Automated Extraction of Rich Software Models from Limited System Information

Michael Langhammer[†][*] Arman Shahbazian[§][*] Nenad Medvidovic[§] Ralf H. Reussner[†]

[†]Karlsruhe Institute of Technology, Karlsruhe, Germany
{langhammer|reussner}@kit.edu

[§]Computer Science Department
University of Southern California, Los Angeles, CA, USA
{armansha|neno}@usc.edu

***Abstract*—Reverse engineering a software system is challenged by the typically very limited information available about existing systems. Useful reverse engineering tasks include recovering a system's architectural, behavioral, and usage models, which can then be leveraged to answer important questions about a system. For example, using such models to analyze and predict a system's non-functional properties would help to efficiently assess the system's current state, planned adaptations, scalability issues, etc. Existing approaches typically only extract a system's static architecture, omitting the dynamic information that is needed for such analyses. The contribution of this paper is an automated technique that extracts a system's static architecture, behavior, and usage models from very limited, but readily available information: source code and test cases. These models can then be fed into known performance, reliability, and cost prediction techniques. We evaluated our approach for accuracy against systems with already established usage models, and observed that our approach finds the correct, but more detailed usage models. We also analyzed 14 open source software systems spanning over 2 million lines of code to evaluate the scalability of our approach.**

I. Introduction and Motivation

Reverse engineering a software system is an abstruse task, especially when limited information is available about the system. More often than not, models of a system's architecture or behavior are either nonexistent or antiquated to a degree that are no longer useful [9]. Therefore, the only up-to date and maintained sources of information are the system's code and test cases. This means that very little can typically be established about a system without actually running it on its target environment. In these situations, one class of questions that cannot be easily answered deal with the non-functional properties of the system. To understand these non-functional properties, engineers can monitor the running system in different scenarios and observe how the system is being used and how the system performs. However, scalable, predictive models that can be used for analytical purposes are difficult to construct. We aim to alleviate this problem, by bridging the gap between a system's source code and the models required for the analysis of non-functional properties of a system.

This is especially helpful while working with free and open source software systems. These systems are usually picked off the shelf and all or parts of them get integrated to current solutions which are being used, and sometimes they get customized to meet specific requirements. In such cases it is very important to be able to predict the behavior, such as the performance, of such systems before deploying them on actual hardware. Reason being quality deficiencies may require considerable changes in design or even worse at the requirements level. Due to the dearth of the mentioned information, we need to work our way from source code and test cases of a system to the system's architecture. Architectural recovery techniques provide meaningful abstractions of the system. However, they omit the dynamic information regarding how a user would interact with the system. Obtaining the dynamic behavior of a system manually is prohibitively expensive. It requires monitoring users of the system or interviewing the architects or developers of the system and in general it requires domain knowledge and expertise.

Therefore, we propose an approach, *Extract*, for automatic usage model extraction. *Extract* extracts the static architecture of a system, augments it with the behavioral information obtained from control-flow analysis of its components and test cases, and generates the system's usage models. These usage models enable researchers to conduct a broad area of experiments which otherwise requires having access to already available usage models or manually crafting them by either running the system, or going through code. These experiments fall into a large spectrum spanning fault tolerance analysis, reliability analysis, stress testing, performance prediction, and even more subjective quality related aspects of a system such as responsiveness and usability. Our approach helps software architects and developers on two fronts: (1) improving

[*] Michael Langhammer and Arman Shahbazian contributed equally to this work.

their existing systems, and (2) understanding third party components they want to integrate with their systems.

Our approach also enables reverse-engineers to start with the source code of the system, which is the only reliable up-to-date artifact most of the time, and recover the static and the use case-based view of the system. Hence, the time consuming and expensive tasks of deploying and measuring the software can be avoided. Furthermore, evolution tasks can be eased due to the existence of an up-to-date architecture, which can be used to specify which components are affected by which evolution task and to predict the possible performance impacts of the evolution task [31].

The contribution of this paper is twofold: (1) we show how the output of structural reverse engineering approaches can be enriched with behavioral information. The result is a component-based architecture analysis model for non-functional properties; (2) we propose a method for extracting static usage models from fully available test cases. We implemented our approach using *ARCADE* [17] as the reverse engineering approach to get a structural architecture representation from Java source code and the Palladio Component Model (PCM) as the component-based software architecture model.[1] PCM is proven to be useful in analyzing quality aspects of a software system model [2]. We evaluated our approach in multiple ways. First, in order to establish a ground for its accuracy we compared our usage models against a system with an already available set of use cases. For scalability analyses, we ran *Extract* on 14 free and open source software (FOSS) systems which we used to evaluate our recent work [17]. The remainder of the paper is structured as follows: In Section II, we introduce foundations. In Section III, we present our approach in detail and describe how we transformed the *ARCADE* output to PCM as well as how we extracted usage models from test code. In Section IV, we present the evaluation of our approach. Section V gives an overview of the related work. Section VI concludes the paper and gives an overview about future work.

II. Background and Foundations

A. Running example

In this section, we introduce a running example which we will use throughout the paper to explain our approach. This example's deliberate simplicity allows us to clarify the functionality of our approach, and to crisply display the complete input and output data, which otherwise would haven been impossible to do. Our running example is a small calculator program for computing the greatest common divisor (GCD) of two very big numbers.[2] The calculator we use consists of two packages, each containing one class (see Figure 1). The first class is the `CalculatorTool` that contains a main method and calls the `Calculator` class.

[1]https://sdqweb.ipd.kit.edu/wiki/Extract

[2]https://sdqweb.ipd.kit.edu/wiki/Calculator_Example

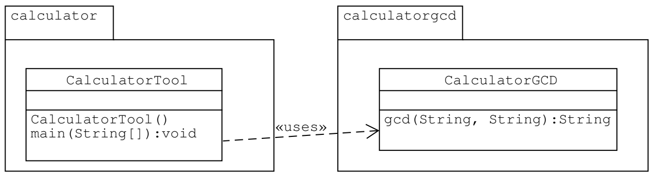

Figure 1. The UML class diagram of our running example. The `main` method of the `CalculatorTool` class uses the `gcd` method of the `CalculatorGCD` class. Each of the classes is contained in its own package.

`Calculator` performs the calculation of the GCD. Input and output are of type `String` to accommodate very large numbers. Furthermore, we have one test that tests the GCD calculation for the `Calculator` class.

B. Architecture Recovery, Change, and Decay Evaluator

In order to obtain the static architecture of a system from its source code, we built on our recent work which resulted in a novel approach, called *ARCADE* [17]. *ARCADE* is a software workbench that employs (1) a suite of architecture-recovery techniques and (2) a set of metrics for measuring different aspects of architectural change. It constructs an expansive view showcasing the actual (as opposed to idealized) evolution of a software system's architecture.

ARCADE's foundational element that we used is its architecture recovery component, called *Recovery Techniques*. The architectures produced by *Recovery Techniques* are directly used in the subsequent steps for extracting usage models. *ARCADE* currently provides access to ten recovery techniques. Two of these techniques are *Algorithm for Comprehension-Driven Clustering* (*ACDC*) [25] and *Architecture Recovery using Concerns* (*ARC*) [10]. Our previous evaluation [8] showed that these two techniques exhibit the best accuracy and scalability of the ten. *ACDC* leverages a system's module dependencies and recovers a primarily structural view of the system, while *ARC* uses information retrieval methods and recovers a semantic view of the system. We incorporated both of these techniques in *Extract*.

C. Palladio Component Model

PCM [2] is a component-based architecture model that can be used to design a system and predict its quality attributes such as performance, reliability, and cost. PCM is able to do this without having the actual implementation of the system. In order to predict the performance of a system, different user roles have to create five different models: *Repository*, *System*, *Resource environment*, *Allocation*, and *Usage*. The first model is the *Repository* model. Within the *Repository* component developers define all components and interfaces of a software system. Each interface contains signatures with parameters and return types. Components can provide multiple and require multiple interfaces. To model the relationship between components and interfaces, provided and required roles are used. A component that is providing a specific interface has to provide an abstraction

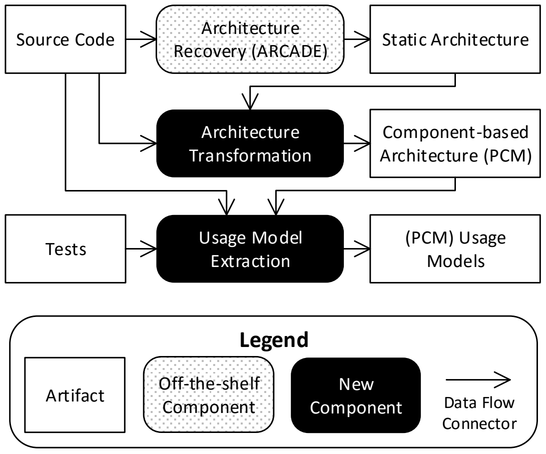

Figure 2. Overview of *Extract*. From the existing source code and unit tests our approach automatically extracts a component-based software model and its usage models. The gray boxes are the contributions of this paper. To realize the Architecture Transformation and the Usage Model Transformation we had to write 5500 Source Lines of Code (SLoC).

of the behavior of its implementation methods. Therefore, a Service Effect Specification (SEFF) [2] has to be created. An SEFF consists of different control flow elements, such as loops, branch actions, as well as internal actions. It also contains external call actions, which are calls to a signature of a required interface. Within the *System* the architects specify which components are combined to create an actual system. They also specify the interfaces a software system provides to its users. The third model is the *Resource environment*, where the person in charge of system deployment has to specify the environment in which a system runs. Therefore, she has to specify the resources in terms of servers that consists of CPUs, HDDs, and linking resources to other servers. In the *Allocation* model, the actual deployment from an assembly to the resource on which it is executed has to be specified. The last model that has to be created is the *Usage* model, which consists of at least one *Usage scenario*. Within the *Usage scenario*, domain experts have to specify how users interact with the system. Therefore, they can use *Loops* and *Branch* actions. To model the thinking time of the user, a *Delay* can be used. To model the concrete interaction with a system, *EntryLevelSystemCalls* have to be modeled. These are calls to the provided roles of a system.

For this paper, PCM's *Repository* including the SEFFs, *System*, and *Usage* models are important. We will clarify why below.

III. Approach

The goal of *Extract* is to extract architecture-based usage models to enable the analysis of non-functional properties (NFPs) of a system. To that end, we have developed an end to end approach, that automatically extracts PCM compliant use cases of a system using only its tests and source code. A high level overview of *Extract* is displayed in Figure 2. The extracted use cases can be directly incorporated in approaches built on top of PCM for analyzing a system's performance, reliability, and cost. However, our approach can easily be modified to use other notations for describing usage models. *Extract* can be divided into two distinct phases. In the first phase—*transformation*—we transform a system's source code to its PCM models. The second phase—*extraction*—combines the information obtained from a system's tests with its PCM models and generates the corresponding usage-models. In the remainder of this section, we discuss each of these phases in more detail.

A. Architecture Transformation

During the architecture transformation phase, *Extract* creates an up-to-date PCM instance comprised of Service Effect Specifications, Repository, and System models. This is used in the subsequent phase to extract a system's dynamic behaviors. In order to get the list of components and the static architecture of a system, we use *ARCADE* [17]. The output produced by *ARCADE* is not readily usable for our purpose and needs to be processed. In general, for a given system *ARCADE* generates two types of outputs: the system's (1) clustering and (2) class dependencies. Figure 3 shows the clustering for our running example. These clusters are the result of agglomerating the semantically and structurally related classes into components based on the architecture recovery algorithm in use. The second output is the list of class dependencies leveraged to cluster classes into components.

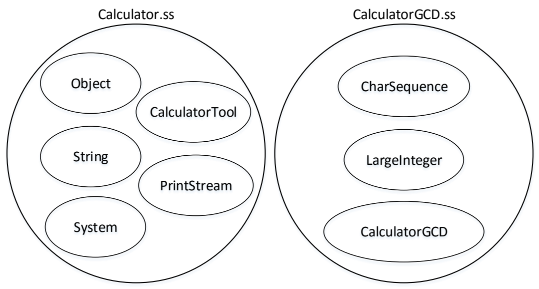

Figure 3. Running example's clustering.

These files are the building blocks of our approach. However, *ARCADE*'s extracted *class-level* dependencies are too coarse-grained to be used for creating PCM components, component-level interfaces, and their signatures. *Extract* needs to identify the dependent *methods* and their signatures. A method's signature consists of its return type, argument names, and data-types. *Extract* uses java-callgraph [1] to identify the method dependencies. After creating a list of method pairs (caller and callee methods) for each class dependency, we extract the involved methods' signatures from the source code. We have developed a signature extractor that is used to this end.

At this point we have all the necessary ingredients to transform *ARCADE*'s output into a PCM instance. A PCM instance is realized using the Eclipse Modeling Framework

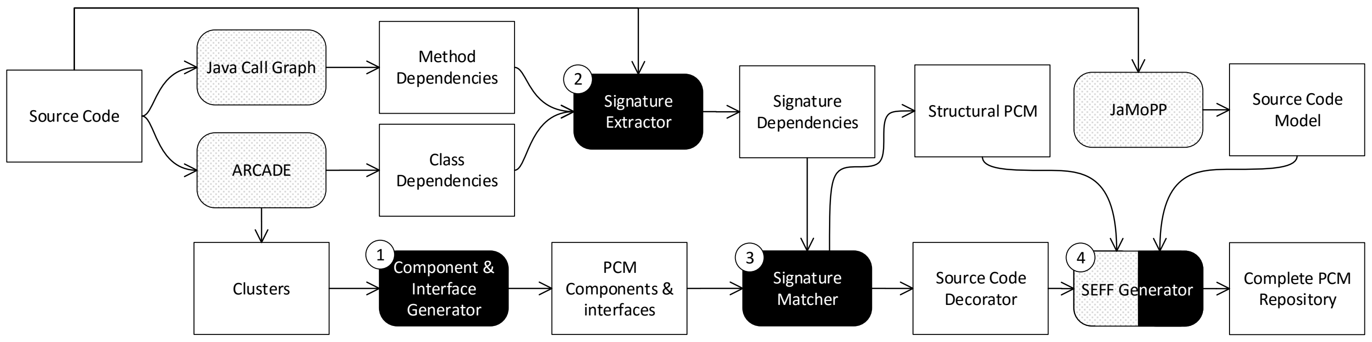

Figure 4. Overview of the transformation phase. We adapted and extended SoMox's SEFF Generator. Connectors and rest of the boxes have the same meaning as Figure 2. Corresponding steps from section III-A are specified with numbers 1 to 4.

(EMF).[3] To simplify things, we use Java Model Parser and Printer (JaMoPP) [11], which allows us to treat a system's source code like an EMF model. Creating the PCM instance includes the following four steps (see Figure 4):

1) Create a PCM repository that contains components and interfaces, as well as their provided and required roles.
2) Extract the dependency method signatures from source code.
3) Assign the extracted signatures and data-types to the provided interfaces.
4) Create Service Effect Specifications for the provided methods of a component.

In order to develop *Extract*, we implemented steps 1 and 4 from scratch. For step 2 and 3 we were able to reuse the already existing data-type generation and SEFF generation from SoMoX [15]. However, we had to make sure that *Extract* generates the same input data for the data-type generation and SEFF analysis as SoMoX does.

To create components and interfaces from *ARCADE*'s output, we transform each cluster to one component in PCM. Since *ARCADE* does not define explicit interfaces for clusters, we create one provided interface for each component as well as a provided role between the component and that interface.

The next step is to create the signatures, parameters, and return types in the provided interface of each component. In order to do so, we traverse the call-graph. If a method calls an external method (i.e., a method of a class residing in a different component from the callee), then the called method is a provided signature for its corresponding component. After creating the signatures, we have to create the signatures' parameters and return types from the code elements. For type creation we reuse the approach SoMoX [15] uses. This approach maps primitive types (e.g., `int` or `long`) to their corresponding primitive PCM data types. Complex data types in the code, such as classes, are mapped to `CompositeDataTypes`. Within the scope of this paper, we extended the SoMoX data type mapping to enable the creation of PCM `CollectionDataTypes` for classes that inherit from `Collection` or for data types that are used within an array type. Applying these two steps to our running example gives us the repository depicted in Figure 5.

[3]https://www.eclipse.org/modeling/emf/

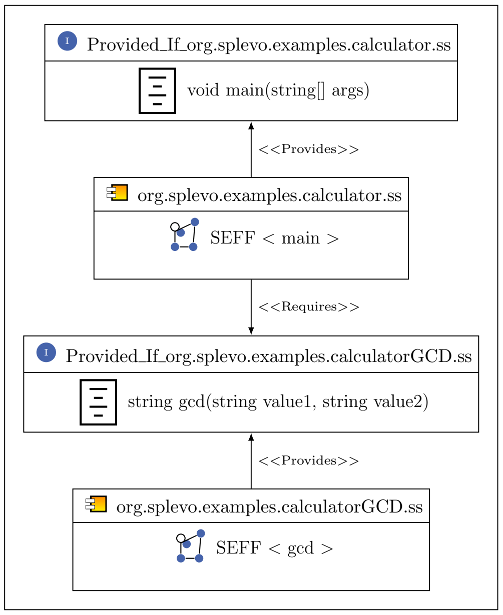
Provided_If_org.splevo.examples.calculator.ss
void main(string[] args)
<<Provides>>
org.splevo.examples.calculator.ss
SEFF < main >
<<Requires>>
Provided_If_org.splevo.examples.calculatorGCD.ss
string gcd(string value1, string value2)
<<Provides>>
org.splevo.examples.calculatorGCD.ss
SEFF < gcd >

Figure 5. Result after applying the first two steps of *Extract* to the running example. During the first step, we create the components and the interfaces. During the second step, the signatures including return types and parameters are created.

To create an SEFF for each provided signature of a component, we have to create a Source Code Decorator Model (SCDM). The SCDM contains the information on how the classes, their interfaces, and methods are mapped to the architectural elements such as components, interfaces and signatures. To create the SCDM within *Extract* the following steps are necessary: Firstly, during the creation of the components and interfaces, we create the mapping from classes to components, and secondly, during the signature creation we create the mapping between code methods and architectural signatures.

After the creation of the SCDM, we run the SEFF generation approach from SoMoX [15]. The SEFF generation approach conducts a control flow analysis on the JaMoPP

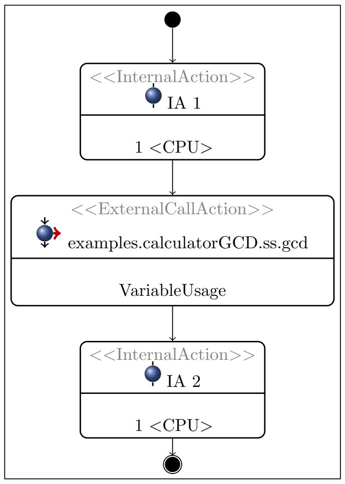

Figure 6. Result after applying the SEFF reconstruction to the main method. The main method conducts some internal actions before calling the `gcd` method from `CalculatorGCD`.

model of the source code for each class or interface method that corresponds to an interface signature. During this analysis each statement is visited and an SEFF action is created if necessary. The most important statements are the method calls to an external component. These calls are mapped to `ExternalCallActions`. Other control flow elements such as `loops`, `if`, and `switch statements` are only mapped to the SEFF if they contain an external call. If this is the case, loops in the source code are mapped to `Loops` in the SEFF, switch and if statements are mapped to `BranchActions` within the SEFF. If this is not the case, the statements are mapped to `InternalActions`. To gain more abstraction from source code, consecutive `InternalActions` are merged into a single `InternalAction`. Component internal method calls, i.e., method calls that call a method of the same class or a method of a class within the same component, are also a part of the control flow analysis. Statements within an internal method calls are treated the same way as above. Applying the SEFF reconstruction approach to the main method of the running example gives us the SEFF depicted in Figure 6.

The application of these steps yields a PCM repository instance that consists of the system's static architecture as well as its behaviors in terms of SEFFs. Finally, we have to create the PCM system. To create a PCM system we instantiate each component from the repository once and connect the provided and required roles. This method has the limitation of not supporting multiple component instantiations. Since our current focus is to reverse engineer the static structure of a system, we do not need to create multiple instantiations. However, *Extract* is extensible to include them.

B. Usage Model Extraction

In this section, we explain how *Extract* generates PCM usage models from tests by creating the four main control flow elements of a usage model. The first control flow element within a PCM usage model is its entry-level system calls. These are the cases where a user calls an interface signature of a system. The second control flow element is the loops, which specify how often a user executes a specific task. The third control flow element is the set of branch actions, which are used to decide which one of the possible actions a user will take next. The last control flow element is the delays, which represent the user's thinking time before calling or executing the next action. Even though we present an approach to extract PCM usage models from source code, *Extract* can be adapted to generate other usage models as well. For example, we could create UML use case diagrams by interpreting the tests as actors and the calls to the system as use cases. An overview of the usage model extraction process can be found in Figure 7.

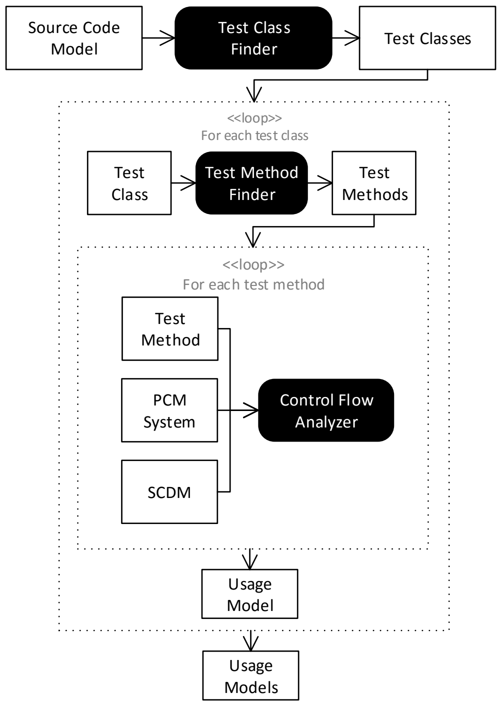

Figure 7. Overview of the usage model extraction process. The inputs are the source code model, the PCM system and the Source Code Decorator Model (SCDM) we generated in the previous step. Legend is the same as Figure 2.

To extract the usage model from the source code, three steps are necessary. First, we have to get the tests from a given project. To do so, we follow the approach Maven[4] uses. Classes that start with `Test` or end with either `Test` or `TestCase` are considered as test classes. We create one usage model for each test class. The second step is to identify the test methods. In JUnit4 they are annotated with `@Test`, and in JUnit3 they start with `test` and reside in classes extending the `Testcase` class. For Listing 1, this step identifies `CalculatorTest` as test class, and `testCalculator` as test method. For each test method we create one usage scenario in the usage model for the test class. The third step

[4]https://maven.apache.org

is to figure out which PCM signatures are called by tests. Our strategy is to visit all method calls within each test and check whether these calls are *test internal* calls, *library calls* or *test external* calls. If they are test external calls, we check whether the callee belongs to a class or interface in one of the system components. If so, we mark the call as an *entry-level system call*. The application of this step to the code in Listing 1 identifies `gcd` as a *test external* method call. It also reveals that the callee is one of the system's components. Hence, we mark this call as an *entry-level system call*. Afterwards we conduct a control flow analysis for each test method. The control flow analysis as well as the method call visiting strategy is similar to the SEFF control flow analysis implemented in SoMoX [15]. The goal of this analysis, however, is to get the call sequence of the *entry-level system calls* as well as the test behaviors.

```
public class CalculatorTest{
    @Test
    public void testCalculator() {
        CalculatorGCD gcd = new CalculatorGCD ();
        for(int i = 0; i < 100; i++){
            String gcdResult = gcd.gcd(i, i*7 );
            assertNotEquals(gcdResult, null);
            assertTrue(Integer.valueOf(gcdResult) > 0);
        }
    }}
```

Listing 1. Test method for the `gcd` method of the running example.

In summary, *Extract* considers calls, as well as other control flow elements such as `loops`, `switch`, and `if` statements. `Loops` in the test code are translated into for `Loops` in the usage model. `If` and `switch statements` are translated into `branch actions`. Calls to an interface signature are translated to `entry-level system calls`. All other calls and statements, e.g., calls to helper methods, are translated into `Delays`. To gain more abstraction from the code, we follow the same approach for `Delays` that is done for `InternalActions` in the SEFF reconstruction: As long as the former statement in the code corresponds to a `Delay`, we do not create a new `Delay`. Instead we aggregate all consecutive `Delays` into one. In Listing 1, the constructor call is not considered an *entry-level system call* and is translated into a `Delay`. Since the next `loop` statement contains an external call, `gcd`, we create the corresponding usage model `Loop`, with an *entry-level system call* inside of it. Finally, the assertions are translated into a `Delay`. The result of the usage model extraction applied to Listing 1 of our running example is depicted in Figure 9.

The advantage of this approach is twofold: (1) during the reverse engineering, we get usage models from a system without having to run the system; and (2) developers can write tests within the code to get usage models instead of creating them within the PCM. Hence, developers do not need to learn how to create usage models within an architecture description language with which they may be unfamiliar. Instead, they can use their regular tools (source code editors) to create them.

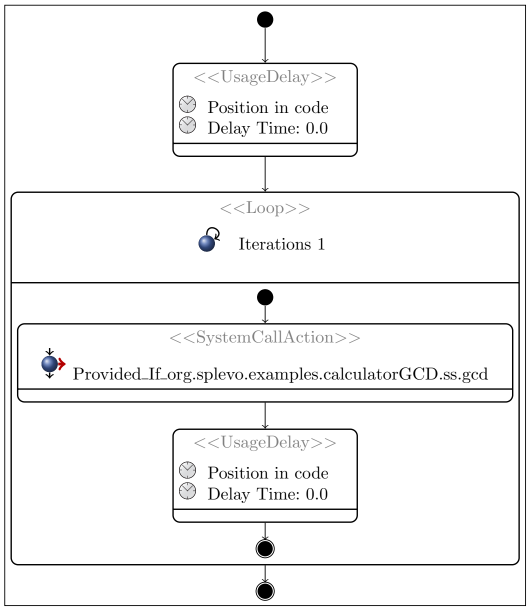

Figure 8. *Extract* applied to the test case from Listing 1.

To use the extracted usage models for NFP analysis, users have to enrich the models with the following pieces of information. Firstly, they have to add dynamic information, such as loop counts for loops and parameters of the entry-level system calls as well as probability for branch actions. Secondly, they have to specify whether the workload is an open workload, i.e., new users arrive after a specified arrival time, and the number of users is not limited, or a closed workload, i.e., fixed number of users constantly interact with the system [24].

C. Limitations

In this section, we discuss several of the current limitations of *Extract*. First, the approach has the flaw that not every test is going to be representative of a usage model for NFP analysis. A lot of tests are designed to provide good code coverage or to test "corner" cases. This limitation however, can be addressed by relying on statistical test cases, which can be used directly for creating usage models. Furthermore, integration tests implemented to cover a system's use cases are another good source of tests.

Since we only consider statements and method calls in the control flow analysis *Extract* cannot reconstruct event-based communication among components. Furthermore, anonymous classes are not considered in our current work. Those limitations apply to the SEFF reconstruction as well as to the usage model reconstruction. Reconstruction of event-based communications, however, can be added to the SEFF reconstruction by using the event [22] extension for the PCM. Since we use JaMoPP, we are limited to Java 5 constructs. Hence, new language features, such as lambdas, are not supported yet.

A further limitation arises with recursion. Currently, we are not able to reconstruct recursive method calls correctly.

Instead of a recursive call we create an empty behavior. For the SEFF reconstruction this problem can be solved by using *ResourceDemandingInternalBehaviour* from the PCM. *ResourceDemandingInternalBehaviour* can be seen as component-internal *SEFF*s that can be called from within an SEFF or a *ResourceDemandingInternalBehaviour* itself. For the usage model reconstruction the problem can not be solved easily since there is no "internal usage model" call. Hence, to recover tests that use recursive method calls correctly such a construct has to be introduced in the PCM.

IV. Evaluation

We have empirically evaluated *Extract* to measure its accuracy and scalability. In the remainder of this section we first present a case study of usage model extraction using CoCoME [12] as our subject system, then present the scalability analysis performed on 14 open source systems.[5]

A. Usage Model Extraction

To validate the usage model extraction we used the Common Component Modeling Example (CoCoME) [12]. It represents an example system that could be used in supermarkets and retail stores, and supports regular processes, such as buying and paying for goods, and administrative processes, such as ordering new products and inventory management. CoCoME was originally designed to compare different modeling approaches in component based software development and modeling. Since Herold et al. described CoCoME and its use cases in detail [12], it is a good fit for our evaluation. Furthermore, there are written unit-tests for all of the eight described use cases.

These unit tests are implemented using a *TestDriver* and several test interfaces to allow testing of multiple implementations of CoCoME. For this reason our approach was not directly able to extract the usage models since the calls to the test interfaces are not considered as *EntryLevelSystemCalls*, i.e., calls to components of the system. To circumvent this problem we modified our approach to consider calls to those interfaces as *EntryLevelSystemCalls*. In summary, we found out that our extracted usage models cover all the interactions between users and the system, plus some additional *EntryLevelSystemCalls*, which were not specified in the original use cases. These *EntryLevelSystemCalls* are generally assertion-initiated calls to a component, and are only used to verify a result, but not to interact with the system as a user would. The general convention is to have one test per method. Therefore we do not expect these calls to be overwhelming for engineers.

To give an example of a usage model *Extract* extracted for CoCoME, let us consider the initial steps of use case 1 – Process Sale from [12]. This use case describes the payment procedure at a cashier's desk of a supermarket. The first four steps are: 1) a customer arrives with her goods at the cashier's; 2) the cashier presses the *Start New Sale* button on the cash box; 3) the cashier enters item IDs, either manually (using the keyboard) or automatically (using a barcode scanner); 4) price of that product as well as the running total are presented to buyer. Steps 3 and 4 are repeated until all products are registered. A complete description and the rest of the use case can be found in [12]. Figure 9 shows the usage model we extracted for this use case if a barcode scanner is used.

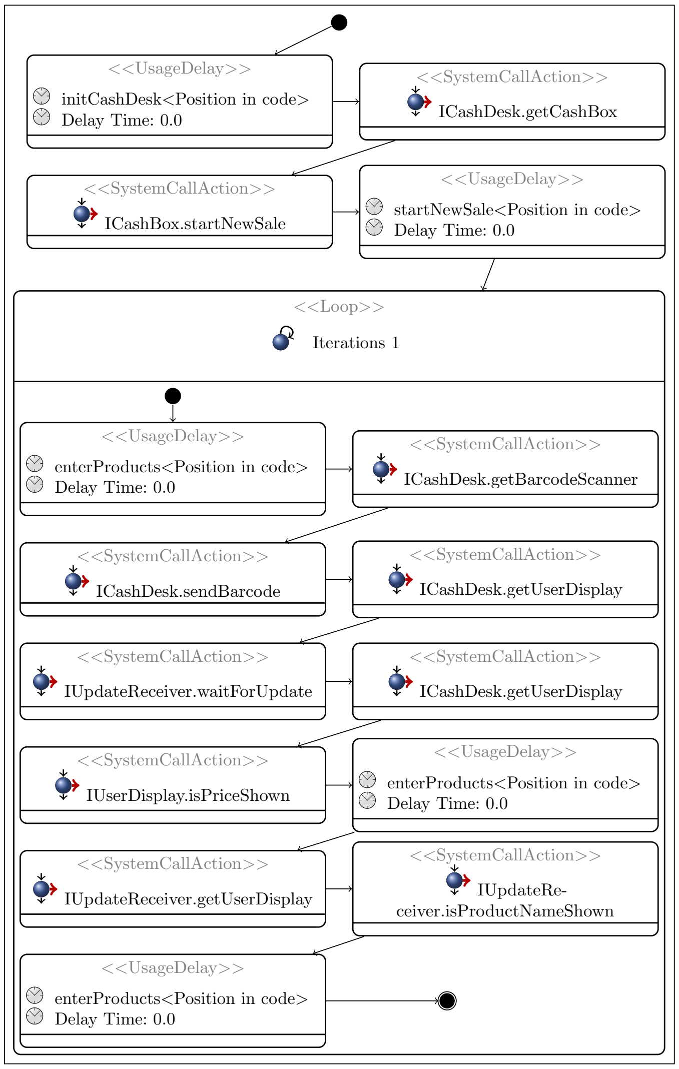

Figure 9. *Extract*'s output for the first three steps of use case 1.

Our approach successfully extracted usage models for the test cases of the eight defined use cases in CoCoME, and was able to determine all the necessary *EntryLevelSystemCalls*. However, as we mentioned above, we found more *EntryLevelSystemCalls* than specified in the original use cases. For instance, before sending a bar code, the method *getBarcodeReader* is called, which is not shown in the original use case where the card reader is shown as an actor. Moreover, the last four *EntryLevelSystemCalls* within the `for` loop (*getUserDisplay, isPriceShown, getUserDisplay, isProductNameShown*) are test assertions. A more detailed usage model can be beneficial for developers since it

[5]The results of the case studies are available online: https://sdqweb.ipd.kit.edu/wiki/Extract

provides more insight about how it translates into code. However, more detailed usage models can be onerous if the purpose is to gain general understanding of the system. As we also mentioned in the section above not all test cases are a good fit for usage models. For CoCoME, our approach extracted a usage model for `StoreQueryImplTest` that is not part of the original set of use cases.

We expect better results compared to the case study, if the developers are aware that the tests are used to create usage models in a subsequent step because they can, e.g., leave out unnecessary verification calls and write tests in a way that would more faithfully reflect a user's behavior.

B. Scalability Analyses

In the first part of the evaluation, we showed that our approach works for CoCoME. For the second part we analyzed 14 open source software systems comprising over 2 millions lines of code to convey the applicability and scalability of *Extract* for analyzing real world systems. Table I, lists the systems and their version numbers. We used the same set of systems used for a previous publication to cover both a good range of domain, and sizes [17].

Extract obtained the same number of usage models regardless of whether it relied on ARC or ACDC. The reason is that the starting point of our approach is the set of test cases, which are independent from the architecture recovery algorithm. The projects we investigated result in a combined number of 2868 clusters when recovered using ARC and 696 clusters using ACDC. *Extract* was able to transform each cluster to its corresponding *BasicComponent*. *Extract* also created a provided interface for each component. For the test classes we created a total of 3804 usage models. *Extract* was able to conduct the control flow analyses to create the *SEFF*.

One problem arises when a call to an abstract method or an interface method occurs and more than one class in the same component implements that interface or abstract method. Currently we take the first implementation in the component for the interface or abstract method. This, however, can be wrong if another implementation of the interface or abstract method is used during runtime. A solution would be to ask users during the reconstruction phase to specify which implementation should be used.

The scalability analysis showed that *Extract* can be used in principle to analyze real world software system and is able to deal with systems with different sizes and large numbers of components (see Table I). For smaller systems such as *Log4j*, usage model extraction takes less than five minutes. For the larger ones it can take up to five hours (Windows 7 with 2.4 GHz Intel core i7, and 8GB RAM). This is significantly lower than the time needed for dynamic analysis techniques, which require running the system. The performance can be further enhanced by replacing JaMoPP with a more performant Java parser such as MoDisco [5].

C. Threats to Validity

A threat to validity for the usage model analysis is that we only investigated one system in detail for which the developers of the tests were not aware that the tests will be used for usage model extraction. This however generalizes our case and eliminates the possible bias that might get introduced if developers are aware of the future uses of their designed tests. Other threats to validity are that we investigated a limited number of systems and that all projects we investigated were Apache projects. However, as stated in [17] these systems are from different domains and of different sizes, and are widely used.

V. Related Work

In this section, we will highlight the most closely related work. Our previous works in prediction and assessment of non-functional properties of a system [2], [19], [20], static analysis techniques designed to uncover surreptitious defects in a software system [23], and our experience with architecture recovery [17] have both enabled, and motivated us to conduct this research. Our work in this paper relieves approaches such as Becker et al. which uses PCM for model-driven performance optimization [2], and Martens et al. [19] which takes advantage of PCM models for reliability, performance and cost analysis from having to manually craft usage models. Safi et al. proposed a static analysis technique to detect race conditions in event-based systems [23]. Their technique can benefit from automatically extracted usage models in several occasions. It can partly alleviate the need for manual inspection and help the engineers in pruning the race condition scenarios that are unlikely to happen or be harmful to the system. It also eases the discovery of application entry point required for that technique.

Researchers have long been using static analysis techniques to uncover useful information about a software system. Ducasse and Pollet [7] presented a process-oriented taxonomy for software architecture reconstruction approaches. In their taxonomy *Extract* would fit in the following categories: Our *Goal* is to support *Analyses* based on the reconstructed architecture, and the *Evolution* of software systems. We use a *Bottom-up Process* to analyze the source code. As our inputs we use *Source Code* as well as *Physical Organization* in terms of package structure. Our *Technique* is *Quasi-Automatic*. Whether we use *Clustering, Concepts, Dominance,* or *Matrix* depends on the recovery algorithm. The *Output* of *Extract* is an *Architecture.*

Whaley et al. [29] proposed a technique comprising dynamic and static analysis techniques to automatically extract component interfaces and transaction models. Unlike our approach, their technique requires dynamic instrumentation and outputs class level components, analyzing which will be a tedious task for developers of a large scale software systems. Approaches like [28], [6] also rely on run-time monitoring to build software usage models which will later be augmented with obtained measurements to

System	Domain	Version	KSLoC[6]	ARC	ACDC	UM	US	ELSC
ActiveMQ	Message Broker	3.0	95	116	88	140	3.5	4.3
Cassandra	Distributed DBMS	2.0.0	184	285	52	132	4.0	11.5
Chukwa	Data Monitor	0.6.0	39	74	43	86	1.6	5.2
Hadoop	Data Process	0.19.0	224	388	101	292	2.4	12.3
Ivy	Dependency Manager	2.3.0	68	128	40	117	8.0	9.8
JackRabbit	Content Repository	2.0.0	246	294	72	518	5.8	1.4
Jena	Semantic Web	2.12.0	384	766	36	414	15.4	3.4
JSPWiki	Wiki Engine	2.10	56	77	31	92	10.3	5.0
Log4j	Logging	2.02	62	187	61	240	3.7	4.8
Lucene Solr	Search Engines	4.6.1	644	115	41	1253	5.3	2.7
Mina	Network Framework	2.0.0	46	93	44	63	3.1	10.3
PDFBox	PDF Library	1.8.6	113	127	41	88	3.8	7.6
Struts2	Web Apps Framework	2.3.16	153	75	26	337	5.6	2.1
Xerces	XML Library	2.10.0	125	143	20	32	6.1	1.8
Total			**2439**	**2868**	**696**	**3804**		

Table I
Overview of the analyzed systems. ARC, and ACDC columns display the number of recovered components for each system. UM is the number of extracted usage models. US is the average number of usage scenarios per usage model. ELSC is the average number of extracted entry level system calls per usage scenario; it is used as an indicator for usage model sizes.

generate system performance models. Several approaches are concerned with the extraction of PCM instances from code. The tool Java2PCM [14] extracts component-level control flow from Java code. The assumption in that paper, however, is that the component detection has been done already. Krogmann et al. introduced SoMoX [15] and Beagle [15], [16] to extract a static architecture, static behaviors and dynamic information from source code. In our approach, we reuse parts of SoMoX to create the static behavior of a components' SEFFs. However, SoMoX does not extract usage models, and requires users to determine the weights for the metrics, which are used to recover the static architecture, upfront. Brosig et al. [3], [4] created an automated method for the extraction of architecture-level performance models of distributed component-based systems, based on monitoring data collected at run-time. The extraction algorithm they use is limited to EJBs (Enterprise Java Beans) [4], Servlets, Java Server Pages and Message-Driven Beans. Van Hoorn et al. looked at probabilistic and intensity varying workload generation for web-based software systems [26]. Their approach relies on use cases and therefore can be used in tandem with ours for software timing behavior evaluation, anomaly detection and automatic fault localization, as well as runtime reconfiguration of component-based software systems. Vögele et al. [27] and [30] have also looked at using dynamic traces to generated workloads and create performance models based on them automatically. Li et al. [18] discover use cases based on dynamic information, which is obtained by instrumenting the code. Qin et al. [21] reverse engineer use case diagrams by generating Branch-Reserving Call Graphs via a statical analyses and refining them using metrics in later steps. Hirschfeld et al. [13] treat use cases as first class entities and embed them into the code to help developers understand the use cases and their mappings to code. They introduced a new language construct (*@usecase*) for Python programming language. They also proposed a semi-automatic approach to recover use cases from acceptance tests, however, they execute test cases for use case recovery, and do not support any NFP analysis.

[6]Source Lines of Code including test code and counted with CLOC (see: https://github.com/AlDanial/cloc)

VI. Conclusions and Future Work

In this paper, we presented *Extract*, an approach that automatically extracts the system's architecture and usage models from source code and tests. We leveraged architecture recovery algorithms provided in *ARCADE* in tandem with the component-based approach PCM.

First, we mapped *ARCADE*'s extracted clusters to basic components and interfaces in PCM, enriched the architectural model with static behavior information, and generated the PCM SEFFs. Second, we developed an approach to extract usage models from unit tests. To this end, it is necessary to find the tests within a Java project and analyze the control flow of each test method. During the control flow analysis the statements in a test method are mapped to their corresponding PCM usage model elements.

In order to validate our approach, we used an existing project, with an existing use case and unit tests for the use case, and compared the use case with the extracted usage model. Furthermore, we evaluated the scalability and applicability of our approach by extracting an architecture and the behaviors of 14 open source software projects. The results show that it is possible to extract a reasonably accurate architecture as well as usage models from source code. We expect even better results for the usage model extraction if developers are aware that their tests are used to generate usage models.

In future, we plan to enhance the control flow analysis to consider event based communications and inner anonymous classes, and enrich the extracted usage models with dynamic performance information by integrating it with existing approaches such as Beagle [16].

Acknowledgments

We greatly appreciate the helpful comments of the anonymous reviewers. This work was partially funded by the German Research Foundation in the Priority Programme SPP1593 Design For Future: – Managed Software Evolution as well as by the InterACT (International Center for Advanced Communication Technologies) program. This work has also been supported by the U.S. National Science Foundation under award numbers 1117593, 1218115, and 1321141.

References

[1] Java call graph. https://github.com/gousiosg/java-callgraph, 2015.

[2] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[3] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE, 2011.

[4] F. Brosig, S. Kounev, and K. Krogmann. Automated extraction of palladio component models from running enterprise java applications. In *Proceedings of the Fourth International Conference on Performance Evaluation Methodologies and Tools*, 2009.

[5] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010.

[6] M. Courtois and M. Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2nd international workshop on Software and performance*, pages 105–114. ACM, 2000.

[7] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, 2009.

[8] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 486–496, 2013.

[9] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic. Obtaining ground-truth software architectures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 901–910. IEEE Press, 2013.

[10] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai. Enhancing architectural recovery using concerns. In *ASE*, 2011.

[11] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and java. In *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[12] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. Cocome - the common component modeling example. In A. Rausch, R. Reussner, R. Mirandola, and F. Plášil, editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer Berlin Heidelberg, 2008.

[13] R. Hirschfeld, M. Perscheid, and M. Haupt. Explicit use-case representation in object-oriented programming languages. In *ACM SIGPLAN Notices*, volume 47, pages 51–60. ACM, 2011.

[14] T. Kappler, H. Koziolek, K. Krogmann, and R. H. Reussner. Towards automatic construction of reusable prediction models for component-based performance engineering. *Software Engineering*, 121:140–154, 2008.

[15] K. Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*, volume 4. KIT Scientific Publishing, Karlsruhe, March 2012.

[16] K. Krogmann, M. Kuperberg, and R. Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *Software Engineering, IEEE Transactions on*, 36(6):865–877, 2010.

[17] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. *12th IEEE Working Conference on Mining Software Repositories*, pages 235–245, 2015.

[18] Q. Li, S. Hu, P. Chen, L. Wu, and W. Chen. Discovering and mining use case model in reverse engineering. In *Fuzzy Systems and Knowledge Discovery, 2007. FSKD 2007*. IEEE, 2007.

[19] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In *Proceedings of the first joint WOSP/SIPEW*. ACM, 2010.

[20] M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In *Component deployment*, pages 1–17. Springer Berlin Heidelberg, 2004.

[21] T. Qin, L. Zhang, Z. Zhou, D. Hao, and J. Sun. Discovering use cases from source code using the branch-reserving call graph. In *Software Engineering Conference, 2003. Tenth Asia-Pacific*, pages 60–67. IEEE, 2003.

[22] C. Rathfelder and S. Kounev. Model-based performance prediction for event-driven systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS 2009)*, New York, NY, USA, July 2009. ACM.

[23] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic. Detecting event anomalies in event-based systems. In *Fundamentals of Software Engineering (ESEC/FSE), 2015*. IEEE, 2015.

[24] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *NSDI*, volume 6, pages 18–18, 2006.

[25] V. Tzerpos and R. Holt. ACDC: an algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering (WCRE)*, 2000.

[26] A. Van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In *Performance Evaluation: Metrics, Models and Benchmarks*, pages 124–143. Springer, 2008.

[27] C. Vögele, A. van Hoorn, and H. Krcmar. Automatic extraction of session-based workload specifications for architecture-level performance models. 2015.

[28] D. Westermann and J. Happe. Towards performance prediction of large enterprise applications: An approach based on systematic measurements. In *Proceedings of the 15th International Workshop on Component Oriented Programming (WCOP 2010), Interne Berichte, Karlsruhe, Germany*, 2010.

[29] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ACM SIGSOFT Software Engineering Notes*, pages 218–228. ACM, 2002.

[30] F. Willnecker, A. Brunnert, W. Gottesheim, and H. Krcmar. Using dynatrace monitoring data for generating performance models of java ee applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 103–104. ACM, 2015.

[31] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE, 2007.