Replace this file with prentcsmacro.sty for your meeting, or with entcsmacro.sty for your meeting. Both can be found at the ENTCS Macro Home Page.

Towards Reverse Engineering Components from Java Code *

Pascal André, Gilles Ardourel[1]

LINA - University of Nantes, France

Petr Hnětynka, Tomáš Poch, Ondřej Šerý[2]

DSRG - Charles University, Prague, Czech Republic

Vladiela Petraşcu, Dragoş Petraşcu[3]

LCI - Babeş-Bolyai University, Cluj-Napoca, Romania

Jean-Claude Royer[4]

ASCOLA/LINA/INRIA - Ecole des Mines de Nantes, France

Abstract

Component-based software engineering has become a common approach in many areas of software development. With respect to other approaches, it offers fast development by composition which, thanks to explicit provisions and requirements of components, can be easily verified. However, there is a gap between component systems focusing on abstract models and component systems focusing on implementation. In this paper, we present an approach to overcome this gap using reverse engineering. The approach has been designed during the Econet international project and allows extracting architectural and behavioural information from plain Java applications.

Keywords: Components, reverse engineering, architecture extraction, behaviour extraction.

1 Introduction

Component-based software engineering is still a challenging topic in both industrial and academic research. Most of the academic approaches focus on abstract models

* This research was partially supported by an Egide project (Nr 16293RG).

[1] Email: {pascal.andre,gilles.ardourel}@univ-nantes.fr

[2] Email: {hnetynka,poch,sery}@dsrg.mff.cuni.cz

[3] Email: vladi@cs.ubbcluj.ro, petrascu@cs.ubbcluj.ro

[4] Email: Jean-Claude.Royer@emn.fr

©2008 Published by Elsevier Science B. V.

and specification of components (sometimes close to architectural description languages) with verifiable properties such as safety and liveness; some of them deal with refinement and code generation, but many times these systems support only specification of components and no implementation. On the contrary, the industrial systems such as CCM, EJB, or OSGI are implementation-oriented. They define flat components only, without hierarchical structures. They have strong and mature runtime infrastructure, but provide no support for verification of any properties. Therefore, reuse of components in different contexts may be difficult, as correctness of their usage cannot be assured. Applications specified in the formerly mentioned systems are many times implemented using the latter ones (or even using no components but objects only) and also vice-versa. But such a situation directly leads to a gap between component specifications and component implementations, as uncontrolled changes can happen on each of these two levels. Sometimes this gap is called architecture erosion [7].

A major issue is to fill this gap and tie the specification and implementation together. A way to address this gap is to define model transformation techniques in order to generate code from the component specifications. This can be qualified as the *engineering* way and it is similar to MDA and MDE approaches. It is quite complex since we should, in theory, prove the correctness of the translation and also because there are various target frameworks and languages. Another way is to focus on program code analysis in order to extract component specification from the component actual code. This can be qualified as the *reverse engineering* way. This way is even more complex than the previous one due to many reasons like there exists (1) no widely accepted common component model for specifications, (2) no direct structural information in the code, in order to extract an application architecture, (3) no common approach for abstracting behavioural description of components. Both the engineering and reverse engineering ways remain open research issues. The latter approach is even more important when dealing with existing legacy applications, which should be integrated with new ones.

The goal of the paper is to contribute to the reverse engineering way by developing techniques for extraction of structural and behavioural descriptions from code and for the verification of these descriptions back against the new/modified code. The presented work has been done in the scope of the Econet international project.

This paper reports the current state of the project. The target contributions are: (1) an open architecture to tackle the problem of re-engineering Java programs to software components, (2) a common component metamodel that supports a general component API, (3) processes for reverse engineering of structure and behaviour, and (4) a prototype created as an Eclipse plugin, which integrates all the mentioned pieces together.

The paper is organized as follows. Section 2 overviews the Econet project and details the architecture of the whole approach. The architecture exposes a common part related to models (Section 3) and two abstraction processes (Section 4). An implementation of the integrated prototype and also performed experiments (done using the CoCoME benchmark [24]) are in Section 5. A comparison of our approach with related work is described in Section 6, and we conclude the paper in Section 7.

2 Project Overview

The goal of the project is to establish a link between component codes (the source model) and component specifications (the target abstract model). The advantages of existence of the abstract model are mainly to provide a more accessible documentation, to find convenient tools for verifying various properties of the component systems such as safety and liveness, to improve the architectural restructuring and evolution. Instead of studying only the structural features of the system, we also work on *behavioural* abstractions. Behaviour is related to the dynamic and functional features of the components and services. The mechanisms used for component specifications are grounded on different formalisms: design by contract, algebraic specifications, state machines, regular expressions and so on. Each of above mentioned formalisms offers a set of advantages and has some drawbacks. Design by contract, a declarative specification only, supports an "incomplete" behaviour specification. Algebraic specifications generally have sound semantics but are, in most cases, difficult to understand by people and not all kind of components can be specified. The *labeled transitions systems* (LTS) and the *regular expressions* (RE) formalisms are suited for dynamic descriptions and have formal semantics.

Complexity of the abstraction process depends on a "distance" between the source concepts and the target concepts. To manage complexity, we set the following starting assumptions:

(i) The source model is limited to Java code.

(ii) The target models are abstract component models such as Sofa [11], Kmelia [4], Kadl [22], Fractal [10]. These models consider both structural and behavioural features and provide tool support for verification.

Figure 1 shows the used architecture of the presented approach, which is structured as follows:

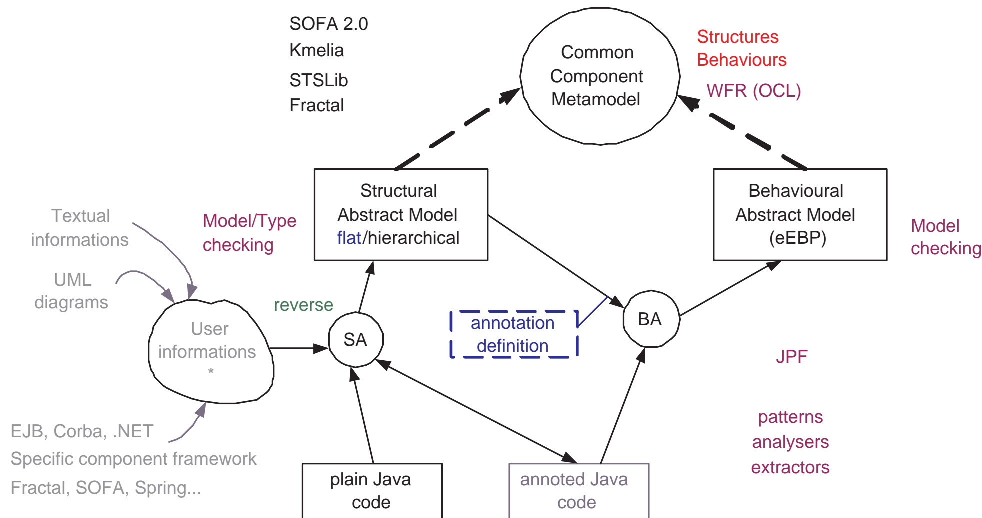

Fig. 1. Econet Project Architecture

- The metamodel part provides the foundation API (Application Programming Interface) for the component model processing. It addresses both the problem of

handling several component models in a generic way and the problem of linking models and codes (traceability).

- Process SA is an iterative process that extracts and infers architectural and typing features from a plain or annotated Java code.
- Process BA extracts a dynamic behaviour specification for the components identified during the process SA. The idea is to make the reverse engineering as general as possible but only variants of *LTS* (*eLTS* of Kmelia [4], *STS* of STSLib [15]) and the *regular expressions* (EBP of Sofa [23]) are currently targetted.

Each of these parts is described in detail in the following sections.

3 Model Management

By managing models we mean: providing a generic component model API, handling source programs, and linking both levels. We define a metamodel to describe models (section 3.2) and annotations to implement a kind of traceability (section 3.1).

3.1 Annotations

In order to let the source-based code analysers benefit from the cumulative information gathered from other tools without having to query the model, it was decided to store model information in the source code. Since JDK 1.5, the Java language has a support for annotations [6], that store information which can be processed either at compilation time or at runtime. Similarly to Javadoc elements, the annotations are meta-tags that you can add to your code by applying them to package declarations, type declarations, constructors, methods, etc. A library of annotations has been developed to establish the link between Java structural features and model elements. For example, the Java definition for the component annotation is:

```
/**
 * One or more Java classes can be assigned to a single component.
 * Such an assignment is specified by this annotation.
 */
@Target(ElementType.TYPE)
public @interface InComponent {
  /**
   * @return the array of sources for this annotation
   */
  String[] annotationSrc();
   /**
   * @return the array (one entry per annotation source) containing component
   *    Names which the annotated class is assigned to. If a single
   *    source declares the class to participate in several components,
   *    its entry should be a comma-separated list of component names
   */
  String[] componentName();
}
```

Since a single Java element cannot be annotated more than once by an annotation type, every annotation we designed has arrays as fields, in order to store the information extracted by the different tools or users. Filtering or combining that information can then be done in subsequent steps of the process.

3.2 Metamodel

Within the envisioned reverse engineering process, there is need for a metamodel that defines the required component modelling concepts and their interrelationships. Such a metamodel has to satisfy at least three basic constraints. The first constraint concerns genericity; the metamodel should abstract over different concrete component models (Fractal, Sofa, Kmelia, and Kadl), by gathering a common set of concepts and postponing specific concepts to concrete model mappings. Second, it has to include means for managing the tight connections between model elements and the corresponding (Java) features implementing them. Third, the metamodel must be rigorously specified, including all the necessary *Well Formedness Rules* (WFRs) and useful query operations. Moreover, appropriate tool support for metamodel testing and repository code generation should be ensured.

The above mentioned requirements have led to the definition of a *Common Component MetaModel* (CCMM), whose overall architecture is illustrated on Figure 2. While designing it, we have used *abstraction* in order to solve the genericity problem; common aspects of the considered component models have been extracted as shared concepts, while particular ones have been included by means of specialization relationships. Moreover, we have tried to reach a compromise between *minimality* (limiting the number of concepts by avoiding unnecessary ones) and *extensibility* (staying as general as possible, in order to be able to include other concepts later).

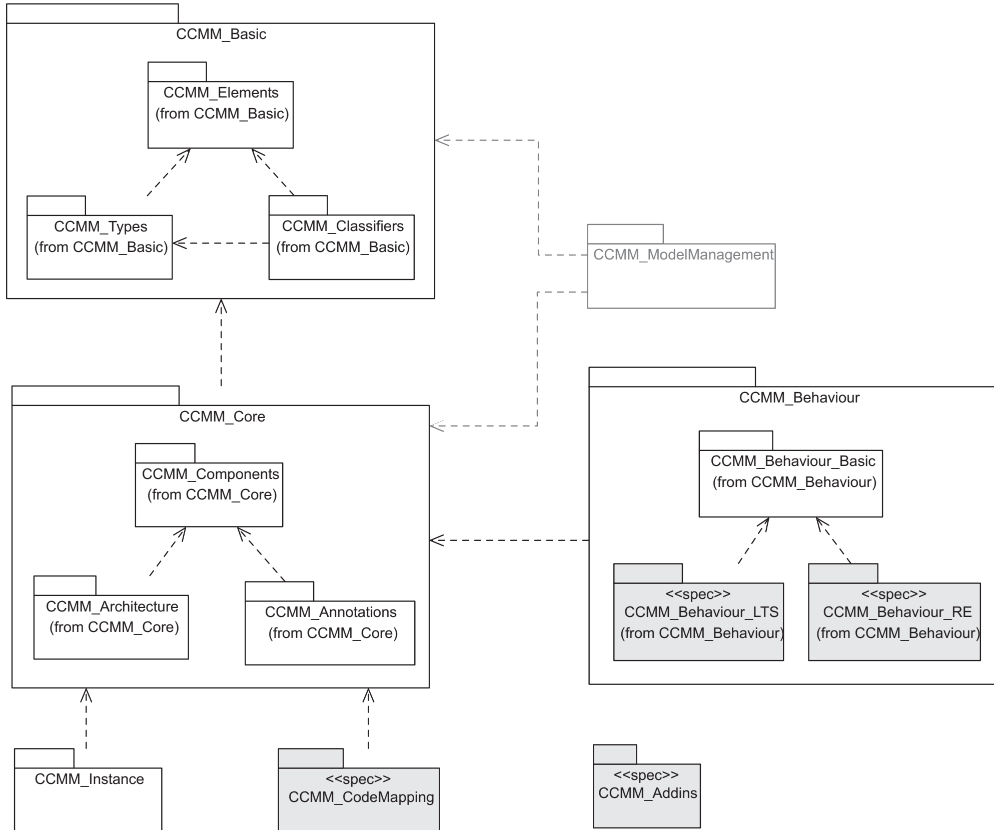

Fig. 2. Common Component MetaModel (CCMM) v1.1

Following the *separation of concerns* principle, the metamodel is organized

on packages according to the specification document [5]. CCMM_Basic gathers all needed elementary modeling concepts; it is inspired by Ecore and UML 2.0 and includes metaclasses that define (model) elements, types, and classifiers. Based on CCMM_Basic, CCMM_Core is the fundamental metamodel package. By means of its three subpackages, it defines the structure of components and component architectures, together with some annotations-related aspects. Within it, the links to the Java code are managed by means of special metaclass attributes, which have been prefixed by s_ (from source) to distinguish them from regular attributes. While structural aspects of components are handled within CCMM_Core, the description of their associated dynamic behaviour is isolated inside the CCMM_Behaviour package. The specific formalisms are defined within CCMM_Behaviour_RE and CCMM_Behaviour_LTS, respectively. CCMM_Instance describes component instances' management, while CCMM_ModelManagement details the structure of a component repository. The CCMM_Addins and CCMM_CodeMapping packages have been designed having in mind a possible evolution of the metamodel. The former includes concepts related to properties and constraints/assertions. The latter offers an alternative to the solution adopted within CCMM_Core (special attributes) in order to handle the model-code mapping. This is achieved by introducing metaclasses corresponding to Java features and representing the mapping by means of associations. The least solution would be to implement a full code metamodel (e.g. JMI) but it is too heavy and subject to standard compliance and evolution. All <<spec>>-stereotyped packages have been ignored while generating the first version of the metamodel API.

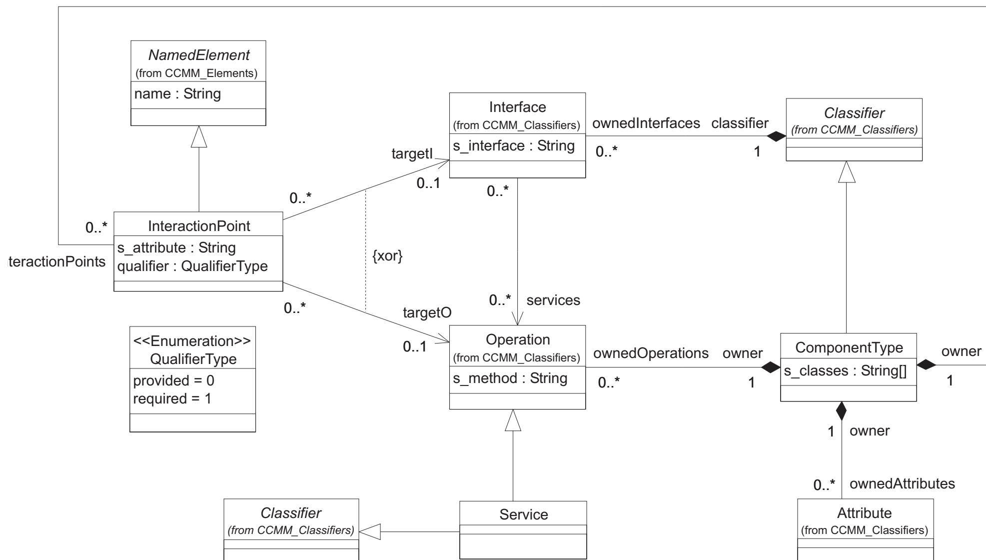

Fig. 3. CCMM_Components package

The class diagram in Figure 3 details the CCMM_Components package. A ComponentType is a black-box entity, defined as a specialization of Classifier. It may own Attributes and Operations, and the latter ones may be grouped to form Interfaces. The s_classes attribute holds a model-code link, by storing the names of all Java classes that implement a certain ComponentType. Any ComponentType interacts with the environment through a number of InteractionPoints. Each of these expresses either a provision or a requirement, and may target either

an `Interface` or an `Operation`. The type of target depends on the concrete component model considered; it is an interface in case of SOFA, and an operation in case of Kmelia. Nevertheless, all `interactionPoints` owned by a certain `ComponentType` should have the same target type, fact expressed by means of the `consistentInteractionPoints` OCL invariant below.

```
context ComponentType inv consistentInteractionPoints:
 -- if at least one interaction point targets an interface,
 self.interactionPoints->exists(ip:InteractionPoint|ip.targetsInterface()) implies
 -- all interaction points should target interfaces
self.interactionPoints->select(ip:InteractionPoint|ip.targetsOperation())->size()=0
```

For sake of readability, the above WFR uses two query operations defined for `InteractionPoint`, namely `targetsInterface` and `targetsOperation`. We give the former's body in the following, the OCL expression for the latter being similar.

```
context InteractionPoint::targetsInterface():Boolean
 body: self.targetO.oclIsUndefined()
```

4 Behaviour and Architecture Abstraction from Code

The abstraction is split in two orthogonal processes: abstraction of structures (types, services, components, and architectures) and abstraction of dynamic behaviours (flows and communications).

4.1 Architecture Extraction

From a plain Java code and user interaction, the architecture extractor (called process SA) produces an annotated Java code and the corresponding component model. Many parameters have an influence on this process (do we have a component model or not, do we have existing annotations or not, is the implementation the result of some component translation, do we have several sources for the annotations, do we have user informations or not...). In order to fix a general but customisable schema we set some restrictions:

- Input: (1) the annotations are those related to CCMM (in the future we may consider roundtrip engineering and accept specific annotations) (2) UML models are not accepted as direct inputs (only user information).
- Output: (3) only flat component behaviours are targeted, (4) process SA is not directly responsible of the consistency between a model and the corresponding Java annotated code, (5) the conformance of the produced component model is checked at the metamodel level.

Process SA is designed as an iterative process over a toolbox architecture (Figure 4) where one iteration applies one transformation. The idea is to combine primitive transformations in a customised human driven process. The input *jac* may be a plain or annotated source code. The input *cm* (a CCMM instance) may be empty or disconnected from any *jac*. Examples of primitive transformations are: annotate a Java program from user information, build a component model from a plain or an annotated Java source, analyse a distributed program to detect components (deployment), analyse dependencies using graph tools and extract clusters.

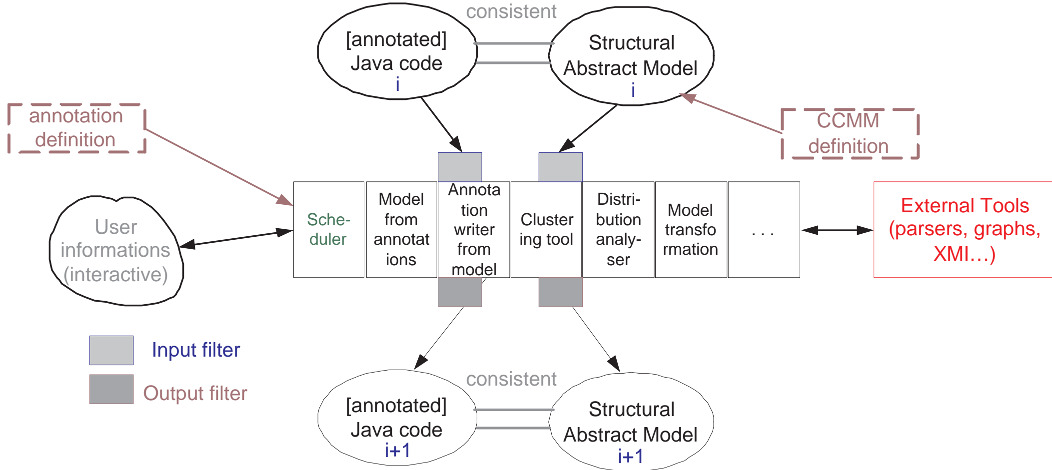

Fig. 4. An architectural view of the process SA

As an example, let us consider the case of finding abstractions from a plain Java code without annotations. At first sight it is not easy because of the distance between the object and component concepts. Let us define some simple tasks allowing to build a first and minimal architecture which can be refined later in another iteration.

- Identify or extract the boundaries for primitive and composite components. The resulting architecture is a tree structure where the primitive components are the leaves and the composites are the nodes.
- Identify or extract the communications and classify services as required or provided. In such applications, communications are usually reduced to message send.
- Establish the interfaces of each component and classify. Interfaces are sets of methods.

Hence, we propose extraction rules and some controls which can help in detecting potential problems or non-expected situations. Due to lack of space, this section describes only the main rules we have defined.

The first rule we consider is that a component cannot be passed as an effective parameter of a method but it is possible for constructors. The main reason is: we expect to respect encapsulation or communication integrity [1]. This is an important property since it implies the existence of some communication channels which can be identified with a static code analysis. Thus we analyse a Java project and extract the main types defined in it, forgetting external data types or primitives ones. These types are called the *types of interest*. Then a type of interest which is used as parameter type in a method is flagged as a data type. Furthermore, the subclasses of this type are also flagged as data types, since their instances could be used as effective parameters using subtyping rules.

The second rule is to extract a possible composite structure from the analysis of the fields. Ideally, the composite structure (or part-of relationship) should be a tree or a forest. The analysis simply browses the candidate types and collects recursively their structure. However, the process has to collect the inherited structure from the super classes, that means to collect private, public, protected and default-package

fields in the inheritance up to `Object`. We collect all kinds of fields since, even if a field is not accessible in a class, a public accessor can be defined to read or write it.

The third rule is to extract communications from the code of methods. We consider that there is a communication from type `E` to type `R` with message `mesg`, if and only if `mesg` occurs in the implementation of `E` and its local sender type `R` provides the `mesg` service. Here `mesg` is a signature in the Java sense.

The fourth rule is twofolds since it computes the required and provided services of each type. For the required services: They come directly from the previous analysis of communications. For the provided services: an analysis of each type is done to extract the public or default package methods of the type.

The above rules apply under some hypotheses on the Java code. The program is a source code (binary Java code could be analysed in more or less the same way, either directly or after decompiling). We consider that the code to analyse is contained in a unique project (an Eclipse project) which can import predefined projects or libraries. We assume that in a Java file, what JDT [18] called a compilation unit, there is only one main type which can be a component type or a data type. This constraint is simple to relax, but we don't want to analyse code which is too badly structured. We don't consider generic types, this is a main future extension of this work. To relax this restriction is not easy since Java 1.5 introduces a sophisticated type system and we have to distinguish generic definition, instantiation of them, their use in fields, inheritance and methods. A component type in Java could be an interface, a concrete or an abstract class. However, a component type must be instantiated and this is only possible with concrete classes. We also do not analyse static methods since, until now, we do not get a precise feeling of their use regarding a component approach. Nevertheless, the component extracting task remains a challenge.

4.2 Behaviour Extraction

As an input, behaviour extraction (process BA) takes (i) the source codes, (ii) the architectural information, and optionally (iii) user hints for the extraction. The architectural information can be either in a form of source code annotations or retrieved from the CCMM repository. If provided, user hints are specified in the form of source code annotations. The result of behaviour extraction is a high-level specification of component's behaviour. Targets of the extraction include family of formalisms based on behaviour protocols (BP, EBP, and TBP) as well as formalisms based on LTS, e.g. eLTS and STS.

Since the expressive power of the considered target formalisms is smaller than the power of implementation languages, obviously, the complete information cannot be preserved. However, with user's assistance and a clever use of over-specification, reasonable results can be achieved. The envisioned extraction process is depicted on Figure 5. In short, the automated extraction process is applied on the annotated source codes. For simple enough code, the extraction succeeds and provides the desired behaviour specification. However, in the case of too complex code, the extraction may fail. Should this happen, user may manually refine the annotations hints and/or the implementation and retry.

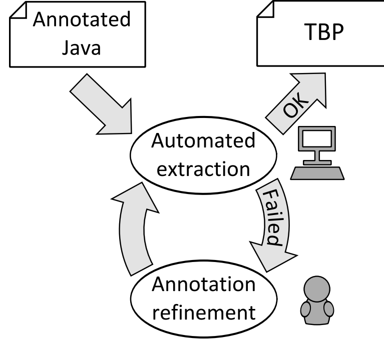

Fig. 5. Iterative behaviour extraction

In well-designed component systems, complex computations are often encapsulated inside primitive components. For such systems, the corresponding behaviour specification of components is likely to exist and be discovered in only a few iterations. On the other hand, successive failures of the automatic extraction may help identify that the complexity is inappropriately exposed throughout the architecture.

The extraction process is implemented by the jAbstractor tool. From a high-level view, the jAbstractor tool works in three steps. First, the Java sources are parsed. Second, a number of transformations is applied to the parse tree. The transformations use the architectural information. Goal of the transformations is to simplify the parse tree enough to be directly transformable to the target formalism while keeping as much information as possible. As the third step, the simplified parse trees are transformed to the target formalism.

Since users may have different requirements on the way the architectural information is passed to the tool, and on the target formalism used at the final step we have focused on the modularity of the whole tool. The result is that there is a number of extension points defined by particular interfaces that can be implemented in different ways to fulfil different goals of particular users. There is one interface providing an architectural information to the tool. Currently, the available implementation is able to extract the architectural information from source code annotations (see Section 3.1). Another interface encapsulates the final transformation. It takes a simplified Java source code and produces a high level specification. Currently, there are two implementations. One produces labelled transition systems and the other produces a behaviour protocol specification [23]. An important point is that the complex transformations are applied on the Java parse tree, so that those do not have to be implemented for each output formalism separately.

Figure 6 depicts the overall architecture of the jAbstractor tool. First, the Java parse tree is produced by the Recoder tool. Recoder [26] is a parser of Java sources and provides an in-memory representation of the Java parse tree, which is then used by the jAbstractor modules. Recoder also provides information about declaration references within the tree and a support for writing user-defined transformations over the tree. Usage of third-party parser eased the development a lot. Then, there are some transformations which make the parse tree simpler. Currently, the control flow is inspected and only those parts that are related to the observable external behaviour of the component are kept. More precisely, there is a notion of *important parse tree nodes* which actually influence the component external behaviour. The

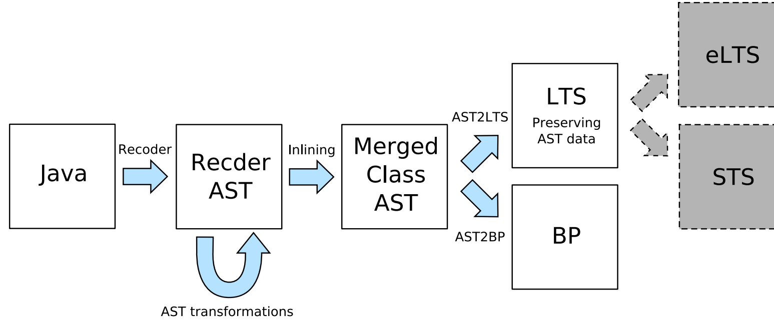

Fig. 6. behaviour extraction work flow

important nodes are method declarations on provided interfaces, method calls on required interfaces, accesses to important variables and control flow statements that connect those statements together. Another transformation is method inlining to create a new class which directly corresponds to the primitive component. There is a single method for each method on a provided interface. The method bodies contains bodies of methods invoked on objects belonging to the same component, which is achieved by method inlining. Also, member variables of other classes must be merged to the result class. Finally, since the merged class corresponds to the primitive component, the final transformation to the target specification language is supposed only to map Java language constructs to the constructs of the target language. Here, overspecification may be employed, to overcome expressiveness gap. If the overspecification is not enough to overcome the gap, the whole transformation fails and suggests to the user to refine annotations.

5 Integration, Implementation and Experimentations

The current prototype is implemented as a set of libraries and programs that can be called either in command line or through a set of Eclipse plugins. The main plugin is the process plugin, which allows the user to select a Java project, then launch and connect the different tools.

The process is started by the `ComponentFinder` (a JDT [18] plugin) which uses rules described in page 8 to annotate Java elements with model information. That information is not stored as Java annotations yet, for time reasons rather than complexity. In our experiments (see below) the annotations were added by hand. The next tool called is the `AnnotationToModel` extractor. The `AnnotationToModel` extractor is parameterised by an `AnnotationSelector` module which determines the annotations to be taken into account for generating the model. The fusion between annotations is customizable by defining subclasses of `AnnotationSelector`. The prototype currently uses a naive implementation which should be replaced by something more elaborate like a user-defined priority order, for instance.

The `AnnotationToModel` extractor uses the CCMM API plugin (implemented as an Ecore metamodel, in order to be able to benefit from the strong EMF tool support) to generate the model, then it checks it against the OCL rules of the CCMM metamodel and stores it to XML. Violations to the metamodel rules are stored, but they are not currently automatically used because they might indicate errors in the annotations or in the selection of annotations. A more complex process

will have to be used to obtain meaningful feedback if the sources are multiple and the `AnnotationSelector` elaborate.

Components that comply with the CCMM metamodel rules are then given as parameters to the jAbstractor tool which generates a behavioural specification from its constituent Java classes. The specification is then added to the component in the model.

Experimentations cover a large-scale set of tests including feasibility, software testing, pattern discovery, API testing, etc. The experimentation benchmark is the CoCoME case study [25], which includes a Java application and a documentation with UML component diagrams. The extraction part of the `ComponentFinder` has been tested using the full CoCoME benchmark. Other experiments worked on subsets only.

A first subset (including only three components) was the basis for the annotation management experimentation. Annotations were put manually in the code in order to generate a model from scratch using the `AnnotationToModel` extractor. A trivial code generator was also implemented an tested in this case.

A second subset included the `CashDesk` composite component. The CCMM tests and validation (covering all specified WFRs and query operations) have been performed on that subset. It also served as an experimentation field (1) to study the link between the component level (UML here) and the implementation level (how close is the implementation to the component model), (2) to investigate the discovery of annotations from UML descriptions and Java programs (how can we use manually or systematically UML information to find the annotations, what to look for in the Java programs). This experimentation [5] showed how far were the abstract and the implementation models, but also discovered patterns (unfortunately with exceptions) that were *implicitely* used in the engineering process.

6 Related Work

Reverse engineering component applications is quite an unexplored topic, while reverse engineering object oriented applications is quite rich (UML abstraction, pattern abstraction, model refactoring, etc) [12,21]. We thought we could proceed from UML models instead of Java code, but actually, the class model does not provide rich structural abstractions and the dynamic aspects are unfortunately poorly captured. In the same way, we could use the UML 2.0 component model as the reference CCMM but it is really too complex, the WFRs do not apply, there are many missing WFRs and the semantics is informal. Reverse engineering techniques such as clustering, slicing and trace exploration [20,8,17] are also useful for specific computations. In CBSE reverse-engineering, the concepts of component and architecture varies from one approach to another. For example, in [19], design components are high level concepts close to design patterns, but they are not abstract components. In [13], the authors claim that time has come to investigate this field. The difference with our approach is that they start from CB legacy code (CCM, EJB, COM). A simple component metamodel and some structural rules are

[5] `http://www.lina.sciences.univ-nantes.fr/coloss/projects/econet/test1.pdf`

provided in [14]. We go one step further with more expressive component models including behavioural features. The work presented in [30] is relevant, since it provides details for the abstraction (clustering algorithms) of Java programs. The target model is JavaBeans, while we target more generic and hierarchical models. Only the structure is abstracted, while, in addition, we consider behaviours to cope with the communication integrity property. Our analysis of the composite structure is comparable to their structural clustering algorithm but we simplify by assuming that component structure is built from the class fields.

Component recovery and *architecture recovery* are the main issues for abstracting structures. Here, we summarize some previous work which has been done in the area of extracting architectural information from Java code. In their technical paper [9], the authors study the various ways to extract some model information from Java code. In fact, it can be done with three different approaches: parsing the source code, disassembling the byte code, or profiling the application execution. They found that these three techniques have complementary advantages. Parsing the source code, using the classic grammar-ware technology, is the most complex to implement and it can lead to detailed models. Disassembling Java byte code gives similar results to parsing but, since the language is simpler, it is technically less complex. Profiling consists in getting some feedback from application execution, it strongly depends on the precise context of execution, but it is easy to do and provide accurate information about polymorphic call, dynamic types of objects, and information related to the use of the reflective Java API. The conclusion from [9] is that: if static analysis is sufficient thus disassembling is probably the best choice. However, if we want to exploit some comments and code annotations, it is only possible with source code. These comments and annotations may be really important to help the extraction of the structure and architecture for components. If we need really accurate information, source code analysis is better, since compilation may omit some relations which could be important from a more abstract point of view. The problem is still open. For instance, [16] considers that runtime analysis or profiling is needed, since types and objects may be dynamically created.

So far, the work related to automatic architecture extraction has been discussed. Regarding the behaviour extraction, the means are similar. One can either monitor the behaviour of a running software or analyse the software statically. In [27], the authors use instrumentation of Java bytecode to obtain execution traces from a running program. Then, the traces can be analysed by various *observers* capable of detecting deadlocks, race conditions, and user-specified LTL properties. In [3] and [31], the authors use those traces to infer automaton-based high level specifications. Moreover, in [31], static analysis is also considered and both approaches are compared. In general, the techniques based on monitoring are useful for inferring a “common” behaviour. However, since they are not exhaustive, there is no guarantee that the inferred specification covers all possible behaviours. The techniques based on static analysis are presented in [29] and [2]. In these papers, static analysis is used to derive protocols describing the correct use of components. When compared, the techniques based on static analysis employ abstraction possibly resulting in a specification that represents a superset of the actual behaviours of the system. If such a specification is used for reasoning and automated detection of errors, false

positive may occur due to the over-approximation. This is in line with our proposed method.

Another way to reduce the distance model-code in the domain of CBSE is simply to merge the levels. This way is illustrated by architectural programming languages such as JAVA/A [7], ArchJava [1] and ComponentJ [28]. These approaches are not really satisfactory in our case. Among the limitations, architectures in those languages are really more concrete than architectural description languages (ADLs). The separation between model and code can be difficult. ArchJava and ComponentJ lack of abstraction (no behaviour specification). JAVA/A provides a specific translation framework using the code generation engine HUGO/RT, it does not support plain Java.

7 Conclusion and Future Work

In this paper, we have presented an approach allowing to extract component abstractions from plain and/or annotated Java source code. The abstractions are extracted in two processes — one for the structural information (business types, components, architectures) and one for the behaviour (dynamics and computations). A common component metamodel has been designed to make this approach generic enough over the "abstract" component models. We have implemented the presented approach as a set of Eclipse plugins and tested it on the CoCoME case study.

Future work concerns at first the finalisation of the implementation into a full chain of tools. Further research is also necessary in both the structural and behavioural extraction processes (in order to cover complex patterns, communication mechanisms, etc.).

References

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[2] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.

[3] Glenn Ammons, Rastislav Bodik, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1):4–16, 2002.

[4] Pascal André, Gilles Ardourel, and Christian Attiogbé. Defining Component Protocols with Service Composition: Illustration withe Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, volume 4829 of *LNCS*. Springer, 2007.

[5] Pascal André and Vladiela Petraşcu. ECONET Project - CCMM Specification v. 1.1 , June 2008.

[6] Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[7] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.*, 160:75–96, 2006.

[8] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE CS Press.

[9] Ivan T. Bowman, Michael W. Godfrey, and Richard C. Holt. Extracting source models from java programs: Parse, disassemble, or profile? http://plg.uwaterloo.ca/ itbowman/pub/paste99.pdf, 1999.

[10] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.

[11] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48. IEEE CS, 2006.

[12] Serge Demeyer, Yann-Gaël Guéhéneuc, Anne Keller, Christian F. J. Lange, Kim Mens, Adrian Kuhn, and Martin Kuhlemann. Object-oriented reengineering. In Michael Cebulla, editor, *ECOOP Workshops*, volume 4906 of *LNCS*, pages 142–153. Springer, 2007.

[13] J. M. Favre, H. Cervantes, R. Sanlaville, F. Duclos, and J. Estublier. Issues in reengineering the architecture of component-based software. In *SWARM forum (Software Architecture Recovery and Modeling) at the Working Conference on Reverse Engineering (WCRE'2001)*, Stuttgart, Germany, 2001.

[14] Jean-Marie Favre, Jacky Estublier, Frédéric Duclos, Remy Sanlaville, and Jean-Jacques 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.

[15] Fabricio Fernandes and Jean-Claude Royer. The stslib project: Towards a formal component model based on sts. In Markus Lumpe and Eric Madelaine, editors, *Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS 2007)*, volume 215, pages 131–149, June 2008.

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

[17] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[18] *Java Development Tooling*, 2008. `http://www.eclipse.org/jdt/`.

[19] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick 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.

[20] R. Koschke, J.-F. Girard, and M. Würthner. An intermediate representation for reverse engineering analyses. In *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, page 241, Washington, DC, USA, 1998. IEEE CS.

[21] Matthias Merdes and Dirk Dorsch. Experiences with the development of a reverse engineering tool for uml sequence diagrams: a case study in modern java development. In *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 125–134, New York, NY, USA, 2006. ACM Press.

[22] Sebastian Pavel, Jacques Noy, Pascal Poizat, and Jean-Claude Royer. A java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*, pages 115–125. Springer-Verlag, 2005.

[23] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[24] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, 2008.

[25] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, Heidelberg, 2008.

[26] Recoder. http://recoder.sourceforge.net/.

[27] Grigore Rosu Riacs and Klaus Havelund. Monitoring java programs with java pathexplorer. In *In Proceedings of Runtime Verification RV01*, pages 97–114. Elsevier, 2001.

[28] J. C. Seco and L. Caires. A basic model of typed components. In Elisa Bertino, editor, *ECOOP 2000 - Object-Oriented Programming, 14th European Conference.*, number 1850 in LNCS, pages 108–128. Springer-Verlag, 06 2000.

[29] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static Specification Mining Using Automata-Based Abstractions. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 174–184, New York, NY, USA, 2007. ACM.

[30] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Sci. Comput. Program.*, 56(1-2):99–116, 2005.

[31] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes*, 27(4):218–228, 2002.