Towards Automatic Construction of Reusable Prediction Models for Component-Based Performance Engineering

Thomas Kappler[1], Heiko Koziolek[2], Klaus Krogmann[1], Ralf Reussner[1]

[1]Chair Software Design & Quality, Universität Karlsruhe (TH)
Am Fasanengarten 5a, 76131 Karlsruhe
{kappler|krogmann|reussner}@ipd.uka.de

[2]Graduate School TrustSoft, Universität Oldenburg*
Uhlhornsweg, 26111 Oldenburg
heiko.koziolek@informatik.uni-oldenburg.de

Abstract: Performance predictions for software architectures can reveal performance bottlenecks and quantitatively support design decisions for different architectural alternatives. As software architects aim at reusing existing software components, their performance properties should be included into performance predictions without the need for manual modelling. However, most prediction approaches do not include automated support for modelling implemented components. Therefore, we propose a new reverse engineering approach, which generates Palladio performance models from Java code. In this paper, we focus on the static analysis of Java code, which we have implemented as an Eclipse plugin called Java2PCM. We evaluated our approach on a larger component-based software architecture, and show that a similar prediction accuracy can be achieved with generated models compared to completely manually specified ones.

1 Introduction

Model-based performance prediction methods [BMIS04] aim at analysing the performance (i.e., response time, throughput, resource utilisation) of large software systems already during early development stages. They support developers by allowing performance analyses of architectural models even before starting the implementation of code. This approach avoids realising architectures with poor performance properties, which can cause expensive redesigns and/or re-implementations. Many of these approaches use annotated UML diagrams [Obj05] to specify performance properties and transform these models into established analysis models, such as a queueing networks, stochastic Petri-nets, or stochastic process algebra [BMIS04].

*This work is supported by the Germany Science Foundation, grant GRK/1076.

A particular challenge for performance prediction arises for component-based systems, where software architects reuse existing components besides new ones, which are designed for specific requirements. Performance models (e.g., annotated UML diagrams) for reused software components allow answer sizing questions or determining the impact of relocations and extensions, but are usually not available from component developers. They provide their component's source or binary code, but software architects cannot execute and measure the performance of these components, if they require other components, which are designed (i.e., defined via their provided and required interfaces), but not yet implemented or if the deployment environment is not yet available.

Software architects can apply reverse engineering techniques on existing components to derive information needed for a performance model (i.e., resource demands and a behavioural abstraction of the component services) from component code. However, existing reverse engineering tools do not support creating performance models from code, but instead focus on functional properties [Kos05]. They are able to reconstruct static code structures as UML class diagrams and often also (partially) dynamic behaviour as UML sequence diagrams [BLL06], which however lack a specification of resource demands needed for performance models. Furthermore, the resulting sequence diagrams often contain information not necessary for performance analysis and are too fine-granular for an analytical performance model.

To tackle this problem, we propose a new reverse engineering approach for deriving performance models from implemented software components. We target components implemented in Java and use code analysis techniques to create instances of the Palladio Component Model (PCM) ([BKR07, RBK$^+$07]). This performance specification language for software components allows composing individual component performance models into a complete architectural performance specification. It is supported by model transformations and simulation tools to predict performance properties. In this paper, we focus on one specific step of the reverse engineering approach, namely the static analysis of Java code to derive abstract behavioural performance models of component services. A special benefit of our approach is that software architects can reuse the resulting performance models in different architecture models.

The contributions of this paper are (i) a generic process for reverse engineering reusable performance models from code and (ii) a static code analysis method implemented as an Eclipse plugin called Java2PCM mapping Java code to PCM instances (as example of a performance model). We have evaluated the applicability of our approach by creating performance models from the code of CoCoME [RRMP08], a component-based system. With the performance models generated by Java2PCM and additional manual treatment, we were able to make performance predictions for CoCoME.

This paper is organised as follows: Section 2 discusses related work, before Section 3 describes the target model of our reverse engineering approach and explains the process model. Section 4 introduces our static code analysis, and Section 5 describes a case study evaluating the proposed benefits. Afterwards, Section 6 discusses benefits and limitations of the approach, before Section 7 concludes.

2 Related Work

This work is related to reverse engineering, automatic complexity analysis, and model-based performance prediction. We argue that a tighter integration of both areas is needed to broaden the scope of performance predictions.

Reconstruction of software architectures is a rich research field [Kos00, pp. 351][TTBS07]. Also, several commercial CASE tools support reverse engineering, in particular design recovery, from code. IBM Rational Software Architect [IBM07] and Borland Together [Bor07], for instance, create UML class and sequence diagrams from Java code. The resulting sequence diagrams do not contain control flow guards or forks modelling thread invocations, however, which are important for performance models. Flowchart4J [Cod07] visualises the control flow through code. Briand et al. [BLL06] use source code instrumentation and execution to create more expressive sequence diagrams (e.g., with thread invocations). However, none of these approaches is able to provide a performance specific abstraction of component behaviour.

Automatic complexity analysis such as [Ros90] determines abstract big O notations or average case execution time [HC88] of algorithms via static code analysis. Later approaches like [SF96] support analysis of upper-bound (O), lower-bound (Ω), matching of upper and lower bound (Θ), and asymptotic approximations ("more concise approximate expression in terms of well-known functions that still can be used to compute very accurate numerical results" [SF96, p. 23]) but do not handle general distributions. Wegbreit [Weg75] uses dynamic analysis in a semi-automatic approach designed for LISP, but does not support concurrency in programs nor components and requires expert knowledge for complex programs. The focus is on min, max, mean, and variance values. Our approach tries to determine more refined performance specifications especially for software components with contractually specified interfaces by combining static with dynamic analysis.

Model-based performance prediction methods [BMIS04] mostly support a top-down development process, where developers create a new software architecture and components (including models and implementation). These methods rarely foresee integrating existing components into software architecture models. Although there are many performance measurement and profiling tools, existing software components cannot directly be performance tested when being integrated into a modelled, but yet unimplemented architecture.

3 Reverse Engineering for Component Performance Models

This section first explains the target model of our reverse engineering approach, namely PCM [BKR07] instances, before it sketches the overall process to generate such models from code.

The PCM is a meta model to describe component-based software architectures and specifically enables performance predictions during early development stages. Other than UML, it provides several domain specific modelling languages for different developer roles. Component developers model the control flow through their components, software ar-

chitects can compose the resulting models to an assembly. System deployers model the hardware environment and the mapping of components to hardware, and domain experts specify the usage of the system in terms of parameter values and number of users.

All specifications may contain performance annotations, such as expected execution times or performance-relevant attributes of hardware resources. The PCM itself is specified in Ecore from the Eclipse Modelling Framework. It consists of more than 100 meta-classes, therefore a thorough description is beyond the scope of this paper (cf. [BKR07, RBK+07]).

In this paper, we particularly focus on the semi-automatic derivation of behavioural skeletons (Resource Demanding Service Effect Specification, RDSEFF) for component services from existing source code.

An RDSEFF provides an abstract model of a single component service's behaviour. It is reduced to resource demands (e.g., to CPU, hard disk) of internal actions (i.e., computations by the service) and calls to external (i.e., required) services (Fig. 1). An internal action can subsume a large number of instructions and express its resource demand as a constant value or probability distribution function (not shown here, see [BKR07]). The abstraction from a component service's actual behaviour is necessary to keep the models analysable and to not reveal the component developer's intellectual properties, because the models could eventually be included into public repositories enabling third-party reuse.

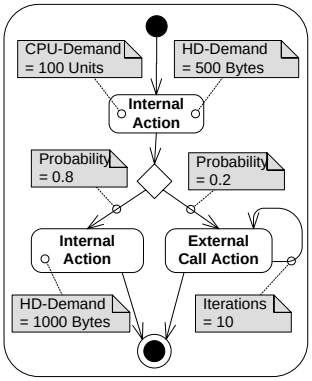


Figure 1: RDSEFF

Besides internal actions and external call actions, RDSEFFs can contain branch probabilities, loop iterations numbers, and control flow forks. All model annotations can be specified in dependency to parameter values. For example, a loop iteration number can be bound to the length of a collection provided to the component service as an input parameter. The length of such a collection is usually unknown to the component developer as the component could be used in many different contexts. An RDSEFF with such parametric dependencies is reusable in different architecture and deployment models.

Our approach for generating RDSEFFs from code combines static and dynamic code analysis (also see [Ern03]) trying to leverage the benefits of both methods (Fig. 2). The proposed process model is generic and can be applied with different programming languages, tools, performance models, and performance prediction methods.

Static analysis starts from source code and parses it to an abstract syntax tree (Fig. 2, 1.1) for example using the Eclipse Java Development tools. Then, component boundaries need to be identified (1.2). This is challenging if the code under study is written in an object-oriented language such as Java, which per se does not support the component paradigm. There is a large body of research dedicated to design recovery from source code [Kos05]. In the context of the PCM, Chouambe [Cho07] has implemented a component detection

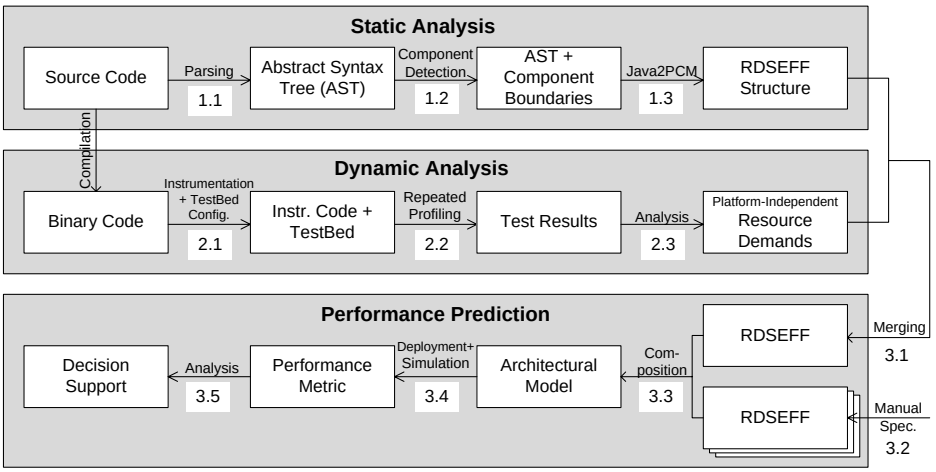


Figure 2: Reverse Engineering and Performance Prediction

tool for arbitrary Java code, which identifies components via different code metrics. For the scope of this paper, we assume this step as already executed. The static analysis implemented by the Java2PCM tool [Kap07] (1.3), which is presented in this paper, operates on the AST representation to reconstruct an initial RDSEFF structure without annotations. It uses the component boundaries identified in the previous step to distinguish between internal and external actions. Section 4 describes this step in more detail.

Dynamic analysis starts from binary code, instruments it, and additionally sets up a testbed (2.1) to execute single component services with different parameter values, which is needed to extrapolate their full behaviour. We have not implemented this step yet and consider it future work. Dynamic analysis specifically aims at identifying the resource demands for RDSEFFs, which the static analysis cannot determine (2.2). To simulate potentially missing required services from external components, the analysis uses generated dummy-components. Woodside et al. [WVCB01] have set up a corresponding test environment, which could be used for this step. The resulting execution times need to be analysed with statistical methods (e.g., linear regression, regression splines) so that function over the input parameter values express the resource demands (2.3). Krogmann [Kro07] proposes using genetic algorithms to determine complex functional dependencies from the measurement data in this step. The resource demand functions can further be parametrised for different platforms [KB07].

Tools can merge the RDSEFF structures from static analysis with the resource demand functions from dynamic analysis to create complete RDSEFFs (3.1). For the scope of this paper, we specified the resource demands manually and merged them into the RDSEFFs by hand, as the encapsulation of this step into tools is still missing.

Software architects can then use these parametrised specifications to model complete software architectures for *performance prediction*. They may compose modelled components containing RDSEFFs resulting from code analysis with manually specified ones

[RBK+07] (3.2, 3.3). Adding deployment information to the architectural model enables simulating or mathematically analysing the model for different usage profiles and hardware environments (3.4). In the PCM context, a discrete-event simulation based on extended queueing networks is able to derive performance metrics, such as response times, throughput, and resource utilisation, from the models [BKR07]. The results can then reveal the infeasibility of certain performance requirements and support design decisions with quantitative data (also see [BCdK07]) (3.5).

4 Static Code Analysis to reconstruct RDSEFFs

This paper focuses on the static analysis of Java code to reverse engineer RDSEFFs. The static analysis reconstructs an abstraction of the control and data flow through a component service and produces an initial RDSEFF model. RDSEFFs hide internals of a component service as long as they do not influence calls to required services. Therefore, the goal of the static code analysis is to hide internal component instructions that do not concern the interaction with other components.

Furthermore, it tries to create boolean expressions for branch conditions and arithmetic expressions for loop iterations numbers, which do not reference local variables, but only parameters declared in the component interfaces. The static code analysis is not concerned with resource demands to the CPU or memory, which shall later be added by the dynamic analysis.

The Java2PCM transformation tool implementing the static code analysis is a plugin for Eclipse. The user selects classes in the Eclipse package explorer and starts the transformation. First, the Eclipse Java Development Tools (JDT) parser creates an Abstract Syntax Tree for the selected code, which is then processed by Java2PCM using a visitor provided by the JDT. After the transformation, a Palladio instance with the created RDSEFFs is serialized as XMI, based on the Eclipse Modelling Framework (EMF).

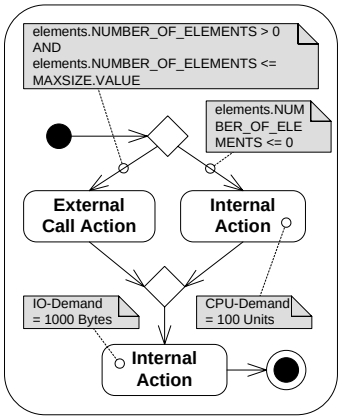
Listing 3 shows a Java code fragment and the RDSEFF created for it. Following this example, we explain the four main tasks of Java2PCM's mapping. RDSEFFs are centered around calls to external services and resource demanding internal actions, which we call *relevant actions* in the following.

Mapping Control Flow. When parsing source code, the transformation creates the appropriate Palladio elements for the Java elements in the RDSEFF. In Listing 3, one of the "if" statements shows as a branch (the omission of the other is addressed in the next paragraph).

Java2PCM also maps thread invocations to RDSEFF forks. They are recognized as methods named `start` when the providing object extends the class `Thread` or implements the interface `Runnable`, both given by the Java API. It recognizes loops iterating over a collection as there is a designated Palladio element for this case, and re-orders switch/case branches to match the Palladio semantics.

```
1  public void someMethod(List elements) {
2      storeHelper(elements)
3  }
4
5  private void storeHelper(List data) {
6      if (data.size() > 0) {
7          if (data.size() < MAXSIZE)
8              wrapper.externalCall(data);
9      } else {
10         // some non-relevant statements
11         if (anotherCondition)
12             // some more statements
13     }
14     FileOutputStream f =
15             new FileOutputStream(LOGFILE);
16     PrintStream ps = new PrintStream(f);
17     ps.println("List " + data + " stored.");
18 }
```

(a) Code



(b) RDSEFF

Figure 3: Example

Raising Abstraction. Java2PCM raises the level of abstraction by mapping control flow constructs not containing relevant actions to summarized internal actions, and by inlining private helper methods.

If, for example, an if statement only has actions in its body that are deemed non-relevant by the analysis, it need not explicitly be present in the RDSEFF. The "else" branch starting on line 9 of Listing 3, although containing another "if" statement, is summarized to one internal action, because the inner branch does not contain any relevant action. The dynamic analysis or a developer later annotate this action with resource usage information, so that the model contains the effect of the summarized statements. Also, consecutive branch conditions with no relevant actions in between can be merged into one branch. The two branches on lines 6 and 7 are merged into one branch.

The analysis inlines the relevant actions of private methods, analogous to the inline expansion method known in compiler construction [GW95]. In the example, the actual contents of the RDSEFF are in the local `storeHelper` method, and the resulting RDSEFF contains them at the location of the invocation of `storeHelper`.

Locating Resource Usages. Java programs access resources either implicitly, like CPU or memory, or explicitly via API calls. Java2PCM addresses the implicit resource usages by summarizing blocks of statements to one internal action. It classifies Java API calls according to a predefined list of packages and methods which are likely to use resources such as storage or network, for instance, `java.io` and `java.sql`. For such invocations, the tool creates separate internal actions. In Listing 3, the invocation of `println` from the `java.io` package in line 17 is mapped to an internal action in the RDSEFF. The dynamic analysis or a developer later quantify both kinds of internal actions.

Parameter Tracing. When a service takes input parameters, the analysis must trace them through the code to capture their influence on branch conditions, loop iterations, and the data flow to other components. If a parameter occurs on the right hand side of an assignment, for instance, and the assigned variable is used later in a conditional, the chosen branch depends on the parameter although the it does not explicitly occur in the conditional.

Java2PCM records assignments and the passing of parameters to local methods. Whenever a Java expression (including simple variable names) is translated to a branch condition or loop expression in the generated RDSEFF, its variables are checked for being influenced by a service parameter. If so, their occurrences are replaced by the expression currently assigned to them until all service parameters affecting the value of the variable are present.

Parameter tracing is a data flow analysis and has similar goals as symbolic execution [Kin76]. It statically determines boolean expressions for branch conditions and arithmetic expressions for loop iteration numbers, which only reference service parameters but does not expose local variables. This adheres to the information hiding principle of software components. Other than def-use-chain or use-def-chain analysis [Wol96] and program slicing [Wei81], it tries to determine boolean or arithmetic expressions over service parameters and not only code lines influencing a parameter usage.

In the running example, the branch conditions include the parameter “elements”, even though in the actual variable name in the “if” statement is “data”, because the service parameter was traced through the method invocation.

For simple assignments, this algorithm works as it simply exchanges one variable name for another name, the parameter name, which is also valid in the PCM instance. For more complex variants, such as expressions including method calls, the simple replacement renders long Java expressions which are not valid in RDSEFFs and have to be manually edited. However, their presence aids the developer in understanding what is happening at that point in the model.

5 Evaluation

We applied Java2PCM to a component-based software architecture (CoCoME). We wanted to find out whether the models produced by our tools enhanced with manual additions achieved the same prediction accuracy as fully manually specified ones, and how much time could be saved opposed to a fully manual specification.

Several CBSE research groups have recently jointly specified the ”Common Component Modelling Example” (CoCoME[1], [RRMP08]). CoCoME is an application managing a supermarket chain by providing billing and storage information and allowing the exchange of goods between different stores. It intends to provide a reference model to compare different component modelling and analysis models. The specification consists of a set of UML diagrams involving more than 20 software components, textual descriptions, and

[1] see http://www.cocome.org

performance annotations. A Java implementation of CoCoME is available, enabling the comparison between model-based predictions and code-based measurements to improve current prediction methods.

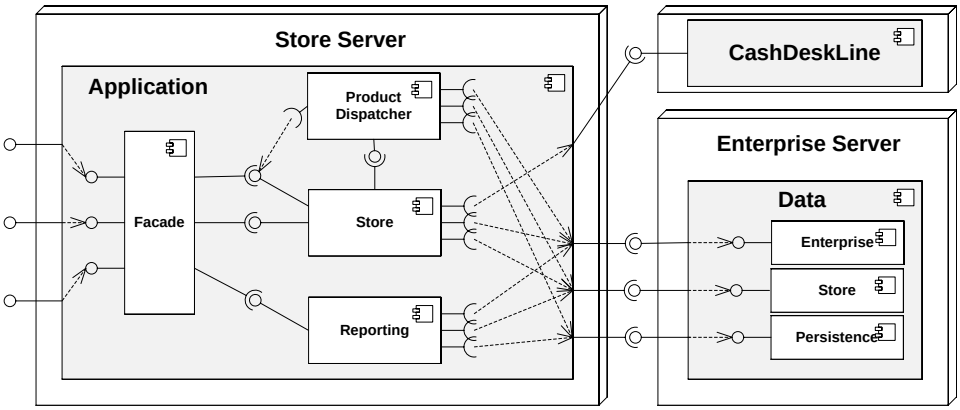


Figure 4: Extract from the CoCoME Software Architecture

In our case study, we focused on use case 8 of CoCoME, which models the exchange of goods between different stores of the supermarket chain. Fig.4 shows an excerpt of the CoCoME software architecture relevant for this use case. Multiple store servers interact with a single enterprise server, which manages the overall storage information of the supermarket. The components `Application`, `CashDeskLine`, and `Data` are composite components, consisting of nested inner components.

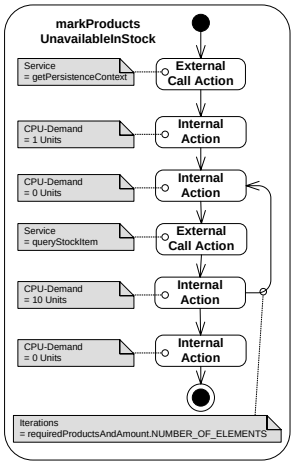
We ran our Java2PCM tool on the Java implementation of CoCoME to generate the RDSEFFs of all services involved in this use case. Notice, that we only used generated RDSEFFs in this case study and did not combine them with manual specified ones. As the dynamic analysis part of our reverse engineering approach is still incomplete and not able to produce the resource demands needed for complete RDSEFFs, we used performance annotations from the CoCoME specification to manually add missing resource demands and incomplete parametric dependencies to the RDSEFF structures generated by Java2PCM.

Java2PCM produced 10 RDSEFF structures for this use case. As an example, Fig. 5 shows the Java code and the corresponding RDSEFF of the service `markProductsUnavailableInStock`. The service invokes two external services (lines 4 and 15), which are mapped to two `ExternalCallActions`. It includes a loop iterating over a list given as a parameter (line 13), visible in the RDSEFF as a backwards connection from the third to the second `InternalAction`, annotated with the number of iterations: the number of elements in the given list. The `InternalActions` stem from summarizing code blocks that do not contain relevant actions. The only manual work required in reconstructing this RDSEFF was giving the concrete resource demands for the `InternalActions`.

In addition to the RDSEFFs, we manually added models of the servers and the number of concurrent requests in the modelled use case. Together, the models formed a complete PCM instance, which we used as input for the Palladio simulation tool. As one of the

```
1 public void markProductsUnavailableInStock(
2         ProductMovementTO requiredProductsAndAmount)
3         throws RemoteException, ProductNotAvailableException {
4     PersistenceContext pctx = persistmanager.getPersistenceContext();
5     TransactionContext tx = null;
6     try {
7         tx = pctx.getTransactionContext();
8         tx.beginTransaction();
9
10        Iterator<ProductAmountTO> productAmountIterator =
11              requiredProductsAndAmount.getProducts().iterator();
12        ProductAmountTO currentProductAmountForDelivery;
13        while(productAmountIterator.hasNext()) {
14            currentProductAmountForDelivery = productAmountIterator.next();
15            StockItem si = storequery.queryStockItem(
16                    requiredProductsAndAmount.getDeliveringStore().getId(),
17                    currentProductAmountForDelivery.getProduct().getBarcode(),
18                    pctx);
19            if(si == null) {
20                throw new RuntimeException( <...> );
21            }
22            // set new remaining stock amount:
23            si.setAmount( si.getAmount() -
24                    currentProductAmountForDelivery.getAmount() );
25            System.out.println( <...> );
26        }
27        tx.commit();
28    } catch (RuntimeException e) {
29        <...>
30    }
31 }
```

(a) Java Code



(b) RDSEFF

Figure 5: CoCoME Service `markProductsUnavailableInStock`

simulation results, Fig.6 shows the end-to-end response time in this use case for the given number of requests as a cumulative distribution function (CDF, blue line, median 6200 ms).

In a former case study [KR08], we had modelled the same use case fully manually, therefore we were now able to compare the results from manual model construction with the results from semi-automatic model construction involving Java2PCM. The CDF from simulating the manual model is shown in Fig. 6 (red line, median 6150 ms). As the curves are widely overlapping, we conclude that the prediction based on semi-automatically built model with Java2PCM achieved a similar accuracy as the prediction based on manually built models.

Table 1 summarizes the most important observations from comparing the manually created models with the semi-automatically constructed ones. The code analysis captured all relevant actions, while the developer modelling the services had omitted some external service calls, or captured their estimated impact in `InternalActions`, thereby losing accuracy. On the other hand, the generated models contain additional actions in comparison to the manually created ones, as the developer could reason that certain parts of the code would have no measurable impact on performance. However, by setting the resource demand of these additional actions to zero, they had no impact on the prediction results except a slightly slower run time of the simulation itself. Java2PCM showed weaknesses in tracing parametric dependencies. Although most of the dependencies were roughly captured, they mostly had to be corrected. For instance, when an element from a list given as a parameter was used in a service, Java2PCM modelled a dependency on the whole list.

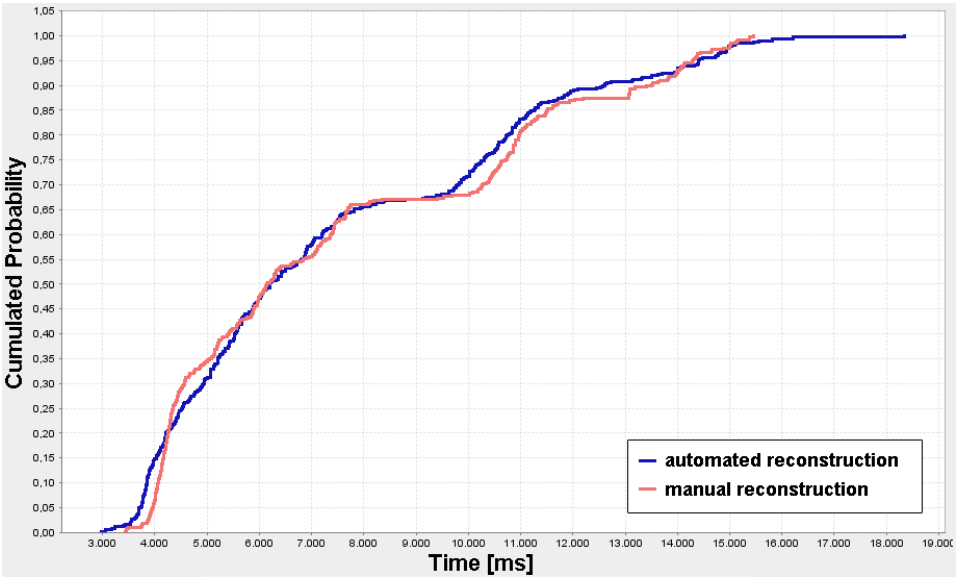


Figure 6: Predictions for execution times as cumulative distribution functions: manually specified versus automatically reconstructed

	Generated model	**Manual model**
Static structure	+ Complete.	− Developer overlooked actions.
Abstraction	○ Too many internal actions, yet not distorting pred. results.	+ Developer could omit non-relevant parts altogether.
Parametric Dependencies	− Roughly captured, with errors and omissions.	+ Correctly modelled.
Time needed	4 person-hours.	40 person-hours.

Table 1: Comparison of generated and reference service models

6 Discussion

This section discusses the benefits of using Java2PCM, open issues of our approach, and general limitations of static code analysis in this context.

Benefits. Using Java2PCM has several benefits over manual modelling. The automatic code analysis is less error-prone that human modelling, which was especially evident in our case study, which involved complex models. During manual modelling, several simplifications had been made to reduce the model's complexity, which had lead to inconsistencies not present in the automatically generated model. Errors still present in the generated models are systematic and can be fixed by for example reanalysing the component boundaries or debugging the tool. Besides higher correctness, using Java2PCM is less labour-intensive than manual modelling (4 vs. 40 person-hours for the use case in our case study) and thus can substantially reduce the costs for performance modelling.

Open Issues. Concerning the current implementation of Java2PCM, the tool so far only supports primitive data types and requires manual effort for structs and objects. As the PCM supports including arbitrary complex data types into RDSEFF specifications, we will improve Java2PCM to include such data types. For example, Java Collections could be translated into collection data types from the PCM. Furthermore, parametric dependencies found by tracing input parameters through the code are still limited to simple cases and require manual effort for complex cases to be processable by the simulation tool.

General Limitations. In general, the halting problem and polymorphism limit static code analyis and require a combination with dynamic analysis techiques. Due to the halting problem, it is not possible to determine loop iteration numbers or dependencies to them in the general case. Polymorphism allows data types that are only known during run time and therefore limits static analyses. Furthermore, tracing parametric dependencies with static analysis is limited to simple cases, as a whole program slice may influence the values of a variable at a particular point in code, which cannot be used as a parametrisation of a performance model and requires manual abstraction.

7 Conclusions

We have presented an approach to reconstruct reusable performance models (called RDSEFFs, from the Palladio Component Model) from Java source via static code analysis. This approach is embedded into a larger reverse engineering framework targeting the integration of the performance properties of existing components into model-based performance predictions. We validated the applicability of our Java2PCM tool by applying it on CoCoME, where less erroneous models could be created in fewer time compared with completely manual modelling.

Our method benefits software architects and performance analysts, who want to assess the expected performance of a component-based system, which includes existing components besides newly designed ones. The resulting performance predictions can reveal performance bottlenecks in an architecture during early design stages and reduce the required effort for fixing performance problems in code.

We intend to further improve the Java2PCM tool to enable analysing more complex architectures and more complex parametric dependencies in single services. The approach will be combined with dynamic code analysis techniques to create a complete tool chain for automatic construction of reusable performance models from code.

References

[BCdK07] Egor Bondarev, Michel R. V. Chaudron, and Erwin A. de Kock. Exploring performance trade-offs of a JPEG decoder using the deepcompass framework. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 153–163, New York, NY, USA, 2007. ACM Press.

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February5–8 2007.

[BLL06] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.

[BMIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[Bor07] Borland. Together. `http://www.borland.com/us/products/together/`, 2007. Last accessed 2007-09-23.

[Cho07] Landry Chouambe. Rekonstruktion von Software-Architekturen. Master's thesis, Institute for Program Structures and Data Organisation, Chair Software Design and Qualitiy (SDQ), Faculty of Informatics, Universität Karlsruhe (TH), Karlsruhe, Germany, May 2007.

[Cod07] CodeSWAT.com. Flowchart4J. `http://www.codeswat.com/cswat/index.php?option=com_content&task=view&id=34&Itemid=55`, 2007. Last accessed 2007-09-23.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9, 2003.

[GW95] Gerhard Goos and William Waite. *Compiler Construction*. Springer, 2nd edition, 1995.

[HC88] Timothy Hickey and Jacques Cohen. Automating program analysis. *Journal of the ACM (JACM)*, 35(1):185–220, 1988.

[IBM07] IBM. Rational Software Architect. `http://www-306.ibm.com/software/awdtools/architect/swarchitect/`, 2007. Last accessed 2007-09-23.

[Kap07] Thomas Kappler. Code-Analysis Using Eclipse to Support Performance Prediction for Java Components. Master's thesis, Institute for Program Structures and Data Organisation, Faculty of Informatics, Universität Karlsruhe (TH), Germany, September 2007.

[KB07] Michael Kuperberg and Steffen Becker. Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs. In Ralf Reussner, Clemens Czyperski, and Wolfgang Weck, editors, *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*, July 2007.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(Issue 7):385–394, July 1976.

[Kos00] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. phd thesis, Institut für Softwaretechnologie, Abteilung Programmiersprachen, Fakultät Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, Germany, Stuttgart, Germany, August 2000.

[Kos05] Rainer Koschke. Rekonstruktion von Software-Architekturen – Ein Literatur- und Methoden-Überblick zum Stand der Wissenschaft. *Informatik – Forschung und Entwicklung*, 19(3):127–140, April 2005. Springer Berlin / Heidelberg.

[KR08] Klaus Krogmann and Ralf Reussner. Palladio - Prediction of Performance Properties. In *The Common Component Modeling Example: Comparing Software Component Models,*, To Appear in LNCS. Springer, 2008.

[Kro07] Klaus Krogmann. Reengineering of Software Component Models to Enable Architectural Quality of Service Predictions. In Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors, *Proceedings of the 12th International Workshop on Component Oriented Programming (WCOP 2007)*, July 2007.

[Obj05] Object Management Group (OMG). UML Profile for Schedulability, Performance and Time. http://www.omg.org/cgi-bin/doc?formal/2005-01-02, January 2005.

[RBK+07] Ralf H. Reussner, Steffen Becker, Heiko Koziolek, Jens Happe, Michael Kuperberg, and Klaus Krogmann. The Palladio Component Model. Interner Bericht 2007-21, Universität Karlsruhe (TH), Faculty for Informatics, Karlsruhe, Germany, October 2007.

[Ros90] Mads Rosendahl. Automatic complexity analysis. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 144–156, New York, NY, USA, 1990. ACM Press.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models*, volume to appear of *LNCS*. Springer, Heidelberg, 2008.

[SF96] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1996.

[TTBS07] Paolo Tonella, Marco Torchiano, Bart Du Bois, and Tarja Systä. Empirical studies in reverse engineering: state of the art and future trends. *Empirical Software Engineering, Springer*, March 2007. Published online first.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Reading, MA, 1996.

[WVCB01] M. Woodside, V. Vetland, M. Courtois, and S. Bayarov. *Performance Engineering: State of the Art and Current Trends*, volume LNCS 2047/2001 of *Lecture Notes in Computer Science*, chapter Resource Function Capture for Performance Aspects of Software Components and Sub-Systems, pages 239–256. Springer, Heidelberg, Heidelberg, 2001.