FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Static Validation of ConQAT Architecture Descriptions

Moritz Marc Beller

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Computer Science

Static Validation of ConQAT Architecture Descriptions

Statische Validierung von ConQAT Architekturbeschreibungen

Author:	Moritz Marc Beller
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens
Date:	September 15, 2011

I assure the single handed composition of this bachelor's thesis, only supported by declared resources.

München, September 4, 2013 Moritz Marc Beller

Acknowledgements

This thesis would not have been possible without the continuous support from my friends, my parents Friedhelm and Erna Beller and my sister Nora.

I am deeply grateful to Fabian Streitel, David Fu, Thomas Kinnen and Jessica Golladay, who reviewed this thesis with great attention to detail.

Finally, I want to express my gratitude to my supervisor Dr. Elmar Jürgens, who created a challenging and always positively motivating working environment, which helped me develop academically and personally. And thanks for the cake!

Abstract

Architecture descriptions are a means of denoting a software project's building blocks on an abstract level. ConQAT is a toolkit with which it is possible to perform a matching between the architecture description and the source code of a project.

In this thesis, we develop two contributions to ConQAT's static architecture validation features. The first contribution defines a restrictivity metric on architectures. The second is an automatic validator for overlapping regular expressions that constitute a common problem of architecture specifications in practice.

We investigate the relevance and runtime of our contributions in a case study on eleven real-world projects. Our results indicate that restrictivity is comparable, yet differing among projects. Moreover, 45% of the architectures contain overlaps.

Further improvement of the proposed algorithms remains an interesting topic for future work: Precision of the restrictivity could be increased and runtime of the overlap detection decreased.

Contents

1 Introduction

"I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind"
Lord Kelvin, 1891 [Kel91]

Every software system can be described on a high level by its architecture, an abstract plan of what the system looks like.

A system architect has to design and maintain the architecture specification separate from the source code. As the project evolves, architecture and source code begin to diverge when no countermeasures are taken [FRJ09].

Software quality measurement systems have been developed which can — among other things — perform an automatic matching between the architecture specification and the actual architecture realised in the source code. One such system is ConQAT, "an integrated toolkit for creating quality dashboards that allow to continuously monitor quality characteristics of software systems." [Con]

In this thesis, we develop two contributions to ConQAT's architecture validation features. The first contribution defines a restrictivity metric on architectures. The second is an automatic static validation checker for overlapping regular expressions that constitute a common problem of architecture specifications in practice.

1.1 Architecture Restrictivity

ConQAT provides the ability to define and evaluate architectures through its Architecture Editor. It features an architecture analysis which can detect violations of the architecture in the source code.

However, having an unviolated architecture is not the key to a well-written software system per se, much less an indicator for a restrictive architecture specification. In practice, we observe that system architects start with a restrictive architecture to begin with. As the system evolves and more code is added, violations in the architecture analysis begin to appear. Often, instead of rethinking the dependency introduced, the programmers modify the architecture specification to allow the new dependency. Over the time, the architecture loses its restrictivity and therefore its expressiveness.

Our goal is to define a restrictivity metric on architectures to be able to identify such misuses.

1.2 Codemapping Overlaps

ConQAT's architecture model comprises components and policies. Policies model allowed and denied accesses between components. Each component must have either a subcomponent or a codemapping. A codemapping is a regular expression that defines which types from the source code belong to the component the codemapping is associated with. Having studied architecture specifications of real-world software systems, we found these contain up to hundreds of codemappings.

A problem arises when there is an ambiguity between the codemappings of different components. The ConQAT architecture analysis then fails because it cannot decide which component the type belongs to. This happens for example when two components have been given the same codemapping by mistake. More subtly, it can also happen if two codemappings overlap. For example, given the codemappings `org.junit.*` and `org.junit.tests.*`, the type `org.junit.tests.validationTester` is matched by both.

It is currently only possible to detect such overlaps when the architecture analysis has failed during runtime.

Thus, our aim is to develop a static validation analysis to guarantee the components do not have overlapping codemappings.

2 Fundamentals

To understand this thesis, profound knowledge in automata theory and in software engineering is required. In this chapter, we briefly summarise the main concepts from these fields of computer science. Additionally, an introduction to ConQAT is given.

2.1 Automata Theory

In this section, we summarise theorems from automata theory that are essential to chapter 4. [HMU79] is an excellent resource on automata theory.

Transformation of Regular Expressions to Automata

Regular expressions and deterministic finite automata (DFA) have the same expressiveness [HMU79]. The former can be translated to the latter by converting the regular expression to a non-deterministic finite automaton (NFA), and then converting the NFA to a DFA [Feg, HMU79].

Overlap of Regular Expressions

Definition 2.1 (Overlap) *An overlap of two regular expressions r_1, r_2 is defined as a string s which lies both in $L(r_1)$ and $L(r_2)$. The two r_1 and r_2 are thus overlap-free iff $L(r_1) \cap L(r_2) = \emptyset$ holds.*

Intersection, Union and Subtraction on Automata

An *intersection* on automata $M = M_1 \cap M_2$ is the operation that returns the automaton M, which accepts $L(M_1) \cap L(M_2)$. In other words, every string accepted by both M_1 and M_2 is also accepted by M.

A *union* over automata $M = M_1 \cup M_2$ is the operation that returns the automaton M, which accepts $L(M_1) \cup L(M_2)$. In other words, every string accepted by either M_1 or M_2 is also accepted by M.

A *subtraction* on automata $M = M_1 - M_2$ is the operation that returns the automaton M, which accpets $L(M_1) - L(M_2)$. In other words, M accepts every string accepted by M_1 but those strings accepted by M_2.

2.2 ConQAT

ConQAT consists of a frontend, its IDE, which is realised as an Eclipse plugin, and its backend, the ConQAT engine. In this thesis, we have expanded both parts, with an emphasis on the engine.

ConQAT Engine

Among other features, the ConQAT engine is capable of calculating metrics on software projects. Usually, these metrics are written as ConQAT processors. ConQAT processors can be combined to form complete analyses in so-called ConQAT blocks. A detailed explanation of this is given in [DFH+10, p. 69f.].

System Architectures

A description of the assembly of ConQAT architecture specifications was already given in chapter 1. In the following, we will further detail architecture specifications and clarify what the assessment of an architecture is.

Architecture Specification

Generally, there are two types of codemappings, *inclusive* and *exclusive* codemappings. If a component does not have at least one inclusive codemapping, then it must by definition have at least one subcomponent, so that the component is not empty. An inclusive codemapping defines which types from the source code are mapped to the component, whereas an exclusive codemapping does the opposite: From the set of all included types, the types that match against the regular expressions of the exclusive codemappings are subtracted.

Definition 2.2 (Component Access) *For two arbitrarily chosen components a and b ($a \neq b$), we say that a accesses or depends on b iff at least one type from a imports or otherwise uses at least one type from b.*

To model which components are allowed to access which other components, *policies* in the form of “allow”, “deny” or “tolerate” edges between components are used. By default, every access between components is forbidden and has to be explicitly allowed through a policy. However, if the components are connected by hierarchy — one is the parent of the other — access is allowed.

Assessment of an Architecture

An architecture specification can be matched against the source code of a project. The artefact generated by an architecture analysis is called the *assessment* of an architecture. The assessment file is an XML description of which types in the source are mapped to which component in the specification. It also shows so-called architecture violations. An architecture violation occurs when one component depends on another component in the source code, but is not allowed to do so according to the architecture specification.

To be able to assess projects in different programming languages, there are predefined architecture analyses in ConQAT (e.g. the “JavaArchitectureAnalysis”).

3 Architecture Restrictivity Metric

This chapter describes the design of an architecture restrictivity metric and its implementation into ConQAT.

3.1 Approach: Design of a Restrictivity Metric

The architecture restrictivity metric, despite its name, is a combination of an architectural metric and a program code metric. This provides the possibility to see how well the architecture design is realised in the program code. A low restrictivity could be caused by three reasons: A low-restrictive architecture, a source code which puts almost all code into one component, or a combination of both. A valid criticism to this design may be that it obfuscates the clear separation between architecture and program code. However, little value lies in an architecture that is not taken into account by the programmers, and vice versa.

The architecture restrictivity metric is based on an architecture specification — from which the access policies are extracted — and an assessment of the source code of the project. From the assessment we retrieve the types of the system and which components in the architecture these are mapped to.

Definition 3.1 (Architecture Restrictivity) *By "Architecture Restrictivity", we understand the probability that for two arbitrarily chosen types a and b ($a \neq b$), type a may not access type b in the underlying architecture.*

Let ρ be the symbol for the Architecture Restrictivity of a system.

With ρ being a probability, $0 \leq \rho \leq 1$ holds. The extreme values can be explained as follows: A value of $\rho = 0$ characterises a completely unrestrictive system, where all types can access each other. $\rho = 1$ characterises a perfectly restrictive system, in which every type can only access itself. These extreme values are of course only of theoretical interest and not to be found in practice.

To calculate ρ on an architecture, we use the following formula:

Definition 3.2 (Architecture Restrictivity)

$$\rho = \frac{\textit{Number of Denied Pairs}}{\textit{Number of All Possible Pairs}} = 1 - \frac{\textit{Number of Allowed Pairs}}{\textit{Number of All Possible Pairs}}$$

One of our explicit goals in designing the metric was to ensure that it behaves semantically correct upon changes in the system: As more allow policies are introduced into an otherwise unchanged system, the metric should reflect this and decrease. More allow policies translate to more allowed type pairs between components that previously were not allowed to access each other. The second numerator in definition 3.2 increases. As a consequence, ρ decreases. As the metric is transitive regarding the allowed types, the inverse is also true: ρ increases as allow policies are removed from the specification.

Nota bene: The opposite, "as more types are introduced into an otherwise unchanged architecture, the metric should reflect this and decrease", is not always true.

Definition 3.3 (Component Cardinality) *Given a component a, let $|a|$ be the number of matched types of the component a.*

When n more types are introduced into a component a, the number of allowed inner-component type pairs increases by $|a+n| * |a+n-1| - |a| * (|a|-1)$. The number of externally available types from a increases, too, adding more allowed pairs for every component that depends on a. However, because the total number of possible type pairs might increase along with it, this increase might be greater than the increase caused in the allowed pairs. Therefore, ρ might increase.

Changes in the architecture specification usually have more impact on ρ than changes in the source code. Imagine two components of size 100 each. Introducing a single allow policy in the specification between the two components leads to 10,000 new allowed pairs. Introducing one more type in the source code leads to an increase of only 100 new allowed pairs. Furthermore, a source code change relevant to ρ is usually associated with hundreds of man hours, whereas the architecture specification can be changed in a minute. This emphasises that changes to the architecture specification be well thought about.

Manual Calculation of the Restrictivity

To get a better understanding of the metric, we give a manual validation of the calculation for ρ in the following. The scenario was also integrated as a system test called "accessToLib" with a freely invented architecture to test whether our processor is inline with the manually verified value for ρ.

Component Name	Matched Types
App	2
special-app	3
Lib	4

Table 3.1: The matched types of the components in figure 3.1.

We calculate ρ by summarising all allowed and then all disallowed type pairs: All types which are in the same component may access all other types of the same component. That makes for $2 * 1 + 3 * 2 + 4 * 3 = 20$ allowed type pairs (App, special-app, Lib). Allowed access through the policy from App to Lib is granted for $(2+3) * 4 = 20$ type pairs (special-app is a subcomponent of App, and the policy stretches to subcomponents, too). Allowed accesses because of a sub or super type account for $(2 * 3) * 2 = 12$ pairs (special-app and App). This makes a total of 52 allowed type pairs.

Denied pairs are only from Lib to App, which are $(2 + 3) * 4 = 20$ type pairs. Thus, the number of all possible pairs is $52 + 20 = 72$.

Using definition 3.2, we receive $\rho = 1 - \frac{52}{72} \approx 0.278$, which is exactly the result of our "RestrictivityMetricAnalyzer" processor.

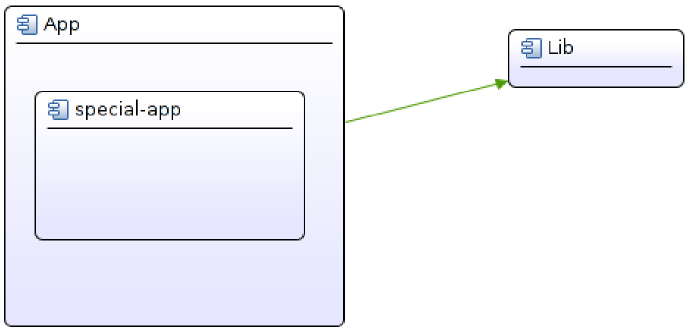


Figure 3.1: The architecture of the "accessToLib" scenario.

3.2 Implementation

The restrictivity analysis was implemented into ConQAT as the "ArchitectureRestrictivityAnalyzer" processor.

To check whether the requirements specified against the metric hold, eight system tests were added. Additionally, the scenario described in section 3.1 was added as a system test.

To be able to conveniently use the processor within ConQAT, a number of predefined blocks were added. The "ArchitectureRestrictivityMetric" block is the base building block for all restrictivity analyses. In this block, the complete chain of reading an architecture, setting up the "ArchitectureRestrictivityAnalyzer" processor, assessing and colouring of the result and filtering the output is taking place. Since setup of the "ArchitectureRestrictivityAnalyzer" processor differs substantially based on the source for the analysis, there are two further blocks that perform a restrictivity analysis based on either a Java source ("JavaSourceRestrictivityAnalysis") or an assessment file ("AssessmentFileRestrictivityAnalysis").

Algorithm

Our algorithm calculates the number of allowed and possible pairs and then calculates ρ by means of a simple division. To infer the allowed and denied access policies, the architecture specification is used. The assessment is then only needed to infer how many types are matched to the components. Thus, an architecture violation in the source code — for example caused by a dependency of types from components that may not access each other according to the architecture — is not taken into account for the calculation of the restrictivity. The value of the metric can therefore only be considered valid if the architecture is not violated.

We prefer to count the number of allowed pairs over the number of denied pairs, as it is a more straight-forward and intuitive approach to count the allowed edges. Furthermore, most architectures have more (implicit) deny policies than allow policies, so that computation of the allowed pairs is shorter, too.

Instead of comparing every matched type with every other — the intuitive approach to calculate the metric — ρ is calculated on the more abstract component level. A comparison is made between all two components a, b.

- If $a \neq b$, then the number of possible types from b a type from a can access, is simply $|b|$. Hence, the total number of possible pairs for the two components is $|a| * |b|$.
- If $a = b$, then the number of possible type pairs is $|a| * (|a| - 1)$. This is because for any type x in a, there are only $|a| - 1$ other accessible types in a.

If b is either a subcomponent of a, or there is an allow policy from a to b (or any of a's parents to b or any of its parents, as long as there is no deny policy), then the calculated type pairs are also added to the allowed types. Otherwise, the pairs are not allowed and only added to the possible type pairs.

Having counted the number of all possible and all allowed type pairs, we use the formula from definition 3.2 to calculate ρ.

In comparison to the naive algorithm of traversing every type pair (which lies in $O(n^2)$, with n being the matched types), this algorithm is several orders of magnitude faster. However, it is still in $O(m^2)$, where m is the number of components ($m \ll n$).

Validation of the "ArchitectureRestrictivityAnalyzer"

To validate the result of the optimised processor, a naive implementation of the algorithm was created as a second processor. This processor builds a pair list of all possible matched type pairs in the system. Hence, the number of all possible types is simply the size of the pair list. It then iteratively checks for each type pair whether it is allowed given the architecture specification. We performed checks against nine of the study objects from chapter 5 (for two systems the algorithm ran for longer than an hour and was thereafter stopped) and against the system described in section 3.1. The calculated restrictivity was identical to the metric obtained from the "ArchitectureRestrictivityAnalyzer".

4 Codemapping Overlap Analysis

This chapter describes the design and implementation of an overlap detection for architecture descriptions in ConQAT. An algorithm in pseudo-code reveals the details of the implementation.

4.1 Approach: Design of an Overlap Detection Algorithm

In this section, we first describe the basic idea behind the algorithm for the overlap detection. Hereafter, the step in the algorithm where excluded codemappings are taken into account is described.

The Overlap Algorithm Principle

To detect overlaps in the architecture, it is principally necessary to compare each codemapping r_1 with each other codemapping r_2. This iteration procedure is shown in algorithm 1 on the component level.

Algorithm 1 IterationAlgorithm

```
Require: $allComponents$ containing all components of the architecture in a list
  for $component \in allComponents$ do
    for $comparisonComponent \in allComponents$ do
      if $component \neq comparisonComponent$ then
        checkComponentsForOverlaps($component, comparisonComponent$)
      end if
    end for
  end for
```

Then, a check for overlaps of the two codemappings has to be performed. Since the codemappings are not mere text strings, but regular expressions, a direct comparison of the two is impossible. Instead, a conversion to finite automata is applied to r_1 and r_2 separately. The automata R_1 and R_2 are semantically equivalent to the regular expressions they were converted from (cf. section 2.1). As depicted in section 2.1, an intersection of two automata returns the commonly accepted language of the two intersected finite state machines. We intersect the automata R_1 and R_2 and call this resulting automaton the "intersection automaton" of the two regular expressions. This step is depicted in figure 4.1 and described in algorithm 3.

For the intersection automaton, we can then infer the set F of accepting states. There are two cases:

Algorithm 2 checkComponentsForOverlaps(component, comparisonComponent)

```
Require: $component \neq comparisonComponent$
  $componentIncludeRegExen$ ← include codemappings from $component$
  $componentExcludeRegExen$ ← exclude codemappings from $component$
  $comparisonIncludeRegExen$ ← include codemappings from $comparisonComponent$
  $comparisonExcludeRegExen$ ← exclude codemappings from $comparisonComponent$

  $componentExlucdeAutomaton$ ← exclusionAutomaton(componentExcludeRegExen)
  $comparisonExlucdeAutomaton$ ← exclusionAutomaton(comparisonExcludeRegExen)

  for $regEx \in componentIncludeRegExen$ do
    $R_1$ ← toAutomaton(regEx)
    $R_1 \leftarrow R_1 - componentExlucdeAutomaton$
    for $comparisonRegEx \in comparisonIncludeRegExen$ do
      $R_2$ ← toAutomaton(comparisonRegEx)
      $R_2 \leftarrow R_2 - comparisonExlucdeAutomaton$
      $overlap$ ← detectOverlap($R_1, R_2$)
      if $overlap$ then
        return $overlap$ {Returns the first found overlap with a warning text.}
      end if
    end for
  end for
```

- If $F = \emptyset$, then there are no overlaps between the two regular expressions. Search for overlaps is proceeded with the next codemapping r_2 not yet compared to r_1.
- If $F \neq \emptyset$, then there is at least one possible path through the intersection automaton. In this case, we save the shortest possible path through the automaton along with the overlapping codemappings as an error message for the components containing each codemapping. The shortest path through the automaton serves as an example with which system architects can conveniently see the overlap between the two codemappings. The search for further overlaps for codemapping r_1 is aborted.

The loop continues until all codemappings have been compared to each other with the above algorithm. Codemappings from the same component are not compared, as it does not pose a problem when there are overlaps: All types would be matched to the same component anyway.

Algorithm 3 detectOverlap(R_1, R_2)

```
  $R \leftarrow R_1 \cap R_2$
  if hasNoAcceptingStates($R$) then
    return
  else
    return shortestAcceptingExample($R$)
  end if
```

We deduce from the depicted algorithm that our algorithm is in $O(n^2)$, where n is the total number of inclusive codemappings in the architecture.

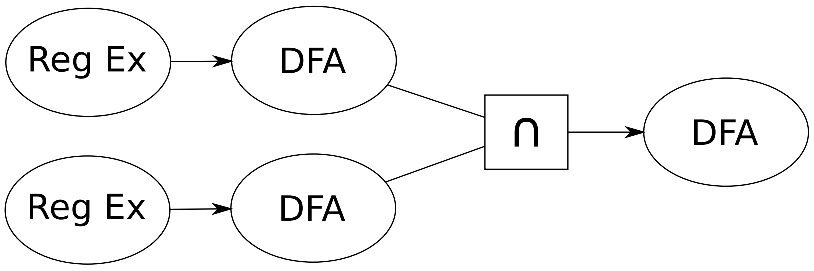


Figure 4.1: Principle of the algorithm: Conversion to DFAs, performing the intersection on the two DFAs, returning an intersected DFA. [Mø11] has built-in support for the conversion from a regular expression to a DFA, the intersection of DFAs and for returning a given DFA's shortest accepting example.

Exclude Codemappings

The actual algorithm implemented differs from the simplified textual description given in section 4.1 in that it has to consider not only inclusive, but also exclusive codemappings. Exclusive codemappings specify which types are not mapped to the components (for a more detailed description refer to section 2.2). To generate valid overlap results, it is crucial to incorporate the exclusive codemappings into the algorithm.

Definition 4.1 (Exclusion Automaton of a Component) *Given a component a, each exclusive codemapping in a is converted to a DFA. All DFAs are joined to create one common DFA that matches any of the excluded expressions. We call the resulting automaton a component's "Exclusion Automaton".*

Therefore, for every codemapping checked — be it r_1 or r_2 — the exclusion automaton of the component it belongs to is constructed. The exclusion automaton is then subtracted from the automaton the codemapping has been converted to. This way, the exclude codemappings are regarded in algorithm 2. The resulting automata overwrite R_1 and R_2 which are then analysed as described in section 4.1.

Algorithm 4 exclusionAutomaton(regExList)

```
$excludeAutomaton \leftarrow$ new Automaton

for $regEx \in regExList$ do
    $R \leftarrow$ toAutomaton($regEx$)
    $excludeAutomaton \leftarrow excludeAutomaton \cup R$
end for
return $excludeAutomaton$
```

4.2 Implementation

Use of `dk.brics.automaton` Library

For the algorithms 2, 3 and 4, functions from the automaton theory are indispensable, namely toAutomaton, shortestAcceptingExample and the "$\cup$", "$\cap$" and "$-$" operations on automata. Instead of writing our own implementation of an automaton library, we used the third party package `dk.brics.automaton` from [Mø11]. The package is distributed under a BSD license, so its integration into ConQAT is legally allowed.

Integration of Overlap Algorithm into ConQAT

One of this thesis' main goals was to be able to use the overlap detection from the engine as a processor as well as to integrate it into the IDE of ConQAT, namely the Architecture Editor. The core functionality of the algorithm is implemented in the ConQAT engine. Since access from the IDE projects to the engine is allowed, we can thereby keep code duplication to a minimum. Code has been added to ConQAT in the `org.conqat.engine.architecture.overlap` package for the engine and `org.conqat.ide.architecture.databinding` and `org.conqat.ide.architecture.model` for the IDE.

However, the IDE and engine projects use different type hierarchies so that the algorithm had to be kept generic by introducing interfaces. Fortunately, the IDE closely resembles the type hierarchy from the engine so most of the functionality we relied on in the engine was already implemented. Existing types from the engine and the IDE were changed to implement the new interfaces. This way, the actual algorithm implemented in the engine is independent of both type hierarchies. It is also granted that the results from both analyses be identical.

The integration into the IDE is shown in figure 4.2. The validation for overlaps is automatically performed on every save of the architecture. By clicking the "Cancel" button, a validation that takes too long can be aborted by the user.

Integration into ConQAT Tests

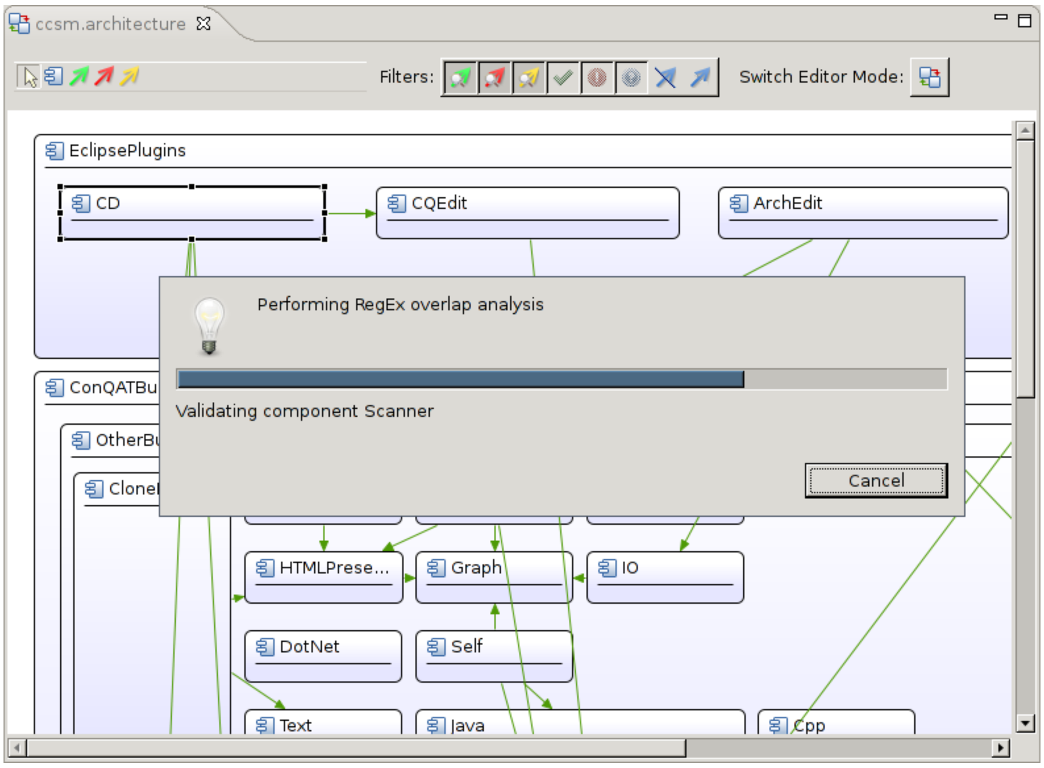
To ensure the correctness of the generated source code, both unit and system tests were added to ConQAT.

System Tests

The system tests consist of one smoke test and one regression test. The smoke test analyses a large architecture to check the stability of the algorithm. In order to ensure none of the many refactorings employed during the code review phase has broken the overlap detection, a regression test on a small, freely invented architecture was introduced.

Unit Tests

Following the usual ConQAT programming paradigm, JUnit tests were added to test parts of the functionality created for the overlap detection. The IDE code could not be tested, as



Performing RegEx overlap analysis

Validating component Scanner

Cancel

Figure 4.2: Architecture Editor displaying the overlap analysis progress monitor.

there is no support for proper UI testing in ConQAT at the date of writing this thesis.

Syntax Comparison Between `java.util.regex` and `dk.brics.automaton`-Style Regular Expressions

The regular expression syntax supported by `java.util.regex` and `dk.brics.automaton` has a few subtle differences. More specifically, the regular expression syntax supported by [Mø11] is somewhat limited compared to [Jav]. It is important to find and eliminate those differences since the mapping process described in 2.2 is performed using Java's `regex`-package — whereas the conversion to automata is done using the library [Mø11]. If the overlap detection algorithm interprets a regular expression differently than the mapping process, possible overlaps might not be detected. Furthermore, if the predefined character sets are not the same, false examples might be returned. Therefore, a conversion utility class was written that realises the transformation rules summarised in table 4.1.

Construct	`java.util.regex` syntax			Equivalent `dk.brics.automaton` syntax
Standard RegEx Operations	`| &`			Supported
Character Class Operations	-, ^, &&, inner-[]			-, ^, unsupported intersection, unsupported concatenation
Predefined Character Classes				Converted
	`\d \D`			`[0-9] [^0-9]`
	`\w \W`			`[A-Za-Z_0-9] [^A-Za-z_0-9]`
	`\s \S`			`[ \t\n\x0B\f\r] [^ \t\n\x0B\f\r]`
Quantifiers	x?	x??	x?+	x, once or not at all, syntax supported
	x*	x*?	x*+	x, zero or more times, syntax supported
	x{n}	x{n}?	x{n}+	x, at least n times, syntax supported
	x{n,m}	x{n,m}?	x{n,m}+	x between n and m times, syntax supported
	greedy	reluctant	possessive	No support for matching strategy. Unsupported greedy and possessive quantifiers
Capturing groups	`((a))`			Supported
Backreferences	`(\d\d)\1`			Not supported
Boundary matchers	`^, \$, \b, \B, \A, \Z`			Not supported

Table 4.1: Table depicting the main syntax differences between `java.util.regex` and `dk.brics.automaton`.

5 Case Studies

To evaluate the restrictivity and overlap analysis, we conducted a case study with eleven current software projects.

5.1 Research Questions

We investigate the following five research questions:

RQ 1.1 *How do the investigated systems compare regarding their restrictivity?*
Since we do not know which values we could expect from our metric for real-world specifications, we have to analyse and compare them. How well does the metric distribute across different systems?

RQ 1.2 *Is the restrictivity dependent on the system size?*
Due to different effects in the design of the metric, it could be that larger or smaller systems automatically have a better chance of receiving a higher restrictivity metric. Ideally, to make the metric comparable amongst systems of all sizes, we would not want such effects.

RQ 1.3 *How long is the runtime of the restrictivity analysis?*
The runtime has a relevance on the possible usage within a ConQAT environment. Could it be that the processor takes so long that a restrictivity analysis can only be made on a nightly basis for large projects?

RQ 2.1 *Do overlaps occur in real-world architecture specifications?*
The second sub-goal of this case study is to find out whether there exist overlapping codemappings in real-world architecture specifications. If so, do they constitute an actual fault in the architecture specification?

RQ 2.2 *How long is the runtime of the overlap analysis? Is it dependent on characteristics of the analysed system?*
The runtime has a relevance on the possible usage within a ConQAT environment. Could it be that the processor takes so long that an overlap analysis can only be made on a nightly basis for large projects? This would also make the instant validation check in the IDE pointless. The runtime of the processor is of interest to identify possible algorithmic improvements.

5.2 Study Design

To conveniently analyse the study objects, we generated a ConQAT block that comprises a statistics part with the "ArchitectureStatisticsAnalyzer" and "MatchedTypesAggregator", a restrictivity analysis — the "AssessmentArchitectureRestrictivity" — and an overlap detection — the "ArchitectureOverlapAnalyzer" (cf. figure 5.1). This block is executed via a ConQAT run configuration on each project.

For the reported runtimes, only the exact processor execution time is measured. The minimum of five successive runs is taken. This is because hard drive operations or other processes can slow down execution of ConQAT blocks. The basis for the runtime results was a 2.2 GHz Core2Duo processor with 3 Gigabyte RAM.

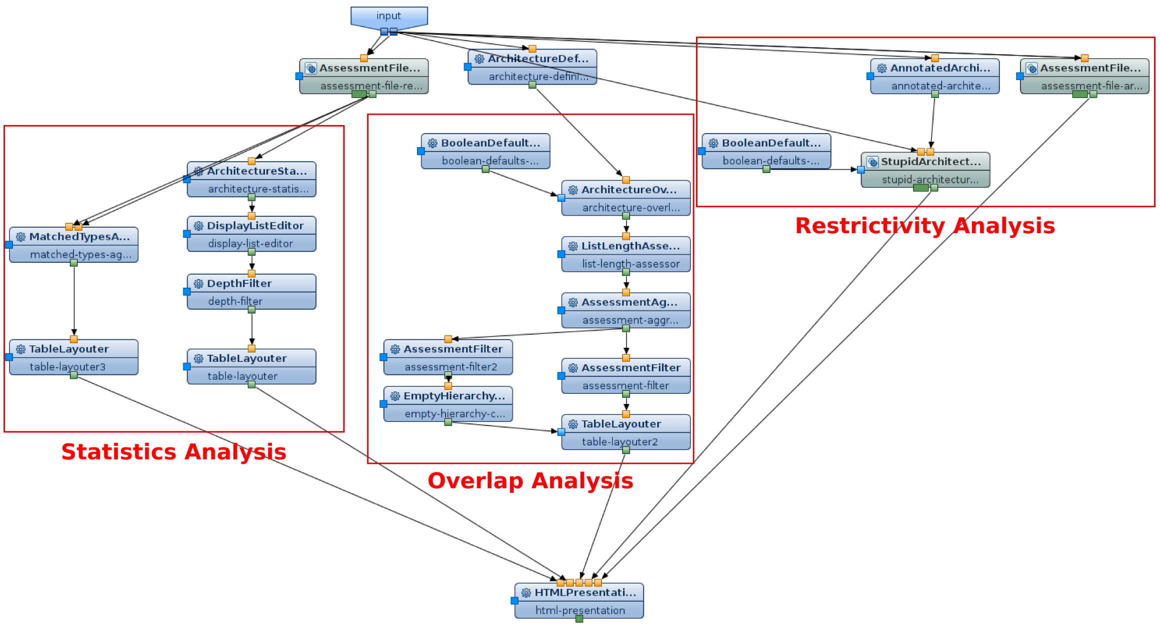


Figure 5.1: The CaseStudy block used to analyse every study object.

RQ 1.1 The restrictivity was determined by the created "AssessmentArchitectureRestrictivity" processor. Based on the collected ρ values we calculated the mean and made an assessment suggestion for three restrictivity categories. The mean might only be considered within the scope of this case study as it is not representative of classifying other projects into the three proposed categories. An outlier is defined as any ρ which is not within $\mu \pm \sigma$. We explain outliers from the mean by a manual inspection of the assessment file in the Architecture Editor, supported by the results of the "ArchitectureStatisticsAnalyzer".

RQ 1.2 We performed a correlation of the variables "matched type size" and "restrictivity" (as obtained from **RQ 1.1**).

RQ 1.3 We extracted the runtime of the created ConQAT processor from the ConQAT HTML execution time map and estimated whether it could become too time-consuming.

RQ 2.1 We measured the number of overlaps by virtue of our created "ArchitectureOverlapAnalyzer" processor. The reported overlaps are the number of overlapping components, and not the number of overlapping codemappings (which would be greater, or equal to the number of overlapping components). Thus, if two components have one overlapping codemapping, the reported number is two. However, if the two components share more than one overlapping codemapping, these additional overlaps are not counted, and the reported overlap number remains two.[1] We examined the overlaps detected and classified them into clusters, if they follow the same pattern.

RQ 2.2 We extracted the runtime of the created ConQAT processor from the ConQAT HTML execution time map and estimated whether it could become too time-consuming. If the architecture contained overlaps, these were removed to the best of our knowledge and the runtime on the overlap-free architecture was measured. Comparing runtimes on some architectures which contain overlaps and some which do not would have led to wrong results due to algorithmic optimisation upon the discovery of an overlap.

5.3 Study Objects

Ten of the examined systems stem from our industrial partner Munich Re and are C# projects. The remaining system is ConQAT itself, written in Java. The systems are used in a productive environment, yet still undergo maintenance and development. To conform to a non-disclosure agreement with Munich Re, the source data is not publicly available and the results have been anonymised. The data we received from Munich Re consisted of an architecture definition and an assessment file (as obtained by running an "ILArchitectureAnalysis") for each project. The ConQAT architecture and assessment files were taken from the internal overnight analysis run on August, the 23rd of 2011. Although these are not publicly available either, the ConQAT source itself is (see section A.1).

To obtain an overview of the analysed systems, the ConQAT processor "ArchitectureStatisticsAnalyzer" was written. It gathers statistical data from the assessment and architecture files. An overview of the systems' characteristics, generated by this processor, is given in table 5.1. To get a general idea of the analysed systems, we measured the number of components as well as codemappings and matched types.

The number of components ranges from 30 to 99 ($\mu = 49.5$, $\sigma = 20.3$), the number of codemappings from 23 to 214 ($\mu = 71.7$, $\sigma = 53.4$). The matched types parameter, indicative of the actual size of a project, ranges from 307 to 8,407 ($\mu = 3025.4$, $\sigma = 2455.8$). The smallest and largest systems differ greatly in their matched type size, with System I being more than 27 times the size of System B. Due to the large standard deviations present in codemapping and matched type sizes (σ is 75% and 81% of μ respectively), we can conclude that we research a broad variety of heterogeneous systems ranging from small to big projects. The number of matched types loosely relates to the number of codemappings (Pearson product-moment correlation coefficient $r = 0.63$, confidence $\alpha = 0.05$), so that

[1]The "ArchitectureOverlapAnalyzer" processor reports the additional overlaps in the overlap list. Report is limited to one overlap per codemapping (see section 4.1).

larger software systems show the tendency to have more codemappings in their architecture specification.

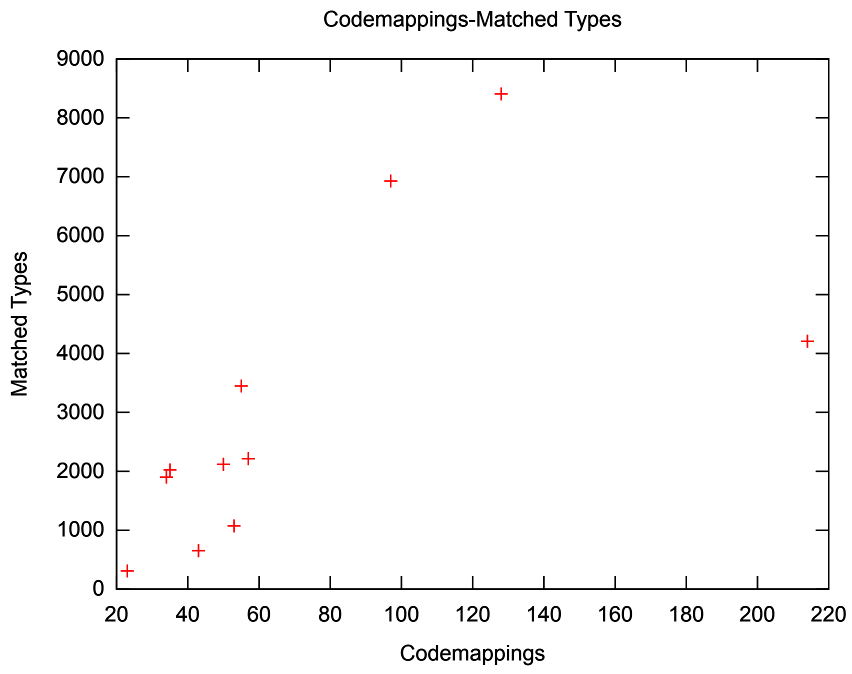


Figure 5.2: Plot of the variables "number of codemappings" and "matched types". The variables show a significant linear correlation (Pearson's $r = 0.63$).

5.4 Architecture Restrictivity

Results: RQ 1.1

At a range of 0.483 to 0.804, the metric distributes nicely for the examined study objects. The calculated average for ρ is $\mu = 0.65$, the standard deviation $\sigma = 0.1$. Subsequently, we assess projects with $\rho <= 0.55$ as unrestrictive (red), $0.55 < \rho <= 0.75$ as restrictive (yellow) and $\rho > 0.75$ as very restrictive (green). In the following, we will inspect Systems D and H which have a significantly lower restrictivity than average, and System C which has an above-average restrictivity.

System C

System C, although it is the second-smallest system analysed in this study, has a relatively large number of components at 37. The greater the number of components an architecture has, the smaller is the number of dependencies allowed, as long as not too many access policies are used in return. The architecture is split into eight top level components, all of which have a roughly equal size of 30 to 90 matched types. The only exceptions are one slightly bigger component A at size 273 and one very small component B at size 5.

System Characteristics				Restrictivity Analysis		Overlap Analysis	
System Name	Components	Codemappings (Inclusive/Exclusive)	Matched Types	Restrictivity	Runtime (in ms)	Overlaps	Runtime (in ms)
System A	41	34 (32/2)	1,900	0.631	85	10	4,920
System B	34	23 (22/1)	307	0.714	39	0	4,830
System C	37	43 (32/11)	653	0.804	55	0	9,299
System D	39	53 (34/19)	1,072	0.533	84	2	30,514
System E	35	55 (44/11)	3,449	0.598	93	0	10,563
System F	99	214 (119/95)	4,207	0.641	294	76	65,751
System G	69	57 (57/0)	2,215	0.577	203	0	7,792
System H	30	35 (31/4)	2,024	0.483	61	2	7,536
System I	71	128 (105/23)	8,407	0.711	296	0	105,889
System J	46	97 (73/24)	6,927	0.774	160	11	26,853
ConQAT	43	50 (50/0)	2,118	0.722	103	0	6,797

Table 5.1: The 11 study objects (System A-J, ConQAT) along with results from the overlap and restrictivity analysis

In contrast to System D, no allow policies between the top-level components are specified. Fine granular, subcomponent access has been modelled instead. All of the high level components (with the exception of B) are further divided into two to six subcomponents, increasing ρ even further.

System D

In System D we find an architecture that is essentially split into three big components A, B and C, containing 491, 292 and 260 matched types each. Component C with 260 matched types is not further split up, thus considerably decreasing the restrictivity of the architecture. The components B and C are given top-level access to the A-component, so that every type they contain may access every type in component A. This generates a total of $260 * 491 + 292 * 491 = 271,032$ allowed dependencies (this is 23.6% of the total type pairs, and causes a decrease of ρ by 0.236), a good deal of which is probably unnecessary, if access to A had been modelled more fine granularly.

System H

System H features five top level components, three of which have roughly the same size with 650 types. Allow policies between three of the top level components are modelled. Removing these three edges leads to $\rho = 0.798$ ($\Delta\rho = 0.315$). This is a situation comparable to System D. Additionally, there are two rather large subcomponents — at 505 and 338 matched types respectively — with no further subdivision. After removing these, ρ rises by 0.065.

Results: RQ 1.2

The two variables "matched types" and "restrictivity" show a weak, insignificant correlation (Pearson's $r = 0.25$, $\alpha = 0.1$). We fail to reject the null hypothesis that matched types and restrictivity do not correlate and can therefore answer **RQ 1.2** negatively: The metric is not related to the system size.

Results: RQ 1.3

At a maximum runtime of merely 0.3 seconds, the optimisations in the algorithm for the computation of the architecture restrictivity show effect. The runtime of this processor is so short, that we dismiss **RQ 1.2**: Analysis-runtime even for very large software projects is well under a second.

Discussion

As we have seen in **RQ 1.1**, outliers (both higher and lower than average values) can be explained by manual inspection, supporting the requirement that the metric be comprehensible. Given the results from the manual inspection of our study objects, in order to receive a higher ρ, it seems to be better to have several top-level components of equal size than only two to three which are then specialised. If these top-level components then have allow policies between them, ρ decreases even further, a pattern observed in both below-than-average performing Systems D and H.

RQ 1.2 shows no significant correlation between the system size and the metric. This enables us to apply the metric across all systems and compare them among each other.

Furthermore, runtime of the restrictivity analysis is negligible (cf. **RQ 1.3**) so we can analyse projects of arbitrary sizes.

5.5 Overlapping Codemappings

From our experience in working with ConQAT and communicating with others working with ConQAT, we know that overlaps are a significant problem in architecture specifications. With the help of this case study, we want to verify and quantify this statement based on empirical data.

The architecture files used in this study had been generated using ConQAT releases without support for automatic overlap detection (ConQAT 2011.7 is the first version to support overlap detection). They were then analysed with the help of the newly written ArchitectureOverlapAnalyzer.

Results: RQ 2.1

We found that five out of the eleven architecture files analysed show overlaps in their codemappings. All of the found overlaps are real errors. No false positives were found, i.e. all the reported overlaps represent genuine problems. Some of the overlaps are caused by wrong syntax (forgetting to escape a character), whereas most are caused by regular expressions that are too greedy. With the exception of System F, overlaps are mostly limited to a small subset of components. Often, these components are subcomponents and share a common parent. When building parallel structures in architecture descriptions, there seems to be a tendency to copy the mistakes (System J, System A).

Therefore, **RQ 2.1** can be answered positively.

System A

System A has four clusters of overlaps, with overlappings limited to the scope of each group. Two analogous clusters are of size three and the other two analogous clusters of size two. All of the overlaps reported show the same pattern: Subcomponents have identical codemappings associated with them.

System D

The reported results for architecture D appear to be false positives at first. There is a component “No namespace” which should match every type that has no namespace and thus belongs to the default package. An overlap is reported to have occurred with the “No namespace” component and another component “APackageComponent” from the system. However, when inspecting the actual codemappings of the “No namespace” component we find that the regular expression used for this is

Listing 5.1: Faulty regular expression for component "No namespace"

```
[^.]+
```

This is a somewhat peculiar construction which is difficult to interpret at first: It is automatically resolved to

Listing 5.2: Corrected regular expression for component "No namespace"

```
[^\.]+
```

By escaping the dot, it is clear that a simple, plain dot is meant – and not the all-matching character ".". This is an interpretation of the regular expression according to `java.util.regex`. However, our Automaton library is not capable of auto-escaping, thus interpreting regular expression 5.1 in a different manner.

Component "APackageComponent" contains a regular expression

Listing 5.3: Faulty regular expression

```
APackageName..*
```

Because `.` bears a special meaning, it is necessary to escape every dot that should be interpreted as a plain dot. Thus, the correct regular expression should read

Listing 5.4: Corrected regular expression

```
APackageName\..*
```

We found the corrected regular expression to be in line with all other regular expressions in this architecture: They all feature a leading backslash before a dot. So, it was forgotten to add the escape character in this regular expression.

Summarising our findings for System D, we detect two problems:

1. The dot in the character class in regular expression 5.1 of component "No namespace" should have been escaped, as shown in regular expression 5.2.
2. A dot in the codemapping for component "APackageComponent" was unescaped, when it should have been escaped, as shown in regular expression 5.4.

System F

System F has one cluster of overlaps. A component, called default component, has several regular expressions, one of which is:

Listing 5.5: Regular expression

```
[a-zA-Z]+
```

Since there are 75 other components with codemappings that consist of only letters in the architecture, there are 76 overlappings. A solution is to remove regular expression 5.5. The system architect has to be asked whether there are types he wants to match with this expression. The assessment file shows there are none at the moment.

System Characteristics		Overlap Analysis	
System Name	Codemappings (Inclusive/Exclusive)	Runtime (in ms)	Δ Runtime (in ms)
System A	34 (32/2)	4,935	15
System B	23 (22/1)	4,830	
System C	43 (32/11)	9,299	
System D	53 (34/19)	32,378	1,864
System E	55 (44/11)	10,563	
System F	213 (118/95)	307,766	242,015
System G	57 (57/0)	7,792	
System H	36 (32/4)	6,409	-1,127
System I	128 (105/23)	105,889	
System J	122 (73/49)	123,746	96,893
ConQAT	50 (50/0)	6797	

Table 5.2: The 11 study objects (System A-J, ConQAT) with overlaps removed from System A, D, F, H and J. We are convinced the negative delta for System H is insignificant and lies within the normal variability of the runtime of ConQAT processors.

System H

The overlaps for architecture H are analogous to those reported for System D.

System J

System J has three clusters of overlaps in total. The architecture has several non-obvious overlaps spread over the whole description. In particular, there is one cluster with analogous subcomponents that share expanded regular expressions with five top-level components in the architecture. Additionally, there is another cluster consisting of two subcomponents with overlaps.

Results: RQ 2.2

The runtime results of the algorithm show room for improvement. The runtime for overlap-free architectures ranges from a few seconds for some systems (System A, B, G, H) to a few minutes (System F, cf. table 5.2).

A very strong, significant correlation between the number of codemappings and the processor runtime could be determined (Pearson product-moment correlation coefficient $r = 0.97$, confidence $\alpha = 0.01$).

Overall, **RQ 2.2** must be answered differentiatedly: For many architectures in small to medium-sized projects, the runtime of the overlap analysis is well under 20 seconds. Architectures with many codemappings ($n > 50$) will experience longer runtimes that might grow problematic for instant validation of architectures upon saving them.

After removing the overlaps, the runtime showed two different behaviours. For some systems it stayed nearly identical (System A, D, H). Others experienced considerably longer computation times (System F, J).

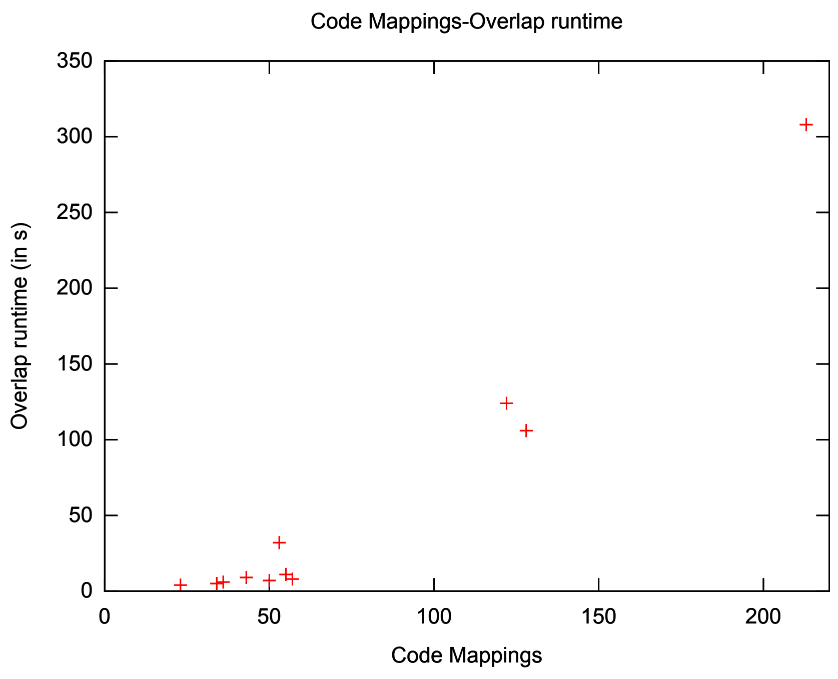


Figure 5.3: Plot of the variables "number of codemappings" and "overlapruntime". The variables show strong linear correlation (Pearson's $r = 0.97$).

Discussion

In this study we counted the number of overlapping components and not the number of overlaps itself. If we wanted to receive a correct number for all overlaps, algorithmic optimisations for the bad case would have been needed to be removed. Therefore, we regressed to the number of overlapping components.

As depicted in section 4.1, for algorithmic reasons, the "ArchitectureOverlapAnalyzer" reports a maximum of one overlap per codemapping. This algorithmic design is reasonable, since in our study we did only encounter a single case where a codemapping had an overlap with more than one other codemapping (System J). Readability of error results might drop if more than one overlap per codemapping were reported. Our study shows that after removing the reported overlaps, all of the architectures but one (System J) instantly reported overlap-free. Only for System J repeated execution of the overlap analysis was necessary to resolve all overlaps. This indicates that there is not usually more than one overlap between codemappings, justifying the algorithm design.

The answers we received for **RQ 2.1** show that overlaps are a common problem in real-world architecture specifications. On average, nearly every second architecture description contains potentially error-causing overlaps.

The results from **RQ 2.2** demonstrate that runtime can increase considerably when architectures that contained overlaps are made overlap-free (cf. table 5.2). Thus, optimisations for the "good case" — when no overlaps are present — are strongly recommended as future improvements of the algorithm. The fact that most runtimes are shorter than 10 seconds, though, justifies the decision to include an automatic overlap validation when saving an architecture file with ConQAT's Architecture Editor. Validation can be aborted by a single click of the user in case it takes too long.

6 Related Work

In this chapter, we summarise and compare similar work by other researchers on architecture metrics and overlapping regular expressions.

6.1 Architecture Restrictivity

In the software engineering community, software metrics have acquired a somewhat bad smell about them. Metrics have often been poorly backed by practical validation, difficult to understand or misinterpreted, as is the case with cyclomatic complexity, to name a renowned example [She88].

Principally, according to [SI93] there are three different kinds of software metrics: Code metrics, design metrics and specification metrics. Code metrics are calculated on the basis of the source code — the lines of code metric (LOC) is a simple example. In contrast, design metrics are mostly centred on the architecture of a system. Specification metrics deal with the product specifications — the functional requirements for example. The restrictivity metric from chapter 3 can be considered a design metric, for it is mainly a calculation on the architecture, even though it takes abstract information from the source code into account, namely the number of matched types per component. A categorisation as a combination of a design and code metric is equally reasonable with the same argument (cf. section 3.1).

Design metrics can be further specified into intra-modular and inter-modular metrics, and a mixture of both [SI93]. Intra-modular metrics capture information internal to a module, whereas inter-modular metrics capture the relationship between modules. Since the restrictivity combines both aspects, it can be considered an intra- and inter-modular design metric. [SI93] notes that "a number of different design metrics have been proposed. They differ mainly in the detail of how best to capture coupling and cohesion". Cohesion is the "singleness of purpose or function of a module" and coupling "the degree of independence" which a module has. By module, [SI93] refers to the top-level building blocks in the architecture, in our case the components on the first hierarchy level. The restrictivity is not based on the concept of coupling and cohesion, although sound architectural design with a high degree of cohesion and little coupling will lead to higher ρ values.

Since the release of [SI93], little has changed in this area of research. In an attempt to capture the analysability of an architecture, [BCvDV11] proposes a metric that takes the number of top-level components and their relative sizes into account. Apart from the aspect that the size of a component is considered and that the metric is architectural, there is no further similarity between the so-called "Component Balance" metric [BCvDV11] and the architecture restrictivity.

Therefore, the restrictivity represents a novel idea for the definition of a design metric.

6.2 Overlap Analysis

While it is a generally known fact that regular expressions can overlap, there is no dedicated research done on it. We found that overlap detection is only scarcely mentioned in [BSCV06, BJP06]. [SVW09] describes extensible languages whose terminal symbols are represented by regular expressions. They identify the problem of overlapping regular expressions between the terminal symbols. However, no detection algorithm for such overlaps is presented.

Therefore, our thorough approach to overlap detection is novel.

7 Conclusion and Future Work

This chapter draws conclusions from the conducted work and suggests possible improvements for the future.

7.1 Conclusion

In this thesis, we have developed ideas and algorithms for the static validation of architecture descriptions. Specifically, we integrated a restrictivity metric analysis and an overlap detection into ConQAT.

Having evaluated eleven real-world projects in a case study, we found that the mean restrictivity is $\rho = 0.65$ and that significant outliers can be explained by an inspection of the architecture and its assessment. Our results of the overlap analysis verify that overlaps are a frequent problem in architecture descriptions and occured in 45% of the examined systems.

A literature research revealed that there exist very few suggestions for metrics semantically similar to the restrictivity metric. No related work had been done on overlap detection in the context of architecture specifications before this thesis.

7.2 Future Work

In this section, we describe potential enhancements of the implemented algorithms.

Restrictivity

To improve the precision of the restrictivity, a differentiation between package-visible, private and public types could be introduced. For two types x, y, it could be checked whether x can access y — this is only the case when y is public or when they are in the same package and y is package-visible. Only then would the type pair be counted to the allowed type pairs.

To retrieve more experience how the metric behaves in practice, an evaluation of a larger set of systems regarding their restrictivity could be done. It is also recommended to include developers' and system architects' feedback and how they feel the metric fits to their system.

Overlap Analysis

As we have seen in the case study, an improvement of the algorithmic runtime for the good case is needed. A profiling analysis of the overlap algorithm shows that almost

50% of the runtime in the package `dk.brics.automaton` is spent on the creation of automata, the other 50% for the conversion to regular expressions. We can reduce the calls to `Automaton()` with the following optimisation: As we compare components with each other, an incremental "checkedComponentsAutomaton" is built. This automaton contains the union over all codemappings checked until that point. Every remaining component is checked for overlaps with the "checkedComponentsAutomaton". If there are none, then we can in turn add the component's automaton (and thereby its codemappings) to the "checkedComponentsAutomaton". If there exist no overlaps in the architecture, all codemappings are joined into one large automaton at the end of the loop.

Algorithm 5 architectureHasOverlaps()

```
Require: $allComponents$ containing all components of the architecture in a list
  $checkedComponentsAutomaton$
  for $component \in allComponents$ do
    $componentAutomaton \leftarrow$ buildMatchingAutomatonForComponent($component$)
    {Creates an automaton for the component as a union over all the include codemappings and a subtraction of the joined exclude codemappings}
    $overlap \leftarrow$ detectOverlap(componentAutomaton, checkedComponentsAutomaton)
    if $overlap$ then
      return true {The architecture has overlaps, use algorithm 1–4 to find out the exact overlap}
    else
      $checkedComponentsAutomaton \leftarrow checkedComponentsAutomaton \cup componentAutomaton$
    end if
  end for
  return false
```

This means that for the good case we only have to perform one comparison per codemapping, namely a comparison to "checkedComponentsAutomaton". Runtime for the bad case decreases slightly, as we still have to find out which exact codemapping and component the overlap is occuring with. This can be achieved with the current algorithm.

A Appendix

A.1 Reproducing Results

To reproduce the results obtained in this thesis, the following tools are necessary:

- **Apache SVN,** the software versioning and revision control system used by the ConQAT project. Subversion command line client, version 1.6.17.
  `http://subversion.apache.org/`
- **Eclipse Modeling Tools**, the modeling edition of the Java integrated development environment Eclipse, version 3.7.
  `http://www.eclipse.org`
  The following plugins are used:
  - **Subclipse**, an SVN client integrated into Eclipse.
    `http://subclipse.tigris.org/update_1.6.x/`
  - **CCSM Development Tools**, providing features essential to the development process, such as the ConQAT Rating Support.
    `http://www4.in.tum.de/~ccsm/eclipse_update_site`
  - **Eclipse Test & Performance Tools Platform Project (TPTP)**, a Java profiler.
    `http://www.eclipse.org/tptp/`
- **Apache Ant**, the Java-based software build system ConQAT employs.
  `http://ant.apache.org/`
- **Checkout of ConQAT trunk**
  ```
  svn co https://svnbroy.informatik.tu-muenchen.de/ccsm/conqat
  -root/trunk conqat-trunk
  ```

You can the import the projects from the ConQAT trunk into your Eclipse workspace. Further assistance can be found under `http://conqat.cs.tum.edu/index.php/Start_Developing`.

A.2 Affected Change Requests

Integration of this thesis' artefacts into the codebase of ConQAT has been conducted under the following Bugzilla change requests:

- Bug 3876: EXCLUDE code mappings from parent components are not regarded in the OverlapAnalysis

- Bug 3869: A codemapping containing `<.*>` throws an IllegalArgumentException during OverlapAnalysis
- Bug 3812: Integrate ArchitectureOverlapAnalyzerProcessor into ArchitectureAnalysis.cqb
- Bug 3753: Create Conqat analysis for restrictivity metric measurement
- Bug 3742: Show progress dialogue when performing architecture validation
- Bug 3705: Unhandeled Event Loop when typing { into code mapping's regex field
- Bug 3702: Add statical RegExen overlap analysis

All of the change requests have been reviewed and set to closed, effectively making the commited changes part of the ConQAT distribution.

Bibliography

[BCvDV11] E. Bouwers, J.P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. 2011.

[BJP06] Z.K. Baker, H.J. Jung, and V.K. Prasanna. Regular expression software deceleration for intrusion detection systems. In *Field Programmable Logic and Applications, 2006. FPL'06. International Conference on*, pages 1–8. IEEE, 2006.

[BSCV06] J. Bispo, I. Sourdis, J.M.P. Cardoso, and S. Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126. IEEE, 2006.

[Con] ConQAT. `http://conqat.cs.tum.edu/index.php/ConQAT`. Accessed 2011/06/01.

[DFH$^+$10] F. Deissenbock, M. Feilkas, L. Heinemann, B. Hummel, and E. Jurgens. *ConQAT Book*. Technische Universität München, 2010.

[Feg] Leonidas Fegaras. `http://lambda.uta.edu/cse5317/notes/node9.html`. Accessed 2011/09/07.

[FRJ09] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 188–197. IEEE, 2009.

[HMU79] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 3. Addison-wesley Reading, MA, 1979.

[Jav] JavaDoc. java.util.regex class pattern. `http://download.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html`. Accessed 2011/06/03.

[Kel91] W.T. Kelvin. *Popular lectures and addresses*. Number Bd. 1 in Nature series. Macmillan and co., 1891.

[Mø11] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2011. `http://www.brics.dk/automaton/`.

[She88] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[SI93] M. Shepperd and D. Ince. *Derivation and validation of software metrics*, volume 9. Oxford University Press, USA, 1993.

[SVW09] A.C. Schwerdfeger and E.R. Van Wyk. Verifiable composition of deterministic grammars. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 199–210. ACM, 2009.