AND MOTIVATIONS

This section presents the CBSE background as well as our motivations and the need to asses CBSE quality.

2.1 CBSE Principles

CBSE is not only a pure programming concept but also an architectural approach. It introduces a more strict way to program applications than Object-Oriented Programming (OOP), for example to allow automatic checking of architectural properties. CBSE aims to improve software development practice by proposing a development model where systems are assembled from components rather than programmed from scratch. CBSE claims to reduce development costs and improve the reliability of the resulting system. We propose here a basic and generic definition of the main CBSE elements. There are many proposed models that implement these main elements and others about which we will not enter into details here (*e.g.* the notions of ports and connectors). For a recent review of component models see [15], but here we focus on models with interface and hierarchical composition. In CBSE, systems are built from reusable *software components* that require or provide *services*, and that have well defined *communication channels* between them. Bosch, Szyperski and Weck [9] propose the following definition of components: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties," as quoted in [14]: this definition is the most frequently used today. This definition implies that *i)* a software component is a unit, *i.e.* it cannot be divided and it is self contained (except for its declared required interface), *ii)* it specifies an interface (or interfaces) of services it provides, and it is bound by contract to implement at least this (these) interface(s), *iii)* it specifies context dependencies, which means services it requires to work properly, and *iv)* it may be part of a larger *composite component*. A composite component is built from other components, a component which is not a composite is called a *primitive component*. The specification of the *provided* and *required* services make up the *interface* of the component. Typically, a *service* will be implemented by a routine or a method accessible from outside the component. These properties make any component *substitutable* by other components that provide or require the same or compatible services. We should distinguish *component types* and *instances* [29]. A component type is a generator for component instances. Finally, as in [5, 16], we find it valuable to consider the notion of *subtyping* as a formal way to organize types in the applications.

Concomitant to CBSE is the notion of component based architecture [14]. A component based architecture specifies how components will be arranged to form the system. It includes the specification of the components and their interfaces (provided and required services); the *communication channels* between them, that is to say, from what other components a component may require services (and which ones), and to what other components it may provide services (and which ones). One important property a component architecture must satisfy is the *communication integrity* property [27]. That is to say, it must ensure that the implemented components do not communicate between themselves in ways that violate the intended control flow rules of the architecture. In other words, two components can communicate only if a communication channel has been formally defined between them, there are no hidden communication channels. This is one of the strengths of the component approach; it allows designers to explicitly specify and automatically check some of the architectural decisions, thus actively limiting the chances of architecture erosion. Regarding the maintainability of applications, the concept of architecture plays a role similar to general documentation. The architectural description can be outside the source code, inside the source code, or both. The use of the CI property forces this document to be part of the code and consistent with it. A previous discussion related to software quality criteria and the CI property appeared in [3].

2.2 Assessing CBSE Quality

In the context of reverse engineering we need tooling support to evaluate the quality of applications and to guide the restructuring process. To assess the quality of CBSE code, we could use either metrics tools, or components extracting tools, or source code analyzers, or architecture compliance checkers. Software components and architectures metrics are still a research area. To the best of our knowledge, there exist two surveys [28, 21] on software component metrics. In its conclusion, [28] states that "more work is needed" and the authors note a lack of automated tools. Goulao and Abreu, in [21] reviewed and compared the existing approaches. Amongst other conclusions, they note: "there is a lack of maturity in existing proposals." Metrics tools could give some useful information about the code structure. Nevertheless, there are several important tasks for which they are inadequate. First, these tools are relevant to measure global qualities of a code such as reusability, maintainability or evolvability. They are not adequate to check semantic properties such as those ensuring that: component types do not appear in public services, component types do not inherit from a data type, etc. Currently, existing approaches do not cope with component subtyping: this could seriously affect metrics definitions. Component and composite types extracting tools [13, 12, 8] are generally based on metrics. They could be used to extract components and then to evaluate the quality of the result. However, they are devoted to find new component types in source code that is not necessarily CBSE code, or not designed with the required rigor. These kinds of tools are able to collect components services and to suggest clusters, aggregating classes, even if there are no components in the code. Our study in Section 5 confirms that they are not suitable for our purpose. The type of semantic properties we want to check are usually verified by tools for the static analysis of the source or byte code, as in the PMD [2], or findbugs [1] tools. Our survey of these tools do not reveal an adequate candidate to assess CBSE quality, or even to ensure good programming style with hierarchical components.

Finally, architecture compliance checkers, like SAVE [24], are devoted to verifying the conformity of a given architecture with the one extracted from the source code. They rely on a light component model with neither subtyping nor first-class components. Furthermore they require a given architecture which we may not have at hand.

2.3 The Communication Integrity Property

A tool is required to guide the restructuring process that identifies types respecting the communication integrity property and related good programming practices. The CI property is a simple, abstract and yet characteristic property of CBSE. Our idea is to consider a type as a component type if it is not responsible for CI violations. Conversely a data type can be used to build hidden channels. CI prevention covers various cases and some are known to be good programming principles, for instance, to prohibit public fields in a type definition.

Listing 1: Sample Code

```
public class A {
        public A(B b) { this.theb = b; }
        public A getIt() { return this; }
        private void setIt(A a) { ... }
} // end A
```

To be more concrete, let us consider the Java code example in Listing 1. Assume we want class `A` to become a component type. Following previous work, [27, 6, 5], it is possible to statically verify the CI property if a component type never appears as parameter or resulting type in a service. Thus, the developer should note that the `getIt` method violates this CI rule and should modify the interface of this type. Private or protected methods, for example `setIt`, can have component types in their signatures provided that these methods are only called on `this` or `super`. This rule prevents the exchange of component references between components of the same type. Let us consider a more complex example

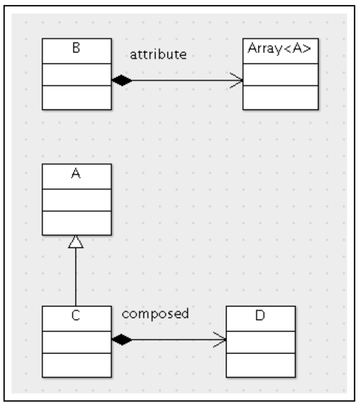

Figure 1: A UML Sample.

with the UML diagram in Figure 1. Arrays allow direct access to their inner values, thus owning an array of `A` enables direct access to an `A` reference. Another component could capture such an `A` reference. Then it could send a message defined in `A` and build a hidden channel from itself to the `A` component instance. Thus `A` should be considered as a data type. But, this is also true for class `C`, since by polymorphism a `C` reference could be used in place of an `A` reference and then the same construction as above is possible. This problem occurs also for the composition part `D` of class `C`. Methods of class `C` can access the `D` instances, and a longer hidden communication channel can be built. As we can see here, violations propagate along subtyping and composition links. Note that there is no violation regarding the `B` type, it could be a component type. Even supposing that an OO program is implemented with a clear separation of components and data types, there are numerous ways to establish uncontrolled communication channels: *i)* passing a reference to a "component" object as a method's argument to another "component"; *ii)* encapsulating a "component" object into a "data type" object, and passing this one around; *iii)* having a "component" inheriting from a "data type" object and casting it to "hide" its component nature; *iv)* using data sharing between two "component" objects as a hidden communication channel; or *v)* one could even use the reflexive API (in the case of Java). We should add to this list some specific cases related to language features, such as inner classes, exception classes, or generic definitions. These CI violations are, without a tool, difficult to evaluate precisely. The reader might think that in most of the programs there are only data types, never component types. Our experiments in Section 5 show that this is not true: often, many component types exist. They usually appear in the top layer of any OOP applications, but they could be more numerous if the application was designed carefully or with components in mind. Summarizing our requirements for such a tool: *i)* It has to analyze Java code and qualify component and data types; *ii)* It has to be based on some strong semantic property, the CI property seems well-suited to that; *iii)* It has to provide information about potential violations of the CI property and its consequences; *iv)* It has to be as efficient as possible; *v)* It could provide more information about various related problems, such as communications outside of the component boundary, cycles in the compositions of components, subtyping problems, and direct access to attributes, among others.

3. RELATED WORK

In CBSE reverse-engineering, the concepts of component and architecture vary from one approach to another. For example, in [23], design components are high level concepts close to design patterns, but they are not abstract components. There has been a lot of research on architecture and component recovery in the reverse engineering community (see Koschke, [26] and [25], for a review of the field). However, the problems they tackle are different. *Architecture recovery*: typically tries to partition the set of software elements of a system into various coherent subsets. For this, it may use clustering to group together the elements that have more things in common. *Component recovery*: typically tries to group constants, variables and routines of a procedural source code into objects or classes. For this it considers what variables or constants are accessed by routines, and may also use clustering of variables and constants. This is far from our preoccupation, as we work from an object-oriented system assuming that the classes correspond (or may correspond) to components. We are not trying to re-group things together, but rather to identify types that have component properties. Washizaki *et al.* [34] propose to extract JavaBeans components of Java programs. Only the structure is abstracted, while, in addition, we consider the communication integrity property. Our analysis of the composite structure is comparable to their structural clustering algorithm but we simplify by assuming that the component structure is built from the class fields. Favre *et al.* [18] describe how they (manually) build a meta-model from a component based source to help understand the system and help build reverse engineering tools for that system.

Component or architecture recovery tools suitable to component languages with component hierarchies are closely related to our work. We are interested in static analysis since it provides abstract and stable information about the structure and behavior of the code. [13, 12, 11, 8] propose automatic and static approaches, while [33] proposes a dynamic one. Chouambe *et al.* have objectives very similar to ours, however, they chose to extract the components from metrics. SoMoX [8] (SOftware MOdel eXtractor) is a part of the Palladio tool suite developed at the Karlsruhe Institute of Technology. It is the successor of the ArchiRec tool which was described in [13]. This tool is able to analyze C++, Delphi, and Java source code. The tool defines an iterative reverse engineering process based on the Palladio Component Model. It collects the provided and required interfaces associated to every class in the Java application. A primitive component type is a class with at least one public service. This leads to a first list of primitive component types. The extraction of composite types considers composition hypotheses, but for complexity reasons the combinations are restricted to pairs of component types. These pairs are compared based on several metrics dedicated to measure various aspects, for instance, coupling, name resemblance, instability and abstractness. SoMoX uses a formula taking into account the metrics inter-dependencies to rate the composition hypothesis. Once the primitive component types are extracted, an iterative process is responsible for the identification of the composites using the combined metrics to compute a global score. This process is also able to detect composition of more than two components. The discovering process is iterated several times to suggest higher-level compositions until no new composite is found. We did some comparisons with SoMoX in our evaluation section, see Section 5. The work of Chardigny [12, 11] is relevant for our study. Their approach, called ROMANTIC, focuses on a quasi-automatic extraction of a component-based architecture from an existing object-oriented system. The ROMANTIC tool also uses metrics, but relies on a different approach than clustering. The first step of the extraction is defining a correspondence model between object concepts and architectural ones. This correspondence is elaborated by the architect. Then the tool validates this correspondence using predefined guides based on semantic and qualities of the architecture. The process selects among all the architectures that can be abstracted from a system, the best one according to the set of guides. The guides are assumed to be measurable constraints to model the extraction process as a balancing problem of these competing constraints. The extraction problem is a search-based one and uses the Low-Temperature Simulated Annealing algorithm. The currently available information does not provide performance measures, and the approach is costly, at least from a theoretical point of view. However, ROMANTIC is not yet publicly available to compare it with others.

In [7] we presented a first version of our tool with the objective to extract component types from Java applications. Experimenting on several middle size case studies we concluded that the tool could recognize primitive component types. But we could not recognize composite types since they were often implemented in a different way.

In their paper Bowman *et al.* [10] study various ways to extract models from Java code. Two approaches are possible: static or dynamic analysis. Static analysis usually gives more abstract information. Dynamic analysis depends on the execution context and may provide very accurate information about polymorphic call, dynamic types of objects, and information related to the use of the reflective Java API. However, it usually produces a huge quantity of information that one must filter. The model sought by [10] is a simple entity relationship model but it is not too far from a component model. The conclusion from [10] is that: if static analysis is sufficient then disassembling (Java byte code) is probably the best choice. Choosing between static or dynamic analysis is still a matter of opinion. For instance, [19] considers that runtime analysis or profiling is needed, since types and objects may be dynamically created.

We should also mention work around ArchJava [6, 5], a Java extension which enforces integrity of the architectural flow of communications. AliasJava [4] is as an extension of ArchJava allowing the detection of communication through shared data. In [3], Abi-Antoun *et al.* manually re-engineered a Java legacy system into an ArchJava system with explicit (and enforceable) definition of control flow and data sharing. Enforcing communication integrity and making explicit data sharing is challenging in programming languages with objects and references. This corresponds closely to our final objective although the work presented here is only the first step toward this objective. On the issue of the integrity of the architectural flow, the rules used in this paper are inspired by those used in ArchJava. The rules for method signatures and composition are the same. We are more liberal on the external types analysis and on constructors but stricter on downcast checking. Here we introduce new rules for generics. The main difference with the ArchJava language is that we define a kind of inference system mining for data types in pure Java code and propagating this information along inheritance, composition, and coping with most of the Java features.

4. THE QUALIFICATION RULES

In this section we present the model underlying our tool and we detail its set of CI rules. This tool, called JCE, is an Eclipse plugin devoted to analyze and report potential violations of the communication integrity property, its consequences and other related CBSE programming practices problems. This tool is based on a strict component model with a straightforward implementation in Java. All of its features have a direct correspondence to Java constructions. The underlying component model relies on the following assumptions: *i)* component types are true types, *i.e.* that can be instantiated to generate component instances, *ii)* they communicate via a strict message passing policy, *iii)* they can be either concrete or abstract component types, *iv)* they support subtyping, and *v)* composites are built from a class structure containing subcomponents. The qualifying process of our tool is split into two steps: *i)* mining the code to identify the relevant elements of the component model and *ii)* running rules to identify data and component types as well as to collect CI violations.

4.1 Mining the CBSE Model

Since our approach relies on static analysis of source code, it is based on types rather than instances. Our mining process collects classes of any kind (generics and nested classes) and also interfaces, to build the set of *types of interest* in the source code. These types are tagged as abstract or concrete according to the abstract modifier in the source code. We define *the composition structure* of a type as the types of its fields or attributes. We consider the maximal structure, which means we group all the defined attributes and the inherited ones, except the inherited private fields since, in Java, they cannot be accessed in the subclasses. A *service* is a public or a default method. A *profile* is a method signature, that is, a method's name with its parameter types and resulting type (as usual we use `void` for procedures). We call a *wrong profile* a profile in which a type of interest appears as parameter or as resulting type. These signatures are important since they can be responsible for a component reference capture and must be removed from the component system if we want to ensure the communication integrity property. *Provided services* are all the available services defined in the component type. The *required services* of a type are those methods that are called in the source code of the type. We compute *subtyping relationships* from the language inheritance and subtyping relationships. In Java, there are two such relationships: `extends` and `implements`. Note that CBSE usually makes little use of subtyping, nevertheless [5, 16] are counter-examples. Subtyping provides a formal way to organize types [16] and is convenient to use in CBSE. We also decided to deal with subtyping because some implementations may use it to represent component types and their communication interface. If we want component subtyping we need to check the compatibility between provided and required services of the component types. Provided services follow the same rule as inheritance in OOP [31]: the sub-component must provide at least all the services provided by its super-component. However, it is different with required services, which obey the so-called contra-variance rule: the sub-component must require at most all the services required by its super-component. The intuition behind this is: if a component subtype has more required services than its ancestors, then an execution could trigger a message which cannot be recognized by the execution context. A similar analysis can be done for parameters and return type of this service. Practically speaking, along subtyping, the provided set can be enriched and strengthened while the required set must be relaxed. Obviously, these rules should be checked by our tool but we avoid a precise discussion of this in this paper.

4.2 Static Checking of the CI Property

Few languages are able to enforce basic CBSE properties (*e.g.* [4, 5, 27]). One important idea behind components is that we have a static map of its construction, this is the so-called architecture. This map exposits the structure and communications between components, and in a good implementation, only those communications can arise. It is difficult to statically check the CI property [27, 5]. The principle of the control in ArchJava relies on strict rules prohibiting components from escaping from their enclosing component. In addition the type system checks that the communications defined between (statically or dynamically created) instances are compliant with the connection information (connect and connect pattern constructions). There are three exceptions to the stated static checking of ArchJava: *i)* there are dynamic controls of casts, *ii)* the legacy mode relaxes some rules, for instance, inheriting from external types, and *iii)* communications via shared data are not checked. As in the ArchJava language, we avoid hidden communications via data sharing, see AliasJava [4] for a solution, and we ignore the use of the Java reflexive API. We do not consider connect pattern as in ArchJava since they would not occur in legacy code. Our communication model relies only on direct calls and we have no explicit connection information. Thus the best we can do is to define rules disabling components escaping from their parent component. We qualify the types of interest of the project as either a component type or a data type using a set of rules defined in Section 4.3. Basically, a component type ($CType$) must respect the rules, if not it is qualified as a data type ($DType$). We call $ETypes$ the set of external types not defined in the project under analysis. An instance of a $CType$ is a *component*, while a *value* is an instance of a $DType$. Following the work on ArchJava [5, 6, 4], our tool considers that a component can neither be passed as a reference in a service, nor be a visible part of a type. We do not check all the possible violation cases, we choose some rules we think are the most important. Our goal is rather to identify the types which are violating the most critical rules. Furthermore, in our context (legacy code) we think it valuable to weaken some rules and to have complementary information about the related infractions. We slightly change some rules about subtyping and we restrict some casts rather than adding dynamic control for casts. Constructors are not checked and can enable component sharing and violations of CI. Static fields and static nested classes are not yet addressed, since we are still not sure how to build components with these features. We also add new rules to restrict $CTypes$ occurring in a visible generic construction, and a $CType$ cannot be an exception class. These rules are complemented with a set of exceptions defining more local problems that we consider less important. They are related to boundary infractions, casts in case of subtyping between component and data types, and non visible members calling on expressions other than `this` or `super`. These infractions are not detailed in this paper.

4.3 The CI Rules

With the information extracted from the source code, our approach checks for some rules in order to statically ensure that components cannot escape from their enclosing parent.

Wrong Profiles.

1-a) This first rule looks for public profiles in the application. If a type of interest is passed as parameter of a service or returned by a service it is considered a $DType$. The method signature is qualified as a wrong profile. This rule is applied to all types, not only to $CTypes$. With this heuristic it would be easier to change a data type into a component type during refactoring.

1-b) Exception to rule 1-a: a component may be passed to or returned by a constructor. In legacy code the use of parameters in constructors is a convenient way to initialize composite structures. The heuristic used here is to report the information about these violations to the user for later consideration. Our experiments have shown that it also renders more iterative the process of fixing violations, as suggested in [3].

1-c) Private and protected methods of $CTypes$ can have component types in their signatures as long as they are called on `this` or `super`. This rule prevents inner components from escaping their enclosing component.

Composition. Encapsulating a component reference into a value opens the risk of a hidden channel. It is not difficult to see that the value can be captured and a method of the data type can indirectly send a message to the internal component. This problem can arise with any attribute modifier, even if it is a private part.

2-a) A type occurring in the structure of a $DType$ is a $DType$.

2-b) Types in public or default package fields of $CTypes$ are $DTypes$ since they are publicly available.

2-c) $CTypes$ can have $CTypes$ as private or protected fields but they are only accessed *via* `this` or `super` to prevent components escaping from their parent.

Subtyping.

3-a) A subtype of a $DType$ is considered a data type. This follows from rule 1-a, since instances of the subtype could be used as parameters or return values using polymorphism resulting from subtyping. However, $DTypes$ are allowed to be subtypes of $CTypes$, but an infraction is reported in case of a cast expression compliant with this subtyping relation.

Arrays and Generics. Array and generic constructions add several complications[1]. Arrays allow access to their inner data, this is also

[1]ArchJava considers array constructions but it says nothing about the use of generics.

true with generics like `Vector` or `ArrayList` provided by the Java library since version 1.5. Array or generics in $DType$ fields should be naturally considered as $DTypes$ and this implies that the types of their stored values are also $DTypes$. Instantiation of arrays or generics used as formal parameters in services or in visible field declarations would allow indirectly accessing the stored references. We use a strict rule and consider that it applies also to generic classes defined by the user.

4-a) We consider the actual parameters of arrays and generics in public or default field declarations as $DTypes$.

4-b) We consider the formal parameter type of arrays and generics in services as $DType$.

4-c) The previous rule applies also to private and protected fields of a $DType$ (from 2-a).

4-d) In case of a generic instantiation used as a superclass or a super interface, their formal parameters should be flagged as $DType$ (from 4-a).

4-e) Addition to 4-d: The subclasses and implementations of the generic instantiation (from rule 3-a) have to be flagged as $DTypes$.

Nested Classes. Nested classes are generally used in GUIs, or to implement some specific features like simulating multiple inheritance. There are four kinds of nested classes: one static nested class and three types of inner classes – non-static member, local, and anonymous class.

5-a) If the inner class[2] is a $DType$, one of its instances could escape from its context and could allow access to the enclosing component reference. In this case the enclosing class should be declared a $DType$.

Exception Classes. Exceptions are class instances enriched with a `throw/try-catch` mechanism. They should be checked like ordinary types, which can be default-package or publicly visible. But an exception type occurring in a `catch` clause is a $DType$.

6-a) A strict and pragmatic rule is to always consider exception types as data types.

External Types. External types of interest ($ETypes$), not defined in the Java project, are ignored. In ArchJava [5], this corresponds to the legacy mode introduced in the compiler. We chose to ignore all types of interest not defined in the Java project (*e.g.*, ignore all external libraries as `java.io.*`, or `org.eclipse.*`). One reason is that we want to extract the provided services of the components, and their structure. This requires having access to the source code (the Java reflective API could help, but we favor a more generic solution). Also, there are good chances that `Object` will be passed as a parameter of, or returned by, some method, turning it into a $DType$ (rule 1-a). This will, in turn, qualify all types of interest as data types (rule 3-a). Nevertheless, it is convenient to extend some external data types. Lastly, we cannot hope to restructure the entire world and we would like to limit ourselves to the application under analysis. Thus we consider that the external world does not introduce communication integrity problems. This is surely wrong and the designer has the responsibility to ensure this. One restricted way is to check for suspicious downcast from an external data type to a component type. To provide more precise checking in the case of extending external type is still an open problem [3].

[2]For technical reasons, local and anonymous classes are not yet identified by our prototype.

In summary the two kinds of types ($CTypes$ and $DTypes$) can neither define wrong services nor public and default fields of component types, generics of component types, or arrays of component types. Protected and private methods can have component types, generics of component types, or arrays of component types in their signatures. A $CType$: *i)* is not an exception class, *ii)* cannot have an inner $DType$, *iii)* does not inherit from $DTypes$ or generic instantiation, *iv)* is not a formal parameter of a visible array or generic, *v)* can have protected or private fields of component types, generics of component types or arrays of component types. A $DType$: *i)* can be an exception class, *ii)* can have inner types, *iii)* can inherit from any other type, *iv)* can be a formal parameter of an array or generic, *v)* cannot have private or protected fields of component types, generics of component types or arrays of component types.

4.4 The Extracting Process

JCE is based on the JDT API [17] which provides a model of the Java source code and procedures to analyze it and to compute information. The JDT API also provides several tools to rewrite the source code, to build an abstract syntax tree of the code, and to help in defining parsers. This approach is currently available for Java only, see the web site [32] for more details and experiments. The JCE tool analyzes the main Java elements: classes (abstract or not), interfaces, inner classes, subtyping, arrays, generics, attributes, and method signatures. The first task of the tool is to extract an abstraction of the source code compliant with our component model principles. Thus the tool mines the source code for the types, their structure, provided and required services, communications and subtyping links. The tool exploits this information to check for CI rules violations. The main rules are devoted to the distinction between component types and data types. By default all the types are initially $CTypes$. The rules which are responsible to flag types as $DTypes$ are run first (1-a, 2-b, 4-a, 4-b, 4-d, 4-e, and 6-a). Then the propagation rules (2-a, 3-a, 4-c, and 5-a) are applied to propagate the $DTypes$ along subtyping, composition, and enclosing inner class.

5. TOOL EVALUATION

The evaluation of our tool is not obvious, as mentioned in the related work there is no other tool devoted to the same purpose for a precise comparison. We tested it on several examples to both validate that our heuristics and set of rules are not too strict and enable effective refactoring of some data types into component types. This section gives a detailed example as well as a summary of other experiments. As a first practical means of evaluation we experimented with the manual refactoring of several small and middle-size applications.

5.1 Using JCE on MineSweeper

In this section we demonstrate how to use the information provided by the tool to help in refactoring a simple Java project. We do not describe a precise refactoring process with a given architecture in mind. The goal is to improve the component structure of the application and the tool will help in making this structure explicit and to show CI violations. This example is based on the `MineSweeper` game: "Minesweeper is a single-player computer game. The object of the game is to clear an abstract minefield without detonating a mine. The game has been written for many system platforms in use today" (from Wikipedia). We use the implementation provided by Tim Van den Bulcke[3]. It has nearly 800 lines

[3]http://timvandenbulcke.objectis.net/minesweeper-in-java

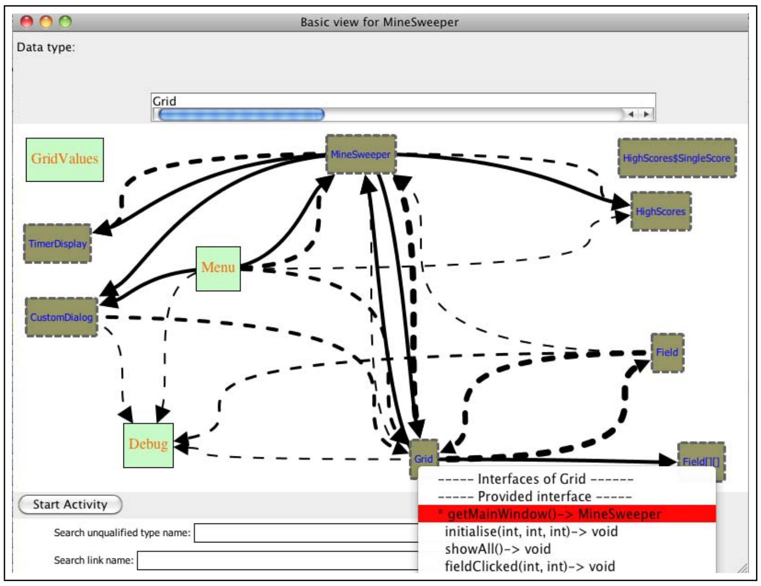
Basic view for MineSweeper
Data type:
Grid
GridValues
MineSweeper
HighScores$SingleScore
HighScores
TimerDisplay
Menu
CustomDialog
Field
Debug
Grid
Field[][]
----- Interfaces of Grid ------
----- Provided interface -----
* getMainWindow()-> MineSweeper
initialise(int, int, int)-> void
showAll()-> void
fieldClicked(int, int)-> void
Start Activity
Search unqualified type name:
Search link name:

Figure 2: Analysis of MineSweeper.

of code and 10 classes. The code was not yet compliant to good Java practices with regards to, for instance, naming conventions, documentation or data encapsulation. The restructured application was tested by playing parties. An initial graphical view of the analysis is depicted in Figure 2. The tool provides also detailed textual information. In this example, the dark gray boxes are data types and the light gray boxes are component types. Arrays and generics instantiated in the code are depicted as data types, for instance the node at the right bottom represents `Field [][]`. Plain arrows go from a composite component type to a component type part, dashed arrows illustrate a communication, *i.e.* one or more method call(s), from the caller to the callee. The width of the dashed arrow indicates the "strength" of the communication link (the number of services called on this communication). A context window on a communication link shows what services are involved in this communication. A context window on a node, lower right window, for node `Grid` in the figure, shows the services it provides and requires. A wrong profile appears as a provided service, colored red, and with a star before its name. We can see in Figure 2 that the method `getMainWindow` is responsible for marking the `MineSweeper` class as a data type. There are three wrong profiles in this application:
`Grid.getMainWindow() -> MineSweeper`,
`Field.getGrid() -> Grid`, and
`SingleScore.isBetterOrEqualScore(SingleScore) -> boolean`. There is also a public two dimensional array of `Field` which is responsible for flagging `Field` as a data type. The above profiles cause `MineSweeper` and `Grid` to be flagged as $DTypes$, then `CustomDialog` and `TimerDisplay` are $DTypes$ because they are parts of `MineSweeper`. `HighScores` is a $DType$ for two reasons: it is part of `MineSweeper` and it has an inner $DType$ (`SingleScore`). There are also several public fields which are responsible for flagging several types as $DTypes$. The manual refactoring process starts with the `Grid` type, trying first to solve the `Field.getGrid() -> Grid` wrong profile. This method is used in `Field` by the `mouseClicked` and `showValue` methods. A solution is to introduce a field `grid:Grid` in `Field`, replacing calls to `getGrid()` by accesses to `this.grid`, and setting the value of this field in the constructor. We also modify the `Grid` initialize method to pass `this` to the created fields. A second step is to solve the wrong profile: `Grid.getMainWindow() -> MineSweeper`. In the `Grid` class, there is already a `mainWindow:MineSweeper` attribute. This method was also used in the `Field` class. Defining a new `lost` method in class `Grid` relaying the `lost` call to the `mainWindow` attribute removes the need of this wrong profile. After manually removing these two wrong profiles, using the tool it is easy to see that the set of $CTypes$ does not change. We can observe that `Field` is a data type since it appears in the public attribute `Field [][] grid` in the `Grid` class. Then this implies by composition that `Grid` is a data type and then so is `MineSweeper`. Four public fields in `MineSweeper` are also responsible for flagging `CustomDialog`, `Grid`, `Menu` and `TimerDisplay` as $DTypes$. Our third step is to fix these public accesses, one in `Grid` and the four others in the `MineSweeper` class. Three of them can become private fields of their class without any further modifications. The `HighScores highScores` field in class `MineSweeper` becomes private, we add a corresponding parameter in the `Menu` constructor, and we modify its call in class `MineSweeper`. The `Grid grid` field requires a similar modification. We add a new corresponding parameter in the `Menu` constructor with the correct initialization in class `MineSweeper`. We also modify the `CustomDialog` constructor to accept a new grid parameter and we modify its call in the `Menu` class. These are quite simple refactorings. As a fourth and final step we extract the inner class from the `HighScores` file. In this case only `SingleScore` is a data type, however it is more difficult to solve this case which is due to the third wrong profile.

There is still some work to make explicit a good architecture: constructors CI violations remain to be solved, communications should be consistent with the component structure, provided or required profiles should respect subtyping rules, etc The JCE tool enables incremental refactoring. The refactoring result can be subjected to more advanced refactorings, for instance targeting the ArchJava language. In a realistic refactoring process designers and architects may need to iterate several times [30]: between analysis

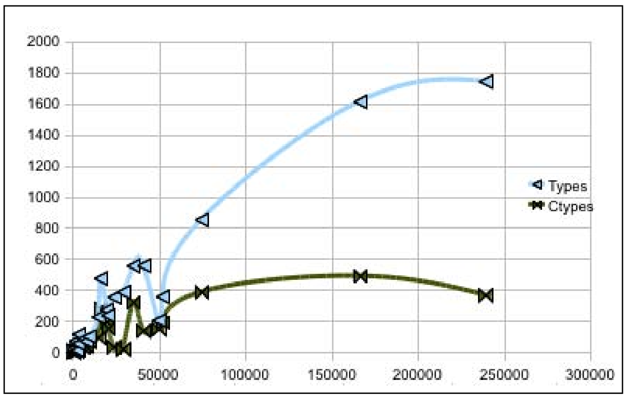

Figure 3: Types and $CTypes$ for some Applications.

of the current code, compliance with the expected architecture, and refactoring. In order to redesign an application with a precise architecture in mind which is not the one they observe in the code, they would need to use different tools conjointly: architecture compliance checker, refactoring tool, metrics based tools, etc. The JCE tool is a new tool to help the designer in checking the compliance of the new architecture implementation with the communication integrity property.

5.2 Experiments

We ran the tools on several examples of various sizes, coming from different repositories and illustrating different application domains. The examples can be found mainly on sourceforge (http://sourceforge.net/) (jabref, metrics, SweetHome3D, jasperreports, StringSearch, Checkstyle, Squirel, Commander4J). But we also collected some specific applications (MineSweeper, Javacalc, prefuse, AirportInternetAccess, TSAFE, and CoCoME examples), and we designed several of them (STSLib, JavaCompExt, NIM game, and simplification). We ran the tools on more than 30 examples ranging from simple examples of 100 LOC to real size applications of 160 KLOC, for a total of more than 500 KLOC. Figure 3 gives the number of types (Java classes and interfaces) and $CTypes$ as a function of the code size (LOC). The percentage of component types relative to the total number of types (%R) gives a partial evaluation of the CBSE quality relevant to our context. As a first remark, types respecting our set of CI rules exist in all these applications even if they are traditional OOP applications. We found a ratio of less than 10% in BECL and ECS projects from http://jakarta.apache.org. For some of them which were designed with components or strict programming rules in mind, the results are better (for instance, the TSAFE application, has a ratio of 67%, and PyrusNGS has 76%). However, for some others which claim to be CBSE applications, the results are poor (for instance, COCOME-RCOS, with a ratio of 31%). Commander4J and StringSearch are even more intriguing since their scores are very high (greater than 70%). Their authors reply negatively to the question: "Do you use any component principles or programming rules?" There are various reasons explaining these results: industrial component approaches have no hierarchical component, often composite implementations use packages and communications are implemented in many different ways. Java component models and implementations are numerous and the JCE prototype is dedicated to our specific component model. Thus it seems difficult to propose a rigorous classification of the component practices based on the above results. However, it raises several issues such as a more in-depth analysis of these results and the defining of a tool which can be configured with a set of CI rules.

Table 1: Results for JCE.

	PC	CC	DT	#W	%R
Nim Game					
OOP Version	2	0	2	4	50
CBSE Version	2	2	0	0	100
Simplification					
OOP Version	2	0	5	18	29
CBSE Version	4	1	2	0	72
MineSweeper					
OOP Version	3	0	7	3	34
CBSE Version	5	4	1	1	90
Javacalc					
OOP Version	1	0	11	5	8
CBSE Version	0	5	7	3	42
REGEXP					
OOP Version	6	1	9	8	44
CBSE Version	10	3	3	3	81
JavaCompExt					
OOP Version	34	5	35	86	53
CBSE Version	37	7	31	80	58
Metrics Plugin					
OOP Version	93	3	133	267	42
CBSE Version	112	8	109	257	52

We analyzed seven applications more precisely, and we will highlight the most important points below. In this case we use the information provided by JCE and refactor the application in a similar way as we did for MineSweeper in Section 5.1. That means, manual refactoring with the help of the tool to identify problems and without any specific architecture in mind. Let us consider Table 1, which represents several analyses performed with JCE on two versions of seven applications. In this table `PC` stands for primitive component, `CC` for composite component and `DT` for data types. In the middle columns, when we go from the OOP version to the CBSE version, several data types become new component types. The two last examples show more consequent remodularizations of non-trivial projects. The #W column gives the number of wrong profiles which is the most important indicator in these examples. We solve some of the rule violations, after which this indicator decreases. The ratio %R column clearly shows the evolution of the component refactoring process. We also note that removing wrong profiles was more difficult with applications like simplification and Javacalc. These applications parse data expressions and have a strong data nature. One might think that extracting tools could help in validating our JCE tool, however, they have very different objectives. We conducted a detailed comparison with SoMoX, see [32], and we report its numeric data in Table 2. We chose the SoMoX tool since it was the only well-supported component extractor tool which is freely available. One first difference is that JCE computes abstract component types, but this concept does not exist with SoMoX. With JCE the total number of data, composite, and primitive types is equal to the number of types in the application. This is not true with SoMoX since it suggests new clustering types and even discards some types. Consequently the measure %R, used above, is not valid for SoMoX. We can observe that the total component type numbers for SoMoX are nearly the same in every version of each application, except the data type numbers, which decrease. While for Metrics there is an increment of 14 component types using JCE

Table 2: Results for SoMoX.

	PC	CC	DT
Nim Game			
OOP Version	4	2	1
CBSE Version	4	2	0
Simplification			
OOP Version	7	1	4
CBSE Version	7	1	0
MineSweeper			
OOP Version	7	1	1
CBSE Version	7	1	0
Javacalc			
OOP Version	8	2	0
CBSE Version	8	1	0
REGEXP			
OOP Version	12	4	6
CBSE Version	11	5	5
JavaCompExt			
OOP Version	51	8	47
CBSE Version	51	8	40
Metrics Plugin			
OOP Version	124	19	109
CBSE Version	124	18	102

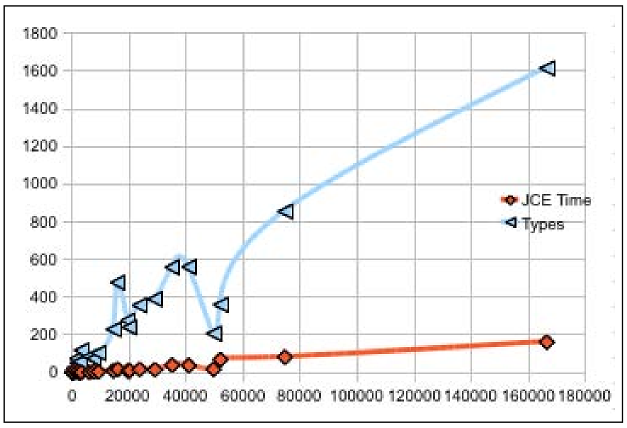

Figure 4: Time for JCE.

(from 96=93+3 to 120=112+8), SoMoX observes few changes in the number of component types (it decreases from 143=124+19 to 142=124+18). This confirms that both tools measure different component information. JCE is devoted to identifying existing component types, really implemented by the construction of the language and it uses a pure semantic approach to qualify component and data types. SoMoX, on the other hand, uses a more syntactic approach based on measures and heuristics to analyze an application and to suggest clusters which could be component types in the next restructuring step.

5.3 Time Performance Issues

The JCE approach needs to parse the source code, its computation time is linear with respect to the code size. The tool propagates data type information which is linear with respect to the number of types. We observe that the number of types for projects with more than 50 KLOC is bound to less than 1% of the number of LOCs ("Type" curve in Figure 4). Given that dynamic environment time measures are not stable in Eclipse, we computed the average of several measures. Figure 4 shows that the JCE time is quite linear (dot square "JCE Time" curve). But the JCE current implementation parses several times the source code and overuses `String` computations. Another point to note is that JDT has some resource management problems and applications under Eclipse are slower than standalone Java applications.

6. CONCLUSION

In this article we examined the problem of architecture erosion, which occurs when a software implementation deviates from its architecture. One way to mitigate the erosion of a system's architecture is to explicitly embed it in the source code. Re-engineering legacy code with software components seems a promising method to implement architectural concerns in the source code. For that designers need a tool to assess the quality of the component refactoring and to guide the re-engineering process. We might think that metrics based software are the right tools. However, the current state of the art is not sufficient and they are relevant only to measure abstract and global properties like reusability or maintainability. Component and architecture recovery is another possible way. However, these also falls short since their objective is not to qualify the existing components but rather to suggest new components by clustering parts of the existing applications. A last option is to use code analyzers or architecture checkers, but amongst the plethora of such tools none of them is really suitable to component programming with hierarchical components, subtyping and ensuring the communication integrity property. To fill this gap, we have provided a tool based on a general and abstract component model with first-class components, subtyping and composition. This component model has a straight implementation in Java. Based on this model we elaborate a set of rules to infer violations of the communication integrity property. These rules allow of separating data types from component types and warning the user about related problems. While experimenting on several open-source applications, we observed that types respecting our CI rules already exist in many applications. We started the process of evaluating the tool by refactoring several middle-size case studies. Our experiments showed that CI violations are sometimes easy to remove but often it is a tricky process. Our JCE prototype is specific to Java but the principles of our approach are reusable for other OOP languages and CBSE models.

In the future we plan to experiment with different set of rules, for example relaxing the rule for methods signature but strengthening the rule for constructor. Another track is to continue the tool evaluation and to compare the results of these different rule sets. We already did some manual experiments in re-engineering a few Java applications but we aim to process larger, industry-sized projects.

Thanks to: Syed Asad Ali Naqvi, Joost Noppen, Arang Rhie and Klaus Krogmann for their help.

7. REFERENCES

[1] FindBugs, 2009. http://findbugs.sourceforge.net/.

[2] PMD, 2009. http://pmd.sourceforge.net/.

[3] M. Abi-Antoun, J. Aldrich, and W. Coelho. A case study in re-engineering to enforce architectural control flow and data sharing. *Journal of Systems and Software*, 80(2):240–264, 2007.

[4] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor,

ECOOP '04 — Object-Oriented Programming European Conference, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, 2004. Springer-Verlag.

[5] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, pages 334–367. Springer Verlag, 2002.

[6] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 187–197. ACM Press, 2002.

[7] P. André, N. Anquetil, G. Ardourel, J.-C. Royer, P. Hnetynka, T. Poch, D. Petrascu, and V. Petrascu. Javacompext: Extracting architectural elements from java source code. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE 2009), tool demonstration*, pages 377–378, Lille, France, October 2009.

[8] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. K. n. Reverse engineering component models for quality predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering, European Projects Track*, 2010.

[9] J. Bosch, C. A. Szyperski, and W. Weck. Component-oriented programming. In *ECOOP Workshops*, pages 34–49, 2003.

[10] I. T. Bowman, M. W. Godfrey, and R. C. Holt. Extracting source models from java programs: Parse, disassemble, or profile? Unpublished paper available at http://plg.uwaterloo.ca/ migod/papers/1999/paste99.pdf

[11] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.

[12] S. Chardigny, A. Seriai, D. Tamzalit, and M. Oussalah. Quality-driven extraction of a component-based architecture from an object-oriented system. In *CSMR*, pages 269–273. IEEE, 2008.

[13] L. Chouambe, B. Klatt, and K. Krogmann. Reverse engineering software-models of component-based systems. In *CSMR*, pages 93–102. IEEE, 2008.

[14] I. Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[15] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *IEEE Transaction on Software Engineering*, Submitted for publishing:1–25, October 2010.

[16] M. C. da Silva Jr., P. A. de Castro Guerra, and C. M. F. Rubira. A java component model for evolving software systems. In *ASE*, pages 327–330. IEEE Computer Society, 2003.

[17] The Eclipse Foundation. *Java Development Tooling*, 2010. `http://www.eclipse.org/jdt/`.

[18] J.-M. Favre, J. Estublier, F. Duclos, R. Sanlaville, and J.-J. Auffret. Reverse engineering a large component-based software product. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 95, Washington, DC, USA, 2001. IEEE CS.

[19] J. Gargiulo and S. Mancoridis. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In *SEKE: Software Engineering and Knowledge Engineering*, pages 244–251, 2001.

[20] D. Garlan. Software architecture: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM.

[21] M. Goulão and F. B. Abreu. *Software Quality Measurement: Concepts and Approaches*, chapter An overview of metrics-based approaches to support software components reusability assessment". Information Technology. ICFAI Books (India), 2007.

[22] L. Hochstein and M. Lindvall. Combating architectural degeneration: a survey. *Information & Software Technology*, 47(10):643–656, 2005.

[23] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE CS Press.

[24] J. Knopel and M. Lindvall. Software architecture visualization and evaluation. SAVE web site: http://www.fc-md.umd.edu/save/about.aspx, 2009.

[25] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Ph.d. thesis, Institute for Computer Science, University of Stuttgart, Stuttgart, 2000.

[26] R. Koschke, G. Canfora, and J. Czeranski. Revisiting the ΔIC approach to component recovery. *Sci. Comput. Program.*, 60(2):171–188, 2006.

[27] D. C. Luckham, J. L. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.

[28] S. Mahmood, R. Lai, Y.-S. Kim, J. H. Kim, S. C. Park, and H. S. Oh. A survey of component based system quality assurance and assessment. *Information & Software Technology*, 47(10):693–707, 2005.

[29] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[30] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[31] J.-C. Royer. Type Checking Object-Oriented Programs: Core of the Problem and Some Solutions. *Journal of Object-Oriented Programming*, 11(6):58–66, 1998. ISSN 0896-8438.

[32] J.-C. Royer. The JCE Checker. http://www.emn.fr/z-info/jroyer/JCE/index.html, 2010.

[33] B. R. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[34] H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer programming*, 56(1-2):99–116, 2005.