Automated Analysis of Software Artefacts – A Use Case in E-Assessment

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

durch die Fakultät für Wirtschaftswissenschaften

der Universität Duisburg-Essen
Campus Essen

vorgelegt von
Name: Michael Striewe
Ort: Langenfeld/Rhld.
Essen 2014

Tag der mündlichen Prüfung: 19.12.2014
Erstgutachter: Prof. Dr. Michael Goedicke
Zweitgutachter: Prof. Dr. Barbara Paech

Abstract: Automated grading and feedback generation for programming and modeling exercises has become a usual means of supporting teachers and students at universities and schools. Tools used in this context engage general software engineering techniques for the analysis of software artefacts. Experiences with the current state-of-the-art show good results, but also a gap between the potential power of such techniques and the actual power used in current e-assessment systems. This thesis contributes to closing this gap by developing and testing approaches that are more universal than the currently used approaches and provide novel means of feedback generation. It can be shown that these approaches can be used effectively and efficiently for the mass validation of exercises, and that they result in a high feedback quality according to students' perception.

Acknowledgements: First of all, I would like to thank my advisor Michael Goedicke for giving me the opportunity to write this thesis and pursue my scientific interests with great freedom. He was always prepared to provide the resources needed and kept the atmosphere at our group warm, friendly and constructive. Many thoughts and comments from him helped me to shape my ideas and enhance my results. My second advisor Barbara Paech deserves thanks not only for her feedback to each and every chapter of this thesis, but also for using techniques developed in this thesis to support her lectures.

Many thanks as well to all my colleagues at our group and everywhere else, who allowed me to have many fruitful and motivating discussions and who gave me lots of valuable feedback. Some people deserve special thanks for their support during the genesis of this thesis: Carsten Köllmann for his initial work on static code analysis via graph transformations, that prevented me from starting with nothing. Christine Zarges for being my guide through the world of algorithms on sequence alignment. Our student assistants Thomas Ziegler, Waldemar Hartwig and Wilhelm Koop for their enthusiasm in implementing some software components. All our students who did project work or wrote their seminar, bachelor degree, or master degree theses in the context of e-assessment, artefact analysis, and feedback generation.

In addition, I'm in deep dept to my parents for all their gentle support and patience through all the years. And finally, many thanks to Katja for everything and her love.

Contents

List of Figures

Part I.

Introduction

1. Motivation and Thesis Outline

Automated analysis of software artefacts is an important topic for several decades. Since today's software systems are often large and complex, they can hardly be understood as a whole by manual inspection. Various techniques and tool support for analyzing them are thus vitally important and applied in many sub-domains of software engineering: During the design phase, artefacts like diagrams and formal specifications can be searched for inconsistencies or ambiguities in order to reflect requirements properly. Artefacts from the design phase can also be checked and verified with model checking techniques. During the implementation phase, code analysis can find undesirable patterns, e.g. low-performance code or evasion of interfaces. In addition, detection of duplicate code can help to improve code quality. Testing and run time verification are core topics in automated software engineering in two ways: The run time behaviour of a program is subject to be searched for unwanted results or execution paths and the input data for tests is subject to be optimized automatically in order to cover as much of the code with at least tests as possible. Finally, code review and re-engineering tasks can benefit from the automated analysis of software artefacts for finding abstractions in order to reveal or discover the formal ideas behind the actual implementation.

Although automated analysis of software artefacts is motivated by the problems of large scale software projects, it is also conceivable to apply at least a subset of these methods in small projects. A special use case with very small projects is the mass validation of exercises in software engineering. Each exercise could easily be reviewed by an experienced teacher, who would have no problem with the size of the solution artefacts, but great reductions in the time needed for marking could be expected by automated mass validation. This effect is already known from automated marking for less complex types of exercise, e.g. multiple choice questionnaires. These less complex question types are easy to check, because the number of possible wrong and right solutions is known beforehand. However, it is known that cognitive skills and application of methods cannot be assessed via multiple-choice tests [20]. Hence courses on programming deal with the problem that the number of possible correct solutions for a programming exercise is unbound in most cases. Similarly, courses in system design or modeling have to deal with assignments with graphical languages and with vagueness or fuzziness that may not be judged in strict terms of right and wrong. Thus methods suitable for artefact analysis in general seem to be a promising approach to handle the complexity of this exercises and allow computer support in this area, too.

Using automated grading is also desirable, because computer support in general is almost an everyday feature in contemporary higher education. Under the umbrella of the term "e-learning" various kinds of techniques are used to support learners and teachers in their work. Besides automated marking of solutions or computer aided assessments (CAA) in general, several other topics like communication techniques for distance learning and ubiquitous learning or intelligent tutoring systems (ITS) became important topics of research and discussion in recent years [26]. Although one might think of automated grading of multiple-choice tests or submitting exercise solutions via e-mail as the first applications of CAA-systems, a grader for punchcard programs on IBM 650 computers published in 1960 [72], followed by two independent solutions for automated grading of ALGOL programs published in 1964 and 1965 [103, 52] can be considered the oldest CAA-systems. On the one hand many different terms exist like "tutoring systems", "exercise systems", "assessment systems" and "examination systems". On the other hand, computer aided tutoring and computer aided assessments can be considered as very closely related with respect to the automated analysis of software artefacts. If a software system is able to give adequate learning hints, it must be able to distinguish between a complete and correct solution where no more hints are needed and an incomplete or wrong solution. Thus it could also be used for marking solutions. Similarly, if a software system is able to mark solutions, it must be able to decide whether a solution is completely correct or partly correct or completely wrong. Thus it could also present this judgement to the learner to tell what is missing and what is expected. Although there might be small differences between straight hints or more subtle clues, the general consideration may be applicable to virtually any learning scenario, independent from topics of an actual course or the methods typically used for assessments on these topics. Nevertheless, careful examinations are desirable to find out which kind of feedback or which kind of judgement can be given by a certain analysis method. Thus the technical view on power and limitations of each method has a direct impact on recommendations regarding the use of a certain method in an actual computer aided assessment system.

It is not the intention of this kind of research to fully replace human teachers and tutors in courses on basic software engineering topics. It is doubtful whether this is possible or even desirable at all. However, the core thesis is that automated assessment of software artefacts can relieve teachers and tutors from the burden of tedious and error-prone grading tasks without lowering the quality of feedback provided to the students. In fact, the idea is to provide even better feedback, because automated feedback is fast, while teachers and tutors gain more time available for truly individual and dialogue-oriented feedback to individual students. Automated analysis of artefacts may also help teachers and tutors to get quickly alerted about common errors or typical misunderstandings and thus allow them to respond precisely to them.

In summary, this thesis is motivated by an ideal scenario in which automated assessment systems provide fast and detailed feedback to individual solutions, while teachers and tutors focus on the creation of high quality exercises and the discussion of individual misconceptions and problems. The scenario seems realistic, as many software engineering methods for the automated analysis of software artefacts already do exist. Thus the ultimate goal is to make appropriate methods available for automated assessment. This goal is based on the underlying assumption that the ideal scenario is not yet reached and that there is a gap between the potential power of such methods and the actual power used in currently available systems. A justification for this assumption will be provided based on a literature review in chapter 2. In particular, the literature review shows that the suspected gap has several dimensions: analysis techniques, inputs needed for these techniques, feedback produced by these techniques, and systems implementing these techniques. The literature review provides an assessment of progress in all these dimensions.

The literature review will suggest that there is no point in closing the gap by creating yet another e-assessment system just to bring additional methods into action. Instead, all implementations provided by this thesis will be designed as extension to an existing e-assessment system. Hence before presenting concepts and implementations, the system JACK will be introduced in chapter 3 of this thesis. The development of this system has partly happened in parallel to this thesis. Although the actual system development is not within the scope of research of this thesis, the work triggered by this research has contributed much to the development of JACK. In particular, JACK provides a flexible architecture that offers the opportunity to plug-in new modules for automated analysis and feedback generation.

1.1. Contribution and Thesis Structure

With the premises from motivation and literature review in mind, part II of this thesis will make two main contributions as well as some additional contribution in order to fill the suspected gap both in the dimension of techniques and in the dimension of feedback:

- First, approaches to static analysis of software artefacts by pattern matching will be studied in chapter 4. The aim is to define a robust mechanism that can be applied to a broad range of different types of artefacts and that goes beyond the current state of the art in the use of static analysis in e-assessment tools. The latter is the main contribution of this chapter, approaching primarily the dimension of feedback in order to provide better results than it is possible with the tools integrated in common e-assessment systems today. However, the dimension of techniques is touched as well by discussing the data structures representing the artefacts and different approaches to query these structures. It is described how the methods work using appropriate data structures, which feedback can be generated from their application and which issues regarding configuration, robustness, scaling and performance have to be observed.
  The main contribution of this chapter is an universal approach to static analysis that allows more flexibility and covers are broader range of artefacts than current approaches.

- Second, approaches to dynamic analysis of software artefacts by trace analysis will be studied in chapter 5. The aim is to explore new techniques that go beyond the state-of-the-art of automated analysis of artefacts and that provide specific benefit for e-assessment and intelligent tutoring. Thus this chapter will specifically tackle the dimension of techniques by creating novel approaches to automated artefact analysis and testing these approaches with prototype implementations. The dimension of feedback is used to evaluate the quality of results gained by these novel techniques.
  The main contribution of this chapter is a novel approach to dynamic analysis that provides more detailed and individual feedback than current black-box testing approaches.

- As additional contributions, applications that extend or complement the work in the previous chapters will be studied in chapter 6. The aim is to demonstrate that the techniques from the main contributions are not limited to a certain scope, but can be reused and recombined in different contexts. In addition, the chapter stresses that the field of feedback generation in e-assessment systems cannot be covered by just a few approaches, but offers room for applying various software engineering techniques.
  The main contribution of this chapter are new features yet uncommon in

e-assessment systems in terms of new kinds of feedback, or feedback to artefacts not yet considered by these systems.

Although the contributions focus on the dimensions of techniques and feedback, the dimension of input is not neglected. Instead, the complexity of input will be examined in the main chapters and it will be shown to which degree generic input or automated generation of input is possible. In the main contributions, the techniques discussed in the respective chapters are applied to different types of artefacts, at least Java source code and UML diagrams of different types. As far as program code is concerned, the thesis is limited to the Java programming language in Version 7. Where appropriate, cross connections between the different techniques will be discussed to create even more advanced feedback generation methods by combining simpler ones.

The thesis is concluded by part III that sums up the results by giving evaluations, reviewing the achievement of goals and naming future work that goes beyond the scope of this thesis. Specifically it presents experiences from using some of the implementations discussed in part II in practice to provide additional evidence for their usability and impact. This part of the thesis also covers additional concepts that have not been implemented in prototypes and thus identifies remaining gaps in either of the four dimensions.

1.2. Previous Publications

The research presented in this thesis has partially been published in several conference and workshop papers. This section provides an overview on these publications in the same order as the topics are tackled in this thesis.

Work on static analysis of artefacts from chapter 4 has been published in:

- Michael Striewe, Michael Goedicke: *A Review of Static Analysis Approaches for Programming Exercises*, Proceedings of International Computer Assisted Assessment (CAA) Conference 2014, Zeist
- Michael Striewe, Michael Goedicke: *Automated Checks on UML Diagrams*, 16th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2011), Darmstadt

Work on dynamic analysis of artefacts from chapter 5 has been published in:

- Michael Striewe, Michael Goedicke: *Using Run Time Traces in Automated Programming Tutoring*, 16th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2011), Darmstadt
- Michael Striewe, Michael Goedicke: *Trace Alignment for Automated Tutoring*, Proceedings of International Computer Assisted Assessment (CAA) Conference 2013, Southampton
- Michael Striewe, Michael Goedicke: *Automated Assessment of UML Activity Diagrams*, 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2014), Uppsala

Work on visualization of data structures from section 6.1 has been published in:

- Michael Striewe, Michael Goedicke: *Visualizing Data Structures in an E-Learning System*, Proceedings of the 2nd International Conference on Computer Supported Education (CSEDU) 2010, Valencia, Spain, volume 1

Work on code reading exercises from section 6.2 has been published in:

- Michael Striewe, Michael Goedicke: *Code Reading Exercises Using Run Time Traces*, 19th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2014), Uppsala

Work on using software product metrics from section 6.3 has been published in:

- Michael Striewe, Michael Goedicke: *Analyse von Programmieraufgaben durch Softwareproduktmetriken*, Proceedings of SEUH 2013, Aachen, Germany
- Michael Striewe: *Generierung von Zusatzinformationen in automatischen Systemen zur Bewertung von Programmieraufgaben*, Proceedings of the First Workshop "Automatische Bewertung von Programmieraufgaben" (ABP 2013)

Evaluation data on performance and behaviour of students from chapter 7 has been published in:

- Michael Striewe, Michael Goedicke: *Feedback-Möglichkeiten in automatischen Prüfungssystemen*, Proceedings of "DeLFI 2010 - Die 8. E-Learning Fachtagung Informatik", Duisburg, 2010
- Michael Striewe, Michael Goedicke: *JACK revisited: Scaling up in multiple dimensions*, Proceedings of Eighth European Conference on Technology Enhanced Learning (EC-TEL 2013), Paphos, Cyprus

2. Research Context and Literature Review

The goal of this chapter is to justify the assumption that there is a gap between the potential power of software engineering methods for artefact analysis and the actual power used in currently available systems. From the motivation it can be deduced that the problem of automated assessment and feedback generation for software engineering artefacts can be tackled from more than one side. An earlier review on that topic shows that research and development progress is different in different areas [36]. Hence this chapter provides a detailed review and identifies four dimensions for the suspected gap, as depicted in figure 2.1:

- The **dimension of systems** deals with the features and design of actual e-assessment and tutoring systems. This dimension answers the question on whether there are tools available that can cover a wide range of artefact types and assessment methods. Weak progress in this dimension can be reported, if there are few systems available, or if most

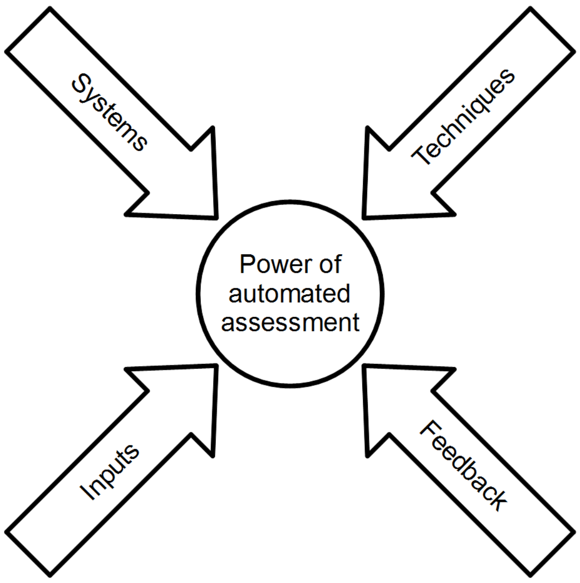


Figure 2.1.: Four dimensions of automated grading and feedback generation: Robust and flexible *systems* are necessary to make automated assessment useful in practice; matured *techniques* are necessary for a detailed analysis of artefacts; the amount and complexity of *inputs* determines the effort necessary to set up individual exercises; the quality of *feedback* determines how useful automated assessment is for students in learning scenarios.

of the available systems are just prototypes, or if there is no common understanding of core features and useful architectures of automated assessment systems. In turn, good progress can be reported, if there a many tools available that share common basic features, that provide alternate implementations for similar techniques, or that are based on mature architectural concepts.

- The **dimension of techniques** deals with methods and algorithms used for artefact analysis. This dimension answers the question about the different properties that can be checked on a particular artefact in general. Besides the plain amount of available methods it is in particular interesting in this dimension, which methods are actually in use in existing assessment systems. Hence weak progress can be reported in this dimension, if there are only few methods available in these systems, or if systems use methods that turn out to be inappropriate or insufficient for the goals of these systems. In turn, good progress can be reported if the majority of known appropriate methods is actually in use. Note that it is neither the goal of this part of the review to report on the amount of methods for software artefact analysis in general, nor to make any statement on the necessity for more basic research in that area. It is solely focused on the usage of existing methods in existing e-assessment systems.

- The **dimension of inputs** deals with the data needed as input for the techniques used in a system. This dimension answers the question how much preparation and configuration of the tools is necessary for individual exercises. This is considered a key indicator for the amount of work that has to be done by teachers when preparing exercises for automated assessment systems. Weak progress in this dimension can be reported, if existing tools and techniques require much exercise specific input, or if it is hard to reuse input data, or if no means of automated input generation are available. In turn, good progress can be reported, if generic techniques and systems are available, possibly by automated input generation, or if modular reuse of inputs across exercises is possible.

- The **dimension of feedback** deals with the kind and content of feedback given to the students. This dimension answers the question on how much information and advice a student can receive from an automated assessment and tutoring system. This is considered a key indicator for the support students get from using automated assessment systems. Weak progress in this dimension can be reported, if the feedback provided by existing systems does not go beyond plain fault detection, or if students consider the feedback messages as not helpful, or if a large amount of false positives or false negatives is produced. In turn, good progress can be reported, if feedback quality is considered to be sufficient by the students, and if detailed comments for solutions can be produced with a high precision.

The following sections provide a brief overview about the history for each of the dimensions and the current state of research in the respective area in order to justify the hypothesis about the existence of a gap. Each section is going to point out unanswered questions and open spaces for improvement beyond the current state of research.

2.1. The Dimension of Systems

As already mentioned in the introduction, the oldest systems for automated assessment of programming exercises date back to the 1960's. They can be considered as the oldest systems for automated assessment of software artefacts in general. Since then, many new programming and modelling languages have been invented and hence more tools for automated assessment of them have been developed as well. Especially in recent years, a large amount of different systems for e-assessment and automated grading has been developed and published. In 2005, a survey of automated assessment approaches for programming assignments conducted by Ala-Mutka [8] referenced about two dozen systems or approaches. Some of them have not seen further development since then, and others have been newly created later. The same is true for another review published by Ihantola et al. in 2010 [75], that still did not cover all systems available at that time and obviously others developed later. In addition, this survey explicitly excluded a large amount of e-learning systems that are not primarily intended to perform assessment, but to provide some guidance in the learning process. However, the survey by Ihantola et al. observed the tendency that there are many new systems developed that do not provide additional capabilities compared to the already existing systems.

From a software engineering point of view, reasons for this tendency can be found in the system design and the requirements the systems are aiming to fulfill. Many older grading systems have been designed as monolithic systems, intended to be used in one particular environment (e.g. Ceilidh [14], Assyst [77], Praktomat [163], or OnExSy [13]). Many of them can be described as course-oriented, trying to cover the whole process of student management in one particular course. For technical reasons, they possibly consider different parts of the systems as separate processes, but they do not define explicit interfaces between them for changing parts of the system. The same is true for e-learning environments that do not provide capabilities for mass assessment, but that are nevertheless monolithic systems, designed to be used as a standalone application supporting a particular course (e.g. Jive [91] or JavaVis [106]) or even just one topic within a course (e.g. iList [53]).

Possibilities to use more than one type of user interface were first steps towards modularity, supported by using modern web frameworks and technologies (e.g. BOSS [81], PASS [24], or DUESIE [71]). These developments were driven either by the requirement of supporting more than one programming language or by the requirement of offering web-based services. However, there was no major tendency to support other artefacts than programming languages. Systems for teaching and assessing other subjects were still designed as monolithic standalone systems (e.g. Exorciser [146]).

However, in recent years, much progress was made in creating more general and modular architectures at least for automated grading of programming exercises. For this approach, grading and feedback generation can be identified as exercise-centric services, advocating for a multi-agent architecture [108] or a service oriented design for assessment tools [33]. By this approach, individual services are integrated using frameworks like the JISC e-learning framework [80]. An implementation conforming to this actual framework is e.g. ASAP [37]. However, the same service-oriented approach has also been used without given frameworks, aiming to be even more general. The architectural concept of JACK, as already published in 2008 [126] and elaborated on in chapter 3 of this thesis, is one example for this direction. It can be considered as almost similar to other architecture proposals, e.g. eduComponents

[11] or FW4EX [111].

Facing these recent developments, it can be stated that architectures for e-learning and e-assessment systems have reached a mature state. Modular concepts are developed and successfully realized. This allows for incremental extension and improvement of the whole systems. As an aside, this also makes those systems ready for future trends like mobile learning and new input interfaces. With the right architectures as hand, these new elements can easily be combined with existing grading engines. Thus the observation by Ihantola et al. can be confirmed that there is no real need to develop more systems without new capabilities. Instead, it can be considered beneficial to develop useful techniques and feedback generation mechanisms that can be integrated into existing systems and frameworks. The suspected gap is thus not a gap in terms of missing systems in general, as good progress can be reported for the dimension of systems.

2.2. The Dimension of Techniques

Research in software engineering has come up with many different techniques that can be used to perform automated analysis of software artefacts. First of all, parsing artefacts of any kind based on a language specification is the most basic kind of static analysis and thus as old as the formal definition of languages. It is necessary to determine whether an artefact is at least syntactically correct. Since this is a necessary condition for any further step, this technique can be considered to be used in any automated assessment and feedback system.

2.2.1. Static Analysis

For program code, data flow and control flow analysis can be considered the largest fields of applied static analysis, that have been explored in great depth since the 1970s. As the most basic application, data flow and control flow analysis are used by compilers to find optimizations for the generated machine code in order to reduce code size and increase performance. This can be done by detection of unreachable code, reordering of common statements or elimination of unnecessary calculations [93, 105]. Common modern programming environments like the ECLIPSE IDE perform these checks on-the-fly while the programmer writes source code and present their results as warnings, including detection of undefined variables. Especially for the Java programming language, the tools FINDBUGS [73, 49] and JLINT [12] provide verification based on data flow analysis, control flow analysis, and structural pattern matching for Java byte code [73, 12]. Structural pattern matching on Java source code instead of byte code is performed by the tools PMD [110] and CHECKSTYLE [21]. Both tools are also able to detect duplicate source code [28].

Integration of one or more of the tools mentioned above is a common approach for recent modular assessment systems. Table 2.1 gives an overview on some automated grading and tutoring systems for Java which use such tools. The use of other external tools than CHECKSTYLE, FINDBUGS, and PMD could not be found in the literature. All three tools are open source and non-commercial projects. The techniques used by these tools include explicit traversals of the syntax tree, Xpath queries to the syntax tree, and regular expressions for string analysis with only slight differences, e.g. regarding explicit inclusion or exclusion of code comments from the analyses. Each of them has some individual benefits and drawbacks with respect to e-assessment:

Name	Source Code Analysis	Byte Code Analysis
ASB [101]	yes (CheckStyle)	yes (FindBugs)
DUESIE [71]	yes (PMD)	no
EASy [62]	no	yes (FindBugs)
Marmoset [125]	no	yes (FindBugs)
Praktomat [163]	yes (CheckStyle)	no
Web-CAT [121]	yes (CheckStyle/PMD)	yes

Table 2.1.: Static analysis capabilities of some automated grading and tutoring systems for Java exercises.

- FindBugs searches for patterns in Java byte code, which allows fast and easy checks, because the byte code representation of a program is simpler than its source code representation and slight syntactical divergence in the source code such as the difference between `i = i+1;` and `i++;` may be unified by the compiler. However, some information is not available after compilation. For example, it is hard to determine in byte code, which type of loop statement was used in source code, because all types of loops are encoded by `goto`-statements in byte code. It is even not possible to find out whether the source code contains loops by looking for `goto`-statements, because other constructs like `switch-case`-statements may result in `goto`-statements in the byte code as well.
- CheckStyle analyses the source code instead of the byte code and is thus more powerful than FindBugs regarding detailed source code elements. However, it was started as a project for observing code conventions and is still limited to checks of single files. Thus CheckStyle cannot check whether a `public` method is called from classes defined in other files. This is a major drawback, because this way exercises dealing with recursion cannot be checked completely. For the same reason, CheckStyle cannot observe inheritance hierarchies of more than two classes or interfaces, which is also a serious limitation in its usefulness for sophisticated system design exercises.
- PMD also analyses the source code and is also limited to analyze a single source file at once. Rules can be implemented as a single Java class, mainly following a visitor pattern. While rule implementations tend to become complex and hard to read, this is a very powerful approach in terms of patterns that can be described this way. As an alternative, patterns can be described by XPath expressions, which are easier to write, but also more limited in their expressiveness. Because of the limitation on single source files, the same concerns as with CheckStyle apply for PMD.

More advanced commercial tools exist as well (e.g. NDepend [3]) that offer pattern matching via code query languages. However, the Code Query Language 1.8 [1] used by that tool focuses on high level queries for professional code analysis and is for example not able to return all methods that contain loops or all local variables that use upper case letters. Moreover, no integration for commercial tools like this one into e-assessment tools exist at the moment.

Although CheckStyle lacks the capability of checking more than one file at once because it was initially intended to observe code conventions, code style in general is indeed an interesting subject of static code analysis in e-learning systems. Besides tools integrating CheckStyle for this purpose, there are also tools focusing solely on this task (e.g. Style++ [7]) or doing it by their own means (e.g. submit [154]). However, checking for good style must always stay a side topic and minor criterion in assessment scenarios since style is a non-functional property and the best style is worthless if required functional properties are missing. Almost the same applies to style checks for other software artefacts, i.e. diagrams, but it has to be considered that in design the distinction between "good" and "bad" design is more relevant [68].

An interesting approach to static analysis that is different to plain pattern matching is used in a system for checking Haskell programs published in 2008 [57]. From a set of sample solutions, strategies are derived in terms of necessary steps in writing a sufficient program. Each step is considered a graph production [44] for the graph grammar defining the Haskell syntax, and a sequence of steps allows to transform an empty start graph into the complete syntax graph of a correct solution. Any solution submitted to the system is checked whether or not it is a valid prefix of one of the strategies, i.e. whether the syntax graph of the solution is a proper subset of any syntax graph belonging to a sample solution. This way hints for action for improving an insufficient solution can be given based on the context of the strategy. This technique can be considered to be a very precise way to generate feedback in an tutoring system, but obviously it is less helpful for plain grading. Moreover, the structured development of a program following some strategy is possibly just possible for functional programming languages, but not for imperative ones. At least, there exists no evidence in literature that this approach could be applied successfully to imperative programming languages.

Besides analyzing an artefact based on its syntactical structure, it can also be analyzed in a more abstract way by metrics. Several metrics for source code have been developed during the 1970s. The simplest way are lexical metrics like counting lines of code or counting operands and operators as the Halstead metric [66] does. More preprocessing is necessary for the McCabe metric, which is based on the control flow graph [98]. These examples represent the two categories that are typically used for metrics, namely metrics of size and metrics of complexity [27, 48]. Since the early metrics were designed for procedural programs, they had to be extended to object-oriented design starting in the early 1990s [56]. There are several e-assessment tools for source code that facilitate metrics: In the early 1980s, a first metric-based approach was used to measure style for PASCAL programs [114]. In the early 1990s the tool Ceilidh [14] compared the complexity of solutions against the complexity of the sample solution and used the C utility lint for detecting structural weaknesses based on patterns. Since then, structural checks based on metrics of the abstract syntax tree are used in different tools (e.g. CourseMarker [67] or ELP [144, 145]). Metrics can not only be used to determine the size or complexity of the source code, but also to check whether a piece of code contains a reasonable amount of comments [81]. Complexity metrics are not only used to judge solutions, but also to give feedback to teachers regarding the incline of difficulty among exercises in a course [99].

Comparing metrics from solutions to metrics from a sample solution is one special case of comparing artefacts as a whole. While pattern matching as discussed above does not require the existence of a complete sample solution and can be done with patterns that are re-usable for several exercises, full comparisons always require the existence of a complete sample solution. An extension to ELP proposes to translate both student's solution and sample solution into pseudo-code before performing a similarity analysis [113]. Another approach also inspired by ELP performs some semantic preserving program standardizations first and then creates a system dependency graph (SDG) that is compared to SDGs derived from sample solutions [159]. The approach has a very high precision, but unfortunately it is so far limited to procedural languages like C or Pascal and has not been extended to object-oriented languages like Java.

Comparison to a sample solution can also be used for other software artefacts than source code. Diagram checking by comparison of sample solutions and student diagrams has been developed since the early 2000s [147, 139] for single diagram types and been extended to UML diagrams by the system DUESIE which compares diagrams to a minimal sample solution to determine conformance, missing parts and superfluous parts [71]. The analysis in this tool is based on SiDiff [143], that has been developed in the context of versioning of models. It is capable of comparing two models with each other and spotting differences. Some more e-assessment tools using similar techniques exist as well [124, 10]. Regardless of the kind of software artefact handled this way generation of sample solutions and if so derivation of strategies for all of them may be too expensive, if there are many different ways to solve the given exercise. In addition, checks based on this technique will never be able to accept an alternative solution that is correct, but does not match any of the given sample solutions. This is also true for simpler approaches that compare solutions based on element counts like FMS does [120].

The observations so far support the gap hypothesis in the dimension of techniques. The plain integration of existing external tools is useful, but does not cover all aspects that might be relevant in an e-learning scenario, while more sophisticated techniques are either not available for a broad range of artefact types or not widely used in actual systems.

2.2.2. Dynamic Analysis

Dynamic analysis by means of running test cases can be considered the oldest application of program analysis in e-assessment systems, since it has already been used in the ALGOL graders from the 1960s mentioned above. For the most simple setup, each test case can be associated with an amount of credit granted for passing the test case as well as with a feedback message presented to the student when the test case fails. If a test case can fail in different ways, different messages or points granted can be used, creating more advanced setups.

Many tools for test automation handle tests as black-box tests like JUnit [83] does, which is a commonly used tool in many e-assessment systems. However, it does not allow any detailed insight into the program under test due to the black-box paradigm. The complementary solution are profiling tools that offer means for detailed run time inspection, e.g. in terms of method calls and resource consumption. However, these tools do not care about test case execution and are in general not fine grained enough to observe execution on statement level. Currently no e-assessment tool exists that integrates a profiling tool. Besides these two general classes of tools, there are also approaches that allow more insight into the program under test and also more automated support for the test execution:

- JavaPathfinder [155] is a tool for testing and run-time verification of Java programs. It can check multi-threaded programs for deadlocks by automated testing of all possible interleavings of parallel threads. In addition, it tackles the problem of test data generation by providing choice generators, that can generates simulated input according to user defined ranges. Both is done by recording the system state after every byte code step and thus creating the state space of the program under test. The information is available via an API and a listener concept at run time and needs to be stored or post-processed if it should be presented to students in an e-learning system. For large programs, the time needed to run tests and the memory needed to store the state space are very large, thus this tool may be suitable for algorithm oriented exercises, but not for system design exercises.

- The Kieker Framework [152] is a framework for dynamic program analysis, profiling, and architecture recovery. Unlike other profilers, it is not just capable of providing performance measures for the program under inspection, but can also generate sequence diagrams from the observed program behaviour. Since this type of diagram acts at the object level without going into the details of method bodies, it may be suitable for system design exercises, but not for algorithm oriented exercises.

- JavaVis [106] is an e-learning tool that creates graphical representations for traces in terms of UML object diagrams and sequence diagrams, which is again suitable for system design exercises. It allows to step through the trace line by line or from method call to method call and thus facilitates detailed trace inspection in general. However, it does not offer any automated capability for assertion checking and is thus not suitable for grading or feedback generation, but just for visualization.

- Another e-learning tool offering almost similar capabilities is JIVE [91]. It also implements a query based search facility that allows to search for steps that e.g. violate a particular assertion. However, this cannot be done as an automated step in this system, so that this tool also does not offer real automated feedback as well.

Besides looking for errors, dynamic analysis based on tracing can also be used to analyze quality aspects of a solution. In particular, creating traces from solutions and comparing them to sample solutions can be used to detect performance problems and bad algorithmic strategies [63]. Studies on whether this approach is superior or inferior to static analysis for the same purpose do not yet exist.

In most cases, the problem of providing test data is not solved automatically, but it is left up to the teacher to provide adequate test cases. However, systems exist that expect the students not only to provide a solution for the exercise, but also test data for testing this solution [41]. Moreover, approaches exist that try to create test data for programming exercises automatically [74]. The principle of running test cases against student programs can also be inverted, in order to test students capabilities in writing test cases [22]. For this purpose, students have to submit test cases and the teacher provides several buggy programs. The resulting mark is then determined by the number of bugs found by the test cases. Dynamic analysis can also be applied to diagrams, as far as execution semantics can be defined for them. This has been done successfully for circuit simulations and flowchart-diagrams in 2002 [147].

An important issue in dynamic tests in e-assessment scenarios is tolerance in face of marginal divergences between actual solution and sample solution. If the output of the solution contains text, there may be typos that should not result in a failed test case if the actually relevant content of the output is correct. A simple way to cope with this problem is to use regular expressions instead of plain string comparison for outputs. However, regular expressions are hard to write in complex cases and may not be able to solve the issue in every case. Instead, tokenizers can be used, that divide the output into tokens and apply individual rules for comparison to each of them [135]. In general, several techniques from static analysis can be applied to the results of program runs in order to mark them.

The system HoGG tries to carry tolerance in face of minor errors even further by using Java Reflection to cope with errors in method names [100]. In detail, the system tries to identify methods based on their parameters and return type, and uses regular expressions for matching the name only if necessary. The system is also able to replace incorrectly working methods by correctly working ones via inheritance to reduce the number of test cases that fail just because of errors in methods they depend on. While this is an ambitious concept, the author of that system admits that in practice teachers tried to define test cases without dependant methods at all. In addition, it requires some design restrictions, i.e. private fields cannot be used in this approach because they are not available in an inherited class that is supposed to replace a method. Thus this concept can be considered too fragile for complex exercises and only useful with carefully designed exercises of very limited scope.

Another interesting problem in dynamic analysis in e-assessment systems is handling of infinite looping behaviour. A usual strategy is to apply timeouts, that are either fixed for a whole test suite or adaptive for each solution [42]. More sophisticated approaches are known from industrial application [19] and theoretical research [64], but are not used in current e-assessment systems.

Again it can be summarized that the gap hypothesis is supported by these findings from literature. On the one hand, there are e-learning tools that provide advanced test case execution and insight into test cases, but no automated analysis and on the other hand there are advanced tools that allow detailed test control and trace generation, but that are not integrated into e-assessment tools. Only for the basic case of plain black-box tests, a full integration exists. What is missing are implementations that use e.g. trace data collected by tests for detailed feedback generation.

2.2.3. Combinations of Static and Dynamic Analysis

Besides applying static and dynamic analysis independent of each other, it is also possible to combine them explicitly or implicitly. The tool ASSYST explicitly uses static analysis of the syntax tree to prepare dynamic tests. It performs a block analysis, identifying the basic blocks of the program. Based on this, performance in terms of numbers of block invocations as well as test data adequacy in terms of block coverage is measured [77]. Implicit combinations are also known under the general term of "formal verification" such as abstract interpretation or model checking. These techniques in general try to analyse the semantics of an artefact via syntactical and mathematical analysis. For example, formal verification for program code can be used for checking type properties [30], or detection of possible range violations by abstract interpretation developed in the late 1970s [29], which is nowadays used in industrial tools [161]. In object-oriented languages abstract interpretation is especially interesting for possible violations in pointer assignments [69].

Model checking based on finite state systems has been proposed for communication protocols in the late 1970s [160] and soon be extended to model checking for parallel programs using specifications in temporal logic [45]. Explicit model checking suffers from the problem of large state spaces, which has partly been overcome in the early 1990s by partial order reduction [58, 150] and especially the invention of symbolic model checking [18], which is a standard technique for industrial tools like CadenceSMV or NuSMV. Not just programs can be subject to formal verification, but also diagrams can be verified by model checking, at least as far as they have a defined execution semantic as behavioral models [92].

Although there is much potential in formal verification techniques, hardly any published e-assessment system uses strong formal verification techniques [94]. One rare exception is OnExSy that uses formal verification of program behavior based on model checking for assembler programs [13]. Hence there is again a gap between the techniques available in general and in tools for industrial use on the one hand and the techniques available in e-assessment systems on the other hand.

2.3. The Dimension of Inputs

Every technique discussed in the previous section needs some kind of input beyond the actual artefact under analysis to perform the desired analysis. Simple static checks require rules or patterns to be matched, while simple dynamic checks require test cases. More complex techniques may use additional information like assertions, constraints, or other means of formal specification. Current research literature often just names the required inputs without digging into the details of their complexity. In particular, it is rarely discussed how much effort is necessary to create the required inputs for an individual exercise. For plain test cases written in Java for the JACK system used in this thesis, effort of 3 to 10 hours per exercise can be reported. Independent of lacking figures on the actual effort, some research exists on how to ease input generation. Three main approaches can be identified: Generic inputs, derived inputs, and automated input generation.

Generic inputs are typically used when integrating third-party tools such as FindBugs or PMD as discussed for the static checks in the previous section. These tools include a standard library of patterns to be matched on the artefact under analysis and can thus be used without exercise specific configuration.

Adding exercise specific checks requires to include new patterns into the libraries, which can be done either by declaration or programmatically [28]. This requires explicit considerations by the author of an exercise to sort out the required core features of a solution. Some approaches exist to ease this process. One option is to derive the actual rules from more abstract patterns that represent the concepts to be realized by a valid solution [89]. Another approach is to derive input from different sample solutions [57]. In both cases, abstract patterns or sample solutions are not part of the actual input for the analysis method, but only the more detailed rules derived from them.

The third approach does not involve sample solutions as a source for derivations, but generates input for dynamic tests directly from solutions submitted by students [74, 140, 116]. This way, individual test sets are created for each submission. While this is no general benefit or limitation, the authors of Pex4Fun explicitly state that their tool is focused on grading solutions and not grading tests [141].

The variety of approaches shows that there are several ways on how to automated input generation and that there are approaches for generic and thus reusable inputs. Consequently, good progress can be reported for this area. However, the dimension of inputs has to be seen in conjunction with the other dimensions: Appropriate systems may ease input creation, while advanced techniques may require more complex inputs. By definition, generic inputs can never be sufficient to generate exercise specific feedback in any case. Thus research in the dimension of feedback may also require research in the dimension of inputs.

2.4. The Dimension of Feedback

The dimension of feedback is the one at most driven by didactical requirements and the one that can be based on extensive research from outside the scope of e-assessment [102]. In general, it is known by empirical evidence, that instant automated feedback to exercise submissions has a positive effect at least on the motivation of the students [122]. Moreover, general models on the use of feedback in self-regulated learning exist, which identify different purposes of feedback [104]:

- Clarify what good performance is
- Facilitate self-assessment
- Deliver high quality feedback information
- Encourage teacher and peer dialogue
- Encourage positive motivation and self-esteem
- Provide opportunities to close the gap
- Use feedback to improve teaching

Hence it is useful to elaborate on the different possible kinds of feedback that can be generated automatically and the different purposes it may serve. As a general typology, teacher feedback can be divided into two categories, each with two characteristics: The first category is evaluative feedback, which can be positive or negative. The second category is descriptive feedback, which can be achievement feedback or improvement feedback [148]. Evaluative feedback can be considered important primarily in summative assessments, while descriptive feedback can be considered important primarily in formative assessments [97]. In addition to that, extensive guidelines exist in literature on how to design formative feedback [123]. Although most studies on this do not focus on automated feedback systems and e-assessment in particular, the typology and most recommendations can be used in this specific context, too. Moreover, studies show that students struggle on different levels and thus indeed need different kinds of feedback [9, 35]. There is also statistical evidence that fine-grained marking schemes have positive effects on student motivation and performance compared to coarse-grained schemes that allow for just two or three different marks per exercise [47].

Some steps towards classification of feedback techniques for Java exercises and their effects as developed during the genesis of this thesis have already been published in [128] and [129]. In particular, it is relevant how technical approaches to feedback generation can be mapped to the feedback typology, as one technique can cover different categories: For example, compiler messages can both be used for negative evaluative feedback and descriptive improvement feedback. In this way, they fulfill several of the purposes listed above: They help to clarify what good performance is (i.e. lesser compiler errors are better), they may encourage dialogue on causes and solutions to the errors, they are concrete hints on what needs to be changed to come closer to a good solution, and they also help teachers to identify misunderstood syntactical concepts that need to be repeated in class.

The already mentioned survey of Ala-Mutka [8] also included a detailed classification of analysis techniques for programming exercises, but focused on different features to assess (e.g. functionality, efficiency, coding style, and more) and not on the quality of feedback that can be achieved by these techniques or combinations of them. Amongst the different ways of delivering feedback in e-assessment systems, visual feedback for programming exercises has gained some significant attention in recent years [90, 76, 130]. Different feedback techniques for diagrams have also been discussed, but not yet been connected to overall evaluation results for diagram checking [137].

Although the review by Douce et al. [36] includes a short overview on different ways to assess feedback quality, this topic has in general only been a minor point in the scientific discussions in the last years, at least in comparison to the dimensions of systems and techniques as discussed in the previous sections. However, quality of feedback has been identified to be an important property of automated grading systems, especially if static checks based on metrics are used [14].

Currently, the common way to measure quality of feedback is to evaluate user satisfaction with a given system based on surveys. Results of user satisfaction with different systems as collected from literature are listed in table 2.2. As it can be seen from these figures, there is a tendency towards a positive attitude, but neutral and moderately positive attitudes dominate over clear positive attitudes in most cases. Thus it can be concluded that there is space for improvements in user satisfaction. This is also reported by a very detailed study for the system Mooshak recently published by Rubio-Sánchez et al. that concludes the need for improvement in feedback quality [117]. However, more detailed discussions and fine-grained studies on reasons for lacking user satisfaction are rare in recent literature. Thus it is desirable to gain more insight into possibilities and limitations of analysis techniques in order to estimate the possibilities of useful feedback that can be given by these methods. More detailed measurements and figures are available for single aspects of feedback. Especially accuracy of feedback is evaluated in many publications, both for programming exercises [95, 16] and for diagrams [138]. For automated grading of free-form texts there are even detailed experiments on the comparison of results from manual grading and automated grading available, but only focused on marks and not on detailed feedback [136].

Another approach taken in the literature in some cases is to compare results for computer supported courses with results from traditional courses (e.g. for a course using the system RoboProf [32]). However, there is a drawback in the methodology, because for legal reasons there cannot be two groups treated differently in the same term on the same course in most cases. Thus studies like this always suffer from that fact that they compare results from different terms or different courses. In order to avoid this, artificial experiment setups have to be used, which in turn may easily lack essential properties of the real scenarios, like student motivation, pressure of exams, or similar. Because of the complexity in study design involved in this area, any of these studies are considered out of scope for this thesis.

	very positive	positive	neutral	negative	very negative	number of answers
Attitude towards the general use of ASB (data taken from [101])	13%	44%	21%	21%	0%	52
Overall impression of Marmoset (data taken from [125])	30%	46%	17%	7%	0%	70
Students' rating of PASS (data taken from [25])	13.3%	50.4%	24.4%	8.9%	3.0%	?
Attitude towards the use of Web-CAT (data taken from [121])	21%	46%	26%	8%	0%	39
Feedback rating for an automated grader component for Sakai (data taken from [133])	20%	30%	24%	14%	10%	71

	excellently	well	moderately well	badly	number of answers
Perceived feedback quality for Scheme-robo (data taken from [118])	6%	44%	41%	10%	229
Perceived quality of support by Graja (unpublished data kindly provided by Andreas Stöcker)	32%	51%	17%	0%	53

Table 2.2.: Comparison of attitudes towards different e-assessment systems

Nevertheless also the simpler studies show that there is just medium progress in the dimension of feedback and thus indeed a gap to be closed. This gap is based on the difference between the insight that can be generated in general by automated methods, and the quality feedback that is actually available for the students using the system. Improving the feedback quality as perceived by the students has been identified as an important goal by several publications.

2.5. The Existing Gap

The literature review shows that there is indeed different progress in the four dimensions: In the dimension of systems, good progress can be reported and there is no need for yet another e-assessment system. In the dimension of techniques, weaker progress and a clear gap could be identified, since there are sophisticated techniques available but not integrated into e-assessment systems. In the dimension of inputs, good progress could be reported if viewing this dimension on its own. However, dependencies could be identified, so the gap in other dimensions may also reveal gaps in the dimension of inputs. In the dimension of feedback, medium progress can be reported as evaluation results show positive tendencies, but also space for improvements.

In summary, it can be confirmed that there is indeed a gap between the potential power of automated assessment systems and the actually available power. The review suggests to tackle this gap mainly in the dimensions of techniques and feedback, while starting from an existing system.

3. Application Environment

As discussed in the literature review in the previous chapter, there is a large number of automated assessment systems with a relatively narrow focus as well as fewer systems that provide a more open environment or framework. As stated in the goals in chapter 1, this thesis focuses on contributing to the research on techniques and feedback generation that can be integrated into an existing system. Thus the application environment for this thesis must be an automated assessment system that supports both the grading of summative exercises and automated tutoring for formative exercises. Moreover, it must be a system that is able to handle virtually any kind of software engineering artefact. In particular, the system must not be limited to a single programming language or similar. Ideally, the system is modular in order to plug in new modules for feedback generation.

The automated assessment system JACK is such a system that has been used in both ways since 2006. Many development steps of JACK have been done in the context of this thesis and hence this thesis considers usage scenarios that have taken place with JACK at the University of Duisburg-Essen. This chapter presents an overview on JACK and its system design, with focus on the parts and features of JACK that are relevant for the analysis of software artefacts. More details about different versions of JACK have been published earlier as technical reports [131, 50] and an overview paper [126].

3.1. Requirements

The design goal of JACK was to develop a flexible system architecture for e-assessment systems that allows automated grading and feedback generation for solutions submitted electronically. The grading methods created in this thesis and at other places are intended to be used in different domains and contexts. In particular, they should not be tight directly to one particular type of exercise or course. Instead, it is considered beneficial to combine some of them in an actual exercise to get a more detailed analysis of the solution artefacts submitted to this exercise. Thus JACK had to provide a generic and flexible system architecture that allows to add and manage grading components independent of each other and separately for each type of exercise. Consequently, there was no need to restrict the system to specific types of artefacts by design. This is also assumed to be beneficial for students, because they can use one system for different subjects instead of learning how to use different learning, tutoring and assessment systems for different subjects.

The use case of e-learning also proposes additional requirements regarding accessibility and communication interfaces. Especially in the context of distance learning or ubiquitous learning, a software system must be accessible from everywhere for tutoring and self-training, but possibly only from an exam hall for assessments.

Organizational and judicial issues emerging from rules made by examination authorities are not considered here. They are out of the scope of this thesis because they are entirely independent from technical aspects of marking solutions or managing exercises. Although the architecture is intended to be used for various subjects, the definition of interfaces for exporting information to administrational databases or to import information from them is independent from the subject the system is used in. Thus it is assumed throughout this chapter that any information about a student that is relevant for running the system is stored inside the system itself.

3.1.1. Exercise Management

For the purpose of this chapter, each exercise is understood as a task of a certain type, for example a multiple-choice question, a programming task or writing a short essay. Each exercise contains a collection of resources, and information about how the overall result in this exercise is composed of the results from different marking techniques. The latter is necessary since the architecture should allow to attach and detach different grading components to the system and configure them individually for usage in a single exercise.

Each resource represents some artefact that is relevant for an exercise. Four different types of resources can be identified:

- An "instruction sheet" contains meta information about the exercise and explains the task to the student. An instruction sheet is created by the teacher, provided to the student and not intended to be changed by the student while working on the task. Hence it is not included into a submission of a solution.
- A "working sheet" is intended to be changed by the student while working on the task. It is either created by the teacher and provided to the student or has to be created by the student on its own. In any case it has to be included into a submission of a solution. In the subject of computer science, a working sheet may be a source code stub or an incomplete diagram that has to be completed.
- A "reference sheet" represents a resource that is necessary to complete the task but must not be changed itself. Similar to an instruction sheet, it is created by the teacher, provided to the student and not included into a submission of a solution. In contrast to an instruction sheet, it does not contain meta information, but is integral part of the exercise. In the subject of computer science, a reference sheet may be a give Java interface that has to be implemented or an element library that has to be used in a diagramming exercise.
- A "hidden sheet" contains information that is relevant for checking solutions and thus not visible to the student. Thus it is available only on the server and never provided to the student. It may contain information for teachers, resources necessary for running a checking technique (e.g. graph transformation rules) or reference input for checkers like a sample solution (e.g. correct answers in a multiple-choice test).

It is required that all four types of resources can be attached to exercises uniformly.

Exercises can be composed for two different purposes: Either as an exam that is made up of several exercises selected by some didactic criteria or as a course containing an arbitrary number of exercises covering the same topic.

Each exam is hence a collection of exercises in a certain order and enriched with information about how the overall result in the exam is composed of the results of single exercises.

In addition, each exam needs a list of participants for granting them access to the exam and storing their individual results. If necessary, alternatives, variants or random ordering of exercises can be specified in the exam, too. Exams must not claim exclusive rights to use an exercise, thus one exercise can be included in several exams.

A course is a less restricted environment that does not use participation lists to restrict access and that does not contain a fixed set of exercises. Instead, it selects all available exercises based on some filter criteria.

3.1.2. Exercise Delivery and Solution Submission

As direct consequence of the requirements regarding flexible exam and exercise composition shown above, a high flexibility in the frontend or client part of the system is required in order to deliver exercises. Basically, there is a choice between internal (browser-based) and external presentation as well as manual or automated communication. Since internal communication can always be automated, three different scenarios to be supported can be identified:

- The simplest case are exercises that can completely be presented and solved using a web page. In this case, the presentation of the exercise is prepared internally, either on the e-assessment server or with client-side code delivered by the server. Exercise meta information may contain formatted text and embedded media elements, while solutions may require plain text input or usage of checkboxes or drop down lists known from HTML forms. Theoretically, JavaScript and HTML5 allow a large amount of interactions with a web page and thus allow to solve even complex exercises this way. In addition, plug-ins embedded in a web browser (e.g. a flash player) allow even more exercise specific interaction. In the subject of computer science this may be sufficient for e.g. simple diagramming tasks or programming. However, advanced features that can help students during the creation of a solution (e.g. a compiler or a run time environment for a programming language) may be not available inside a browser because of general limitations of a browser environment or because of user specific security settings. Thus using web pages is the most native option for a server-based e-assessment system and also the one usable on virtually any web-enabled device, but also the one bearing the most uncertainty, because many different browsers with many different capabilities and settings exist.

- These limitations can be overcome in general by providing the resources of an exercise as downloadable files that can be opened and manipulated in an external application. As a consequence, the e-assessment system has to provide a possibility where these files can be uploaded for submitting a solution. In this case, only parts of the exercise are presented directly by the e-assessment system via a web page, while others require to involve an external application. This method can be considered to be sufficient for any kind of exercise. However, it requires the students to manage the downloaded and modified files very carefully and is thus not very comfortable. Moreover, it requires the availability of appropriate external applications. Thus with this method there is a risk that students do not have access to an appropriate application or that they are not able to run it on the devices available to them.

- The most elaborated way is to provide dedicated communication interfaces on server side and plug-ins for external tools to allow direct download of exercises and submission of solutions. In this case, the external application must be able to present all necessary exercise information. This requires in any case not only to provide the e-assessment system itself, but also at least plug-ins or completely configured external applications. As above, there is still a risk that these do not run on a particular device.

It is not desirable to limit the choice which way is used to deliver a certain exercise by the type of this exercise. For example, a diagramming exercise could be provided in a way suitable for a browser plug-in, but also as a set of files for free download or for downloading via a specialized IDE plug-in. Consequently, each implementation of an user interface for delivering exercises is required to specify and document which types of exercises it can handle.

Another important aspect in delivering exams and exercises is security. In closed exam scenarios it has to be ensured that only authenticated students get access to exercises and that all exercises that have to be delivered to a specific student are indeed delivered. Similarly, solution submission may only be allowed to authenticated students. In less closed self-training or tutoring scenarios, there is no need for restricted access, but complete delivery of exercises has to be assured as well. In both scenarios, each solution must be stored in a way that makes comprehensible from whom it was submitted, from where it was submitted, when it was submitted and to which exercise it belongs.

In terms of usability, each possible frontend or client is required to provide a simple user interface to ask for login data in a simple way, provide input for single exercises, allow the creation or modification of solutions and the submission of solutions to the assessment system. When an exam is composed of more than one exercise or one exercise contains more than one sheet, the client must ensure that a student is given notice of all of them. In addition, the client is required to inform the student if working sheets are missing or have not been modified yet.

3.1.3. Grading and Feedback Generation

As a consequence of the goal to analyse artefacts using different techniques, the system must be able to handle different grading components, each implementing one particular technique. In detail, flexibility with respect to two aspects is needed: First, it must be possible to add different grading components to each exercise. Second, it must be possible to configure a particular grading component with different inputs for different exercises or even two instances of the same grading component in one exercise. Since the system is intended to be used for automated grading and feedback generation, each grading component is required to return feedback in terms of human readable error messages and a mark from 0 for a completely wrong solution to 100 for a completely correct solution. Wherever possible, the error messages are required to be related to the solution artefacts as close as possible. Where applicable, it is desirable to offer additional warnings or messages that do not contribute to the overall mark of a solution but add valuable feedback about the quality of an artefact. For example, an error message can point out a `NullPointerException` in Java programming reducing the mark for this solution, while an additional visualization shows the data structure in order to give a better understanding of this error without contributing to the mark.

As a general concept, the system should support synchronous and asynchronous grading. In synchronous grading, a submission should be handed over to the necessary grading components right after submission to receive immediate feedback. In asynchronous grading, solutions may be stored in the database and handed over to grading components later. Synchronous grading is at least required in self-training scenarios where an exercises consists of several steps and feedback should be presented after each step without interrupting the learner for a long time. Asynchronous feedback is at least required in the mass validation of exams where each individual grading process is potentially a long-running task.

All grading components should be easy to develop, integrate and manage at run time. They also must be able to run outside the actual assessment system to allow load balancing: Since the students' access to the assessment system must be as robust as possible and the system resources needed for marking are type-specific and not predictable, it is necessary to loosely couple these tasks. Thus a run time system is required that executes arbitrary grading components and connects to the assessment system for this purpose. The assessment system is responsible for delivering solutions on request of the marking process until every single solution has been assessed and marked. Running grading components in a separate run time system that connects to the assessment system raises additional security issues. To prevent fraudulent attempts, the assessment system must make sure that only valid marking run time systems may access stored solution data and may deliver results for them.

There should be a mechanism allowing grading components to reuse results from other grading components as additional input. To allow for this, proper chaining of grading components must be possible, where some components are only invoked if others have already produced results.

3.1.4. Review Management

Independent from any automated grading process, it is required that a teacher can access any submitted solution in order to review it and to add a manual result to it. Moreover, it must be possible for a teacher to review any comments made by an automated grading component and to remove any single error message or warning generated. This way, “false negatives” in automated grading can be detected and removed manually.

Obviously a well presented list of errors and warnings for a given solution has to be a core feature of a tutoring system. As already required in the previous section, each message should be related to a solution artefact as close as possible and this relation has to be comprehensible in the review interface. Since students have to learn just by studying the results from the grading components in pure e-learning scenarios, the feedback has to be presented in a way that is clear enough to be understandable without further explanation. To facilitate quality assurance, students should be allowed to vote up or down on feedback, indicating whether a particular line of feedback was helpful or not to them.

Also in exam scenarios it should be possible to let students review their own solutions afterwards and see a justification of the results, both for educational and judicial purposes.

3.2. System Design

The requirements sketched above can be turned into a data model representing the entities handled by the system and an architectural model representing the system components. Both models are discussed in short in the following subsections.

3.2.1. Data Model

The data model of JACK is shown in figure 3.1. The entities present in this model can roughly be divided into three groups: (1) Entities concerned with exercises and their arrangement into courses and exams, (2) entities concerned with solutions to exercises and feedback given to these solutions, and (3) entities managing users and their rights.

Exercise and Examination Data

The central concept to manage exercise and examination data within JACK is the entity "AbstractExercise" shown on the top left in figure 3.1. It is specialized for actual exercises and courses and holds meta-data about the exercise (such as a title or rules for submission). For exercise delivery, each exercise states a display type in the meta-data. Depending on this type, the user interface can decide how to display the exercise to the user or even to not display it at all because of missing capabilities to show it properly. Files belonging to the exercise are attached as resources, belonging to one of the four different types of resources identified in the requirements. Resources are never shared among exercises, but always belong to exactly one exercise. The checker configuration attached to the exercise states which resources are handed to which grading component for which purpose. Consequently, it is not necessary to make all resources visible to all grading components and it is also possible to invoke the same grading component with different configurations and resources on the same exercise. As part of the meta-data of an exercise it is specified how the final mark for each solution in calculated from the individual marks given by each of the grading components configured for this exercise. It is not necessary to use each grading component in this calculation, so a component can also be used to create additional feedback that does not influence the mark or is not visible to the student at all, but just to the teacher.

Exercises can be grouped to courses based on their display type and their tags, and they can be assigned to exams. Both entities carry their own meta-data.

Solution Data

In analogy to the entity "AbstractExercise" there is an entity "AbstractSolution" (top right in figure 3.1) which is the core concept for handling the user submission. Besides containing meta-data like a submission timestamp, it is mainly a container for the resources submitted by a student and the feedback generated by the grading components or by manual results added by a teacher. For the files attached to a solution there is a distinction on whether they are user generated (thus submitted by the user) or generated by a checker (thus attached later during the checking process). Feedback is stored in error records, where each individual line of feedback can be ruled out by a teacher. Moreover, a student's vote on the feedback can be stored. The marks given are in all cases handled as integer numbers from 0 to 100. Grading components may also return additional attributes that are stored alongside the solution. This can be used to pass additional information from one grading component to another.

Courses and exams do not carry own attached files, but just meta-data and overall results. For each solution submitted by a student and each asynchronous checker configured for the associated exercise, a job entity is created, which is used by the grading management and deleted as soon as results from the respective checker are available. For synchronous checkers,

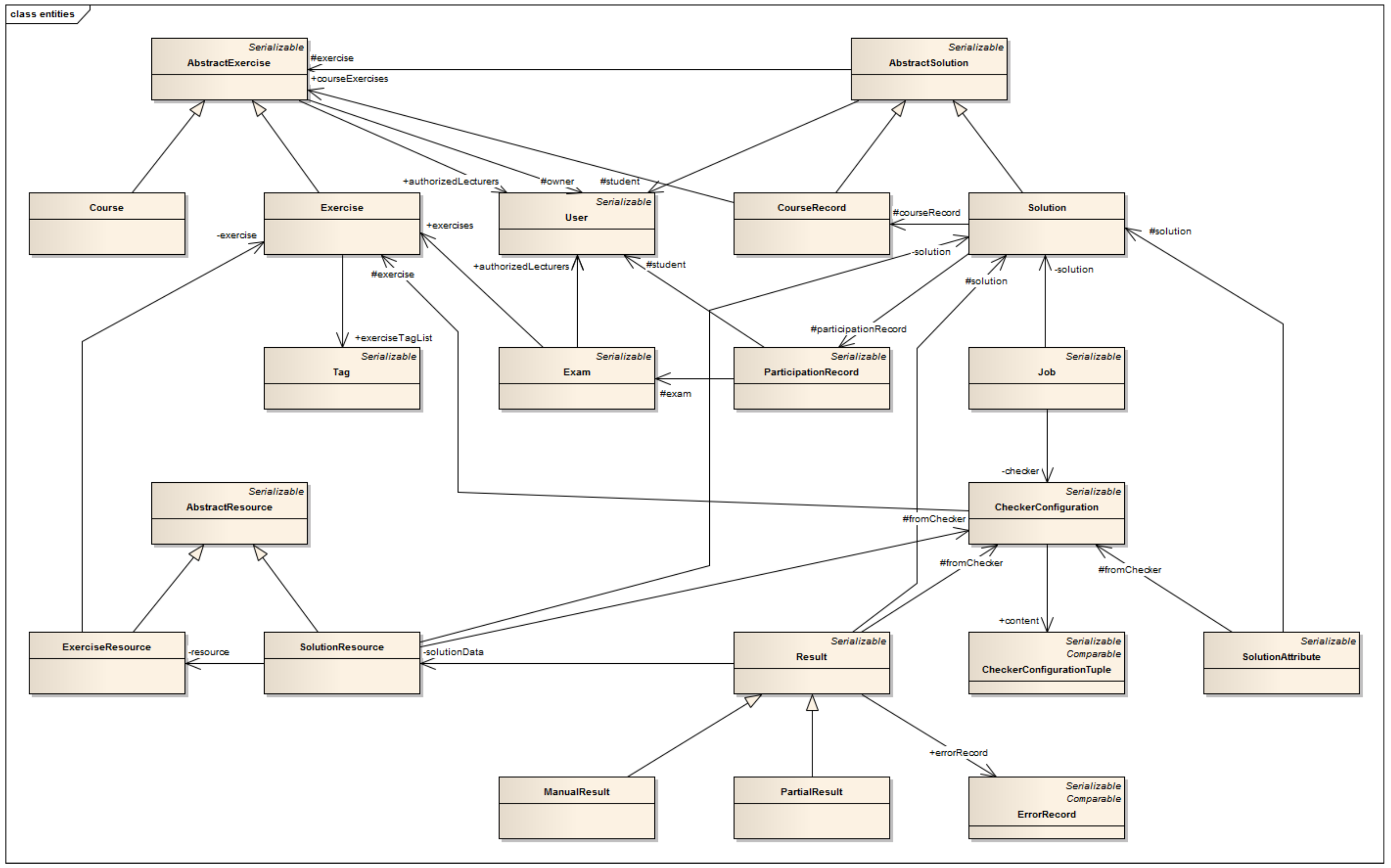


Figure 3.1.: Class diagram for the data model of JACK. Primitive attributes on all entities as well as multiplicities of associations are hidden for the sake of readability.

no jobs need to be created, as the solution can be handed directly to them, obtaining the results immediately.

User Data

User data is the smallest chunk of data handled by JACK. It consists of entities for each user, that can be either students, teachers or administrators. Besides user name, password, and role, user data only consists of information on which entity has been created and is thus owned by which user. For teachers, additional access rights can be granted to exercises, courses, and exams.

Participation of users in exams is covered via an additional entity "PatricpationRecord" that may also contain an individual password required for access to this particular exam.

3.2.2. System Architecture

The component based system architecture of JACK is shown in figure 3.2. The architecture is organized in two parts to be run on different servers: The part running on the core server is designed as a classical three-tier architecture and cares for data storage, business logic, synchronous grading, and various user interfaces for teachers and students. For user authentication, external services can be called. Both the web-frontend and the web-service for ECLIPSE integration are concerned directly with the appropriate presentation of exercises and exercise resources. They also provide to the learners the necessary means to submit a solution. Both the web-frontend and the web-service for workers also deal with presentation of solutions, either for manual review via the web frontend or for automated grading on another server. They also provide the necessary means for submitting results from manual assessment by teachers or automated grading components, respectively. Finally, the web-frontend also provides means for authoring, import and export of exercises, courses and exams by teachers.

All three components call methods of the business logic core to provide or request data. The business logic core cares for processing of this data, e.g. permission checks for access to exercises and exams, running synchronous checks on solutions, and maintenance of the job queue for asynchronous checks. It in turn calls the persistence layer or external services to fulfill its tasks.

The part running on worker servers is designed as service-oriented system and cares for asynchronous grading. Each worker server can be equipped with different marking components. Each of the checking concepts discussed in this thesis can be realized as concrete, complete, runnable implementation in such a component. Several worker servers can connect to the same core server and one worker server can also connect to several core servers. They issue periodic requests for work to the server and use the methods provided by the web-service for workers to retrieve data and send back results later if a job is available for them.

3.3. Implementation

JACK is implemented in Java, using EJB3 for the core server, JSF2 for the web-frontend, OSGi for the worker servers and grading components, and web-services for the communica-

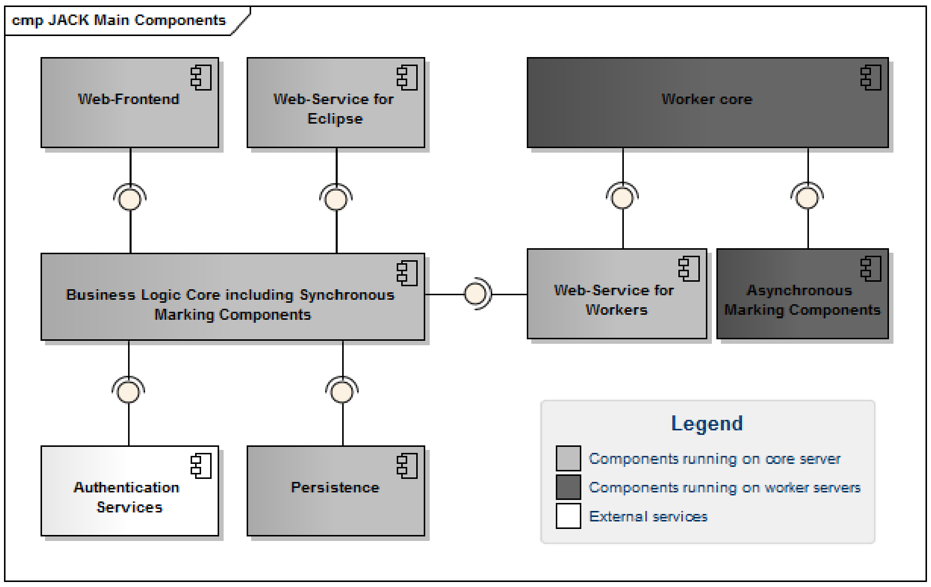

cmp JACK Main Components
Web-Frontend
Web-Service for Eclipse
Worker core
Business Logic Core including Synchronous Marking Components
Web-Service for Workers
Asynchronous Marking Components
Authentication Services
Persistence
Legend
Components running on core server
Components running on worker servers
External services

Figure 3.2.: Architectural overview of JACK showing all main components. One core server can serve multiple worker servers and one worker server can connect to multiple core servers.

tion between core and workers. Access for other clients like Eclipse is offered by web-services as well. Since several marking components will be designed and discussed in this thesis, some more detailed implementation characteristics are presented in the following.

Each marking component (also called "checker component" or "checker" for short) implements a Java interface named `IChecker` which defines an unified interface for all marking components. The interface defines four methods: The method `getCheckerId` takes no parameters and is supposed to return a unique string identifying a marking component. The method `doCheck` takes arbitrary resources and configuration information organized as typed pairs of name and value as input and returns an object containing the checker result and feedback messages. The method `getNewResources` takes no parameters and returns a list of arbitrary resources, if these are created during the grading process. The method `getCheckerAttributes` takes no parameters and returns a list of arbitrary pairs of name and value, if these are created during the grading process.

An illustrative sequence of calls to these methods is shown in the sequence diagram in figure 3.3. It displays a complete possible sequence of calls made during a grading process for a solution to an exercise using an asynchronous marking component. Note that the work of more than one different grading components can happen in parallel if several worker servers are attached to the same core server.

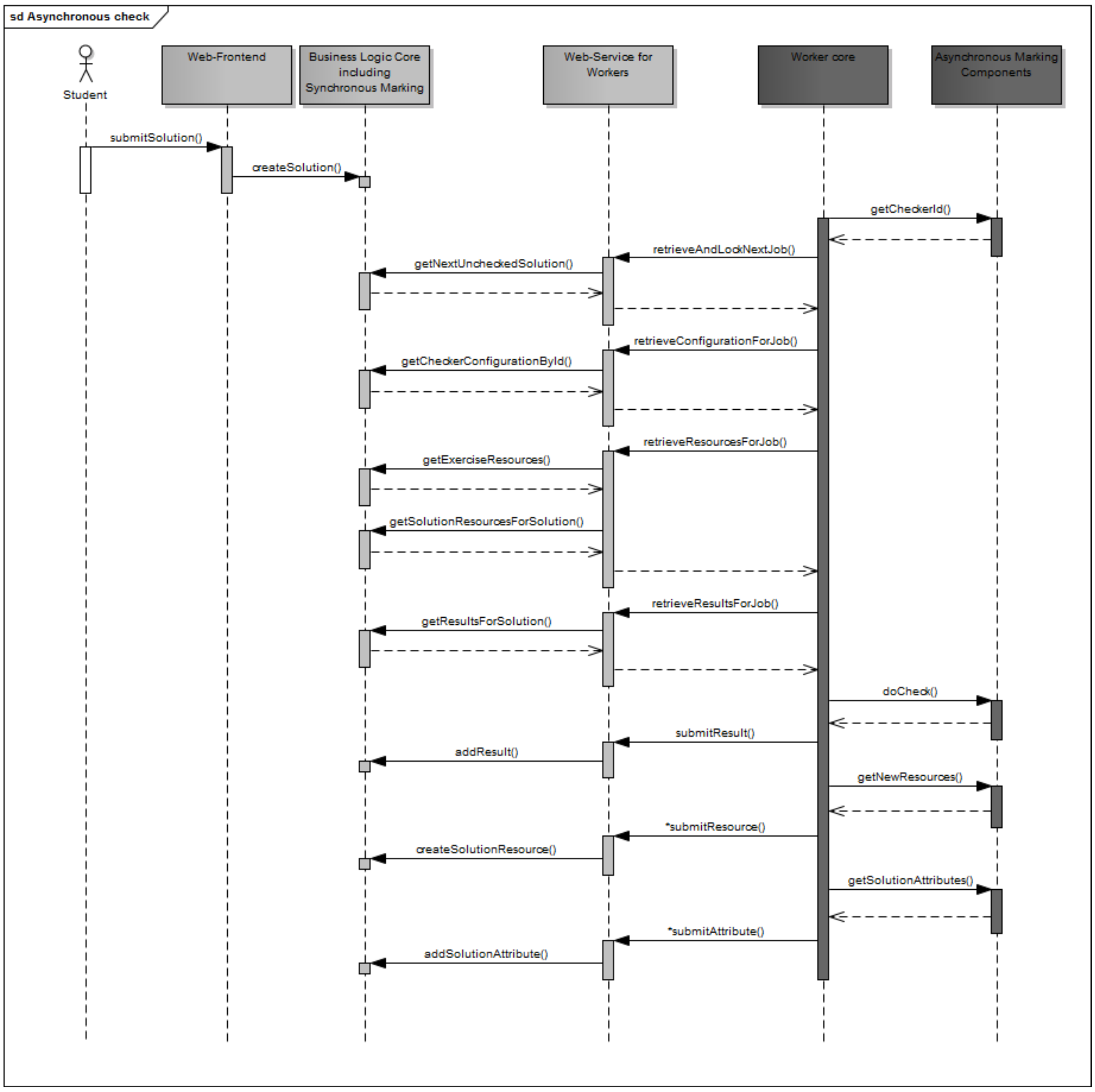


Figure 3.3.: Sequence diagram illustrating the calls issued when performing grading on an asynchronous marking component.

Part II.

Concepts and Realizations

4. Static Checks

One possible and common way to look at any software and design artefact is to look at its written or drawn representation. In this case, the artefact is primary considered based on its syntactical construction. Semantics, e.g. the operational meaning of program statements denoting an algorithm, are not considered in this step in the first place. Instead, the artefact is decomposed into syntactical elements and the relation between these elements and their attributes are examined. It has to be noticed that relations between different representations may be based on semantics but nevertheless both representations are subject of static analysis. For example, from program code a system dependency graph can be derived as it is done for Java by Walkinshaw, Roper, and Wood [158]. This graph contains explicit denotations for control flow and data flow, which concerns the semantics of program code. However, the system dependency graph can be handled as a software artefact itself and be analyzed by static checks based on its syntactical representation. As another example, theorem proof systems can be used to show semantic properties of formulas based on purely syntactic transformations.

This chapter of the thesis discusses techniques that can be used to check those syntactical representations of software artefacts. The results of these checks can either be used to provide direct information about the quality and correctness of the artefact, or they can be used as additional input information for subsequent dynamic checks. As already mentioned above, checking syntactical representations does not mean to limit checks on syntactical issues. For example, defining recursive methods in an object-oriented program is a semantic issue, since method references have to be resolved. However, the necessary semantic analysis can be done based on the syntactical structure of the program and thus recursive methods can be found by static analysis.

As pointed out by the literature review in chapter 2, current usage of static analysis in e-assessment tools suffers from two problems: First, any implementation currently in use is limited to a specific kind of software artefact, such as program code in a particular language. Second, most implementations are focused on particular features to assess, such as searching for style errors or functional bugs. What is missing is an implementation that is general enough to (1) handle a broad range of artefacts, (2) detect a broad range of features and properties of the artefact under examination, and (3) reuse the same schema for input and feedback generation independent of the actual type of artefact and property to check. To fill this gap, this chapter looks for appropriate techniques, implements them and demonstrates their usefulness and suitability.

This chapter is organized as follows: Sections 4.1 and 4.2 focus on the concepts needed for static analysis of software artefacts. Section 4.3 presents actual implementations of these concepts, which in turn are used for the application examples discussed in section 4.4. The chapter is concluded by a discussion of remaining open problems in section 4.5 and a chapter summary in section 4.6.

4.1. Suitable Data Structures

Static analysis based on the syntactical construction of artefacts is tightly connected to the language and grammar the representation is written in. However, several data structures can be used to express a given set of syntactical constructs of a given language. For example, a piece of source code can be represented textually using strings as data structures, but also using an abstract syntax tree. A diagram can be represented graphically using an attributed graph, but also textually by listing all elements and associations between them. Strings, trees and graphs can be understood as high level data structures that can in turn be represented by low level data structures, e.g. strings as arrays of characters and graphs as lists of nodes and edges. The remainder of this chapter only talks about high level data structures and considers low level data structures as a means of efficient implementation of the high level ones.

At a first glance it seems to be useful to define a primary notation for each software and design artefact. For source code, this could be the textual representation, as program code is usually written and displayed in a text editor. The abstract syntax tree is then a derived representation, generated by a parser while reading the textual representation. However, software engineering tools exist that generate or manipulate programs by working directly on the syntax tree. A textual representation is in these cases generated out of the syntax tree and thus not the primary representation. Consequently, it is not possible to define one representation to be the primary one for any purpose, but it is legal to switch representations whenever a more appropriate one is available for the current purpose.

Even if there is no primary notation for a software or design artefact, there might by a notation that is most appropriate for static checks, independent of the kind of the artefact. The possible choices are as follows:

- **Strings:** The artefact is represented by a linear sequence of tokens where each token has zero or one predecessor and successor, respectively. Each token may have a type and an actual content. Both the token itself and its position in the sequence may carry some meaning. As mentioned above, a string-based representation of a program is the typical representation used by humans while editing source code.

- **Trees:** The artefact is represented by a set of nodes where each node has zero or one parent and an arbitrary number of children. Each node may have a type and a set of attributes containing some content. Both the node itself and its position in the tree may carry some meaning. Moreover, the parent-child-relations may carry additional meaning by attributes as well, e.g. by ordering information. A parse tree is a typical tree-based representation of a program.

- **Graphs:** The artefact is represented by a set of nodes where each node has an arbitrary number of relations to other nodes. Both the nodes and the relations may carry some meaning by having types and attributes. If the graph is directed, the position of a node in the graph may also carry some meaning. Graphs are a typical representation for diagrams such as UML Class Diagrams.

To determine which representation is the most appropriate one, requirements have to be collected. As already stated above, static analysis is concerned with syntactical elements and their relations. Thus as our lower bound, we are only interested in constructs that are at least syntactically correct. A suitable data structure must be able to represent any correct arrangement of syntactical elements. Thus string-based representations are considered insufficient: If four syntactical elements need to be related to each other, this cannot be done with strings because each token in a string can have at most two tokens (a predecessor and a successor) connected to it. Connecting to more than two elements is possible in tree-based and graph-based representations.

Furthermore we are interested in any kind of software or design artefact and thus especially not focused on a particular language with a limited set of features. But we are only interested in artefacts based on a formal language which can be parsed automatically, but not in natural language. Parsing constructs in general results in a syntax tree (or several possible trees if the grammar is ambiguous). Thus tree-based representations may be a sufficient representation for any kind of artefact. Differences between the syntax tree and the abstract syntax tree of an artefact are not considered here, since they do not affect the form of the data structure. However, trees are limited in that way that they do only allow one parent per node. They thus cannot represent different kinds of relations between element, like "is defined in" and "is called by" for methods in program code. Since it might be interesting to check for recursive methods in an object-oriented program, it is necessary to have this information available. Graph-based representations can be used to store this information, because they allow to have an arbitrary number of connections between nodes.

Thus attributed graphs are considered an appropriate representation of any software or design artefact for analysis purposes in thesis. An attributed graph G is considered to represent a software or design artefact A in the following manner:

1. Syntactical elements of A are represented by typed, attributed nodes in G.
2. Relations between syntactical elements in A are represented by typed, attributed edges in G.
3. The (abstract) syntax tree of A is a subgraph of G.

Graphs fulfilling these requirements are called "syntax graphs" throughout the remainder of this chapter. The technical process of how to create a syntax graph for a given artefact is explained by example in sections 4.3.1 and 4.3.3, where the implementation of components for static checks are discussed. Because of requirement (3), any syntax tree provided by a parser is a valid syntax graph, so in general no mandatory actions beyond parsing the artefact have to be performed unless additional information should be stored in the graph. See listing 4.1 for a sample piece of Java source code and figure 4.1 for the syntax graph belonging to it.

The rules of the underlying grammar of each syntax graph can be encoded in a type graph or meta-model for the syntax graph. It covers the possible associations between syntactical elements as well as generalizations such as abstract elements for statements or expressions. Besides generalizations given by the grammar, it can also be used to encode additional information which can be useful to simplify subsequent analysis steps. For example, additional generalizations can be introduced to create an abstract type covering all types of loop statements available in a programming language.

Type graphs for Java 7 and a subset of UML2 are given in the appendices A.1 and A.3 of this thesis. However, alternative type graphs are possible for Java as well, that in

```
class Example {
    String s;

    Example(String s) {
        setString(s);
    }

    setString(String s) {
        this.s = s;
    }
}
```

Listing 4.1: A sample piece of Java source code.

particular may introduce different generalizations not directly given by the grammar [115]. As a consequence, analysis tools or techniques based on a particular type graph are not compatible with other tools and techniques locked to a different one. In order to avoid this limitation, the remainder of this chapter does not rely on a particular form of type graph, but just assumes that a type graph exists. Nevertheless, an actual configuration of a static analysis tool has to include the type graph in some way in any case, thus counting the type graph as part of the input for this kind of static analysis of artefacts.

4.2. Analysis of Syntax Graphs

Once a syntax graph is generated, it can be searched for patterns of nodes representing syntactical constructs. Both the presence and absence of patterns can be interesting, depending on whether they represent a desired or undesired feature of the artefact. Moreover, an interesting pattern does not necessarily imply an error in the artefact. Instead, it is possible to use results from static analysis to get e.g. candidates for lines or blocks of code that need special observation in dynamic tests.

Finding occurrences of a pattern in a graph is a matching or querying problem. Two different solutions will be discussed in the following sections. The first one is the use of a graph transformation tool, which allows for graphical notation of the patterns and which involves constraint solving to solve the matching problem. The second one is the use of a graph query engine, which uses a textual notation of the patterns and which involves traversing the graph structure to solve the querying problem. In addition, both approaches use different ways of defining feedback messages on existence or non-existence of patterns.

4.2.1. Rule Definition by Graph Transformation

The first technique for analyzing syntax graphs implemented in JACK prior to this thesis is based on graph transformations [88]. Graph transformation techniques were chosen because they cannot only be used for pattern matching in graphs, but do also allow to manipulate syntax graphs. This was supposed to be beneficial for aligning slight syntactical divergences as a preprocessing step before applying the actual analysis. For example, `i += 1;` is a semantically equivalent, but syntactically different short version of the longer `i = i + 1;`. To avoid two versions of each checking rule dealing with assignments like this, a general

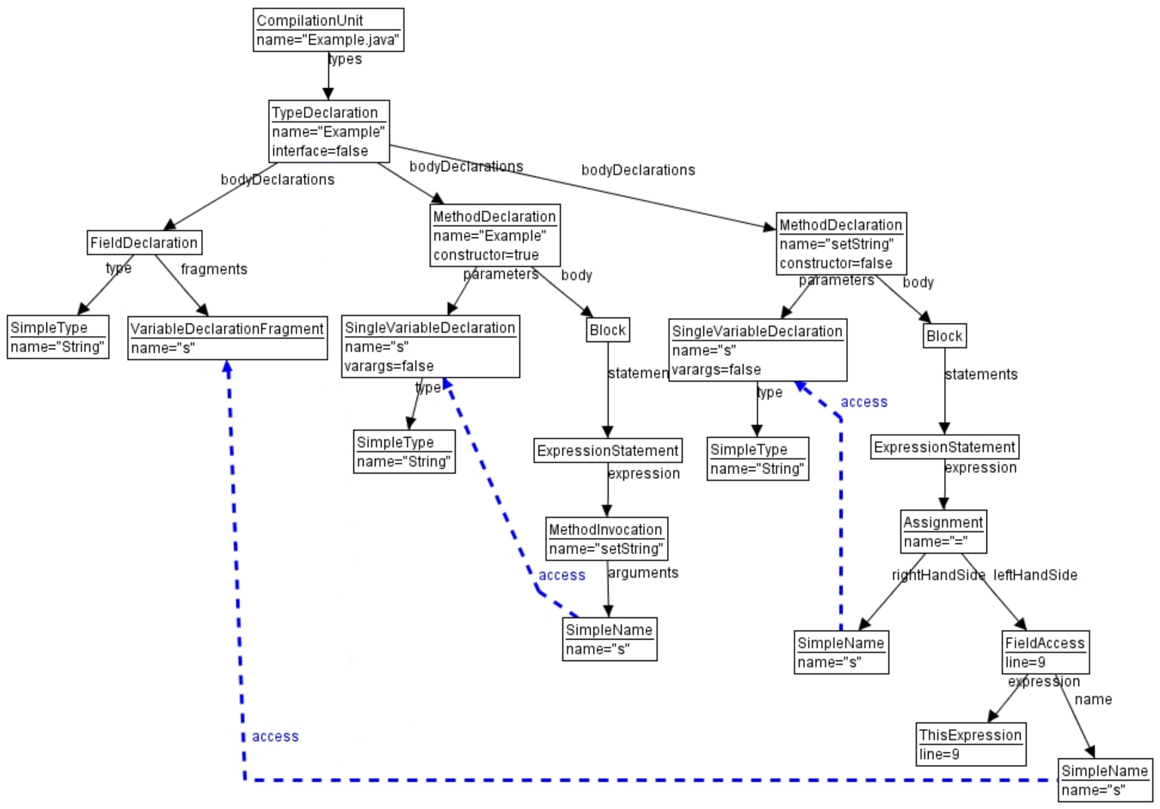


Figure 4.1.: Example of a syntax graph for the piece of Java source code given in listing 4.1. Boxes represent syntactical elements and their properties, where the root node (*CompilationUnit*) represents the whole document. Solid arcs form the abstract syntax tree according to the rules of the Java grammar. Dashed arcs extend the tree to a graph by resolving references.

transformation can be applied before, replacing all long statements with an according short version. These changes are only performed internally for checking the solution, but not written back to the original code submitted by the student. However, writing back such changes is well known from program refactoring and can also be used to merge elements and insert additional statements into the code as an initialization before performing dynamic checks.

Graph transformations are realized by applying a series of transformation rules to a host graph [43]. The first step of the static analysis is to transform the source code into a graph structure by using a tool named *java2ggx*. This tool was designed as general-purpose tool to treat Java source code as an abstract syntax graph in the "GGX" file format of the Attributed Graph Grammar (AGG) system [134, 6] to be used for different kinds of more extensive analyses and transformations. It originally supported the Java 5 syntax and had been extended to Java 6 later on [127].

The general process of syntax analysis by graph transformation rules has been sketched in a technical report [131] as follows: When the solution source code is represented as a graph, the checker can start to apply rules. Each rule consists of a left hand side (LHS) and a right hand side (RHS) and possibly one or more negative application conditions (NAC). First, the transformation engine tries to find a match for the LHS in the host graph, i.e. a set of nodes and edges that are connected the same way and having the same types and attributes as the LHS of the rule. Rules do not have to specify all attributes, so it is possible to make a match on a node representing a method with an attribute with a certain name, but to neglect the attribute telling whether this method is `public` or `private`. If a match is found, the engine tries to find matches for the NACs if they are specified. Whenever a match for a NAC is found, this rule is discarded and the next one is processed. If no NAC matches or none is specified, the rule will be applied. In this case, all nodes and edges that are present in the LHS are replaced by elements from the RHS, if there is a replacement defined for them. If no replacement is defined, they are removed from the host graph. If the RHS contains elements without origins on the LHS, they are inserted into the host graph. Inside JACK this is done by using the AGG API and an additional control script handler. The script handler allows to apply rules in a certain order and to apply rules only if other rules matched before. Typically, checking rules can be designed in one of the two following ways:

- Optimistic rules assume that the solution is correct unless an undesirable structure is present. In this case, the erroneous structure is placed on the LHS of the rule. The RHS contains the same structure, because the original source code should remain unchanged. An additional “error node” is added to the RHS, which is a node of a particular type that does not appear in normal syntax graphs and that contains a message describing the detected error. So whenever the erroneous structure is present in a solution, this rule matches and inserts the error node into the graph. An additional NAC has to be written which contains a copy of the error node to assure that the rule is not applied twice.
- Pessimistic rules assume that the solution is wrong unless a certain structure is present. In this case, the LHS of a rule is empty and correct structures are added as NACs. The RHS contains only an appropriate error node. Because of the empty LHS, this rule can always be applied, except if one of the correct structures is present. Again an additional NAC assures that the rule is not applied twice.

After all rules have been applied, all error nodes created during the process are filtered from the graph and the feedback messages are collected for the final list of feedback. If grades or scores are to be given in e-assessment scenarios, they can be attached to the error nodes as an additional attribute.

More sophisticated rule sets can use preprocessing rules for manipulations to add auxiliary nodes or edges with additional information. For example, they can add counters for recursive method calls that can in turn be used to perform more complex checks. In addition to using NACs, rules can be parametrized by using attribute conditions. For example, a check may be performed for field declaration that are either `private` or `protected`, but not for those that are `public`. Of course this could be done by two versions of the respective rule, each using the appropriate attribute. A more convenient way is to define a rule variable for this attribute and add an attribute condition, forcing this variable to contain either “`private`” or “`protected`”. Rule variables can also be used to record attribute values from the LHS when finding the match and to reuse them on the RHS when placing an error node.

While this solution works without limitation on the conceptual level, there are some serious limitations in practical application: Writing checking rules the way described above can hardly be considered intuitive, because up to three graph patterns (LHS, RHS, and NAC) have to be specified for capturing just one situation. This problem can be solved by advanced tool support, which hides the technical details of graph transformation rules by generating these rules automatically from more general specifications. Details for this process are part of an ongoing research project which is not part of this thesis. Besides tool support for editing, also capabilities of graph transformation engines are critical. The data format used in graph transformation engines typically contains a so-called “host graph” which is the start graph for a graph grammar, a “type graph” describing legal graph structures, and a set of transformation rules. This is reasonable from the view point of graph grammars in terms of languages, where a complex graph is derived from a (possibly empty) start graph by applying rules. However, this does not fit the concept of static code checks, where different sources of input exist: The host graph is the syntax graph generated by the parser, the ruleset may be composed from different sources mixing general and exercise specific rules from different authors, and the type graph must be identical for all sources. Copying rules and host graph together into one file and checking the compatibility of the type graph imposes a significant overhead for static checks and decreases the performance.

4.2.2. Rule Definition by Graph Queries

Another way of handling attributed graphs is the use of TGraphs [40], which were invented for graph based modeling. Similar to the type graphs used in graph grammars, TGraphs are based on a schema definition, describing legal graph structures. Thus the formal foundation of both approaches is equivalent. A new parser had to be implemented (see section 4.3.2) to read Java source files and represent their syntax graphs as TGraphs.

TGraphs are especially designed for efficient handling and analyzing of large graphs and are thus a promising approach to tackle the performance problem of the graph transformation approach. Queries on this graph format can be expressed using a query language named GReQL [15]. This language is somewhat similar to SQL and thus well suited to implement rule-based checks: Queries in this language allow searching for elements of certain type, that are connected in a certain way, and own certain attributes. This is sufficient for our definition of syntax graphs and allows for basic graph queries of the following types:

- Existence of diagram elements based on their type;
- Existence of diagram elements based on the value of their attributes;
- Existence of tuples of elements based on the connections between them.

Executing queries of these types via the graph query engine returns either a list of found elements matching a specified condition, or an empty list. To get rules usable for checking software or design artefacts, queries have to be enriched with additional information.

First, it has to be defined by the author of the rules whether the query specifies a desired or undesired element or element tuple, respectively. Depending on this definition, the result of the query is interpret differently: In case the query specifies desired elements, an error has to be reported if the result list is empty. Contrary, in case the query specifies undesired elements an error has to be reported if the result list is not empty. Consequently, there is a difference in the meaning of “matching query” and “matching rule” in the remainder of this section: A “matching query” is a graph query in GReQL that reports at least one element. A “matching rule” is a checking rule in which either a desired element is found or an undesired element is not found.

Second, it may be necessary to combine different graph queries by logical operators to allow for alternatives. In the simplest case two or more different spellings of an element name or attribute may be allowed. Then a rule must report all elements using any of these spellings. This can be achieved trivially by combining several query expressions, each for one spelling, by using the logical OR. For easy handling of more complex variations, user defined functions can be defined as extensions of GReQL, e.g. for comparing strings while ignoring whitespaces. Examples for such user defined functions are given in section 4.3.2, where details of the implementation of rule-based checkers with GReQL are discussed.

Finally, at least for the use in the context of e-assessment systems, scores and feedback messages have to be assigned to each rule. Scores can be handled differently in optimistic and pessimistic approaches: In an optimistic approach, each artefact gets full credit initially and any checking rule that does not match reduces this credit by the given score. In a pessimistic approach, an artefact gets 0 credit initially and every matching rule increases this credit by the given score. Independent of these two models, textual feedback messages are displayed for any checking rule that does not match. Results from the query execution (e.g. a list of element names) may be used when generating the feedback message.

In contrast to the graph transformation approach described earlier, the graph query approach has no overhead in terms of complex rule formulation or additional preparations for the execution engine. Thus the performance of this approach is much better. On the downside, it does not allow for intermediate transformation steps as the graph transformation approach does. Thus slightly different graph patterns have to be covered by different queries and cannot be normalized by additional transformation rules. On the other hand, user defined functions allow to cover these situations to some extend, although they represent a completely different approach.

4.3. Implementation

In this section, three actual grading components realizing the concepts presented above are discussed in detail: Two graders applying static analysis to Java code and one grader performing static analysis of UML diagrams. The focus of this section lays on the technical details of the implementations rather than on the general benefits and drawbacks of the concepts as before. Actual examples of how to use the components discussed here can be found in section 4.4 below.

4.3.1. Component `StaticJavaChecker`

The marking component named `StaticJavaChecker` implements static checks for Java 6 based on graph transformation rules. As graph transformation engine it makes use of AGG[1]. The component is based on a toolbox API for performing various operations on

[1] `http://user.cs.tu-berlin.de/~gragra/agg/`

AGG graphs in GGX file format. Relevant operations for the checker are performed by three sub-components:

- The *Parser* component knows a type graph for Java programs and takes Java source code as input. It creates nodes and edges according to this type graph and adds them to the host graph of a graph grammar. When the parser has finished, the host graph of that grammar is an abstract syntax graph of the given input source code.
- The *RuleCopy* component takes a list of additional graph grammar files as input and extracts transformation rules from them. It imports all rules into the graph grammar created before.
- The *RuleControl* component takes a script file as input, which provides an order in which rules are to be executed on the syntax graph. It determines the rules for each instruction and invokes the graph transformation engine of the graph grammar.

Design and implementation of this component has partially happened before this thesis. Nevertheless, the component is described in more detail in the subsections below. To fit into JACK's general architecture, there is an additional class implementing the `IChecker` interface. On invocation of the method `doCheck`, it selects all source files, all rule files, and an additional file containing checker instructions from the provided resources and copies them into a working directory. Afterwards the source code is compiled using the compiler provided by Eclipse IDE. If compiler errors are reported, the check is aborted. In this case, the solution gets a score of 0 points and a list of feedback messages is returned, containing all errors reported by the compiler. If no compiler errors are reported, the RuleCopy and RuleControl components are invoked. The latter performs the actual graph transformations that may introduce error nodes into the graph. These are collected as the last step of the process and reported to the user. The sequence of invocations of the different components is depicted in the sequence diagram in figure 4.2. Some method calls are simplified in this diagram compared to the actual implementation to make the diagram more readable.

Parser Implementation

The parser sub-component is based on the Java parser provided by Eclipse IDE. This parser is capable of parsing source files into an object structure representing the syntax tree of the source code. Objects in the tree denoting field access, variable access, method invocations, or similar contain so-called "bindings", which can be resolved to get a reference to the respective field declaration, variable declaration, method declaration, and so on. In order to create a graph based representation of this object structure for use with AGG, a visitor pattern has been implemented to traverse this syntax tree. On visiting each object in the tree it creates a node in an AGG graph grammar. The visitor keeps track of the parent node visited before and creates an edge from parent node to the newly created child node in the resulting graph. It also keeps a list of all binding references encountered during traversal. After all nodes of the syntax tree have been visited, the parser iterates over this extra binding list and adds additional edges to the graph, each one representing on reference from the binding list. Refer to listing 4.1 and figure 4.1 from the beginning of this chapter for an example of a syntax graph created by the parser implementation used in the `StaticJavaChecker` component.

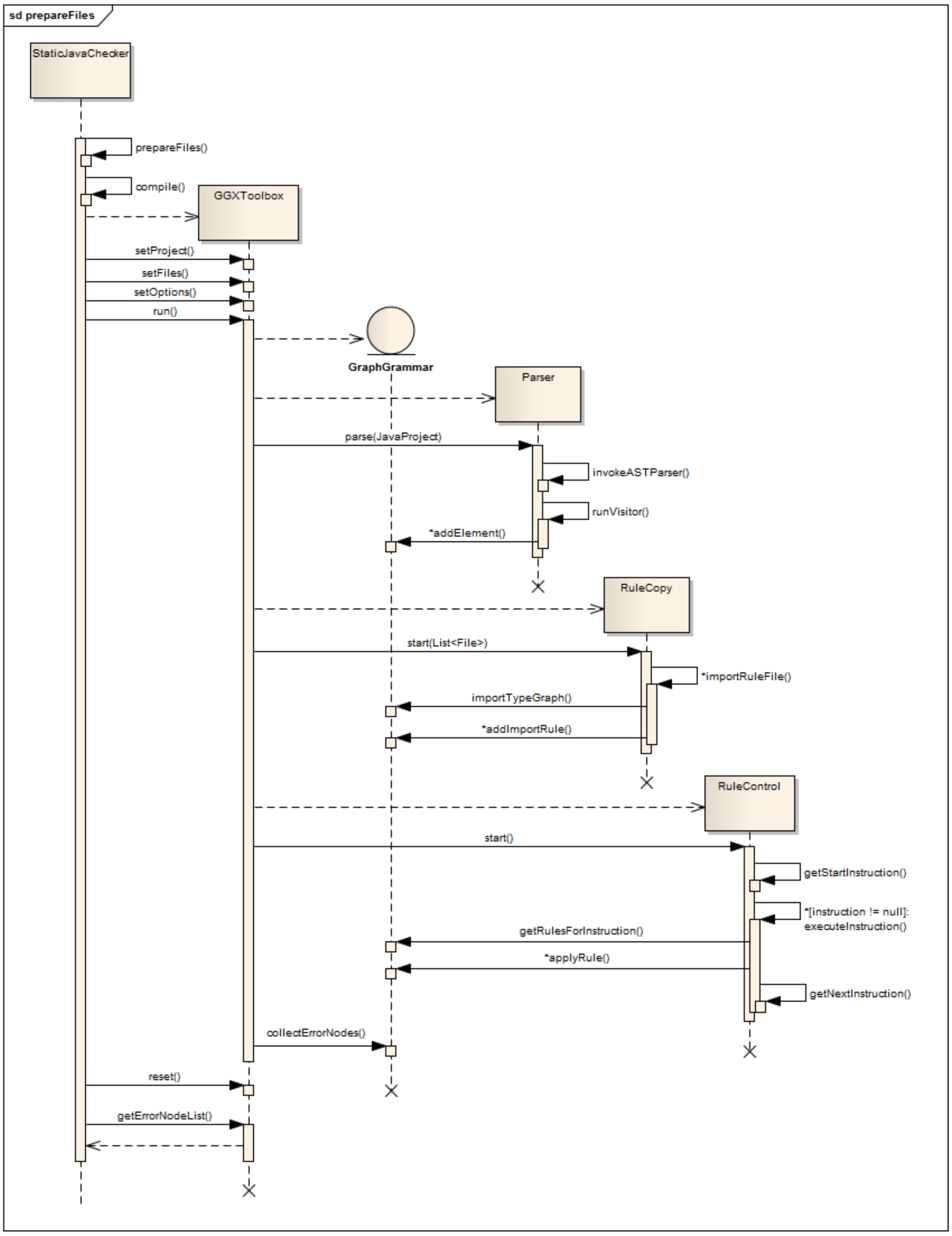


Figure 4.2.: Sequence diagram illustrating the method calls occurring when using the `StaticJavaChecker` component.

RuleCopy Implementation

The output of the parser component is a graph grammar project for AGG, that just contains a graph and its type graph, but no transformation rules. These are provided in one or more resources attached to the exercise and must thus be imported into the graph grammar project. This is what the RuleCopy component is made for. It makes the necessary calls to the AGG-API and especially takes care that the imported rules fit to the type graph already existing in the graph grammar project. If a rule is imported that adheres to a different type graph, applying these rules during the checking process will produce incorrect results.

As can be seen from the sequence diagram in figure 4.2, RuleCopy basically implements two nested loops: It iterates over all rule files and first imports the type graph found in these files. If AGG reports an error because of incompatible types in the type graph at this stage of the process, the import is canceled and no rules are imported. Otherwise, a second loop is started, iterating over all rules in the rule file. Each rule is imported by an appropriate call to the AGG-API. Afterwards, the resulting grammar is checked for errors again and the import from other rule files is discarded if an error occurred. Otherwise, the import proceeds with the next rule file. Only if no error occurs during the whole import process, the resulting graph grammar is considered ready for transformation and can be handed over to the next component.

RuleControl Implementation

The main goal of the RuleControl component is to allow for more flexibility and control over rule execution than the original transformation engine of AGG does. However, some of the features supported by RuleControl have been included in some way in recent versions of AGG, but the use of RuleControl was not changed or abandoned for that reason.

AGG allows to organize transformation rules in layers, where each layer has a number. Layers are processed according to that order. While this is generally sufficient for the work with graph grammars as graph production systems, it is not sufficient for graph based checks of syntax graphs. Thus RuleControl introduces a scripting language that allows to define explicit sequences of layers, including loops and conditional branches. This way, a set of rules can be defined that first performs some preparations, then checks for specific constructs, and finally branches to two different layers of rules depending on the results of the previous checks.

The actual implementation can be seen on figure 4.2 as above: The first instruction of the script is determined, which is the number of the layer to be executed. Then a loop is started that terminates if no instruction (thus no layer) remains. Inside the loop, the rules for this instruction are retrieved and applied, as long as this is possible. If no more rules can be applied, the next instruction is determined from the script. It has to be noted that it can be meaningful to visit one particular layer more than once. Although a layer is left only if no more rules from this layer are applicable, rules from another layer that are executed later may again introduce nodes, edges, or attribute values in the graph that make rules applicable again.

After RuleControl has finished working, the only remaining task is to collect all error nodes introduced into the graph and report the respective error messages.

4.3.2. Component `GReQLJavaChecker`

The marking component named `GReQLJavaChecker` implements static checks for Java 7 based on a different technique than the one discussed in the previous section. It makes use of the Java library JGraLab[2], which provides all necessary facilities to handle TGraphs and queries in GReQL. The marking component splits into two sub-components:

- The *parser* takes Java source code as input and creates a representation of its syntax graph as TGraph in memory.
- The *checker* uses the GReQL library to apply queries to the graph produced by the parser. The checker takes a set of rules as input, where each rule consists of a query and an error message.

Both sub-components are explained in more detail in the following subsections. An additional package provides user defined functions extending GReQL to make writing some checks more convenient. Both sub-components are coordinated by a Java class implementing the `IChecker` interface. On invocation of the method `doCheck`, it selects all source files from the provided resources and copies them into a working directory. Afterwards the source code is compiled using the compiler provided by Eclipse IDE. If compiler errors are reported, the check is aborted. In this case, the solution gets a score of 0 points and a list of feedback messages is returned, containing all errors reported by the compiler. If no compiler errors are reported, the parser sub-component is invoked on the source files and the resulting TGraph is passed to the checker sub-component. In this case, the checker determines the score to be given and the list of feedback messages. The sequence of invocations of the different components is depicted in the sequence diagram in figure 4.3. Some method calls are simplified in this diagram compared to the actual implementation to make the diagram more readable.

Parser Implementation

The parser sub-component for the `GReQLJavaChecker` works exactly the same way as the one for the `StaticJavaChecker` does which was already described in section 4.3.1. It just employs a different visitor implementation, generating TGraphs instead of graph representations for AGG. The TGraph representation of the example used above in listing 4.1 is shown in figure 4.4.

Checker Implementation

The checker sub-component takes an XML file containing checker rules and a TGraph created by the parser sub-component as input. It expects the XML file to be formatted according to the XML schema depicted given in figure 4.5. Allowed values for the parameter “type” of tag “rule” are either “presence” or “absence”. If this parameter is set to “presence”, the checking rule matches if one of its graph queries matches. If the parameter is set to “absence”, the checking rule matches if none of its graph queries matches. The parameter “points” takes an arbitrary positive integer as value. Points are granted for every matching rule. The sum of

[2] `http://userpages.uni-koblenz.de/~ist/JGraLab`

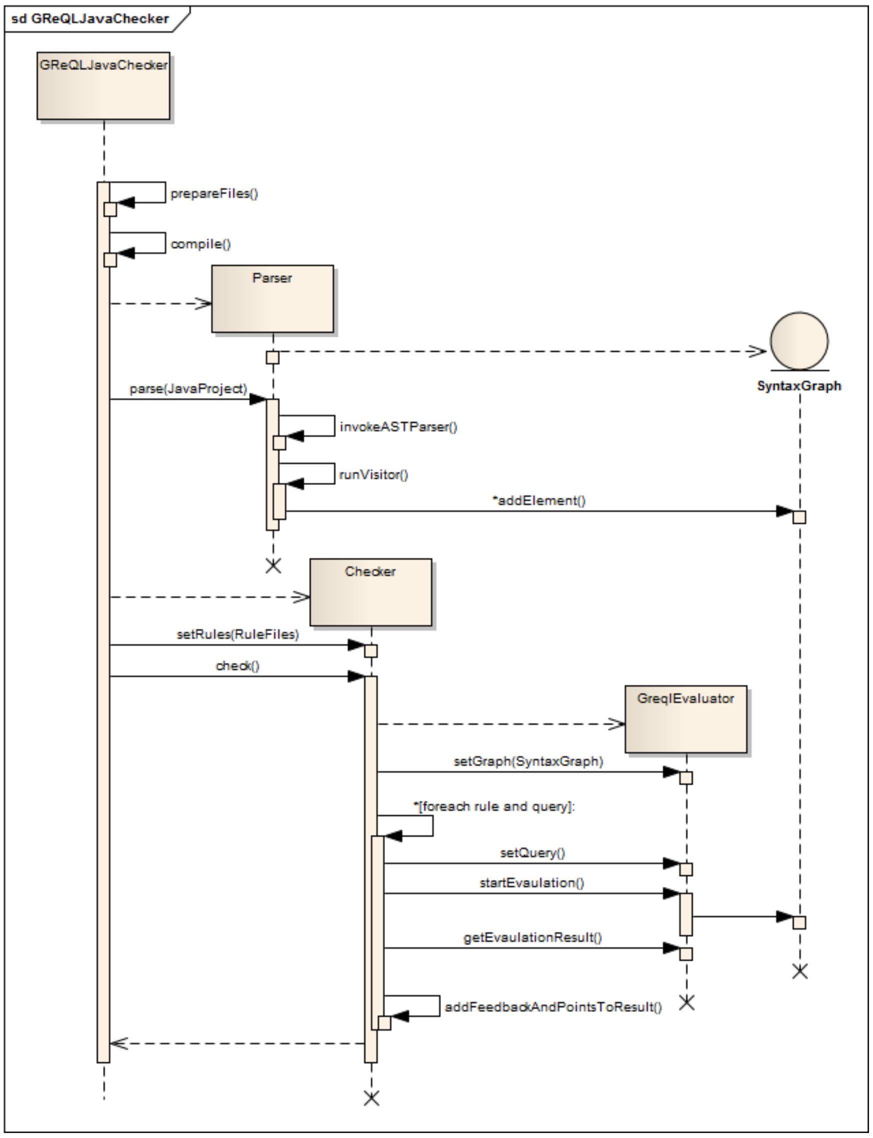


Figure 4.3.: Sequence diagram illustrating the method calls occurring when using the `GReQLJavaChecker` component.

possible points from an XML file is normalized to 100 before creating the final result and the points actually granted are computed relatively to this sum.

The parameter “query” contains the actual graph query in GReQL. The GReQL statement may return one or more graph elements from the result as named data fields. These fields can be reused in the feedback associated with this rule via the parameter “feedback”. Any placeholder `{x}` in the value of this parameter is replaced by the content of a data field `x` in the query result if the latter exists. Each feedback message may have an optional prefix given in the parameter “prefix” of the “feedback” tag. If no prefix is given, a default prefix saying “Erroneous Code Structure” is used.

The checker sub-component handles the list of rules from the XML file straightforward,

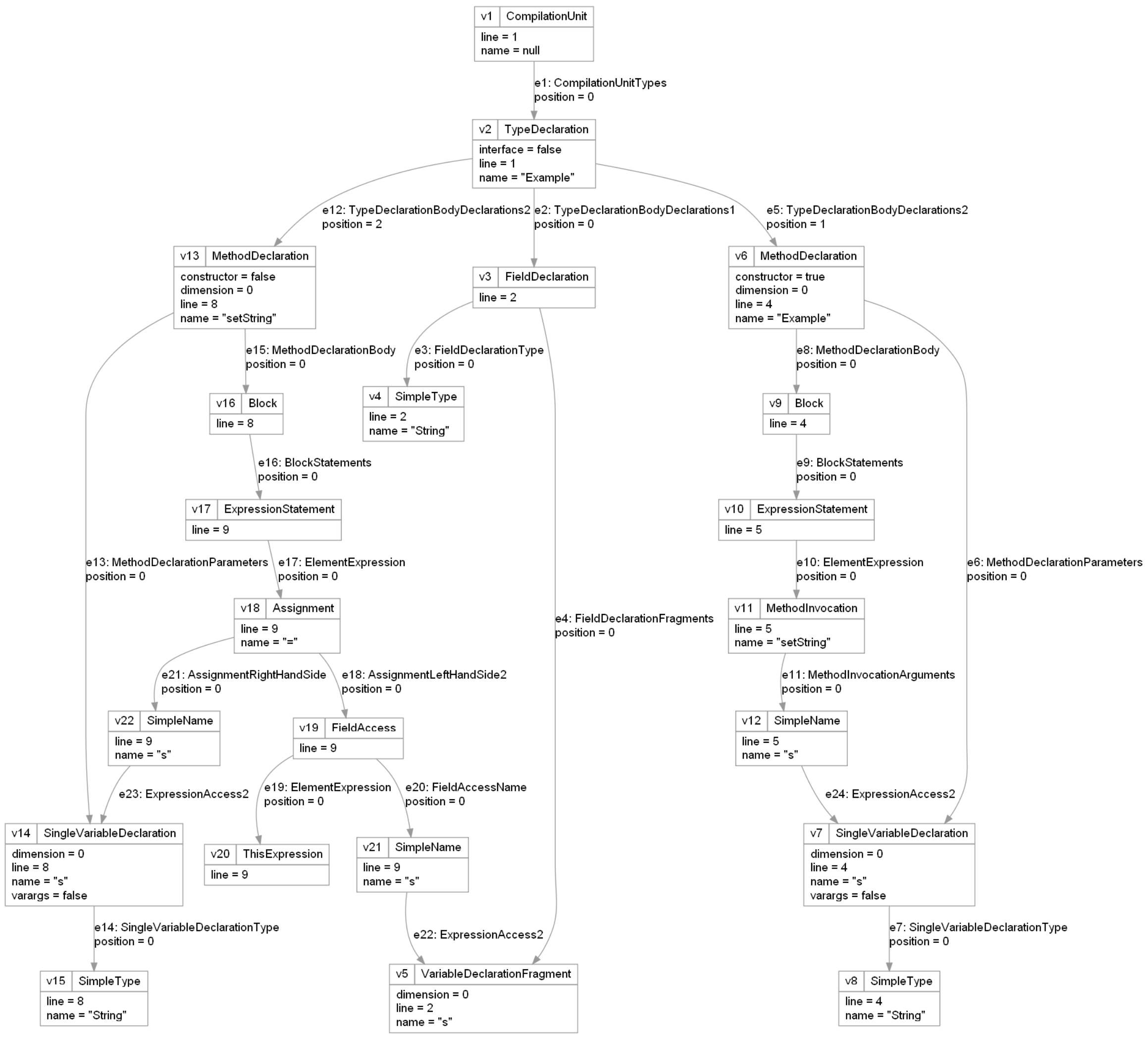


Figure 4.4.: Example of a syntax graph for the piece of Java source code given in listing 4.1 as created by the parser component of `GReQLJavaChecker`. The actual TGraph is stored in a textual file format and visualized using the tool *GraphViz* for this figure. Similar to figure 4.1, boxes denote syntactical elements and their properties, while solid arcs represent their nesting according to the grammar. Arcs for resolving references are not present in this figure.

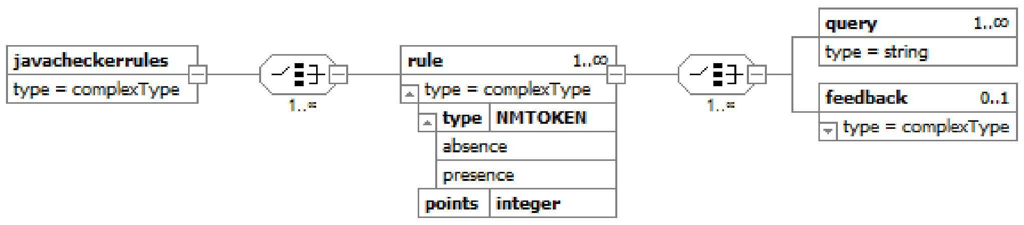


Figure 4.5.: Graphical overview of the XML schema definition for rule files used by `GReQLJavaChecker`.

applies one rule after another to the TGraph and adds the feedback message to the list of feedback if necessary. As the final result it returns both the list of feedback messages and the points calculated as explained above.

User Defined Functions

The GReQL for queries on TGraphs can be extended by user defined functions. This can be used to implement functions useful in the special context of e-assessment systems. Often code templates are provided alongside an exercise and it has to be checked if method signatures have not been changed by the students. This is important in conjunction with dynamic checks, since test cases cannot be executed if expected methods are not present. Since a method signature consists of several parts (i.e. the actual method declaration containing the method name, possibly some modifiers, an arbitrary number of parameters, each having a name and a type, and an arbitrary number of thrown exception types), it is tedious to list all of them in a GReQL query as nodes together with all necessary connections between them.

To solve this problem, a user defined function `hasSignature` has been implemented (see appendix A.2). It takes a single node and a string containing the desired method signature as parameters and returns true iff the node and its subtree represent the syntax tree for the given method signature. The signature needs not to be given in correct Java syntax, i.e. parameter names can be left out if only parameter types are considered relevant by the author of the checking rule. The same is true for visibility modifiers, so if no expected modifier is declared, students are free to chose. This way, this function does not only ease writing rules, but also enables tolerant checks.

Listing 4.2 shows a sample query making use of that function. It searches for all method declarations in some class that have a particular signature. This signature is not given in correct Java syntax, but just with the search string `int someMethodName(int, String)`. The function thus matches on all methods that fulfill the following four criteria:

1. The return type of the method is `int`.

2. The name of the method is `someMethodName`.

3. The method takes exactly two parameters, where the first one is of type `int` and the second one is of type `String`.

```
from t : V{TypeDeclaration}, m : V{MethodDeclaration} with
    t.name="SomeTypeName" and
    t -->{TypeDeclarationBodyDeclarations2} m and
    hasSignature(m, "int someMethodName(int, String)")
report 0 end
```

Listing 4.2: Graph query using the user defined function `hasSignature` to check for method signatures based on partial method definitions

4. The method signature does not have a `throws`-clause for exceptions thrown by this method.

In particular, the function does not care about the names of the parameters and it also does not care about the visibility modifiers of the method. However, if visibility of the method is relevant (e.g. because it is called by test cases), an appropriate modifier can be added to the search string.

4.3.3. Component `GReQLUMLChecker`

Similar to the component for static checks on Java source code, the marking component named `GReQLUMLChecker` implements static checks for UML diagrams. It also makes use of the Java library JGraLab and splits into two sub-components:

- The *parser* is able to parse UML diagrams provided in the XMI data format and represent them as TGraphs in memory.
- The *checker* acts the same way as the checker component for static analysis of source code does.

Parser Implementation

From a theoretical point of view there is not much to say about the parser implementation, because XMI is a standardized language based on XML and thus easy to parse. A type graph covering a subset of UML for Class diagrams and activity diagrams can be found in appendix A.3. It can in general be extended to cover the full UML syntax, although this has not been done in this thesis.

However, there is a major challenge in the fact that most UML tools use their own file format for storing diagrams and thus implement XMI just as an export. Virtually none of these exports (and especially those found in freeware tools) is fully conformant with the XMI specification, which makes it often impossible to import XMI files exported by one tool into another tool.

This is a significant challenge for e-assessment systems, because it has to be assumed that students will use different freeware tools to do their exercises, since they (and often universities as well) are not able to pay for expensive software licenses for more standard compliant tools. Consequently, the parser implementation has to cover as much variants of XMI as possible. In order to do so, it iterates over the XML structure of the XMI file and compares each element encountered with a list of known elements. For each known element it creates a new node in the syntax graph as necessary. Similar to the list of bindings managed by the Java parser as described above, a list of links between elements is maintained by the XMI parser as well. These links are usually represented by ID attributes inside XML tags, that can only be resolved after the complete files has been read.

Checker Implementation

The TGraph created by the parser component is handled by the checker component the same way as in the `GReQLJavaChecker` above. Rule files follow the same XML schema with two deviations: The root element is named “umlcheckerrules” and the text content of query tags has to refer to the syntax graph for UML. The checker handles the list of rules from these file by applying one rule after another to the TGraph and adds the feedback message to the list of feedback if necessary. As the final result it returns both the list of feedback messages and the points calculated as already explained above.

User Defined Functions

As for the `GReQLJavaChecker` above, user defined functions can be used to make checks more convenient when using GReQL. In general, the syntax graphs for UML diagrams are much smaller than the ones for Java programs and the number of different syntactical constructs sharing the same purpose is consequently much smaller as well. Thus the main application area of user defined functions is not to cover different syntactical constructs in one query (although this can of course be done by these functions as well as discussed for Java above), but to cope with the creativity of students during modelling. In particular, user defined functions can be used to define string comparisons that are tolerant with respect to whitespaces or typing errors, as well as functions that cover synonyms.

The possibilities of using user defined functions have not been explored systematically for this thesis, because they are strongly connected to topics like natural language processing and thus far out of the scope of this thesis. However, it has to be stated that there is a very good opportunity to improve the quality of checking results by exploring these possibilities and that there is no need for a change in the architecture of e-assessment systems to get these features included.

4.4. Application Examples

The following section provides examples on how static checks can actually be used in analyzing different kinds of software and design artefacts. For programming exercises, both examples using graph transformation and graph queries are provided. For checks on diagrams, just graph queries are shown, because graph transformations have not been implemented for that case.

4.4.1. Program Check Examples

As already mentioned in chapter 1, from an application point of view two main purposes of static analysis exist: Giving direct feedback on the analyzed artefact and retrieving information about the artefact being used as input for subsequent checking steps. For each of these goals, an example will be given in the following subsections. Both examples are taken

```
int i = 5;
while (i > 0);
{
    // some operations
    i--;
}
```

Listing 4.3: A `while` loop in Java that looks correct on the first glance but actually misses a body due to the extra semicolon after the termination condition.

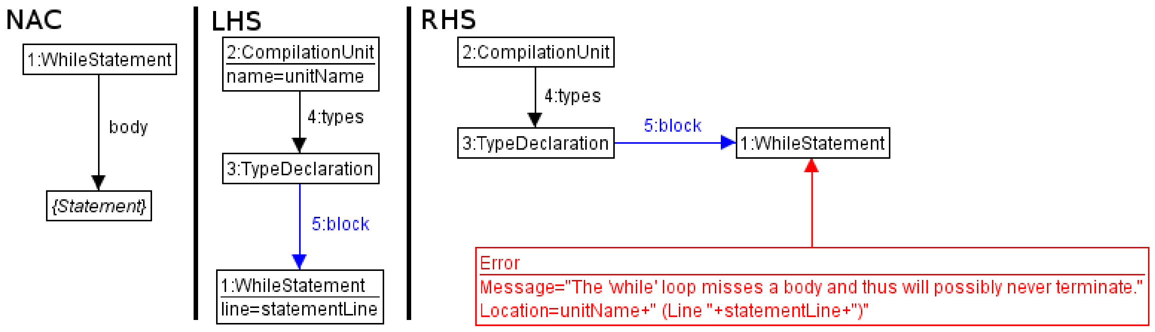


Figure 4.6.: Graph transformation rule checking for a `while` loop that misses a body.

from exercises on programming in Java 7, as taught in first year courses at the University of Duisburg-Essen every term.

Detecting Infinite Loops

A common problem to first year student in programming is struggling with termination conditions of loop constructs. During their tries to create a proper loop statement, many students create infinite loops. Most of these loops may be caused by semantic issues, but there are also cases in which an infinite loop is caused just by syntactic flaws. One of these cases is a `while`-statement in Java, which is terminated by a semicolon directly after the termination condition, thus missing a body (see example in listing 4.3). Experience shows that it is hard for a tutor to point out this flaw rapidly by manual code inspection, and the same is true for the students as well. Thus this situation is a good candidate for automated support. Detecting this kind of logical error in Pascal programs has also been a feature of the system CAP published in 1995 [119], so this kind of flaw is neither new nor typical for Java programs.

The necessary graph transformation rule and GReQL query are shown in figure 4.6 and listing 4.4, respectively. While the graph transformation rule just covers the situation exactly as described above, the query also covers cases in which the body is present, but empty.

The graph transformation rule is an optimistic rule, so the idea is to assume no broken `while`-statement in the code unless the rule matches. However, the nodes used in the LHS of the rule do not depict the complete erroneous structure, but just a kind of anchor for locating the possible error in the code. Using nodes for the compilation unit and the type declaration on the LHS is a typical design, making it possible to use the name of the compilation unit

```
<rule type="absence" points="2">
    <query>from u : V{CompilationUnit}, x : V{WhileStatement} with
            not isEmpty(u -->* x -->{WhileStatementBody}&amp;{EmptyStatement}) or
            (not isEmpty(u -->* x -->{WhileStatementBody}&amp;{Block}) and
              isEmpty(u -->* x -->{WhileStatementBody}&amp;{Block} -->{Child}))
            report u.name as "name", x.line as "line" end</query>
    <feedback>In file {name} there is a while-loop in line {line} that is meaningless
        because it just has a termination condition but no body.</feedback>
</rule>
```

Listing 4.4: GReQL rule checking for a `while`-loop that misses a body.

in the error message. Both nodes are not part of the actual syntactical pattern of interest. Instead, the third node for the while statement is part of the pattern and thus also used in the NAC, where the actual flaw is described. It has to be noted that in the Java syntax graph as generated by the parser, the while statement has a body child of type "empty statement", which is the Java representation of a standalone semicolon. It is assumed here that the empty statement is not considered as a valid concrete type for the abstract node "statement" used in the NAC. Designed this way, the LHS will match on all while statements in the source code, but will just be executed on those where the while statement has no statements as body children in the syntax graph. This exactly matches the situation described above. The RHS in turn does repeat the LHS as usual for optimistic rules, and adds an appropriate error node to the graph.

The GReQL query basically uses the same design. It also starts with looking for compilation units and `while`-statements. From these it considers `while`-statements having the empty statement as body. Note that the empty statement is named here explicitly, rather than excluding it implicitly as in the graph transformation rule. As a second alternative, also situations are considered, in which the body is a block that in turn contains no children. From all situations matching one of these two conditions, the name of the compilation unit and the line of the `while`-statement are returned to be used in the error message.

Both the graph transformation rule and the GReQL query match on the encounter of an infinite loop caused by an extra semicolon. The query also matches an additional situation, which would be covered by an additional rule if graph transformations are used. Both the rule and the query can be used to generate feedback to the students and helping them to find the cause of probably unexpected behaviour. In practice, it is usually not a technical problem to express the desired pattern with one of the approaches this way, but it can become a problem to imagine all patterns that are relevant with respect to the actual checking goal. While the query discussed above is sufficient to cover a good deal of the typical flaws made by first year students, is has also two serious drawbacks regarding its quality.

First, it misses situations in which a `while`-statement exists that has a non-empty block that contains just an empty statement. While this is a somewhat artificial situation, it is not completely unlikely to appear by mistake in students' program code. As the situations covered by the rule, it is also a situation in which the infinite loop happens just because of a syntactical flaw. However, it is hard to imagine or automatically enumerate all these kinds of faulty situations before they occur for the first time. Thus the particular rule in listing 4.4 may cover a lot, but not all faulty situations, thus providing a lower recall than the optimum. Obviously, this problem can be solved by continuous improvement of existing rule sets as they are in use. In theory, it may be possible to generate all possible combinations of syntactical elements relevant to this rule and derive a query automatically from that, but in practice it may be easier to amend an existing query as needed.

Second, the implications of this rule have to be carefully examined. As discussed, the rule is used to detect infinite loops based on syntactical observations. However, a `while`-statement that misses a body is not necessarily an infinite loop. The termination condition of the loop may be designed in a way that it returns `true` finite many times, but eventually returning `false`, causing the loop to terminate. Since this decision is based on execution semantics, it cannot be made by static checks. Consequently, there may be cases in which the query matches, although the artefact is correct with respect to the intent of the rule, thus lowering its precision. In winter term 2012/2013, these kinds of false positives have been reported in just one out of nine cases on a total amount of 3850 solutions.

The problems with false positives can be tackled by combination of checking techniques. As discussed in chapter 5 later in this thesis, trace generation for programs at run time can be used to check for temporal properties. Hence it may be beneficial to use the query as a second step in a checking process: First, the code is executed and terminated by timeout if an infinite loop is suspected. On that event, checking for temporal properties on the trace and queries to check for typical patterns for these loops are used to provide explanations to the student.

Detecting broken `if`-statements

The same flaws as with `while`-statements can also happen with `if`-statements, that either may contain two empty branches or an extra semicolon right after the condition. Two GReQL rules for checking these cases are shown in listing 4.5. They are constructed basically the same way as the rule shown above, checking for the existence of empty statements and empty blocks. Note that an empty second branch (the `else`-branch) is legal in Java and that the combination of an empty first branch with a non-empty second branch causes a compiler error and thus does not need coverage by an additional rule.

Different to the `while`-loops discussed above, there is no case in which the rules will report false positives, because no additional semantics are involved. Even if the condition contains some meaningful method call, it is in any case meaningless to surround it by an `if`-statement with empty branches or no branches at all. Consequently, no false positives have been reported for this rule in winter term 2012/2013, where 50 feedback messages were generated by these rules on a total amount of 5672 solutions.

Detecting recursive and iterative approaches

Besides giving feedback directly on flaws, static checks can also be used to prepare subsequent analysis, whether or not is also static or dynamic. One example to consider here is the detection of problem solving strategies. Both for static and dynamic analysis is can be beneficial to know whether the student tried to apply e.g. an iterative or recursive approach to problem solving. In the special case of exercises that explicitly require to use one of these approaches, this information can of course also be used for direct feedback and grading.

The basic idea of strategy detection is to check for constructs that are necessary for one

```
<rule type="absence" points="2">
    <query>from u : V{CompilationUnit}, x : V{IfStatement} with
            ( not isEmpty(u -->* x -->{IfStatementThenStatement}&amp;{EmptyStatement}) or
                (not isEmpty(u -->* x -->{IfStatementThenStatement}&amp;{Block}) and
                   isEmpty(u -->* x -->{IfStatementThenStatement}&amp;{Block} -->{Child}))
            ) and
            ( not isEmpty(u -->* x -->{IfStatementElseStatement}&amp;{EmptyStatement}) or
                (not isEmpty(u -->* x -->{IfStatementElseStatement}&amp;{Block}) and
                   isEmpty(u -->* x -->{IfStatementElseStatement}&amp;{Block} -->{Child}))
            )
        report u.name as "name", x.line as "line" end</query>
    <feedback>In file {name} there is an if-statement in line {line} that is meaningless
        because both branches are empty.</feedback>
</rule>
<rule type="absence" points="2">
    <query>from u : V{CompilationUnit}, x : V{IfStatement} with
            not isEmpty(u -->* x -->{IfStatementThenStatement}&amp;{EmptyStatement}) and
               isEmpty(x -->{IfStatementElseStatement}&amp;{Statement})
        report u.name as "name", x.line as "line" end</query>
    <feedback>In file {name} there is an if-statement in line {line} that is terminated by
        a semicolon right after the condition. The condition is hence meaningless and the
        subsequent code will always be executed.</feedback>
</rule>
```

Listing 4.5: Two GReQL rules checking for broken `if`-statements.

```
from m : V{MethodDeclaration} with
    isEmpty(m -->{Child}*&amp;{WhileStatement}) and
    isEmpty(m -->{Child}*&amp;{DoStatement}) and
    isEmpty(m -->{Child}*&amp;{ForStatement}) and
    isEmpty(m -->{Child}*&amp;{EnhancedForStatement})
report 0 end
```

Listing 4.6: Graph query checking for the absence of any kind of loops inside a method.

of the strategies but not for the other. For an iterative strategy in Java, there need to be loop statements of some kind, while for a recursive strategy there needs to be a method that directly or indirectly invokes itself. For both ideas, a solution in GReQL is given below, but solutions by using graph transformation rules are possible as well.

Listing 4.6 provides a GReQL query that checks for the absence of any kind of loop statements. More precise, it matches on all methods that do not contain any kind of loop. Instead of using a common super type for syntax graph nodes representing loops, it just lists the four types of loops defined in Java. In contrast to the previous example, where by using `->{WhileStatementBody}` as a path description only direct body children of the while statement were of interest, here by using `->{Child}*` any child of the method in any depth is considered. Since we are only interested in the fact that there are loops, no particular information is returned by the query.

Listing 4.7 provides a GReQL query that checks for a recursive method. In detail, it checks for a method `m` that contains somewhere in its body a method invocation that refers to some method that in turn also contains somewhere in its body another method invocation

```
from m : V{MethodDeclaration} with
    (m (-->{Child}*&amp;{MethodInvocation} -->{Access}&amp;{MethodDeclaration})* -->{Child
        }*&amp;{MethodInvocation} -->{Access} m)
report 0 end
```

Listing 4.7: Graph query checking for recursive methods.

that refers to `m` again. As above, `->{Child}*` is used in the path descriptions to cover all nodes in any depth in the syntax tree. Note that the query implicitly also matches methods that call themselves. Again, no particular information is returned by the query.

By using both queries in combination, it is possible to detect which strategy to problem solving is used in a given piece of code. Obviously, no answer can be given if neither loops nor recursive methods can be found. Similarly, no decision can be made if both loops and recursive methods are found. In these cases, additional checks are necessary. In the other cases, the information gained can be used to apply more specific checks, e.g. for detecting infinite loops, which would be useless on code that uses a recursive strategy.

Rules with Alternative Solutions

In some cases, there is more than one possible solution for a given structural requirement on the code, where each solution involves different code structures or at least a significantly different number of elements. For example, classes in Java may implement the standard library interface `Comparable<T>` or the older version without generic types, which is also named `Comparable`. In the latter case, the syntax just involves a simple type, while in the former case, a parameterized type is involved, where not only the name of that type, but also its parameter needs to be checked. However, the same feedback should be presented to the students, iff neither of these variants is found. A possible solution to this challenge with two queries used in one rule is shown in listing 4.8. The rule is designed for an exercise in which a class `Fraction` was provided and students were asked to extend the given template in an appropriate manner to allow for proper comparison of fractions. Besides other things to do, this involves adding a reference to a standard library interface, either by using `Fraction implements Comparable` or `Fraction implements Comparable<Fraction>`.

The first query in this rules checks for the simpler structure, where only two nodes and one arc are involved. The second query is significantly more complex, as four nodes and three arcs are involved in order to check the variant with the parameterized type. However, both queries remain quite readable, which would not be the case if they are merged into one query by making use of sub-queries for the elements only used in one of the two alternatives. The whole rule is designed as a presence rule, so it matches if neither of the two queries matches.

4.4.2. Diagram Check Examples

Since the techniques for static checks as described above are not concerned about the type of artefact they are applied to, illustrating the same techniques on a different type of artefact again does not add much value. Hence this section sticks to graph queries as discussed in section 4.2.2, but elaborates on different types of artefacts.

```
<rule type="presence" points="2">
    <query>
        from t : V{TypeDeclaration}, pt : V{SimpleType}
        with t.name="Fraction"
        and pt.name="Comparable"
        and t --> {TypeDeclarationSuperInterfaceTypes2} pt
        report 0 end
    </query>
    <query>
        from t : V{TypeDeclaration}, pt : V{ParameterizedType},
        ptt : V{SimpleType}, pta : V{SimpleType}
        with t.name="Fraction"
        and t --> {TypeDeclarationSuperInterfaceTypes1} pt
        and pt --> {ParameterizedTypeType} ptt
        and ptt.name="Comparable"
        and pt --> {ParameterizedTypeTypeArguments1} pta
        and pta.name="Fraction"
        report 0 end
    </query>
    <feedback prefix="Type hierarchy">
        Fraction does not implement the interface Comparable.
    </feedback>
</rule>
```

Listing 4.8: A GReQL rule checking for two possible ways of implementing a type hierarchy.

Static checks based on graph queries have been used in a bachelor-degree course on UML modeling in winter term 2010/2011 at the University of Duisburg-Essen. Two exercises concerned with UML class diagrams were given to the students who had to solve it in teams of two. The student teams were asked to submit their solutions as XMI files in XMI 2 format. Models in XMI format can be parsed by a XML parser to a data structure that fulfills our requirements for syntax graphs. The XMI schema for UML is already a direct representation of the diagram's abstract syntax, thus in the resulting graph each relevant diagram element is represented as a node. Nodes have types and attributes, while arcs connecting these node just have types.

Figure 4.7 shows one of the solutions for the first exercise submitted by one of six student teams. The task was to model customers that get invoices for using telecommunication services. Each invoice is composed of several parts, possibly from different phone companies. Each part consists of several items, each related to a tariff that was offered by a phone company and used by the customer. A correct solution does not only involve to create a sufficient number of classes with reasonable naming, but also to use cardinalities, names, and directions for associations. The exercise description also contained some attribute names, so their correct usage had to be checked, too.

A total set of 17 rules has been written initially to describe the teacher's requirements for a correct solution. The checking process was based on a pessimistic approach, starting with 0 credit for an unchecked solution and giving credit for each matching rule. The set of rules can be subdivided into two subsets: One contains rules specific for this task, e.g. using element names taken from the task description. Another subset contains generic rules, that are concerned with general features of correct UML class diagrams, e.g. directions for

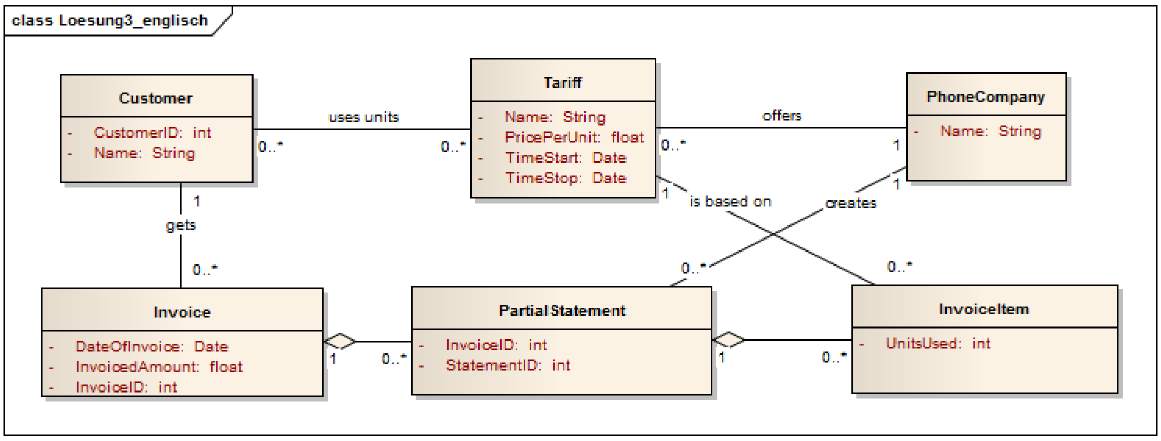


Figure 4.7.: Solution submitted by a student team for the exercise discussed in section 4.4.2.

named associations. Both subsets are explained by examples in the following subsections. The complete ruleset is provided in appendix A.4 in the original German version. Rule examples are translated to English in the following sections.

Task Specific Rules

The set of task specific rules contains 12 rules. Seven of these rules are concerned with attributes and simple relations between elements as shown in listing 4.9 by example: This graph query checks for classes named “Tariff”, having an attribute called “Name” or “Identifier” (which was considered to be an acceptable synonym for “Name”), ignoring the case of the first letter for the attribute. Having this class with the respective attribute is a desired feature of a correct solution, so credit is given if this query matches.

```
from x : V{Class}, y : V{Property}
with x --> y and x.name="Tariff" and (capitalizeFirst(y.name)="Name" or capitalizeFirst(y.
    name)="Identifier")
report x.name, y.name end
```

Listing 4.9: Graph query for checking a class and its attribute

The task description given to the students contained not only necessary information, but also some narrative context, i.e. introducing a phone company named “Pink Panther”. Students were expected to be able to identify this name as an instance name and to not include a class with this name in their diagrams. The graph query used to check this is shown in listing 4.10. The corresponding rule defines to give credit if and only if this query does not match. Hence only solutions having no element named “Pink Panther” get the credit defined for this rule.

Note that several possible spellings are used in the query. Elements with completely different spellings will not be detected by the query. This may cause the rule to match, although a student tried to model “Pink Panther” in some way. At least this is no specific problem in checking UML diagrams, but a common issue in any string-based pattern matching problem.

```
from x : V{Class}
with x.name="PinkPanther" or x.name="Pink Panther" or x.name="Pink_Panther"
report x.name end
```

Listing 4.10: Graph query for checking for an element with a given name. This element is not desired in correct solutions, so an error is reported if this query matches

However, this problem can be solved to some extent by the use of user defined functions. This shortens the query given in listing 4.10 and makes writing rules easier.

The remaining five task specific checking rules are concerned with more complex relations involving at least three elements as shown in listing 4.11 by example: This graph query checks for two classes x and y named "Customer" and "Invoice", respectively. These classes are expected to be connected by an association. Note that this query does not care about the direction of the association, because correct solutions can be designed for both directions. As above, having an association between these elements in general is a desired feature of a correct solution and thus credit is given if this query matches.

```
from x,y : V{Class}
with x <-- V{Property} <-- V{Association} --> V{Property} --> y and x.name="Customer" and
    y.name="Invoice"
report x.name, y.name end
```

Listing 4.11: Graph query for checking two classes and their association

Generic Checking Rules

The set of generic checking rules, which are not specific to the given task, contains five rules. Four of them check for missing names, roles, cardinalities, or directions of associations. The fifth one (see listing 4.12) checks if the diagram contains attribute names staring with a lower case letter and others starting with an upper case letter. This is a stylistic feature, hence it does not check the correctness of the solution, but the quality. Credit is only given if either all attributes start with lower case letters or all attributes start with upper case letters.

```
from x,y : V{Property}
with not isNull(x.name) and not isNull(y.name) and x.name=capitalizeFirst(x.name) and not
    (y.name=capitalizeFirst(y.name))
report x.name, y.name end
```

Listing 4.12: Graph query for checking upper and lower case letters in attribute names

4.5. Open Problems

Although both approaches discussed in this chapter can be considered useful by providing enough power to achieve high quality results of artefact analysis and feedback generation, there are still open problems: Tool support for the generation of rules or queries, decomposition of complex rules and queries, and automated verification of rules and queries.

4.5.1. General Tool Support

Tool support for the creation of rules and queries can be considered one of the most important issues for productive use. As already discussed above, the way of writing the rules in the graph transformation approach can hardly be considered intuitive. It could be supported by an editor that hides the structure of transformation rules and provides a logical structure: Instead of using three graphs for LHS, RHS and NACs, one graph can be used for the pattern to be matched, while all other information (i.e. the feedback message) can be inserted automatically into a proper graph transformation rule from simple text fields. Work in this direction is done by the Henshin project, which offers a graph transformation editor in which both LHS and RHS can be edited in one graph window [2].

Besides editing support, the graph transformation approach is fragile with respect to the use of different type graphs in different rule sets. A single change in the type graph may result in rule sets that are no longer compatible with the graph grammar projects produced by the parser, resulting in wrong output of the `StaticJavaChecker` component.

For graph queries the need for tool support is less urgent, since the structure to be edited by the query author fits the logical structure. However, due to the lack of graphical editing support, graphs cannot be understood as intuitively as in the graphical editor for graph transformation rules. If complex queries are to be written including a large number of elements that form a pattern, the textual representation of GReQL is harder to comprehend than a graphical representation. A graphical editor like the one of AGG that supports as well the constructs needed in GReQL would possibly be a helpful solution.

4.5.2. Decomposition

Comprehension and creation of complex rules and queries cannot only be supported by sophisticated editor support, but also by mechanisms for composition or decomposition of complex patterns. For example, in source code there can be patterns describing some properties of a method, while other patterns describe the method invocations between those methods. Instead of creating a complex rule or query including all these aspects in one, one can imagine to compose the same complex result by arranging simpler patterns in an appropriate manner. These smaller patterns can then also be reusable across several more complex rules.

4.5.3. Automated Verification

As discussed above, the quality of checking results is directly dependent on the quality of rules and queries. Thus an automated verification can be considered helpful. It can be used especially to check whether a rule or query would match always or never due to a malformed pattern. More sophisticated verification can also be used to avoid the creation of pattern that are correct in terms of the underlying type graph, but will never match because the actual syntactical construct represented by this pattern will never occur in an actual software artefact.

4.6. Chapter Summary

This chapter contributes to the goal of thesis in the dimension of techniques by discussing two different techniques: Application of graph transformation rules and application of a graph query language. For both techniques, implementations were given that can be integrated into an existing system, thus contributing to the goal of the thesis in the dimension of systems as well. In both techniques it is possible to create pre-defined generic sets of rules that are independent of specific exercises. Admittedly, tool support and decomposition have just been discussed as open problems, so this chapter of the thesis does not provide progress in the dimension of inputs, but also does not fall back behind the current state of research in this area.

Both techniques can be used successfully and effectively for the purpose of analysis and feedback generation of software artefacts provided in a graph based representation. Hence the chapter also contributes to the goals of the thesis in the dimension of feedback and provides evidence for this by example. In particular, both techniques allow to generate feedback in situations in which external tools usually used by current e-assessment systems cannot provide the same fine grained analysis. Both techniques also allow to handle different kinds of artefacts based on a common representation and are thus not as limited to one kind of software artefact as other approaches named in the literature review are.

5. Dynamic Checks

Unlike static checks, dynamic checks focus on the semantics of software or design artefacts. More precise, they consider the artefacts to be descriptions or specifications of changes in a state space and analyse these occurring changes. Thus dynamic checks are mainly concerned with states and relations between states. While this sounds somewhat similar to the ideas of static analysis as described in the previous chapter, where focus was laid on syntactical elements and relations between them, different techniques are necessary for dynamic analysis. This is mainly true because the state space has to be generated before it can be analyzed. Thus not only the artefact itself is necessary, but also a run time environment in which it is executed or interpreted. In the case of program code written in Java, this is usually a virtual machine which executes the byte code produced by a Java compiler. Both for the virtual machine and for the language specifications exist [5, 4], that ensure reliable semantics independent from the actual run time environment implementation at least to a very high extend. In contrast to that, several different semantics have been defined for some design artefacts in literature, e.g. for UML statecharts in [156, 153, 142]. Thus the semantics and dynamic behaviour of these artefacts depends on the chosen run time environment.

In general, not every state in a state space is reached in every run of a program or interpretation of an artefact respectively. If practical limitations in terms of memory or other limited resources are neglected, there is even the whole class of systems with infinite state spaces where an infinite number of runs or one infinite run would be necessary to cover the whole state space. Given a finite state space, the need for more than one (but finite many) runs can occur because of non-deterministic choices included in the semantics (e.g. the semantic of a random number generator is exactly to have non-deterministic choices) or because of dependence from input data provided from the outside. If a dynamic analysis aims on analyzing all possible relations between state in a finite state space, the full state space has to be generated. This in general incorporates several runs of the tested program or interpretation of the tested artefact respectively, before the actual analysis can take place. Typically, this kind of dynamic analysis is called verification. However, dynamic analysis can also focus on selected analysis of relations between states and thus waive the generation of the full state space. In these cases it is also applicable to infinite state spaces. Most often, this partial analysis of the state space is called testing.

In both cases, attention has to be paid to the design of the test cases, which define the input data for each run. If only selected states and relations should be analyzed, the test cases have to ensure that these states are visited during the test. If the whole state space is to be examined, it may be beneficial to find the smallest set of test cases that is able to cover the whole state space.

This chapter of the thesis discusses techniques in which the state space of a software or design artefact is generated completely or partially. Techniques are discussed both for analysis of single states as well as for analysis of traces, which are sequences of states. The chapter is organized as follows: Section 5.1 discusses dynamic checks for Java Programs by presenting concepts, implementations, and examples. Section 5.2 does the same for UML Activity Diagrams. The chapter is concluded by a discussion of open problems in section 5.3 and a chapter summary in section 5.4.

5.1. Black-Box-Tests and White-Box-Tests for Java Programs

One special relation between states in a program that can be analysed is the relation between input given at the starting state and accumulated output produced when a final state is reached. The output can then be compared to an expected output, that has to be provided as additional information from outside. If all system states visited on the path from starting state to final state are not examined, this test method is called black-box-testing. If intermediate states are recorded and analysed as well, the method is called white-box-test.

Black-box-tests are a very common technique in software testing and well supported by automated tools. In the case of Java, JUnit [83] may be the best known from all of them. Black-box-tests are well suited to match several software engineering principles: First, they are compatible to component and interface oriented design, where the functionality of a component is described by a contract at its interface. The internal organization and implementation of the component is not visible to the outside and thus tests consider it as a black-box, just checking the actual behaviour against the behaviour specified by the contract. Second, black-box-tests allow for the "test first" development style, where tests are written before the actual implementation.

However, black-box-tests are of limited value for the actual debugging process. If a black-box-test reveals a bug inside the tested program, it just states that the actual output is not equal to the expected output. It does not tell any details on how this output was created. Thus subsequent steps have do be done by a developer to reproduce the fault case and observe the program behaviour and state evolution in a debugging tool. Actually, the developer performs manual white-box-testing then, checking each intermediate state against the expectation of correct behaviour. Hence it can be stated that black-box-testing helps to reveal bugs, but white-box-testing additionally helps to explain them. This is what makes automated generation and analysis of traces interesting in the context of e-assessment.

The analysis of traces can be separated into different aspects depending on goals and the point in time the analysis takes place. Goals can be general program properties like the non-existence of infinite loops or task-specific properties like class invariants. The analysis can take place "on the fly" every time a new state is added to the trace or as "post mortem" analysis after the termination of the program. A well known subset of the latter is the post mortem analysis of the Java heap after a system broke. In fact, this analysis is just an analysis of the last state of the trace instead of analysing the whole trace.

As far as program analysis for teaching purposes is concerned, not all of these areas are useful in equal measure. Most certainly, the detection of infinite loops at run time can be considered beneficial, since infinite loops are a common mistake for students being less experienced with loop constructs. Detection for at least some of all different types of infinite loops would allow more efficient testing since programs do not have to be run until a time or step limit is exceeded, but can be terminated earlier. In these cases students can be informed about a more concrete reason than a plain limit and learn criteria for identifying infinite loops additionally. The same applies to deadlocks in thread-based systems.

The check for task specific properties during run time may be less beneficial depending on the actions taken when a specified property does not hold. It can be argued that it is more informative for students if a test is not aborted in these cases so that they can see the consequences of the violated specification. Consequently it is sufficient to perform checks for task specific properties as post mortem analysis.

Even without any further automated analysis it can be considered beneficial to generate traces of the program behavior: If the output of a method is not as expected, a programmer can use a trace to see how it was generated. In the special scenario of automated tutoring, it can be expected that traces can help students to get a deeper understanding of algorithms. Several approaches exist (e.g. [91, 106]) that support teaching algorithms by a step-by-step execution and sequence diagrams, thus using traces in some way. Doing this not only for sample implementations but for the source code provided by students may help them to study the effects of changes in the code in more detail.

In addition to beneficial effects in tutoring traces may also support professional debugging. The Java programming language knows the concept of "exceptions" for handling unexpected system behaviour. Exceptions can be thrown explicitly or implicitly to leave a method without return value. Exceptions can also be caught inside the program code, bringing the system back into a state expected and properly handled by the programmer. If exceptions are not caught, they stop the system. Java provides a "stack trace" in these cases which represents the call stack from the main method up to the method that threw the exception. This stack trace does include line numbers, but no information about field values and variable values. This is very disadvantageous when a "NullPointerException" occurs in a line where several variables or fields are accessed that could be "null". In these cases, full tracing can already add valuable information for further manual inspection.

5.1.1. Conceptual Considerations

In order to apply automated white-box-tests to a Java program, at least three things have to be done: First, states have to be explicitly logged during program execution, forming a trace of the actual program run. In the case of Java, this includes reading objects, field values and local variable values from the heap of the virtual machine and storing these data in an appropriate format. Second, properties to be checked have to be expressed in an appropriate way. This includes both general properties as well as program specific properties like invariants or post-conditions. Third, the occurrence of property violation events or other unexpected events has to be handled. For example, it has to be decided whether the test is completely aborted in case of a violated invariant or just set back to some predefined state. All three tasks are tackled in the subsequent subsections and discussed both in theory and by example.

5.1.2. Generating Traces

As mentioned above, the first step in white-box-testing is to generate a trace of the program run. A trace is considered to be a linear ordered list of all states visited during the execution of the program. Since Java is an object-oriented language and able to handle parallel threads, a state is defined by the following properties:

- The active thread

- The set of all existing threads
- The value of the program pointer for each existing thread
- The set of all objects existing on the Java heap
- The values of all non-static fields for each object
- The values of all static fields for each loaded class
- The values of all local variables in the current scope

Among these properties, the property "value of the program pointer" needs some careful consideration. In general it is possible to write a whole Java program as a single line but obviously its execution will happen in several steps. Thus the meaning of "value of the program pointer" in general is not equivalent to "current line number". However, the definition of "step" is not exactly fixed in the Java Virtual Machine (JVM) specification and thus the exact definition is left to the JVM implementations. Hence a step may or may not consist of the completion of one atomic JVM instruction [84]. Consequently, the definition of "step" may depend on the form of the byte code created by the compiler or even on the source code created by the programmer. However, for the purpose of tracing and trace analysis it is not acceptable to have the number of states depending on the used JVM implementation and the form of the code. Otherwise it could depend on the used JVM whether a property does or does not hold, or how many states a trace contains although the actual behaviour of the program or even the actual source code is the same.

This thesis tackles this problem by considering and implementing one of two possible different ways of trace generation. The first one, which is the one implemented in this thesis, uses the Java Debugging Interface (JDI) and considers steps in the meaning of "step by line" according to the JVM specification. In this case, "value of the program pointer" becomes equivalent to "current line number". Note that it is a natural limitation of the JVM that stepwise execution is not available for native methods so that these are necessarily still handled as black-boxes. This is acceptable in the context of assessment and tutoring, since these methods are not implemented by the students. Details about the implementation of stepwise execution and observation are given in section 5.1.6, but an example is already introduced here for more convenience in the following explanations: See listing 5.1 for a piece of the source code cut out of a programming exercise. Table 5.1 shows the trace generated for this solution using the array `{{1,2,1},{4,3,2},{2,2,7}}` as input parameter to the method. Of course, different traces would be produced if different input parameters have been used. A byte code based trace would be longer, but contain the same information in terms of variable values.

The second alternative uses Java Pathfinder as an underlying framework for generating states. It has not been implemented in this thesis, but is considered conceptually: In this case, the meaning of "value of the program pointer" is more fine-grained at statement level, since Java Pathfinder is able to fork program execution after any single byte code instruction and come back to this state later. Referring to byte code instructions may cause different behaviour for the same source code, if different compilers have been used that create different byte code instructions. The pathfinder framework is able to handle Java programs with multiple threads, so this aspect is omitted in the JDI implementation. Java Pathfinder can also be used to generate test data for exercises, as already discussed in literature [74].

File and line	Variable values					Line of code to be executed
	i	j	mat	temp	vec	
Miniprojekt2:220			{{1,2,1},{4,3,2},{2,2,7}}			`float [] vec = new float [mat.length];`
Miniprojekt2:221			{{1,2,1},{4,3,2},{2,2,7}}		{0.0,0.0,0.0}	`float temp=0;`
Miniprojekt2:223			{{1,2,1},{4,3,2},{2,2,7}}	0.0	{0.0,0.0,0.0}	`for (int i=0;i<=mat.length-1;i++){`
Miniprojekt2:224	0		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{0.0,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	0	0	{{1,2,1},{4,3,2},{2,2,7}}	0.0	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	0	0	{{1,2,1},{4,3,2},{2,2,7}}	1.0	{0.0,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	0	1	{{1,2,1},{4,3,2},{2,2,7}}	1.0	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	0	1	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	0	2	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	0	2	{{1,2,1},{4,3,2},{2,2,7}}	4.0	{0.0,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:227	0		{{1,2,1},{4,3,2},{2,2,7}}	4.0	{0.0,0.0,0.0}	`temp=temp/mat[0].length;`
Miniprojekt2:228	0		{{1,2,1},{4,3,2},{2,2,7}}	1.3333334	{0.0,0.0,0.0}	`vec[i]=temp;`
Miniprojekt2:229	0		{{1,2,1},{4,3,2},{2,2,7}}	1.3333334	{1.3333334,0.0,0.0}	`temp=0;`
Miniprojekt2:223	0		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,0.0,0.0}	`for (int i=0;i<=mat.length-1;i++){`
Miniprojekt2:224	1		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	1	0	{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	1	0	{{1,2,1},{4,3,2},{2,2,7}}	4.0	{1.3333334,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	1	1	{{1,2,1},{4,3,2},{2,2,7}}	4.0	{1.3333334,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	1	1	{{1,2,1},{4,3,2},{2,2,7}}	7.0	{1.3333334,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	1	2	{{1,2,1},{4,3,2},{2,2,7}}	7.0	{1.3333334,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	1	2	{{1,2,1},{4,3,2},{2,2,7}}	9.0	{1.3333334,0.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:227	1		{{1,2,1},{4,3,2},{2,2,7}}	9.0	{1.3333334,0.0,0.0}	`temp=temp/mat[0].length;`
Miniprojekt2:228	1		{{1,2,1},{4,3,2},{2,2,7}}	3.0	{1.3333334,0.0,0.0}	`vec[i]=temp;`
Miniprojekt2:229	1		{{1,2,1},{4,3,2},{2,2,7}}	3.0	{1.3333334,3.0,0.0}	`temp=0;`
Miniprojekt2:223	1		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,3.0,0.0}	`for (int i=0;i<=mat.length-1;i++){`
Miniprojekt2:224	2		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,3.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	2	0	{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	2	0	{{1,2,1},{4,3,2},{2,2,7}}	2.0	{1.3333334,3.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	2	1	{{1,2,1},{4,3,2},{2,2,7}}	2.0	{1.3333334,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	2	1	{{1,2,1},{4,3,2},{2,2,7}}	4.0	{1.3333334,3.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:225	2	2	{{1,2,1},{4,3,2},{2,2,7}}	4.0	{1.3333334,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:224	2	2	{{1,2,1},{4,3,2},{2,2,7}}	11.0	{1.3333334,3.0,0.0}	`for (int j=0;j<=mat[0].length-1;j++){`
Miniprojekt2:227	2		{{1,2,1},{4,3,2},{2,2,7}}	11.0	{1.3333334,3.0,0.0}	`temp=temp/mat[0].length;`
Miniprojekt2:228	2		{{1,2,1},{4,3,2},{2,2,7}}	3.6666667	{1.3333334,3.0,0.0}	`vec[i]=temp;`
Miniprojekt2:229	2		{{1,2,1},{4,3,2},{2,2,7}}	3.6666667	{1.3333334,3.0,3.6666667}	`temp=0;`
Miniprojekt2:223	2		{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,3.0,3.6666667}	`for (int i=0;i<=mat.length-1;i++){`
Miniprojekt2:231			{{1,2,1},{4,3,2},{2,2,7}}	0.0	{1.3333334,3.0,3.6666667}	`return vec;`

Table 5.1.: Trace generated for the source code shown in listing 5.1, when invoked for the array `{{1,2,1},{4,3,2},{2,2,7}}`. If a cell is empty, the respective variable is not in the scope of the current step.

```
public static float[] arithmetic_average(int[][] mat) {
    float [] vec = new float [mat.length];
    float temp=0;

    for (int i=0;i<=mat.length-1;i++){
        for (int j=0;j<=mat[0].length-1;j++){
            temp+=mat[i][j];
        }
        temp=temp/mat[0].length;
        vec[i]=temp;
        temp=0;
    }
    return vec;
}
```

Listing 5.1: Piece of source code cut out from a programming exercise. The method is supposed to return the arithmetic average of each row of a quadratic array. This exercise is used as a running example throughout this chapter. See table 5.1 for a trace created by this solution.

5.1.3. Run Time Assertion Checking

It is good coding practice to use private fields whenever a field represents the internal state of an object. On the other hand, it can be considered beneficial for automated testing to have all fields accessible. Nevertheless, it is not reasonable from a didactic point of view to force students to use public fields in any case. As described above, a trace includes the values for all fields of all objects, thus also the values of private fields are included here. Traces can thus be used for assertion checking on both public and private fields. An alternative without tracing is to use the Java reflection API which allows to access a subset of the information available via the JDI at run time. As the JDI provides more information and more control over the program under test, Java reflection is not considered here in more detail.

Assertion checking based on traces can be used in the following example: A method call to the tested program is executed that creates a new object. Assertion checking is used to check whether private fields of this new object are set correctly, i.e. whether the constructor of this object's class works as expected. If some assertion is violated, the test can be aborted with a detailed feedback message. If aborting the test is not desirable, the spotted violation can at least be marked in the trace. Regardless of the way chosen, students are informed about the location where an error happens and not just about the malicious effects this error causes in the results of subsequent operations. Of course, students can also locate the error manually reading the trace, but assertion checking can help them in this task.

It is not necessary to limit assertion checking to the last observed step in the trace or to private fields. Instead, also permanent checks for a given field or local variable can be performed. For this purpose the tracing framework needs methods to switch on and off assertion checking for a given field or variable name for all subsequent program steps. This can be used for example to check invariants of sorting algorithms. In this case, first some method calls to the student's program can be performed to fill an array with data. Afterwards checking of assertions ensuring data consistency is switched on and a sorting algorithm implemented by the student is invoked. If the algorithm does not perform correctly, the test run can be aborted immediately after the program step that violated an assertion or a marker in the trace can be set. This way the student gets a precise hint on the location of an error instead of just being informed that the overall result of the algorithm is wrong.

In any case it is obviously necessary to know the names of the private fields or variables to be checked. In teaching scenarios it is possible to provide code templates to the students, which incorporate at least all necessary field declarations. However, this does not solve the problem with local variables. A more complex technique suitable both for private fields and local variables is to apply static code analysis methods as discussed in chapter 4. For example, this can be used to identify all variables used in the termination condition of a `while`-loop, allowing to apply assertion checking to them for checking some loop invariant.

5.1.4. Post Mortem Trace Alignment

If a given program is executed twice with different test data or the same test is executed on different versions of a program, comparison between traces can be of value. Both ways of comparisons include to check the actual changes against the expected changes. This task can obviously not be automated. However, automated trace comparison can be performed, if the trace of a student's solution is compared to the trace of a sample solution. In this case, algorithms known from string or sequence alignment [65] can be used to align the trace of the sample solution with the actual solution. This idea is stimulated by the fact that a program trace is basically a sequence of steps of program behavior, where each step is characterized by the values of the variables scoped in this particular step. Thus, heuristic techniques known from sequence alignment can be applied to match two traces with each other, finding out whether these two traces represent a rather similar or rather different behavior in terms of changes in the variable values. Basically, a human reader of a trace would do the same thing, comparing the actual values in the trace with the conceptual ideas on what should have happened.

Technically, these kinds of mappings produce a score that represents the degree of similarity. So the general idea is to analyze and interpret these scores: A step in the trace that causes a sustainable decrease in the score is likely to contain an error. This idea is based on the assumption that each correct solution will run through the same program states than the correct sample solution. Identical states gain a high score during the alignment and thus a decrease in the score means to visit states that are not part of the correct solution. However, it might be possible that two programs do the same things in general, but visit program states in different orders. Consequently, they produce a different sequence of states, but may nevertheless produce the same output. Thus not every decrease in the scores is necessarily related to an error and it has to be checked whether scores increase later on again. If these have been ruled out, the remaining candidate steps are then reported to the students as automated feedback.

Besides these local error spotting, two more patterns can be expected:

- The traces are almost similar in the beginning but from some point on there is hardly any possible alignment. In this case, the actual solution seems to do something wrong that leads into the wrong direction. The region of the last reliable alignment marks the region of program code where the responsible erroneous statement is most likely to be located.

- There are sections with good alignment between sample solution and actual solution all over the trace, which cover almost the complete sample solution trace, but only parts of the (consequently longer) trace of the actual solution. In this case, the solution seems to be unnecessarily complicated. The code executed during the trace steps without alignment are consequently most likely superfluous. Of course, the same may also happen the other way round, if a student finds a solution more compact than the sample solution used for comparison.

In all cases, trace alignment cannot be used for direct error responses, since results are vague. Sequence alignment is based on probabilities and thus this kind of analysis can only sketch regions of program code that are worth a closer manual inspection. However this can guide students to get a deeper understanding of their program code more easily.

Preliminary Considerations

Considering programming exercises, not each and every comparison of two traces is meaningful. If for example the task is to sort elements of an array, students may be free to implement different sorting algorithms. Comparing a student's trace based on Quick Sort with a sample trace based on Bucket Sort can be expected to be largely useless, because there will be many deviations. Consequently, it is considered useful to perform a static analysis of the source code first, trying to find significant bits of source code revealing the strategy used in the program. As explained earlier in this thesis, it is easy to distinguish between recursive and iterative approaches using static analysis. This kind of analysis can help to make sure that only solutions that apply the same basic strategies are considered for trace alignment. In turn, this requires the teacher to create multiple sample solutions using different strategies, if detailed feedback to all solutions should be possible.

If traces are compared, two main questions have to be answered: (1) Which variable in the first trace corresponds to which variable in the second trace, and (2) which step in the first trace corresponds to which step in the second trace. The approach thus consists of three stages, answering the questions first using heuristic methods and then calculating the rating:

1. Split each trace into columns for each variable used within that trace and find the best match for each column.
2. Find the best match for each step in the traces using only columns matched in stage 1.
3. Give a rating to each step in the traces based on the matching found in stage 2.

The stages are discussed in more detail in the subsections below. Examples and experimental results will be given later on in section 5.1.7.

Identifying Variables

In general, it cannot be assumed that traces from two different solutions to a programming exercise use the same variable names in identical situations. In addition, even the type of two variables need not to be the same although they are used for the same purpose in different solutions. Thus variables have to be identified based on the change of their values over time. For this purpose, each trace is split into columns for each variable used within that trace. Each column from the first trace is then aligned with each column of the second trace using a modified version of the algorithm by Ukkonen [149]. This algorithms allows for alignment of sequences of characters (where each character is a variable value in our case) based on a scoring function for matches and allowing alignments with gaps. The latter is important, because two traces may contain the same steps with one trace having some extra steps in between. These extra steps have to be mapped to gaps during sequence alignment.

The scoring function adds or subtracts the following scores on matching two variable values: If the values are equal (ignoring the type of the variable), 1 is added to the score. If the variables are not in the scope of the current step or are `null`-pointers, the score remains unchanged. Otherwise, the score is reduced by 1. Since types of variables have been ignored so far, an extra check on variable types is made in addition: If the types match exactly, 1 is added to the score. If types differ just in precision (as `double` and `float` do), 1 is added to the score, too. In all other cases, the score is reduced by 1. The gap penalty used in the algorithm is -2. Consequently, a match on value and type gains a score of 2, a match on either value or type gains a score of 0, a mismatch of both value and type gains a -2 and a gap gains a -2 as well. The algorithms thus prefers partial maps over gaps or complete mismatches.

Based on the individual scores for each alignment, variables are identified by maximizing the sum of all scores for the matched columns. If the number of variables in the two traces differs, only the smaller set of variables is considered. The remaining columns of the larger trace are then discarded and not used throughout the remaining process. In addition, a threshold can be defined to discard variables with bad scores in general. However, finding an appropriate threshold is not trivial, since the length of the traces has to be taken into account. For a simple approach, the threshold is set to the median of all sums of scores of all matches.

For an intermediate evaluation of the approach presented so far, each solution in a set of 51 unique solutions to a programming exercise has been aligned with the trace of a sample solution. The sample solution (see listing 5.1) contains five relevant variables: One 2-dimensional array of integers (which was not supposed to change during the execution), two integer variables for line and row indices on the array being used in loops, one 1-dimensional array of type double, and one auxiliary variable of type double. The latter is not absolutely necessary, so that some of the student's solutions aligned with the sample solution contained less variables. Others contained additional variables or the same number of variables but with different meanings. Finally, some of the solutions defined the same set of variables as the sample solution.

Every variable matching has been inspected manually to find out whether it is correct. A variable matching is considered correct, if the algorithm mapped two variables which indeed are used for the same purpose in the programs. Results were satisfying, but not perfect: 45 out of 51 variable matches had been considered correct, while in 6 out of 51 cases at least one variable was matched that should not have been matched. The threshold turned out to be largely ineffective, since it ruled out a variable mapping just in one case. Raising the threshold provided better results for the mapping, but worse results for the subsequent stages. More detailed experiments on determining a good threshold are discussed in section 5.1.7 below.

Besides defining a threshold to rule out mappings with bad scores, the approach presented so far can be refined by merging two or more columns prior to matching the columns. An improved quality of results can be expected from that in cases where one trace makes use of auxiliary variables while the other one does the same operations on the original variables. However, checking all possible merges of columns results in an exponential increase of run time.

Identifying Steps

In the second stage of the overall process, an alignment for the two traces as a whole (rather than for individual columns as in stage 1) is calculated. If the length of the input traces differs by more than the factor 10, no matching is calculated, because results would be largely random in these cases. Again a modified version of Ukkonen's algorithms is used for the calculation, now using the complete set of variable values in one step as a single character in the sequence. The scoring function computes the score for a matching on steps by iteration over all variable values contained in this step: For each variable that matches by value, the score is increased by 2. For each variable that is not in the scope of the current step in both traces, the score is increased by 1. For each variable that is not in the scope of the current step in only one of the traces, the score is decreased by 1. In all other cases, the score is decreased by 2. Gaps do not receive a penalty, but a score which is the number of variables divided by 3 in an integer division. Thus the algorithm tries to maximize the overall score for the traces by aligning steps that contain as much similar variable values as possible, but prefers gaps in favor of alignments with poor matching.

In contrast to stage 1, no post-processing is necessary, but the alignment is directly used as input for the next stage. No manual inspection for evaluation was done, because it cannot clearly be said how to consider an alignment correct. It can only be said that an alignment of two traces is sufficient, if it allows to identify significant deviations between the traces.

Rating

The last stage of the overall process is to rate each step in a trace based on the alignment calculated in the previous stage. The goal is to identify steps that cause a significant deviation between the two traces. Candidates are steps where significant differences between the scores before this step and the scores after this step can be observed.

A simple heuristic approach for this task works as follows: For step n, the number of variables is counted ($count_n$) and also the number of variables that got a positive score during stage 2 ($match_n$). The numbers for step n are then compared to the numbers for the next step $n+1$ (thus $count_{n+1}$ and $match_{n+1}$). If $(match_{n+1} - match) - (count_{n+1} - count) < 0$ is true, then step n is rated as a candidate step.

If a correct sample solution and a faulty student's solution are compared, candidate steps found in the student's solution are likely to stem from lines in the program that contain errors, if they are not neutralized by a subsequent step in which $(match_{n+1} - match) - (count_{n+1} - count) > 0$ is true. If such a neutralizing step is found, then the deviation between student's solution and sample solution is only temporarily and does not hint to an error. Thus the candidate step starting this temporarily deviation can be ignored. Otherwise, the candidate step is marked as a potential error in the trace of the student's solution and the annotated trace is delivered as feedback to the student.

Sample trace selection

All considerations in the previous sections were concerned with the comparison of exactly one pair of traces. However, it can be assumed to be beneficial to compare one student's solution to more than one sample solution and rate candidate steps based on the alignment with the closest sample solution. This is also true if static analysis was used to analyze the approaches taken by different solutions, as there might be different possibilities to implement an approach, that has large impact on the form of the trace. For example, loops can count upwards or downwards, or temporary variables can be reset in different places within a loop. Thus some approach needs to be defined for the process of selecting the most appropriate sample solution.

As for the rating above, again a simple heuristic approach is proposed here. For this approach, a total match quality for the alignment is computed as the average of $match_n/count_n$ from the previous step. Thus the student's solution is aligned with all traces and the match quality is computed for all of these alignments. The results are between 0.0 and 1.0, where the latter represents a perfect alignment that will not result in any candidate steps. As this can trivially be achieved in cases where the previous steps decide to not match any variable, traces with a match quality of 1.0 can be ignored. On the other hand, a low match quality represents a great mismatch between the student's solution and the sample solution which might result in random lines being identified as candidate steps. Hence it is also desirable to ignore traces with a low match quality. At the same time, a match quality below 1.0 does not necessarily result in lines marked in the student's trace, since candidate steps that are discarded still have impact on the total match quality. Thus a simple approach for selecting the most appropriate sample trace is to select the trace where the best match quality that is below 1.0 and above some threshold. Based on some experiments, the threshold is set to 0.88, but further experiments on this decision are desirable and will partly be discussed in section 5.1.7 below.

5.1.5. Post Mortem Performance and Coverage Analysis

A second additional possibility to analyse traces after program termination is to gain performance measures from these traces. In contrast to taking performance measure by run time, results from trace analysis are independent of the performance of the underlying hardware and operating system and thus more reliable. Especially, when different grading servers are used that may use different hardware platforms, just using the run time is not sufficient as a performance measure.

The easiest measure based on traces is of course the number of steps in the traces. A solution passing the same test cases with less steps than another solution can in general be considered more efficient. However, this simple metric does not take into account that a line may contain one or more statements and that different statements are of different complexity, so stepping through two simple lines may still be more performant than stepping through one complex line. To avoid this, other performance measurements can be used based on the generated traces, for example the number of changes in variable values. This way, lines changing more than one variable by containing more than one statement provide a higher contribution to the performance count than lines with just one change. Similarly, method calls can be counted separately.

Performance measures gained that way do not contribute to checking the correctness of a solution, unless a certain performance was required by the exercise explicitly. Instead, they can be used to motivate students to improve their solution by giving reference performance figures from sample solutions. At least for ambitious students it might be interesting to know that they can achieve the same program output with more performant code.

Coverage analysis requires slightly more effort than plain line counts. For each line of the source code that contains executable instructions, it has to be recorded whether or not it occurs in the trace generated by at least one test case. Source code lines that are not covered by any trace can be considered superfluous with respect to the given set of test cases. Uncovered code can have two distinct reasons that also influence the expected benefit from feedback generation:

- Some code section may be not covered because it actually implements a feature that is never used. For example, this could be a `for`-loop that iterates over all list elements in a code branch that is only invoked on empty lists. This case is especially interesting if a solution passes all test cases, but there is still uncovered code. This can motivate students to improve their solution although it has already passed all test cases.

- Some code section may be not covered because some condition for entering it is erroneously never fulfilled, although it should be at least in some test cases. For example, a `switch`-statements offers three branches, where the third branch is never entered because the second branch catches more cases than expected. This is especially interesting in conjunction with cases in which not all test cases are fulfilled. This gives a strong hint to students where to search for errors in their solution.

In both cases, careful design of test cases is necessary, because coverage measures can only be used if it is guaranteed that the code is not intended to execute other test cases than the ones used to create the traces. Only in this case it can be said that code not covered by any of these traces is really unnecessary. It also has to be taken into account that there might be methods in the source code that are intended to be used for local testing, but not by the test cases (such as the `main`-method in Java or utility methods for console output).

5.1.6. Component `TracingJavaChecker`

The marking component named `TracingJavaChecker` implements dynamic checks as white-box-testing and tracing for Java programs. It facilitates run time assertion checking as well as post mortem trace analysis. It consists of several Java classes organized in four Java packages: The main package `tracingchecker` contains the core logic for creating traces, the package `tracingchecker.traces` contains classes forming the data structure for storing traces, the package `tracingchecker.analyzer` contains the logic for analyzing traces, and the package `tracingchecker.framework` contains a utility library that has to be deployed alongside with the actual test driver written by the user. The complete class diagram for the component is shown in figure 5.1. The following subsection gives an overview on the workflow of the component, while the subsequent subsections dig into the details of the implementation.

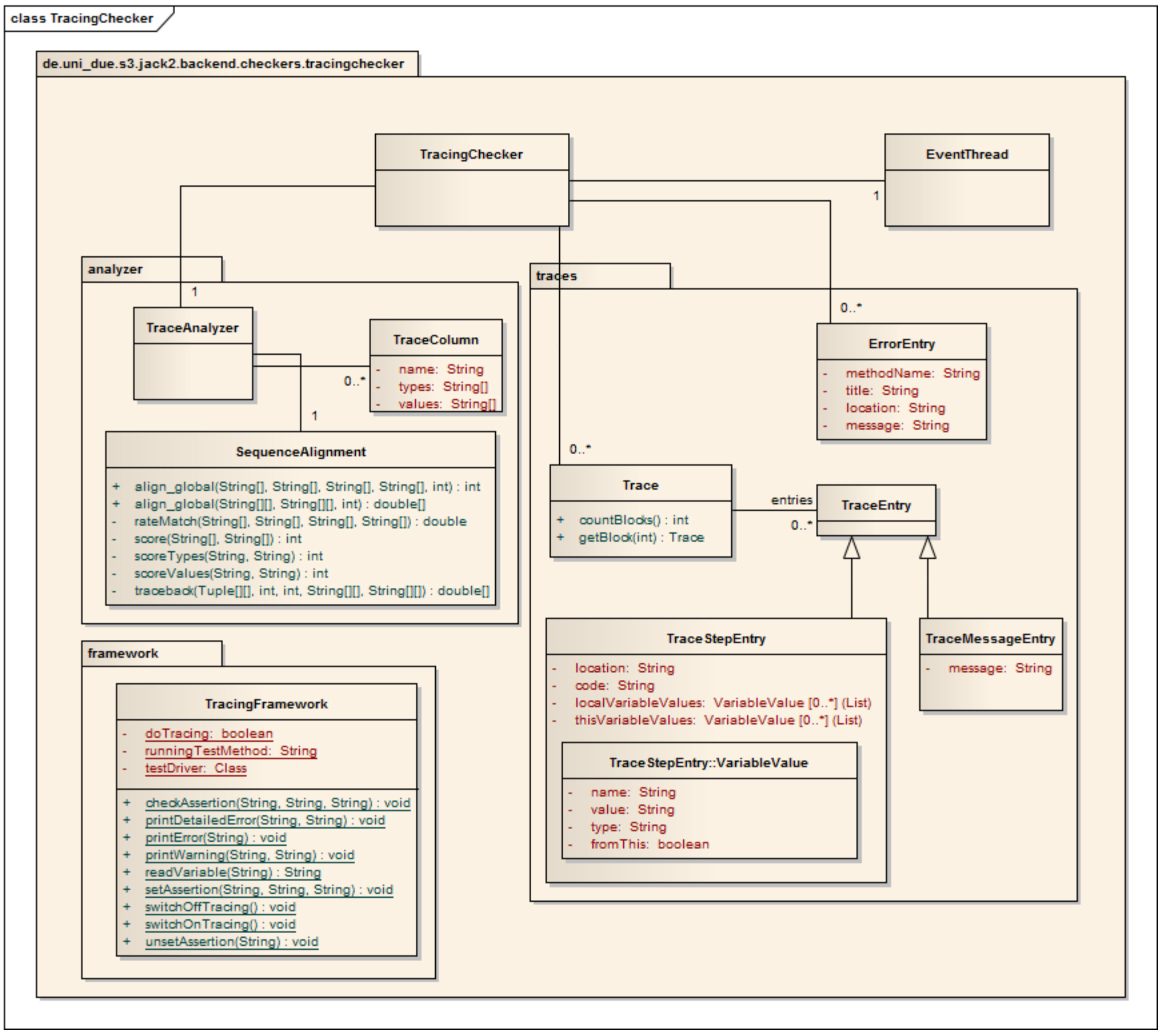

class TracingChecker
de.uni_due.s3.jack2.backend.checkers.tracingchecker
TracingChecker
EventThread
1
analyzer
1
TraceAnalyzer
TraceColumn
- name: String
- types: String[]
- values: String[]
0..*
1
SequenceAlignment
+ align_global(String[], String[], String[], String[], int) : int
+ align_global(String[][], String[][], int) : double[]
- rateMatch(String[], String[], String[], String[]) : double
- score(String[], String[]) : int
- scoreTypes(String, String) : int
- scoreValues(String, String) : int
- traceback(Tuple[][], int, int, String[][], String[][]) : double[]
traces
0..*
ErrorEntry
- methodName: String
- title: String
- location: String
- message: String
0..*
Trace
+ countBlocks() : int
+ getBlock(int) : Trace
entries
0..*
TraceEntry
TraceStepEntry
- location: String
- code: String
- localVariableValues: VariableValue [0..*] (List)
- thisVariableValues: VariableValue [0..*] (List)
TraceStepEntry::VariableValue
- name: String
- value: String
- type: String
- fromThis: boolean
TraceMessageEntry
- message: String
framework
TracingFramework
- doTracing: boolean
- runningTestMethod: String
- testDriver: Class
+ checkAssertion(String, String, String) : void
+ printDetailedError(String, String) : void
+ printError(String) : void
+ printWarning(String, String) : void
+ readVariable(String) : String
+ setAssertion(String, String, String) : void
+ switchOffTracing() : void
+ switchOnTracing() : void
+ unsetAssertion(String) : void

Figure 5.1.: Class diagram for component `TracingJavaChecker`. Some methods and fields are hidden in the class diagram to make it more readable.

General workflow

The general workflow of the component is as follows (see figure 5.2): Class `TracingChecker` is invoked by the backend core and requested to perform a check for a given solution. All source code files, a test driver class and an JAR archive containing the byte code of `TracingFramework` are copied to the working directory and compiled. If no compiler errors occur, a new virtual machine ("TargetVM" in the sequence diagram) is initialized and requested to execute the main method of `TracingFramework`. The actual test driver class is passed as parameter to this method. However, the new virtual machine is started in debug mode and thus it is not active until it receives a command to resume via the debug interface. Prior to sending this command, an instance of `EventThread` is started that listens to any events fired by the virtual machine. A second thread listens to the standard output. When both threads are ready, the virtual machine is resumed.

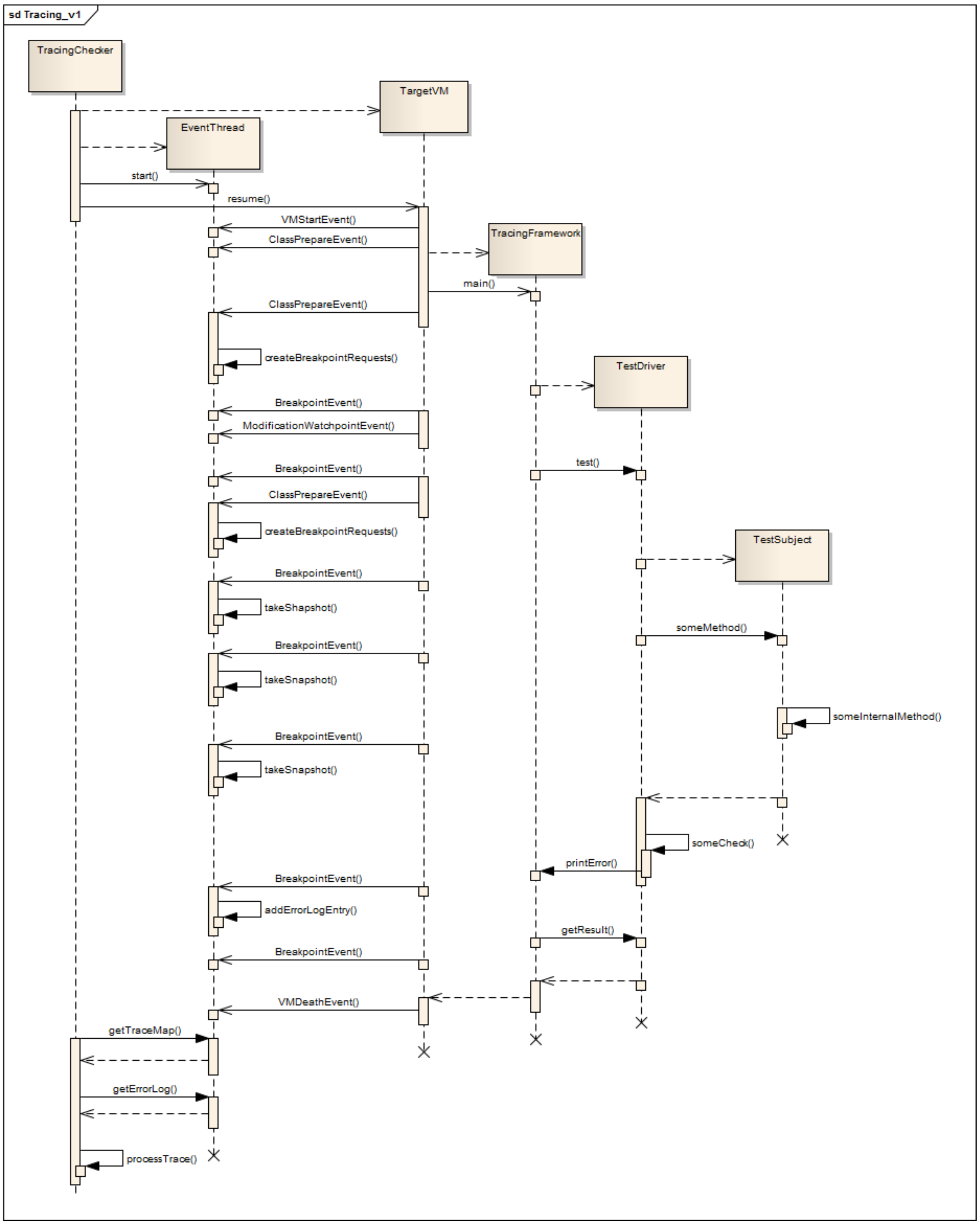


Figure 5.2.: Sequence diagram illustrating the sequence of calls performed when running a white-box test with the `TracingChecker`

The main method of the tracing framework reads the test driver class and interprets it by reflection. Any test method (identified by a `@Test` annotation) is invoked. Each time a class from the tested solution is loaded, breakpoint events are requested for all of its lines. Afterwards, each time a breakpoint is hit, a snapshot is created. Checks on assertions and comparison of actual return values and expected return values of a method have to be performed inside the test driver method. The test driver may request values and generate error messages by calling a respective method in the tracing framework. The event thread is informed about these method calls and stores the messages in an error log, or returns the requested values, respectively. After having executed all test methods, the framework requests the overall grade for the solution from the test driver and return it to the `TracingChecker`. In turn, the `TracingChecker` requests all created traces and error logs and starts post mortem analysis of the traces. As the final step, the backend core requests result data (including overall grade, error messages and any data written to the standard output) and the traces from `TracingChecker` and sends them to the JACK server.

Collection and run time analysis of trace information

Collection of trace information is initiated by class `TracingChecker` and realized by class `EvenThread`. The latter is also responsible for doing run time assertion checking.

According to the steps defined in section 5.1.2, the component is intended to generate traces using the Java debugging interface (JDI) [79]. With JDI the Java virtual machine provides a comprehensive interface for the inspection of the program state during run time. It allows to start, suspend, resume and terminate a virtual machine programatically. It gives access to all objects from the heap, all of their field values and all local variable values. The later is only available if the source code is compiled with the compiler option "`-g`" that includes debug information into the byte code. The information named above is accessible whenever the virtual machine under inspection is suspended. In multi-threaded programs information about the current active thread, waiting threads and thread locks can also be acquired. The only important information that can not be acquired directly is the value of the program pointer. This information can only be accessed in conjunction with so-called "locatable events". These are a subset of all events fired by the JDI in various situations. In order to receive these events, an event request has to be issued, specifying the kind of event of interest. The different types of events and especially those received by the tracing component are:

`AccessWatchpointEvent` Watchpoints can be defined on field declarations. An `AccessWatchpointEvent` is fired whenever a field with a watchpoint is accessed for reading. This is a locatable event and the location of this event is the file and line in which the access occurs. Since read access to fields is of no interest for tracing, the tracing component does not issue an event request for this kind of event.

`BreakpointEvent` Breakpoints can be defined on any executable line in a source file. If program execution comes to a line with a breakpoint, a `BreakpointEvent` is fired before the code of this line is executed. This is a locatable event and the location of this event is the file and line in which the breakpoint is defined. The tracing component registers breakpoints for all executable lines in all files of a student's solution that should be traced. In addition, it registers breakpoints for all lines within public methods of the test driver class and for some methods within the tracing framework. Obviously, event requests are issued for breakpoint events. See further down in this section for more details on handling of breakpoint events.

`ClassPrepareEvent` Before an instance of a class can be created, the class definition has to be loaded by the class loader. After this has happened and before the first instance is created, a `ClassPrepareEvent` is fired. The tracing framework issues an event request for these kind of event, in order to amend the loaded classes with breakpoints or to issue `ModificationWatchpointEvent`s (see below) for its fields.

`ExceptionEvent` Depending on the chosen configuration, any time or sometimes when an exception is thrown inside the program under inspection, an `ExceptionEvent` is fired. This is a locatable event and the location of this event is the file and line in which the exception was thrown in case of exceptions in non-native methods and the file and line of the first non-native method from the stack trace in case of an exception in a native method. Since the occurrence of exceptions is relevant for tracing, the tracing component issues an event request for this kind of event.

`MethodEntryEvent` Any time a method is entered, a `MethodEntryEvent` is fired, before any code in this method is executed. This is a locatable event and the location of this event is the file and first line of the method's declaration. It is alluding to use this kind of event to get informed about the beginning of test methods. However, this involves a significant overhead, because event request cannot be issued just for specific classes. If method entry events are requested, they are generated for all methods, including native methods and standard library methods, and most of them have to be filtered out. Since this decreases performance, method entry events are only requested in the special case the class `java.util.Scanner` is loaded in order to react on calls to its blocking methods, that can cause a test run to hang up.

`MethodExitEvent` Any time a method is exited normally (i.e. without an exception), a `MethodExitEvent` is fired after all the method's code or a return statement has been executed. This is a locatable event and the location of this event is the file and line of the return statement or the last line of the method's declaration, respectively. For the same reasons as with method entry events, no event request for method exit events is issued by the tracing component.

`ModificationWatchpointEvent` Similar to the read access for watchpoints as explained above, a `ModificationWatchpointEvent` is fired whenever a field with a watchpoint is accessed for modifying access. This is a locatable event and the location of this event is the file and line in which the access occurs. The tracing component uses this kind of event to communicate with the tracing framework: The class `TracingFramework` declares fields that store information about the current test run (e.g. the name of the current test method). The tracing component issues event requests for this kind of event and thus gets informed about the contents of these fields as well.

`StepEvent` If a virtual machine is suspended, it can be asked to execute the next executable fragment (e.g. one of several nested method calls) of the current line or the whole line. This behaviour is called "stepping" because it steps from fragment to fragment or line to line. A StepEvent is fired whenever a step is performed and right before the actual source code for this step is executed. However, the location of this event is the file and line reached after the execution of this source code. Step events are used by the tracing component in cases when a test method should be aborted, so event request for step events are only issued in this special case. The handler for step events then determines the return type of the currently active method and calls forceEarlyReturn() on the suspended main thread. The method is hence removed from the stack frame. This is continued until the method removed is the test method to be aborted. After that, no more step events are requested and the tracing framework can run the next test method.

WatchpointEvent This event is no separate event, but the superclass of AccessWatchpoint-Event and ModificationWatchpointEvent and thus fired whenever a field with a watchpoint is accessed for reading or modification. It is not used by the tracing framework.

The handling of BreakpointEvents is the most important step for acquiring the necessary trace information and controlling the trace generation at all. Any time a breakpoint is hit, the virtual machine is suspended right before the according line of code is executed. Since breakpoints are requested not only for the program under test, but also for the test driver and the tracing framework, handling breakpoint events serves three purposes.

First, if the breakpoint occurred in class TracingFramework, the current method inside this framework is determined. For trace generation and assertion checking, all methods of interest within this framework are dummy implementations with just one line, intended just to facilitate communication with the tracing component. Thus the first breakpoint hit on these methods is the only one. Using breakpoints is hence a valid replacement for using method entry events, that would serve the same purpose, but with worse performance. Handling of the breakpoint depends on the name of the method:

- If the breakpoint is on methods printWarning, printError, or printDetailedError, the parameter values from these methods can be read as local variables from the stack frame and used to create a new entry in the error log for the test run. These methods are intended to be called by the test driver to report back any errors found during the test run, e.g. a mismatch of actual output and expected output.

- If the breakpoint is on method readVariable, the current stack frame is searched for a variable with a name given as parameter value of this method. If a variable with this name exists on the stack frame, its value is returned to the caller. If no variable with this name exists, null is returned. This method is intended to be called by the test driver to read a (possibly private) variable from the program under test for assertion checking.

- If the breakpoint is on method checkAssertion, the current stack frame is searched for a variable as above. However, the method takes also an expected value and an error message as additional parameters. If a variable with the given name exists, but does not contain the expected value, the message is added to the error log. If the variable contains the expected value or does not exist at all, nothing happens. This method is a convenience method for simple checks on assertions and intended to be called by the test driver.

- If the breakpoint is on method `setAssertion`, also three parameters for a variable name, an expected value and a message are taken into account. Instead of checking the assertion directly, it is added to a list of assertions to be checked in any subsequent step, including the current one. If there was already an assertion defined for this variable name, it is overwritten. This method is intended to be called by the test driver to start permanent checks for invariants.
- If the breakpoint is on method `unsetAssertion`, any existing assertion for the variable name given as parameter value of this method is removed from the list of assertions to be checked. This method is intended to be called by the test driver to stop permanent checks for invariants.

Second, if the breakpoint occurred in a class belonging to the student's solution, the current method inside this class is determined and also the point in time the breakpoint occurred. In case it is the first breakpoint in a method that has been just called by the test driver, an according message is added to the trace. In any case, the line is recorded for calculating the test coverage later. Finally, a snapshot of the program state is taken that contains information about all objects on the heap with all their fields as well as all local variable values from the current stack frame. The snapshot is added to the trace for the current test method. However, there are two exceptions: (1) The test driver can call a method on the tracing framework that disables tracing explicitly for the running test method or until it is switched on again explicitly. (2) Traces can be limited to a certain length to reduce resource consumption. In both cases, no snapshots are taken and stored.

Third, both on the solution under test and the tracing framework, the total run time of the current test method is checked and the test is aborted if a time limit is exceeded. Moreover, all assertions in the list filled by the `setAssertion`-method are checked on the current stack frame. This is the last action in breakpoint handling and the virtual machine is resumed afterwards.

The breakpoints set on public methods of the test driver do not imply any specific actions, but are needed to determine the beginning and end of test cases.

Post mortem trace analysis

Post mortem trace analysis is performed partly in class `TracingChecker` and partly in class `TraceAnalyzer`. The former calculates test coverage based on the records created by class `EventThread` as mentioned in the previous subsection above. The coverage is simply calculated by dividing the number of visited lines by the number of total lines that could possibly be visited. The result is returned to the teacher as an additional information without further generation of feedback messages. Instead, a list of uncovered lines in the solution is generated and send back to the server at the end of the checking process. The server can use this list of lines to colorcode these lines when displaying the submitted source code to the student. This way the student gets a visual feedback on uncovered lines.

The process of trace alignment is somewhat more complex and illustrated in figure 5.3. Trace alignment is mainly realized in class `TraceAnalyzer`. It provides a method that takes

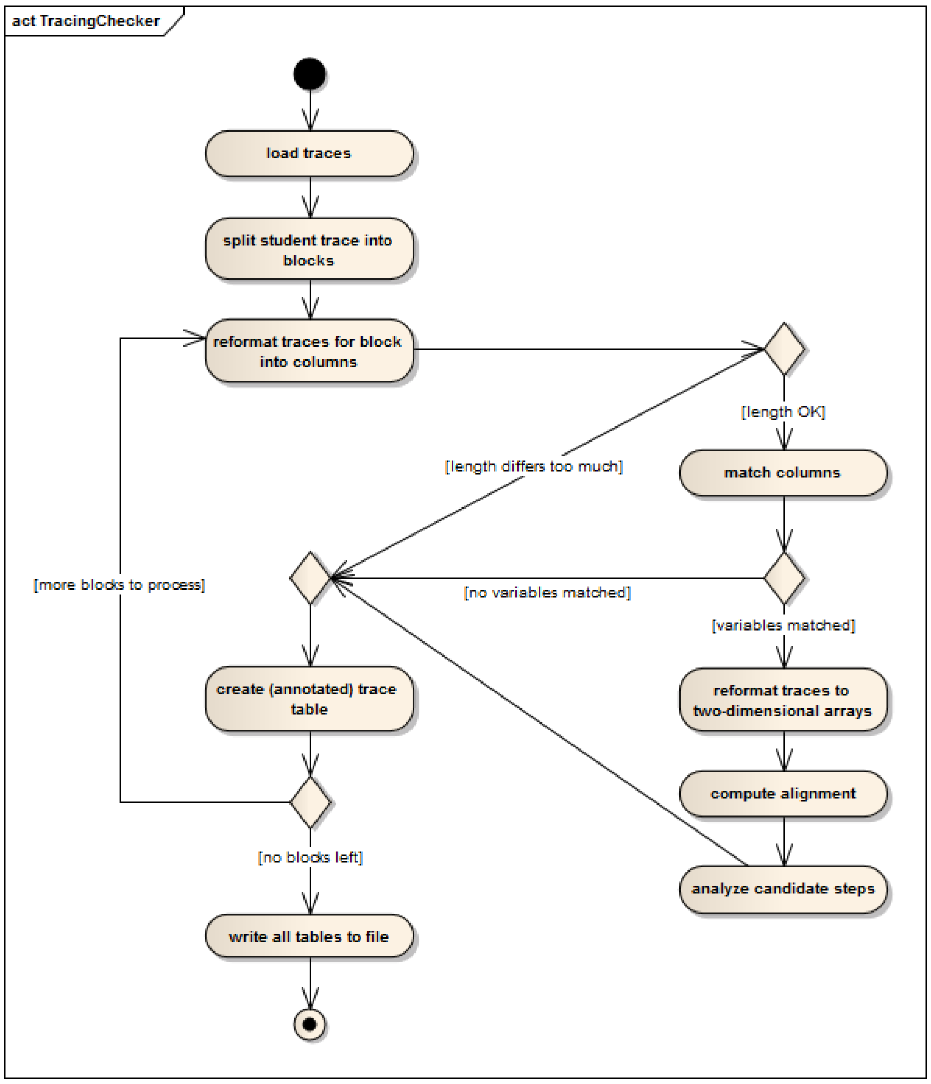


Figure 5.3.: Activity diagram for trace alignment in component `TracingJavaChecker`. Activities on the left hand side are realized inside class `TracingChecker`, while activities on the right hand side are realized inside class `TraceAnalyzer`.

a student's trace and a sample trace as input. As a preprocessing step handled by class `TracingChecker`, the traces are reformatted to a different data structure: Instead of a list of trace entries, where each entry contains one value for each variable scoped in this step, the trace analyzer expects a set of trace columns, where each column contains a sequence of values for one variable. In addition, the trace is split into independent blocks if the test driver made more than one method call to the solution under test, so that traces for each of these calls are handled separately.

For each of these blocks, method `match()` on class `TraceAnalyzer` is called in order to generate a list of markers. It returns immediately with an empty list of markers, if the length of the traces differs too much. Otherwise, it tries to find the best matches for the columns in order to identify variables. Again, it returns with an empty list of markers if no variables could be identified. Afterwards, it reformats the trace representation again to a two-dimensional array with one identified variable per column and one step per row. The ordering of variables is the same both for the student's trace and the sample trace used for comparison. On these arrays, the actual trace alignment is computed. Result of the alignment is a rating with one entry for each step in the student's solution, as defined at the end of section 5.1.4. From these ratings, markers for steps of the trace are generated and returned.

Finally, class `TracingChecker` creates the final trace tables for output and adds annotations by colored lines to them based on the markers. It also assembles several tables for one test run if necessary and writes the final output to a file, which is returned to the server later on as part of the overall result of the checking process.

Two activities in this process deserve special attention: Matching of columns and computation of trace alignment. Matching of columns in turn consists of two steps: Computing alignments for all possible combinations of columns and finding the best combination.

The computation of alignments for columns is based on Ukkonen's algorithm as discussed in the conceptual section. The code of the implementation is provided in annex B.1. It returns an integer value which is the score for the best possible match. Its input are four one-dimensional arrays, where the first and the second one contain the variable values for the trace columns, while the third and fourth one contain the type names for the variables. Consequently, two different scoring functions are used: One for scoring comparisons by value and one for scoring comparisons by type.

Finding the best computation is done differently for small and large traces. On small traces containing not more than six different variables, all possible combinations are tested to find the one that results in the maximum overall score. On larger traces, this would imply a significant loss of performance, because the number of possible combinations is the faculty of the number of columns. Thus for seven columns, more than 5'000 combinations must be checked. Instead, a greedy algorithm is used in these cases that iteratively matches the columns with the best score.

The computation of alignments for the complete traces is again based on Ukkonen's algorithm. The code of the implementation is provided in annex B.2. It takes the two traces as input and uses its own scoring function, that in turn calls the one for comparing variables by value. The output is computed by a separate method that maps the results from the alignment matrix back to the sequences of trace entry and subsequently computes the rating for the aligned entries. The ratings denote the candidate steps that are returned for further analysis.

5.1.7. Example: Program Checks

In this section, dynamic analysis of Java programs by post mortem trace alignment is illustrated by two examples. See table 5.3 for a student's trace and table 5.1 already introduced on page 67 for the sample trace. The student's trace was created from the source code shown in listing 5.2. The first step (matching the variables) in this example decided to ignore variables `rows` and `cols` because of the scores shown in table 5.2. Variable `i` receives a score of 54 because of 30 perfect matches (adding 60 to the score), two gaps, one mismatch (in total reducing the score by 6), and four neutral lines. Variable `j` receives a score of 24 because

```
    public static float[] arithmetic_average(int[][] mat){
        int rows = mat.length;
         int cols = mat[0].length;
         float [] v = new float[mat.length];

         for(int i=0;i<rows;i++){
               int temp=0;
             for(int j=0;j<cols;j++){
             temp+=mat[i][j];
             }
             v[i]=(float)(temp/3);
           }
        return v;
    }
```

Listing 5.2: Piece of source code cut out from a student's solution for the example exercise. See table 5.3 for a trace created by this solution.

	i	j	cols	rows	mat	temp	v
i	**54**	-18	-4	-6	-74	-10	-68
j	-14	**24**	-34	-36	-74	-4	-68
mat	-74	-74	-74	-74	**66**	-74	-74
temp	-58	-58	-62	-64	-74	**-26**	-70
vec	-72	-72	-72	-72	-74	-72	**10**

Table 5.2.: Scores calculated in the first step when comparing the trace shown in table 5.3 with the sample trace from table 5.1. Scores are calculated as explained on page 71: Matches of type and value gain a score of 2, matches of either type or value gain 0, complete mismatches and gaps gain a score of -2. The median of all values in this example is -68. The best matching found is the one indicated by bold font numbers, gaining a total score of 128.

of 18 perfect matches (adding 36 to the score), six gaps (reducing the score by 12), and 15 neutral lines. The other scores can be explained similarly. Scores of -74 are based on the fact that the sample trace contains 37 lines and all of them produce a mismatch with a line from the student's trace.

Based on this mapping of variables, the alignment for the whole trace is computed. The first line gets a score of 6, because it has one perfect match for variable `mat` and four none-scoped variables. The next two lines are mapped to gaps, which is reasonable because they only deal with variables not matched between the traces. The fourth line of the student's solution is then mapped to the second line of the sample solution, receiving a score of 7 because of two perfect matches and three none-scoped variables. Line five of the student's solution is mapped to line three of the sample solution with score 3 because of the different order variables come into scope: `i` is assigned in the student's solution but out of scope in the sample solution and the other way round for `temp`. Consequently, the score is decreased by 2 for these two variables. Two perfect matches and one remaining none-scoped variable contribute 5 to the score as above. Finally, line six of the student's solution mapped to line four of the sample solutions receives a score of 9, because now both `i` and `temp` are assigned

File and line	Variable values							Line of code to be executed
	i	j	cols	rows	mat	temp	v	
Miniprojekt2:164					{{1,2,1},{4,3,2},{2,2,7}}			`int rows = mat.length;`
Miniprojekt2:165				3	{{1,2,1},{4,3,2},{2,2,7}}			`int cols = mat[0].length;`
Miniprojekt2:166			3	3	{{1,2,1},{4,3,2},{2,2,7}}			`float [] v = new float[mat.length];`
Miniprojekt2:168			3	3	{{1,2,1},{4,3,2},{2,2,7}}		{0.0,0.0,0.0}	`for(int i=0;i<rows;i++){`
Miniprojekt2:169	0		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{0.0,0.0,0.0}	`int temp=0;`
Miniprojekt2:170	0		3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{0.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	0	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	0	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	1	{0.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	0	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	1	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	0	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	3	{0.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	0	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	3	{0.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	0	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{0.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:173	0		3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{0.0,0.0,0.0}	`v[i]=(float)(temp/3);`
Miniprojekt2:168	0		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,0.0,0.0}	`for(int i=0;i<rows;i++){`
Miniprojekt2:169	1		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,0.0,0.0}	`int temp=0;`
Miniprojekt2:170	1		3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{1.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	1	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{1.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	1	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{1.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	1	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{1.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	1	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	7	{1.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	1	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	7	{1.0,0.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	1	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	9	{1.0,0.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:173	1		3	3	{{1,2,1},{4,3,2},{2,2,7}}	9	{1.0,0.0,0.0}	`v[i]=(float)(temp/3);`
Miniprojekt2:168	1		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,3.0,0.0}	`for(int i=0;i<rows;i++){`
Miniprojekt2:169	2		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,3.0,0.0}	`int temp=0;`
Miniprojekt2:170	2		3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{1.0,3.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	2	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	0	{1.0,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	2	0	3	3	{{1,2,1},{4,3,2},{2,2,7}}	2	{1.0,3.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	2	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	2	{1.0,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	2	1	3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{1.0,3.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:171	2	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	4	{1.0,3.0,0.0}	`temp+=mat[i][j];`
Miniprojekt2:170	2	2	3	3	{{1,2,1},{4,3,2},{2,2,7}}	11	{1.0,3.0,0.0}	`for(int j=0;j<cols;j++){`
Miniprojekt2:173	2		3	3	{{1,2,1},{4,3,2},{2,2,7}}	11	{1.0,3.0,0.0}	`v[i]=(float)(temp/3);`
Miniprojekt2:168	2		3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,3.0,3.0}	`for(int i=0;i<rows;i++){`
Miniprojekt2:175			3	3	{{1,2,1},{4,3,2},{2,2,7}}		{1.0,3.0,3.0}	`return v;`

Table 5.3.: Trace generated for the source code shown in listing 5.2, when invoked for the array `{{1,2,1},{4,3,2},{2,2,7}}`. The first encounter to source code line 173 is marked as candidate step and indeed contains the relevant error by applying the type cast to `float` at the wrong position in the integer division.

```
137     public static float[] arithmetic_average(int[][] mat){
138         float arith=0; float n = (float) mat.length;
139         float [] vektor =new float [mat.length];
140         for (int i=0; i<=mat.length; i++){
141             for (int j=0; j<=mat[0].length; j++){
142                 arith+= (float) mat[i][j];
143             }
144             vektor [i]= arith/n;
145             arith=0;
146         }
147         return vektor;
148     }
```

Listing 5.3: Another piece of source code cut out from a student's solution for the example exercise. See table 5.4 for a trace created by this solution.

in both traces. Hence there are four perfect matches and one none-scoped variable. The algorithm thus treats the score of 5 on the fifth line of the student's solution as an irrelevant deviation and does not mark it as a candidate step.

Later on, a candidate step is marked (see grey line in table 5.3), because the perfect match of variables `vec` and `v` is lost by executing the source code associated with this line and never gained again later on. Thus the score drops for matching the next line and never increases to the original level again. Indeed the marked line contains the relevant error, because the integer division is used here, but a decimal division was expected.

Another example is shown in 5.4 and listing 5.3. It also refers to the sample solution from table 5.1 on page 67. Here a more serious error occurs because the trace stops immediately on the occurrence of an exception, since 3 as value of variable `j` cannot be used as an index for array access on an array of size 3. However, not the last line where the exception is thrown is marked in the trace, but already the line before where the actual error occurs: This line uses `<=` as a comparison operator instead of `<`, causing the undesired increment of `j`. Consequently, the score for matching drops on the next line, because there is no match for value `3`. Since the method is terminated by exception after that step, there is no recovery from this decrease in the rating and it is not considered as an irrelevant deviation. Thus it is marked as a candidate step. This also happens if the same method is invoked with different test data, e.g. a larger array. If there is no test case that invokes the method with an array of size 0, the exception will happen in all test cases and no code beyond line 142 will be executed. Thus code coverage analysis will report that executable lines 144, 145, and 147 are not covered by any test case. Obviously, this does not happen because they are entirely useless, but because they represent dead code just because of the previous error.

An example where trace alignment fails to produce usable feedback is shown in listing 5.4. Here the student tried to solve the task without using a loop construct, which only works for arrays of a fixed size. Indeed the code is correct if invoked with an array of size 3 times 3. However, the test case using a larger array as above fails, producing the trace shown in table 5.5. It only has 5 lines, whereas the corresponding trace of a sample solution with loop constructs contains 56 lines. Thus no alignment is calculated at all because the trace lengths differ by more than factor 10. Indeed it can be considered useless zu point out some erroneous line in this trace instead of informing the student about the necessity for loop

```
    public static float[] arithmetic_average(int[][] mat){
        float[] bla = {0,0,0};

        bla[0]=(mat[0][0]+mat[0][1]+mat[0][2])/3f;
        bla[1]=(mat[1][0]+mat[1][1]+mat[1][2])/3f;
        bla[2]=(mat[2][0]+mat[2][1]+mat[2][2])/3f;

        return bla;
    }
```

Listing 5.4: Another piece of source code cut out from a student's solution for the example exercise. See table 5.5 for a trace created by this solution.

File and line	Variable values						Line of code to be executed
	arith	i	j	mat	n	vektor	
Miniprojekt2:138				{{1,2,1},{4,3,2},{2,2,7}}			`float arith=0; float n = (float) mat.length;`
Miniprojekt2:139	0.0			{{1,2,1},{4,3,2},{2,2,7}}	3.0		`float [] vektor =new float [mat.length];`
Miniprojekt2:140	0.0			{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int i=0; i<=mat.length; i++){`
Miniprojekt2:141	0.0	0		{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int j=0; j<=mat[0].length; j++){`
Miniprojekt2:142	0.0	0	0	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`arith+= (float) mat[i][j];`
Miniprojekt2:141	1.0	0	0	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int j=0; j<=mat[0].length; j++){`
Miniprojekt2:142	1.0	0	1	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`arith+= (float) mat[i][j];`
Miniprojekt2:141	3.0	0	1	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int j=0; j<=mat[0].length; j++){`
Miniprojekt2:142	3.0	0	2	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`arith+= (float) mat[i][j];`
Miniprojekt2:141	4.0	0	2	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`for (int j=0; j<=mat[0].length; j++){`
Miniprojekt2:142	4.0	0	3	{{1,2,1},{4,3,2},{2,2,7}}	3.0	{0.0,0.0,0.0}	`arith+= (float) mat[i][j];`

Table 5.4.: Trace generated for the source code shown in listing 5.3, when invoked for the array `{{1,2,1},{4,3,2},{2,2,7}}`. Execution stops because of an exception in the last step, since 3 is no valid index for array access. The second but last line is rated as a candidate step. Indeed it contains the relevant error by using `<=` instead of `<`.

File and line	Variable values		Line of code to be executed
	bla	mat	
Miniprojekt2:202		{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}	`float[] bla = {0,0,0};`
Miniprojekt2:204	{0.0,0.0,0.0}	{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}	`bla[0]=(mat[0][0]+mat[0][1]+mat[0][2])/3f;`
Miniprojekt2:205	{1.3333334,0.0,0.0}	{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}	`bla[1]=(mat[1][0]+mat[1][1]+mat[1][2])/3f;`
Miniprojekt2:206	{1.3333334,3.0,0.0}	{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}	`bla[2]=(mat[2][0]+mat[2][1]+mat[2][2])/3f;`
Miniprojekt2:208	{1.3333334,3.0,3.6666667}	{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}	`return bla;`

Table 5.5.: Trace generated for the source code shown in listing 5.4, when invoked for the array `{{1,2,1,4},{4,3,2,7},{2,2,7,3},{-3,2,1,0}}`.

	Exercise 1				Exercise 2			Exercise 3	
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 1	Test 2
True positive	6	7	7	4	30	20	15	15	19
False positive	12	10	13	10	17	6	11	12	10
False negative	7	8	10	12	15	12	7	0	11
Precision	0.33	0.41	0.35	0.29	0.64	0.77	0.58	0.56	0.66
Recall	0.46	0.47	0.41	0.25	0.67	0.63	0.68	1.00	0.63

Table 5.6.: Evaluation results for the simple heuristic approach using the median of all sums of scores of all matches as a threshold during variable identification. One trace of a sample solution was used for alignment in this experiment.

constructs in general. The latter can be done via static checks as shown in chapter 4.

Additional experimental results

As explained in section 5.1.4 above, the threshold used during variable identification and the thresholds for determining the most appropriate trace to compare with have some impact on the overall outcome of the trace alignment. Hence the following sub-section provides more experimental results for different thresholds and evaluates the accuracy gained with each of them. The accuracy of trace alignment in terms of correctly and incorrectly marked steps can be evaluated by manual inspection. If the lines in the program code corresponding to these candidate steps do contain errors indeed, the rating is considered correct (true positive). If some line in the program code corresponding to a candidate step does not contain an actual error, the rating is considered incorrect (false positive). If the code contains at least one error, but none of them is marked as candidate step, the rating is also not correct (false negative). The case of true negatives is irrelevant here, since for correct solutions no markers in the traces need to be generated at all.

Based on the inspection results, quality of the approach can be measured by calculating precision and recall for a given set of traces. Two things can be observed in advance: First, there may be cases in which some line is marked as candidate step, which does not contain an error, while other lines containing an error are not marked. These traces consequently count as false positive and false negative results at the same time. Second, a remarkable number of false negatives can be expected due to the fact that traces are not compared if their lengths differ too much. Such a difference may hint to the fact that the two programs are using different strategies, so it is indeed desirable to give no fine grained hints in these cases. These kinds of false negatives are not counted in the results discussed below. If they are added, recall will decrease. It has to be noted that more decisions of these kind can be taken, lowering or raising the expectations towards the number of candidate steps that should be produced by the algorithm.

The evaluation has been carried out by analyzing traces from nine test cases from three different exercises. The results are shown in tables 5.6, 5.7, and 5.8. The sample trace used in test case 1 for exercise 3 is the one shown in table 5.1 above. So far, all these experiments compare the student's solutions to exactly one sample solution. Experiments one selecting appropriate traces for alignment are discussed later in this section.

Results of the evaluation show a fair performance of the simple heuristic approach when

	Exercise 1				Exercise 2			Exercise 3	
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 1	Test 2
True positive	6	5	8	5	25	20	13	15	19
False positive	5	4	9	3	6	1	3	9	12
False negative	14	16	13	18	31	17	18	3	9
Precision	0.55	0.56	0.47	0.63	0.81	0.95	0.81	0.63	0.61
Recall	0.30	0.24	0.38	0.22	0.45	0.54	0.43	0.83	0.68

Table 5.7.: Evaluation results for the simple heuristic approach using the median of all sums of scores of all matches plus 20 as a threshold during variable identification. One trace of a sample solution was used for alignment in this experiment.

	Exercise 1				Exercise 2			Exercise 3	
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 1	Test 2
True positive	9	9	10	10	30	20	14	16	20
False positive	10	6	8	5	17	3	12	11	9
False negative	6	10	12	11	15	15	7	0	11
Precision	0.47	0.60	0.56	0.67	0.64	0.87	0.54	0.59	0.69
Recall	0.60	0.47	0.45	0.48	0.67	0.57	0.67	1.00	0.65

Table 5.8.: Evaluation results for the simple heuristic approach using the top 40% quantil of all sums of scores of all matches as a threshold during variable identification. One trace of a sample solution was used for alignment in this experiment.

using of the median of all sums of scores of all matches as a threshold during variable identification (table 5.6): Precision varies from 0.29 (which is a very poor value) to 0.77 (which is a rather fair value) with a mean of 0.51 and a standard deviation of 0.17. Recall shows one peak with a perfect 1.0 for test case 1 on exercise 3 and in the other cases varies from 0.25 (which is a poor value again) to 0.68 (which is a rather good value as well). The mean value including the peak is 0.58 with a standard deviation of 0.21. These figures prove the simple heuristic approach to be effective, but they also show that there is a lot of space for improvement. While a recall of about 50% can be considered acceptable, a precision of less than 50% is clearly not yet satisfying for an automated tutoring system. The numbers can also be used to calculate the probability for feedback content: With a mean precision $p = 0.51$ and a mean recall $r = 0.58$, students will receive a correct feedback with probability $p \times r = 0.29$. They will receive a wrong feedback with probability $(1-p) \times r = 0.28$ and they will receive no feedback at all with probability $1 - r = 0.42$. Thus four out of ten students will receive no feedback, while the others will receive either right or wrong feedback with almost the same chance. This is better than highlighting random lines in the trace, but clearly not satisfying. Instead, it is desirable to increase the precision, even if this causes the recall to drop even more.

It can be assumed that the precision can be raised by suppressing more unsure matches during variable identification by using a higher threshold. Table 5.7 provides results from a second experiment using the same sample data as above but a different threshold. Instead of using the median of all sums of scores of all matches, the same value plus 20 was used. Using a fixed value can be considered appropriate, because the match values will get larger the longer the traces are. Thus the influence of a fixed addition to the threshold on short traces will be larger than on long traces. This is reasonable, because confidence in matches for long traces is higher than for short traces. From the figures, the expected effect can be observed: The best precision reached is now 0.95 (which is almost perfect), and the worst is 0.47. The mean value raised to 0.67 with a standard deviation of 0.16. On the other hand, recall dropped to 0.83 for the best value and 0.22 in the worst case with a mean of 0.45 and a standard deviation of 0.2. These remarkable changes have some impact on the feedback probabilities: With a mean precision $p = 0.67$ and a mean recall $r = 0.45$, students will receive a correct feedback with probability $p \times r = 0.3$. They will receive a wrong feedback with probability $(1 - p) \times r = 0.15$ and they will receive no feedback at all with probability $1 - r = 0.55$. Hence more than half of the students will receive no feedback with this approach. In the case feedback is generated, the chance for getting correct feedback is twice as high as the risk of getting wrong feedback. From this perspective, the altered threshold has a clear beneficial effect compared to the probabilities achieved in the previous section.

As an alternative to using a fixed addition to a threshold, another approach was also explored that takes to top 40% quantil instead of the median value as a threshold. This approach pays attention to the fact that there will in general be more bad matches than good matches. Moreover it provides only a small change compared to the median value if the distribution of match values is smooth, while it provides a solid addition if the distribution shows a stronger incline. Table 5.8 shows the result for this approach. For this approach, precision is between 0.47 and 0.87 with a mean of 0.62 and a standard deviation of 0.11. These figures are slightly worse than the ones obtained with a fixed addition to the threshold. As in the original experiment, there is also a peak value for recall with 1.0 for test case 1 on exercise 3. For all other cases, recall varies from 0.45 to 0.67 with a mean value of 0.62 and standard deviation of 0.17. This is the best result for recall obtained so far. In summary, the effects on feedback probabilities are as follows: With a mean precision $p = 0.62$ and a mean recall $r = 0.62$, students will receive a correct feedback with probability $p \times r = 0.39$. They will receive a wrong feedback with probability $(1 - p) \times r = 0.23$ and they will receive no feedback at all with probability $1 - r = 0.38$. Hence this approach gives the largest amount of correct feedback (about four out of ten students), but also more wrong feedback than with using a fixed addition to the threshold.

So far it can be stated, that a threshold higher than the median of all sums of scores of all matches is beneficial, but there is a trade-off between precision and recall. While the existence of the trade-off is not surprising, the results regarding the threshold are true findings from the experiment.

It can also be noticed in the results and especially for the peak value observed on one of the cases that the test cases used to create the traces may have an impact on the performance of the approach. This seems to be obvious because different test cases may reveal different errors, where some of these are easier to spot by trace comparison than others. However, no possibilities to enhance trace alignment with knowledge about the errors to be spotted by a particular test case have been explored so far. Instead, the experiment has been extended to take more than one sample solution and thus more than one trace for comparison into account.

Table 5.9 shows the results for an experiment, in which again the median of all sums of scores of all matches plus 20 was used as a threshold. Unlike the other experiments so far, this

	Exercise 1				Exercise 2			Exercise 3	
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 1	Test 2
True positive	6	5	6	6	28	15	18	12	26
False positive	3	2	2	1	1	1	2	4	3
False negative	16	18	22	19	33	22	13	11	11
Precision	0.67	0.71	0.75	0.86	0.97	0.94	0.90	0.75	0.90
Recall	0.27	0.22	0.22	0.21	0.46	0.41	0.58	0.52	0.70

Table 5.9.: Evaluation results for the simple heuristic approach using the median of all sums of scores of all matches plus 20 as a threshold during variable identification. On exercise 1 and 2, three traces of different sample solutions were used for alignment. On exercise 3, four traces were used. Traces used for marker generation were selected manually, so results are theoretical.

time three to four sample traces for an exercise where used for comparison. No automated selection of these traces has been used for this experiment. Instead, each student's solution was compared to all traces for that test case and the best result was chosen. So if at least one alignment resulted in a correctly marked line, a true positive was recorded. If there was no correctly marked line, but at least one alignment did not produce any markers, a false negative was recorded. Only if all alignments provided erroneously marked lines, a false positive was recorded. The possibility to discard all alignments and mark no lines has not been used explicitly, as it would trivially allow to eliminate all false positives. Hence the results in table 5.9 represent the best possible result that can be obtained from applying the simple heuristic approach to several traces and selecting the most appropriate one.

It can clearly be seen that these results are better than any of the results obtained so far: Precision is between 0.67 and 0.97 with a mean value of 0.83 and a standard deviation of 0.11. Recall is in turn on the lowest level of all experiments, as it is between 0.21 and 0.70 with a mean value of 0.40 and standard deviation 0.18. Based on these results, the following feedback probabilities can be calculated: With a mean precision $p = 0.83$ and a mean recall $r = 0.40$, students will receive a correct feedback with probability $p \times r = 0.33$. They will receive a wrong feedback with probability $(1 - p) \times r = 0.07$ and they will receive no feedback at all with probability $1 - r = 0.60$. With respect to the risk of wrong feedback, these results are the best of all experiments, as less than one out of twelve students will receive wrong feedback. One out of three will receive correct feedback and all others will receive no feedback.

However, these results are theoretical as explained above, because as selection of the most appropriate trace was done manually. Thus another experiment was conducted, using the approach for trace selection described in section 5.1.4 on page 73, also using the threshold of 0.88 for the match quality proposed there. Results of this experiment are shown in table 5.10. It is easy to see, that the approach is not able to reach theoretical results: Precision is between 0.43 and 0.85 with a mean value of 0.67 and standard deviation 0.14. Recall drops to values between 0.12 and 0.57 with a mean value of 0.32 and standard deviation 0.16. Hence both the results for precision and recall are lower than in the theoretical results. In comparison to the results from table 5.4, it can be noticed that there is no gain in precision but a slight loss in recall. While this sounds not much like an improvement, feedback

	Exercise 1				Exercise 2			Exercise 3	
	Test 1	Test 2	Test 3	Test 4	Test 1	Test 2	Test 3	Test 1	Test 2
True positive	5	5	3	5	28	15	16	6	12
False positive	3	2	4	2	5	7	5	6	3
False negative	17	18	23	19	29	16	12	15	25
Precision	0.63	0.71	0.43	0.71	0.85	0.68	0.76	0.50	0.80
Recall	0.23	0.22	0.12	0.21	0.49	0.48	0.57	0.29	0.32

Table 5.10.: Evaluation results for the simple heuristic approach using the median of all sums of scores of all matches plus 20 as a threshold during variable identification. On exercise 1 and 2, three traces of different sample solutions were used for alignment. On exercise 3, four traces were used. Traces used for marker generation were selected automated with a threshold in match quality of 0.88.

	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5
Threshold for variable identification	median	median + 20	40% quntil	median + 20	median + 20
Trace selection	single sample trace	single sample trace	single sample trace	manual selection	threshold 0.88
Average precision	0.51	0.67	0.62	0.83	0.67
Average recall	0.58	0.45	0.62	0.40	0.32
Chance for correct feedback	0.29	0.30	0.39	0.33	0.22
Chance for wrong feedback	0.28	0.15	0.23	0.07	0.11
Chance for no feedback	0.42	0.55	0.38	0.60	0.68

Table 5.11.: Summary of experimental results for trace alignment.

probabilities provide some more information: With a mean precision $p = 0.67$ and a mean recall $r = 0.32$, students will receive a correct feedback with probability $p \times r = 0.22$. They will receive a wrong feedback with probability $(1 - p) \times r = 0.11$ and they will receive no feedback at all with probability $1 - r = 0.68$. In comparison to all results that were gained via full automation, this is the best value for the risk of wrong feedback. In turn it is also the lowest result for the chance for correct feedback, due to the low recall.

All experimental results are summarized in table 5.11 to get an overview about the achieved quality: It can clearly be seen that the changes in the threshold for variable identification have effects both on precision and recall and thus also influence feedback probabilities. It can also be seen that the final experiment showed the best results for precision and the risk of wrong feedback among all fully automated approaches. Only the manual approach to trace selection showed better results. Thus it can be concluded, that on the one hand the heuristic approaches are useful and effective, while on the other hands additional experiments are desirable to reach further improvements in quality.

5.1.8. Preliminary Conclusions

Based on this first use case for tracing and trace analysis, it can be concluded that this technique is indeed useful for creating additional feedback. In particular, two types of additional feedback are generated: First, white-box testing in general provides more feedback information in the form of traces than black-box testing approaches. This offers students the general possibility to examine their solutions in more detail, without providing detailed hints. Second, trace analysis generates feedback in the form of additional annotations to the traces or the source code produced by the students. This way, detailed hints can be produced to attend the general feedback and help students in their fault analysis. The implementation shows that both types of feedback can be generated effectively. In addition, the experimental results provide pointers for further research in this area.

5.2. Testing Diagrams

In contrast to programming languages, not every modeling language defines execution semantics for their diagrams. Obviously, dynamic checks can only be performed on diagrams and models that provide execution semantics. It can be assumed that these kinds of models provide at least a definition of control flow. They may also define data flow and thus be able to handle input and output similar to programs. In this case there can be test cases defined that provide a particular input and check for a particular output. If some execution semantics do not involve inputs, then there is just one test case with empty input. If some execution semantics do not involve output, then checks are just made based on the traces created during the execution of the test case. The latter is also true for models that provide just a definition of control flow.

For the purpose of this thesis, UML Activity Diagrams [107] are used as an example of diagrams that have execution semantics, but the techniques presented throughout this section are not limited to this kind of diagram. This section also focuses on approaches and techniques that differ from the ones used in the previous section. In particular, this section does not repeat the explanations made above regarding marker generation for traces. Nevertheless, this can be done for traces generated from models as well. Instead, more attention is given to parallelism in execution, which has not been covered in the previous section.

The semantics of UML Activity Diagrams are in general defined by token flows: Tokens represent objects, dates, or control and are passed from one activity to another via the edges in an activity diagram. The sequence of activities that accepted a token on its way from its creation to a final node can be considered as a trace. Forks may occur in an UML Activity Diagram that cause several tokens to be passed around in parallel. The resulting trace thus contains parallel sequences as well.

Besides passing a token directly, it can also be passed by explicit object flow via activity outputs and inputs or by explicit communication via activity calls. However, both action definitions in activities and objects in the object flow may be abstract. In particular, they do not need to make explicit statements about variable manipulation happening inside activities or about the variables describing the object's state at all. Hence semantics for UML Activity Diagrams do not involve output in the same way as programming languages do. Thus test results and feedback for dynamic checks cannot be generated based on output comparison for actual input values. Instead, traces have to be compared to find out whether two diagrams induce the same behaviour. Different to the case of programming languages discussed above, trace comparison is thus not an auxiliary means of additional feedback generation, but the primary source for feedback.

5.2.1. Generating Traces

As mentioned above, a trace for an UML Activity Diagram can be understood as a sequence of activities that accepted a token on its way from an initial node to a final node. Thus the generation of traces can be performed as a search for all paths from all initial nodes to all final nodes. However, UML Activity Diagrams allow to use forks and joins that can create parallel sections in a flow. If flows are to be analyzed as sequences, these parallel sections have to be linearized in some way. Thus the creation of traces consists of two steps:

1. For all initial nodes, generate all possible paths to final nodes.
2. For all paths, generate at least one linear sequence by removing parallelism.

Each of these steps is in general independent from the other. Hence, different approaches can be used in the first step to generate the paths without confusing the second step. In the same way, different approaches to linearization can be applied in the second step without depending on the algorithms used for path generation in the first step. Both steps will be explained in more detail in the following subsections. During these explanations, the term "flow" always refers to "control flow" unless stated differently.

Computation of paths

The computation of all paths defined by an UML Activity Diagram corresponds to the implementation of execution semantics for control flows. One possibility to do so is to generate directed graphs, that are initialized with one node representing an initial node of the activity diagram. A stack is also maintained and initialized with the same node as the only element. If there is more than one initial node in the activity diagram, each one is represented by its own graph and stack.

Starting from that setup, and iterative algorithm can be applied: As long as there are nodes on the stack, one of them is removed and used as the current node in this iteration. The element in the activity diagram represented by this node is then examined for outgoing control flow edges. New nodes in the graph are created for each diagram element reached by a control flow edge. Each of these nodes is also placed on the stack. Depending on the type of the current element, the new nodes are connected to the current node in two different ways: If the current element is a fork node, the new nodes are connected as parallel successors. If the current element is a decision node, the new nodes are connected as alternate successors.

Special attention has to be paid to so-called interruptible regions. In an interruptible region, the normal control flow can be stopped and an interrupt event can be fired instead. Each interruptible region defines an action that accepts this event and defines a proper continuation of the control flow. So for all nodes in an interruptible region, an additional alternate successor has to be created, connecting the current node with a node for the action that handles the interrupt event.

For performance optimizations, a lookup table for each graph can be maintained that stores all elements that have been examined. If an element is reached more than one time, no additional graph nodes have to be created, but the already created node can be retrieved from the lookup table. Thus only connections to the already existing node have to be added into the graph.

Computation of sequences

Each of the directed graphs created in the previous step represents all possible execution paths starting from one initial node in the activity diagram. The idea of the next step is to represent all these paths as linear sequences of strings, where each string is a representation of the corresponding diagram element. In some cases, these representation will be the element name or label, while for unnamed elements generic representations are used. The computation of sequences has to cope with two special challenges: First, the graph is not necessarily acyclic, thus loops can induce potentially infinite sequences. Second, parallel path segments have to be linearized by defining some interleaving for them.

Infinite sequences can be avoided by checking whether an element is already part of the sequence before adding it to its end. If an element is already contained in the sequence, it may be added at the end as an indicator of the beginning loop, but sequence generation has to stop at this element.

Linearization of parallel path segments can in general follow several strategies for interleavings. The most complex one is to compute all possible interleavings between two parallel paths. This results in $\binom{i+j}{i}$ sequences for input sequences of i and j, but this is an unnecessary overhead both for generation and computation of comparisons: If two different paths contain the same parallel sequence, all interleavings generated for one path will match an interleaving generated for the other path. But if the two paths differ, e.g. by having elements in different order or by an extra element in one of the paths, none of the interleavings for one path will find a perfect match in the interleavings of the other path. Hence it is theoretically sufficient to calculate just one deterministic interleaving by concatenating the two parallel path segments. However, there is no natural order in which the segments should be handled. Ordering by length fails in cases where both segment are of the same length and may moreover produce completely different sequences for paths that differ in just one or two elements in the parallel section. Using an alphabetical order of element names is also not suitable, because these may be subject to marginal differences, that should not result in completely different sequences.

A mixture of these two extreme options is to calculate all possible interleavings for one path, which only needs to be done once if this path represents the sample solution of an exercise. For a path compared to this one, just one possible linearization is then sufficient. However, if the sample solution is long and thus produces many possible interleavings, this option still results in a large number of comparisons to be made.

Hence another approach can be used to reduce the number of comparisons: Instead of computing all interleavings, only concatenations will be computed. This way, the number of resulting sequences is independent from the length of the sequences and lower than the number of possible interleavings. Again it is sufficient to create all possible concatenations for just one of the two paths to be compared, while for the other just one possible concatenation is created.

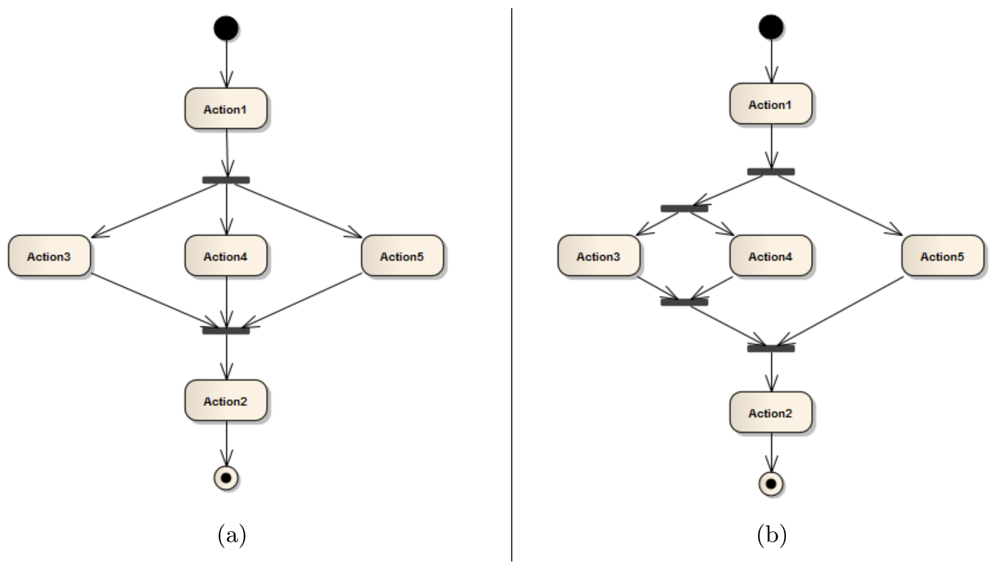


Figure 5.4.: Two different ways of modeling parallelism of three actions in an UML Activity Diagram

See figure 5.4 for two diagrams modelling parallel actions. In both cases, actions 3 to 5 are modelled to be executed in parallel. For the version in figure 5.4(a), the list of linear sequences is as follows:

```
* -> Action1 -> n/a -> Action3 -> -> Action5 -> -> Action4 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action3 -> -> Action4 -> -> Action5 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action5 -> -> Action3 -> -> Action4 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action5 -> -> Action4 -> -> Action3 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action4 -> -> Action3 -> -> Action5 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action4 -> -> Action5 -> -> Action3 -> n/a -> Action2 -> #
```

As can be seen, all six possible permutations for the three actions have been calculated. Each of them has action 1 as prefix and action 2 as suffix. Element `n/a` denotes unnamed and unlabeled elements, i.e. the fork and join nodes in this case. Initial and final node are indicated by `*` and `#`, respectively.

For the modeling version in figure 5.4(b), the list of linear sequences is as follows:

```
* -> Action1 -> n/a -> Action5 -> -> n/a -> Action3 -> -> Action4 -> n/a -> n/a -> Action2 -> #
* -> Action1 -> n/a -> n/a -> Action3 -> -> Action4 -> n/a -> -> Action5 -> n/a -> Action2 -> #
* -> Action1 -> n/a -> Action5 -> -> n/a -> Action4 -> -> Action3 -> n/a -> n/a -> Action2 -> #
* -> Action1 -> n/a -> n/a -> Action4 -> -> Action3 -> n/a -> -> Action5 -> n/a -> Action2 -> #
```

The linearization created two possible sequences for parallel actions 3 and 4, where each of these can either be proceeded or succeeded by action 5. An interleaving where action 5 is executed in between action 3 and 4 is not created. Thus there are just four different permutations instead of the six ones above.

It can be considered useful to keep information on whether a sequence was generated by linearizing parallel segments or not. This can be used in trace analysis later on, if there is a fair match between two sequences, but one was created involving parallel segments, while the other one did not involve parallelism. This may happen if an interleaving was modelled directly in the activity diagram.

5.2.2. Analyzing Traces

With traces represented as linear string sequences, trace analysis can happen straightforward using the same algorithms for trace alignment as in the previous section for programming exercises. Nevertheless the environment is different: Each linear sequence from a sample solution needs to be aligned with all sequences generated from all paths of the student's solution. As a result, the score for the best match can be obtained and used for further analysis. This has to be set into relation to the overall size of the sequence to gain information on whether a perfect match was achieved or just a partial match. For partial matches, additional information like the length of the longest sub-sequence that achieves a perfect match or a list of elements that could not be matched can be retrieved.

Most important for the quality of the alignment is the scoring function. A score of 1 is added to the overall score if two elements are equal by their name or if both are unnamed or unlabeled elements. Matching is not type sensitive, so this also applies to cases in which one of the elements as a named node while the other one is a labeled edge. Moreover, a score of 1 is added if the string distance in the element names is less than a certain threshold. The string distance can in turn be calculated by aligning both strings, where each mismatch or gap adds 1 to the distance. The threshold should be defined in terms of the length of the compared strings. A mismatch of named elements is punished with a score of -1, while a mismatch of edge labels does not change the score. The gap penalty in the sequence alignment is -1 in general, but 0 if a gap is assigned to an unnamed or unlabeled element.

The actual feedback presented to the student and the grades given for the solution can be based on the received scores. Intervals of scores can be defined for different feedback messages and grades. Moreover, additional information as named above can be used for further feedback generation. Other techniques for trace analysis like checking temporal assertions on the traces as discussed in section 5.1.3 can also be applied to the traces as well to generate more feedback. They can either be checked during the trace generation phase and thus at "run time" or as a subsequent analysis step on the generated sequences.

5.2.3. Component `DynamicUMLChecker`

The component `DynamicUMLChecker` implements trace generation and alignment for control flows from UML Activity Diagrams. It consists of the main class `DynamicUMLChecker` that in turn creates instances of class `ActivityPathGenerator` for generation of paths and sequences and class `PathAlignment` for computing the trace alignment. The component also contains two classes `ActivityPath` and `ActivityPathElement` for the graph structures created during path generation. For parsing the input files, it reuses the parser implementation from component `GReQLUMLChecker` as presented in section 4.3.3. See figure 5.5 for an illustration of the interaction between these classes.

Class `DynamicUMLChecker` implements the `IChecker` interface and thus provides a `doCheck` method. Upon invocation of this method, the XMI file containing the student's solution, arbitrary many files containing samples sequences as serialized Java objects, and a XML

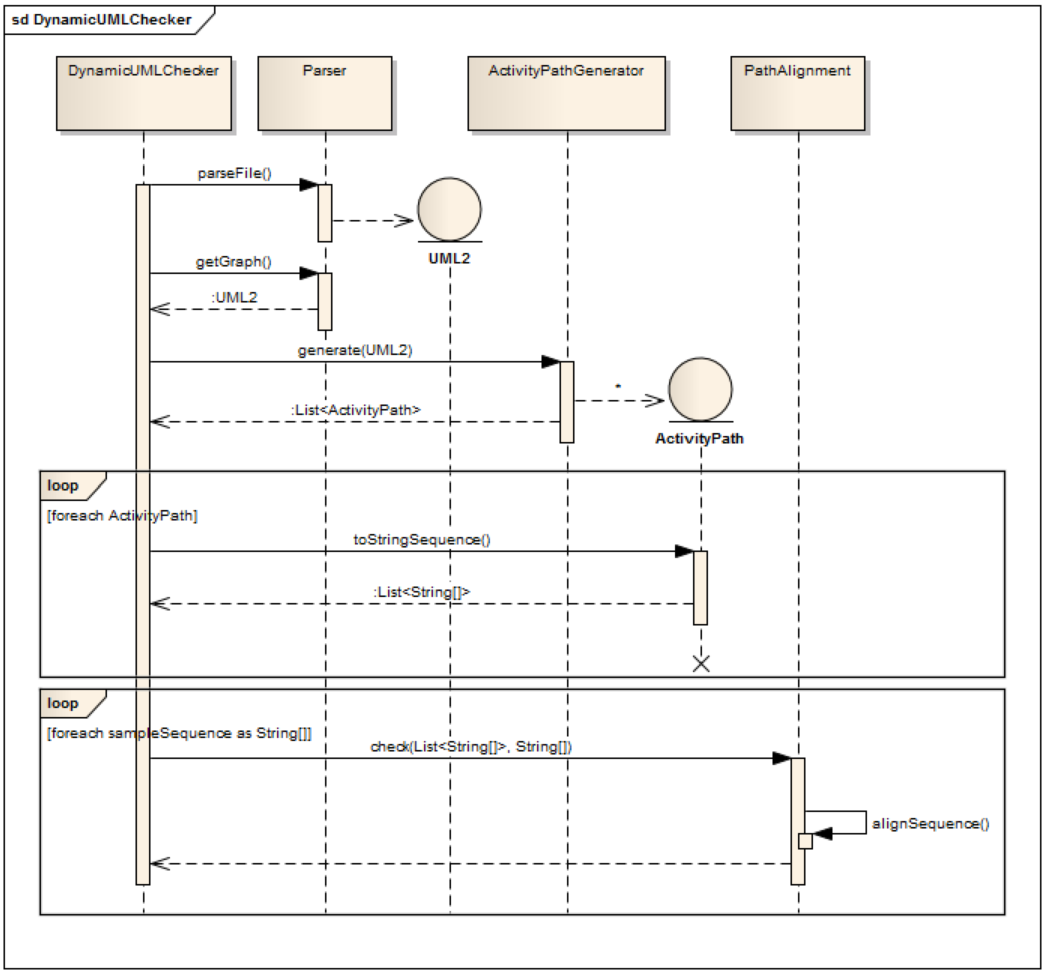


Figure 5.5.: Sequence diagram illustrating the sequence of calls performed when using the `DynamicUMLChecker`

file containing the feedback settings are retrieved from the data provided by the server. In the first step, the XMI file is passed to the parser and an instance of a TGraph for UML2 is returned. In the second step, this graph is handed over to an instance of class `ActivityPathGenerator`. This class implements the algorithm for path generation as presented in section 5.2.1 in a straightforward manner. It iterates over all existing nodes of type `InitialNode`, maintains a stack and a lookup table for each of them and creates and returns an instance of class `ActivityPath` for each of them.

Afterwards, a loop is started as a third step that iterates over all paths created in the previous step. On each of these paths, the method `toStringSequences()` is called in order to obtain a list of string sequences created out of this path. The algorithm used here is a recursive implementation of the ideas sketched in section 5.2.1. Each sequence is represented by an array of type string. If the path does not involve parallelism, this array contains only one element. Sequence generation happens recursively by calling method `toSequence` on the root element of the path. This method inspects the successors of the current element and calls itself recursively on these elements. It maintains a list of elements already visited and creates the actual sequence starting from its end. To do so, it distinguishes the following cases:

1. The current element has been visited before. In this case, a new sequence with the current element as the only element is created and returned.
2. The current element has no successor. In this case, a new sequence with the current element as the only element is created and returned as well.
3. The current element has just alternate successors. In this case, it iterates over all these successors. Again, there are two cases to distinguish:
   a) If the successor represents a join of paths (by being a join node or the event handler for an interruptible region) a new sequence with the current element as only element is created and added to a temporary set of sequences.
   b) Otherwise, the method calls itself on the successor, adds the current element as prefix to all sequences returned by this call and adds all of them to the temporary set of sequences.

   Finally, the temporary set of sequences is returned after all successors are processed.
4. The current element has just parallel successors. In this case, it iterates over all these successors. As above, two cases have to be distinguished that are handled slightly different:
   a) If the successor represents a join of paths a new sequence with the current element as only element is created and added to a temporary set of sequences as above.
   b) Otherwise, the method calls itself on the successor and adds all sequences returned by this call to the temporary set of sequences as well.

   Afterwards, a second iteration is started on this temporary set. If a sequence in this set ends because the next element in this sequence would be a join of paths, it is stored for further processing. A sequence fulfilling this condition is a sequence created from case 3a) or 4a) above. Otherwise, it is prefixed with the current element and stored in another temporary set for completed sequences. For the sequences stored for further processing, permutations of all sequences ending with the same join of paths are computed. The method call itself recursively on the join element, adds all permutations as prefixes to the sequences returned by that call and stores the resulting new sequences in the set for completed sequences. This set is finally returned after all sequences have been processed.

Note that the case of having both alternate and parallel successors cannot occur because of the way the paths are constructed. The implementation is limited in that way that it cannot handle implicit joins of paths without join nodes. This is because implicit joins by having more than one incoming edge to an activity node are ambiguous, as they can either be an implicit join of parallel paths or an implicit merge of alternate paths.

After the generation of sequences, another loop is started as a fourth step that iterates over all sample sequences provided as serialized Java objects. Each of them is deserialized and handed over to an instance of class `PathAlignment` by calling method `check()`. The list of sequences created in the previous step is also passed in the same method call. The method implements the ideas for sequence alignment sketched in section 5.2.2 in a straightforward manner. The code is provided in appendix B.3. As an heuristic threshold for string distances, the mean value of the length of two strings divided by 3 is used based on experiments.

Although the core algorithm used for alignment does not differ much from the one used for trace alignment on program traces, the interpretation of the results is different: The score obtained is divided by the maximum of the length of the sample sequence and the length of the student's sequence to obtain a match ratio. Since a matching element receives a score of 1, this means that a perfect match receives a match ratio of 1.0 after the division. Sequences with worse matches achieve lower scores and thus lower match ratios. Since all sequences generated by the solution are compared to the sample trace, the best result is used as return value.

The last step is thus the final interpretation of this return value for feedback generation. For this purpose, the XML file containing feedback settings is read. In this file, the author of an exercise can define which feedback and grade should be used for which match ratio.

5.2.4. Example: Diagram Checks

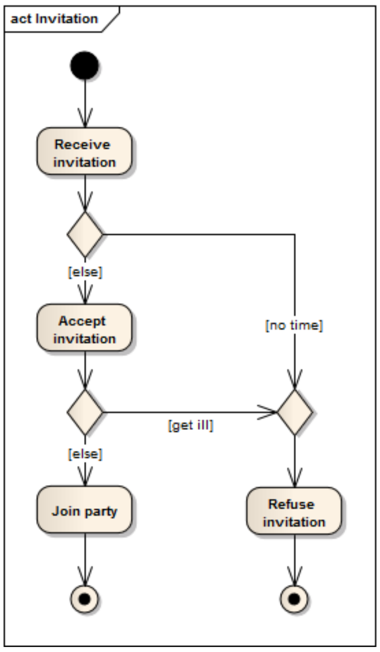
This section illustrates diagram checks via sequence alignment by an example. As in section 4.4.2, the example is taken from a bachelor degree course on UML modeling held at the University of Duisburg-Essen in winter term 2010/2011. The example has been translated to English and slightly amended to be ready for use in this thesis.

The task was to model activities that are likely to happen when one receives an invitation for a party. In particular, the students were asked to consider three different cases: (1) Accepting the invitation and joining the party, (2) Accepting the invitation but not joining the party because of illness, and (3) Refusing the invitation at all because of lack of time.

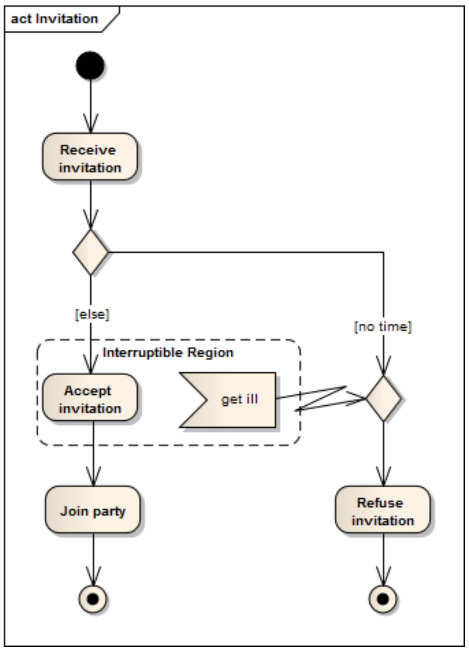
See figure 5.6(a) for a sample solution for this exercise. The solution chains the different activities in a simple way without using complex features. From this solution, three sequences can be derived:

```
* -> Receive invitation -> n/a -[no time]-> n/a -> Refuse invitation -> #
```

```
* -> Receive invitation -> n/a -[else]-> Accept invitation -> n/a -[else]-> Join party -> #
```



(a) Sample solution



(b) Alternate solution with interruptible region

Figure 5.6.: Two different ways of solving the modeling task discussed in section 5.2.4

```
* -> Receive invitation -> n/a -[else]-> Accept invitation -> n/a -[get ill]-> n/a -> Refuse invitation
-> #
```

Part of the criticism on the current state of the art as discussed in chapter 2 of this thesis was that the generation of different possible sample solutions might be to expensive. Consequently, it has to be considered to create sample sequences as the ones shown above by hand. For the approach it does not matter how the sample sequence was generated, so they could indeed be derived directly from the three aspects of the task description, as each of them covers exactly one of the three cases.

A student's solution that solves the exercise correctly is shown in figure 5.6(b). It uses an interruptible region and thus a more complex feature of UML Activity Diagrams. There are also three sequences that can be derived from this solution:

```
* -> Receive invitation -> n/a -[no time]-> n/a -> Refuse invitation -> #
* -> Receive invitation -> n/a -[else]-> Accept invitation -> Join party -> #
* -> Receive invitation -> n/a -[else]-> Accept invitation get ill -> n/a -> Refuse invitation -> #
```

Consequently, nine sequence alignment must be computed to find out all matches between sample solution and alternate solution. Obviously, for the first sequence generated by the sample solution, there is a perfect match in the alternate solution, since its first sequence is the same. It thus receives a score of 11 and a match ratio of 1.0. This is not surprising, since the related parts of the model are the same in both solutions.

For the second sample sequence, the best match scores 10 on the second sequence of the alternate solution. This happens because there are 11 matching elements, but the sample solution has two extra elements (one being the unnamed decision node and one the [else]-edge). While the unnamed one is mapped to a gap without changing the score, the labeled edge reduces the score by 1. This results in a total score of 10 and a match ratio of about 0.77. The first sequence from the alternate solution scores 4 (ratio 0.31) and the third one 8 (ratio 0.57). Thus there is no perfect match. However, a match ratio of more than 0.75 is considered sufficient in this exercise, so no negative feedback is given to the students.

For the third sample sequence, there is a match that scores 13 with the third sequence of the alternate solution. This happens because there are 13 matching elements, while all additional elements are unnamed or unlabeled and can thus be mapped to gaps without decreasing the score. Most notably, this case takes benefit from the fact that matching is not type sensitive, because this way the action accepting the interrupt event in the alternate solution can be mapped to the edge label from the sample solution. Without this feature, the score would be just 11. Considering the length of the sequence, the resulting match ratio is about 0.87 for score 13 and 0.73 for score 11. This is again no perfect match but close enough to the sample solution to give no negative feedback. The other two sequences score 8 (ratio 0.53) in this case.

Considering solutions that do not contain structural differences, but different edge labels, result in different scores depending on the amount of change. For example, a solution that is similar to the sample solution, but misses the two “else” labels on the edges receives a match ratio of about 0.85 for the second sequence and a match ratio of about 0.93 for the third sequence. This is reasonable, since the solutions are still pretty close to each other. Most notably, sample sequences may miss these labels as well, if they are created by hand and not derived from a complete sample solution. In any case, enforcing modeling guidelines that require to have at least on outgoing edge labeled with “else” on each decision node or to have no unlabeled edges on them would be a typical use case for static checks and is thus not covered by the dynamic ones.

Figure 5.7 shows another possible solution to the same exercise. Compared to the sample solution, there are additional elements (one additional action and a third final node), missing elements (no merge node), and different wording (“reject invitation” instead of “refuse invitation”). There are also three sequences that can be derived from this solution:

```
* -> Receive invitation -> n/a -[no time]-> Reject invitation -> #
* -> Receive invitation -> n/a -[else]-> Accept invitation -> n/a -[else]-> Join party -> #
* -> Receive invitation -> n/a -[else]-> Accept invitation -> n/a -[get ill]-> Cancel acceptance -> #
```

The first of these sequences is still the best to be matched with the sample solution, but it just receives a score of 9 instead of 11. Most notably, the different wording on the last named element in this sequence is not responsible for this, because the string distance to the original spelling is still low enough. Instead, the merge node and one edge that are present in the sample sequence have to be mapped to gaps, which reduces the score by 2 compared to the example above.

For the second sequence of the sample solution, there is a perfect match with the second sequence shown above. This is not surprising, because the solutions are identical with respect to this case.

For the third sequence from the sample solution, a match scoring 11 can be found with the third sequence from above. It contains 12 matching elements and one mismatch because

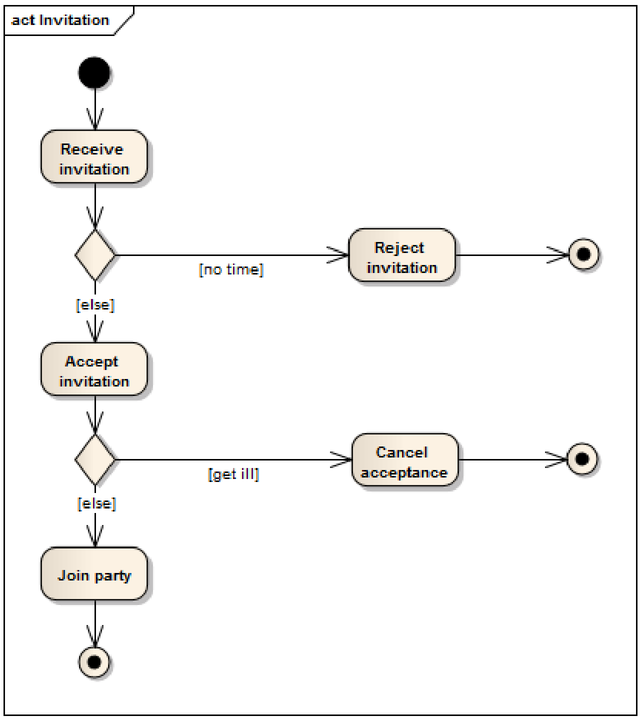


Figure 5.7.: Another possible solution for the modeling task discussed in section 5.2.4

the string distance between “Cancel acceptance” and “Refuse invitation” is too large. Hence the score is reduced by 1. The missing merge node and one edge are mapped to gaps as for the first sequence. The resulting match ratio is about 0.73 and may thus trigger some feedback to the student.

5.2.5. Preliminary Conclusions

The focus of this second use case for tracing was different than for the first one: Instead of generating additional feedback for executed programs, tracing and trace analysis was used on UML diagrams to provide feedback based on execution semantics at all. This proved this technique to be not only useful to extend existing feedback, but also to be used as a basic concept of feedback generation for the artefact type of UML diagrams.

5.3. Open Problems

The work presented in this chapter of the thesis presents some novel approach to the use of trace generation and analysis which can hardly be considered fully mature. Some open problems have already been discussed in the previous sections in conjunction with the quality of the checking results, while some more will be discussed in this section.

5.3.1. Generation and Identification of Sample Traces

If using trace alignment, the quality of checking results depends to some extent on the form of the sample traces as discussed above. Thus better results can be achieved, if more sample traces for various correct solutions can be provided. However, generation of these sample solutions can be a tedious task that cannot be automated in the general case. One possible solution would be to reuse traces from correct student's solutions as sample traces in subsequent checks on new submissions. While this would indeed reduce the amount of work needed to provide a large number of sample traces, it raises a different issue: In order to avoid useless comparisons, traces that are similar to already existing traces need to be filtered out and not to be added to the set of sample traces.

A second issue is how to improve the results in finding the appropriate trace to compare to, if several traces for the same test case exist. As discussed, this can partly be done based on static checks, e.g. by making distinctions between iterative and recursive approaches. However, more subtle details, that are not that easy to catch may be relevant as well. One approach to tackle this problem could be to use more general static analysis methods known from plagiarism detection to find correct solutions that are most similar to the one being checked, and use their traces for comparison.

5.3.2. Implementation of Execution Semantics

As mentioned in the introduction of this chapter, for some models different execution semantics have been defined and formalized. Consequently, it may happen that two models may create the same traces under one particular execution semantic, but different traces when considering some other execution semantics. This is no drawback for the checking process itself, but it limits the reusability of existing exercises, because checks and feedback defined for them may be limited to one particular execution semantic.

Moreover, some types of models may distinguish between different parts of the execution semantics as UML Activity Diagrams do with control flow and object flow. The techniques presented in this chapter are designed to compare two traces with each other. If control flows and object flows are to be analyzed in conjunction, these techniques have to be extended to compare two sets of interrelated traces.

5.3.3. Analysis of Multi-Threaded Program Executions

Similar to parallel sections in UML Activity Diagrams, programming languages may define parallel execution of instructions like Java does in multi-threaded programs. However, the trace generation approach for programming exercises discussed in this chapter of the thesis is only concerned with the trace generation for a single-threaded execution path. Consequently, extensions have to be made to cover multi-threaded executions, which can happen based on truly parallel execution or based on interleavings.

With truly parallel execution, one execution will produce not one single trace but a set of traces, containing one trace for each parallel thread. These threads can be compared to the corresponding threads in the sample solution independent from each other. If there are no non-deterministic interactions between threads and also forks and joins of parallel executions happen deterministically, no changes are necessary to the approaches presented above. The only thing to add is an additional step at the beginning of the process that tries to identify corresponding threads in the student's solution and the sample solution. However, if there are non-deterministic interactions between threads, it cannot be guaranteed that they happen at the same execution step and thus deviations between a student's solution and a sample solution may happen just by random. The consequence of this is twofold: First, a serious deviation may be noticed in one trace, although there is no error in the corresponding part of the program or model and thus no error should be marked in it. Second, an error may be seeded in on thread, but its effect may be visible in the trace of another thread. In this cases, lines must be marked even without spotting deviations in the corresponding trace. Both cases thus require extensions that have not been explored in this thesis.

With interleavings, the execution model reduces parallel execution to a single trace by interleaving steps from different parallel threads. Thus, even for multi-threaded executions there is a single final trace that can be used in trace alignment. This is the approach used above for UML Activity Diagrams. If the interleaving is calculated based on a deterministic scheduling (and no non-deterministic interactions occur as above), trace comparison can be applied without further additions. In all other cases, differences between traces may happen by random again, which makes error detection more complex as well.

5.3.4. Checking of Complex Temporal Constraints

The approach to assertion checking discussed in this thesis and also the approach used by Java Pathfinder is not sufficient to check complex temporal constraints at run time: Run time checks on constraints that are no simple comparisons between an actual variable value and an expected value can be performed only for particular points in time by explicit calls in the test driver, thus without any temporal capability. Run time checks by simple comparisons can be performed after each step if requested, thus realizing some kind of "always"-operator known from temporal logic. This in turn implies that no checks of type "eventually" or "nexttime" as known from temporal logic as well can be performed at run time.

It can be considered both possible and beneficial to implement checks of these kind as part of the post mortem analysis of traces. This can work similar to the query based search mentioned for the tool JIVE above. Obviously, in these cases "nexttime" can only be used in conjunction with an implication, where finding some property on one state must imply some other property for the next state. In addition, "nexttime" could be implemented as run time check, where finding one property on the current step triggers the check of another property on the next step.

5.3.5. Analysis of String Contents

As already mentioned for the static UML checker in the previous chapter in section 4.3.3 and shown by example in section 4.4.2, natural language processing is a general issue in automated analysis of artefacts. It becomes even more important in trace alignment: In program traces, variables are aligned by their content and thus in string variables differences may be spotted that are in reality just a spelling or typing error. In model traces, names given in natural language are the main elements used in the alignment and no supporting information based on syntactical relations between elements is available. While measures for similarity have already been introduced in the approach discussed above, more sophisticated methods like dictionaries require additional implementation work. They may also have a remarkable impact on the performance of trace alignment for model traces. However, experiments in this direction are beyond the scope of this thesis.

But even for more structured string content than natural language additional processing can be considered useful. For example, a string may contain a logical expression. A sample solution may suggest `A && B` as content for that string, but a student's solution may contain `B && A` instead. Since this is clearly a different string, it will reduce the match score. However, the logical content of the string is the same and thus proceeding with full score is more appropriate.

5.4. Chapter Summary

This chapter contributes to the goal of the thesis in the dimension of techniques by discussing a novel approach to dynamic analysis based on trace generation and trace alignment via sequence alignment algorithms. A contribution to the goal of the thesis in the dimension of systems is made by two implementations of this technique that can be used to extend an existing system. For both implementations it has been discussed and shown by example how to generate feedback with this techniques on different types of software artefacts. With respect to the dimension of inputs, using sample traces which can be generated automatically from sample solutions is well inside the borders of current research. As for static checks, the thesis does thus not provide significant progress in this area, but also does not fall back behind the current state of research.

Thus this chapter also contributes to the goal of the thesis in the dimension of feedback. In particular, the dimension of techniques has been focused by performing experiments with the proposed algorithms for sequence alignment on program traces. It could be shown that the novel approach developed in this thesis is able to generate detailed feedback and thus offers more support to the students than current approaches based on black-box-tests. Moreover, several additional options have been discussed how actual feedback can be generated based on traces besides the comparison to a sample solution. As in the previous chapter, it has also been shown that trace analysis is a technique that is not limited to a particular type of artefact but can be used for different kinds of artefacts as long as execution semantics are defined for them. In this case, trace generation and analysis is not necessary an additional means of feedback generation, but can be used as the primary means for feedback based on execution semantics. This is a new application of trace analysis that has not been explored in e-assessment systems before.

6. Additional Contributions

Although this thesis focuses on the automated analysis of software artefacts for direct feedback generation, some of the techniques discussed in the previous chapters are also useful for more indirect feedback, or in other contexts of e-learning and e-assessment. Thus this chapters discusses one additional use case for trace generation and one for trace comparison. It also discusses additional information that can be gained by static analysis techniques. The selection of these use cases does not rely on specific criteria, but reflects current interests and needs of students, teachers and researchers. The aim of this selection is to point out that the techniques discussed so far are not closed, but can serve as a basis for more advance feedback generation techniques or in other related fields. In particular, each section in this chapter will end by naming connecting ideas that related the specific additional contribution to the main contributions of the thesis.

The work on these cases was done in parallel to the research on the core parts of this thesis and was partially conducted by students as part of their seminar, bachelor or master theses.

6.1. Visualization of Object Structures

In chapter 5, dynamic analysis of Java Programs was performed by recording traces of program states. While this is a technique not limited to Java in particular, it does not pay attention to specific features of Java or object-oriented programming in general. Nevertheless, many introductory courses to computer science combine teaching basic programming skills in object-oriented programming languages with basic knowledge about algorithms and data structures. This is typically supported in lectures not only by presenting appropriate examples of program code, but also by presenting graphical representations of typical data structures such as lists or trees. Thus it is alluding to use similar visualization as a means of feedback to solutions submitted to exercises. These visualizations of each individual solution would be an important link between contents of the course and feedback in exercises. Moreover, they can help students in debugging their program code, because an image of the data structure may give them some useful information without any need for inserting debug statements into their program code.

In general, two different approaches exist for creating and using visualizations of data structures in the classroom: The use of visual debuggers, either as standalone tools [55] or integrated into an development environment, or the use of development environments specialized on teaching and visualization [87, 164]. Both approaches are reasonable to some extent, but also come with major drawbacks.

First of all, all approaches require the students themselves to get active in order to generate visualizations for their exercise solutions. In an optimal case, students are motivated and talented in using those tools and consequently get some benefit from using them. In an average case one can expect at least some students having problems in using these tools, resulting in different and possibly misleading visualizations even for similar solutions. In the worst case, this can frustrate any learning success if the visualization does not show the expected result since students cannot distinguish whether they have written wrong program code or have used the tool in a wrong way.

The second problem is complexity. On the one hand, professional visual debugging tools offer many options that are not necessary in introductory courses, making these tools confusing to first year students. Readership skills in interpreting diagrams are known to be an important problem [109] and thus visualizations for exercises should be as close to the visualizations used in the respective lectures as possible. On the other hand, development environments specialized for teaching may be too limited for some kind of exercises, narrowing the teacher in designing exercises for the course. The same applies to students, who may be limited in creating creative solutions.

However, with the data recorded for tracing and trace alignment, it seems possible to create visualizations of object structures and present them to students directly without need for any additional interaction from the student's side.

6.1.1. Concepts of Visualization

Three main aspects have to be taken into account when selecting techniques suitable for visualizing data structures in an e-learning system: First, general layout and design questions have to be answered. For example, data structures like trees or lists already hint towards layout constraints regarding the order of elements in an image. Second, the use case of an e-learning scenario requires special features like displaying objects that are missing in a data structure because they have been deleted unintentionally. Third, visualizations of data structures that are used in the context of algorithms produce sequences of changed structures over time. Thus displaying these changes in an appropriate manner is an additional requirement. The following subsections elaborate more on each of these three aspects.

Data to be Visualized

Data structures in object-oriented programming languages can in general be considered as attributed, directed graphs. Thus, each node of a graph represents an object, while each edge represents a reference from one object to another. In particular, class attributes of primitive types can be represented as attributes of a node, while class attributes of a non-primitive type are realized by an edge between the node representing the object and the node representing the attribute's value object.

While this basic idea covers the core principles of object-oriented programming, it ignores concepts like local variables that are not associated with any object but may nevertheless be crucial for the understanding of an algorithm. Thus it is in general not sufficient to visualize just the actual objects as nodes of a graph, but also to include some special node with attributes for the local variables. It has to be a node indeed, as a local variable may be an object, thus requiring an edge between the special node and the node representing this object. Once in place, the special node can also contain additional information, e.g. a visualization of the call stack of the running program.

General Considerations on Layout

Since graphs as a general data structure do not make any implications regarding a layout, the semantics of the actual data structure to be displayed should be taken into account. So wherever possible, the layout should reflect properties of the data structure. Two main data structures have been considered at first: lists and trees. In both cases it is necessary to keep a direction of reading, which can be achieved by using algorithms for drawing layered graphs [132, 86]. This way the entry point of a data structure, which is the "head" element in case of lists and the "root" element in case of trees, is displayed as the top-most object, while all subsequent objects follow below in the same order as in the data structure. Decisions have to be made if data structures are combined, e.g. a tree structure where each element is the head of a list. In this case, the direction of reading for the first data structure can be vertical, while the elements of the second data structure are arranged in horizontal direction. It is important to notice in this context that arrays are not that easy to display for several reasons: First, arrays may be of primitive type, so their contents are not objects at all and thus cannot be displayed by an approach for object visualization. Second, visualization approaches for 2-dimensional visualizations will naturally fail on arrays of dimension 3 or higher.

Since the visualizations are intended to be used in an e-learning system running as an online service on the web, some additional requirements regarding the display environment have to be considered. In particular, space is limited by screen sizes and resolutions and bandwidth may be limited when many students are accessing one server at the same time. So the layout algorithms should be able to create compact arrangements of objects without wasting much space, resulting in images of small file size that are readable on a screen with minimal scrolling.

Sequences of Visualizations

As already mentioned above, programming courses usually do not just teach a particular programming language and the static parts of data structures, but they also deal with algorithms and thus dynamic manipulation of object structures. Consequently, feedback to solutions on an exercise cannot only contain single static visualizations, but a sequence of static visualizations. This can simply be achieved by taking snapshots of the program state several times during run time. If program behaviour is recorded in traces anyway, this means to create visualizations for each step in the trace or some appropriate subset of states. Despite this simplicity, this raises some additional requirements for the layout of each visualization.

The most important concept in this context is the "mental map" [39] of the student using this visualizations. According to this concept, changes between visualizations should be as minimal as possible. Thus it is not acceptable that objects displayed in one visualization are changing positions or colors in the subsequent one without changes in the relations between them. Even if no changes in the data structure occur it is hard to identify where an object has gone if it has different positions in subsequent visualizations. Consequently, the layout algorithms used in this context need to look ahead with respect to changes in the data structure in order to minimize confusion for the students. In a simple solution to this problem, each object has only one fixed position in all visualizations generated for a program.

This is well suited to preserve the mental map of the student, but may be confusing with respect to the behavior of the implemented algorithm. For example, an exercise may be to realize sorting of elements in a list. In this case, it would be much more helpful if objects in the visualization change place step by step until the algorithm terminates, instead of keeping each in a fixed position and just changing pointers between them.

Note that the requirements for sequences of visualizations may stand in contrast to our general requirements. For example, generating a compact arrangement of objects in order to reduce the size of the visualization would imply to move objects closely together and close gaps between them. However, being foresighted with regards to object positions would imply to reduce this movement and plan with gaps in early visualizations that are filled by objects that appear later on in subsequent visualizations.

Colors in Visualizations

Beside the general layout constraints discussed above, visualizations can also use colors to make them more readable and comprehensible for students. At least three possibilities exist how coloring can be used in visualizations of object structures:

First, colors can be assigned to each object class, thus coloring all objects of the same type in the same color, making it easy to distinguish them from objects of other types. This way of coloring is useful in situations, where many different classes are in action and using objects of the wrong type is expected to be a frequent fault made by students. This way of coloring is of little use if only one or two different classes are in action, e.g. in a list data structure with elements of the same type, where wrong references between objects are expected to be the most common fault. In these cases, the second method of coloring may be beneficial: The coloring of class attributes of non-primitive types and the corresponding edges in the graph. For example, in a binary tree, the left child of a node could always be colored yellow, while the right child is always colored blue.

A third way of using colors is to mark objects that have been removed from a data structure, e.g. by displaying them half-transparent or surrounded with a red border. This way, even a static visualization can give information about changes that happened to the data structure. Furthermore, it helps students to distinguish between a situation in which an object is missing because it has never been created and a situation in which an object is missing because it has been deleted. Being able to distinguish between these situations is assumed to be a crucial point in understanding behaviour of algorithms acting on data structures.

6.1.2. Component `KovidaChecker`

The component `KovidaChecker` realizes most of the concepts discussed so far as a component for the e-assessment system JACK that is able to visualize object structures from exercises on Java programming. It has initially been implemented by Mobasher Ullah in his bachelor thesis. The original component did not provide visualizations for local variables, but a solution to this and various additional features have been implemented later in a project thesis by Christian Heiming.

The basic architecture of the component splits the work to be done into three parts, where each part is handled by a set of classes in a separate package. The first package named `tracer` is responsible for retrieving data from the solution under test and recording objects and their values in a simple data structure. The second package named `visualizer` is responsible for drawing graphs based on the data collected in the first step. The third package named `webpage` renders a HTML page that provides navigation support for sequences of visualizations. Details about these packages will be discussed in the following subsections.

Retrieving Data

Similar to the `TracingJavaChecker` discussed in section 5.1.6, the tracing part of the `KovidaChecker` component is based on the Java Platform Debugging Architecture [82]. Hence it uses the JDI-API [79] for retrieving objects and events from the Java virtual machine running the program submitted by the student.

```
<project>
   <classes>
      <class name="Phonebook" style="list" />
      <class name="Entry" style="list_element" />
      <class name="Profession" style="list_element" />
   </classes>
   <breakpoints name="Miniprojekt5Kovida.java">
      <breakpoint orderId="1">34</breakpoint>
      <breakpoint orderId="2">36</breakpoint>
      <breakpoint orderId="3">38</breakpoint>
   </breakpoints>
</project>
```

Listing 6.1: Sample configuration file for the `KovidaChecker` component.

Besides the source code it requires a file containing configuration information created by the author of the exercise. Listing 6.1 shows a small sample configuration file. The file is written in XML and contains the following information:

- **Class names:** A list if class names indicates which classes should be considered in the visualizations at all. In the sample configuration, three classes have been included. Any instances of these classes will be recorded for visualization, while all instances of other classes will be ignored.

- **Layout hints:** For each class used in the visualization, a hint for layout can be added. In the sample configuration, class `Phonebook` is marked to be a list head, while the other two classes are marked to be list elements. Thus the layout will try to arrange them in a linear sequence. Other possible option are `tree` and `tree_element`.

- **Breakpoints:** To avoid overhead in data processing, visualizations are not created for each and every execution step. Instead, breakpoints can be defined in which execution will be stopped and a visualization of the current state of the program will be generated. The sample configuration defines three breakpoints, but it is possible that more than three visualizations are created if a breakpoint is hit more than once in a program run. Since breakpoints are defined by naming file and line, they cannot be used on source code to be manipulated by students, as this may lead to unexpected results. To avoid this, the configuration allows also to specify method names, where visualizations will be created for each execution step within that method, regardless of line numbers.

Based on this configuration, the Java Virtual Machine for running the students program is instructed to notify the visualization module every time an object of the given classes is created. Each time a breakpoint is reached, another notification is given and the module takes a snapshot of the current program state by collecting the current state for all recorded objects. Each snapshot is stored in a separate file for further processing. At the same time the current state of recorded objects is stored, the current method call stack is inspected as well. For each stack frame, the method name and all available local variable names and values are stored in the same file to be included in the subsequent processing steps.

Drawing Graphs

For creating the visualizations, the `visualizer` package of the component makes use of the GraphViz library [59] for graph drawing. It uses the "dot" routine of this library, because it is able to compute graph layouts that are suitable for the general requirements explained in section 6.1.1. However, it is not able to handle sequences of visualizations, so additional effort had to be made to get the module foresighted with regards to optimal positions of objects. For this purpose, the visualization module first compiles a list of all objects relevant in at least one visualization and calculates positions for each of them. These positions are used in all visualizations, even if this would imply large gaps between objects. Thus the implementations prefers preservation of the mental map against a compact arrangement.

The method call stack is visualized as a separate node that has a fixed position in the lower left corner of the visualization. The node contains stacked blocks for each stack frame, where each block contains a table listing local variables and method parameters alongside their values.

After the list of all objects has been compiled, colors are assigned for coloring attributes of classes. Thus these colors are defined globally and do not change during sequences of visualizations. However, running the same program two times may result in different colorings, because internal mechanisms of the Java Virtual Machine may result in a different order of objects reported to the visualization module and thus a different order of colors assigned to them.

Having done all global settings, the module creates a layout configuration file for each snapshot and runs the GraphViz routine with this configuration in order to generate an image as the final step.

Rendering Final Output

After the individual visualizations for each state are generated, the final output of the component needs to be assembled. This happens by generating a HTML page that contains the first visualization on the left hand side and the student's source code on the right hand side. The source code line corresponding to the program state in which the visualization was created is highlighted. Navigation elements on the page allow the students to step to the next visualization, resulting in a changing image on the left hand side and a different highlighted source code line on the right hand side.

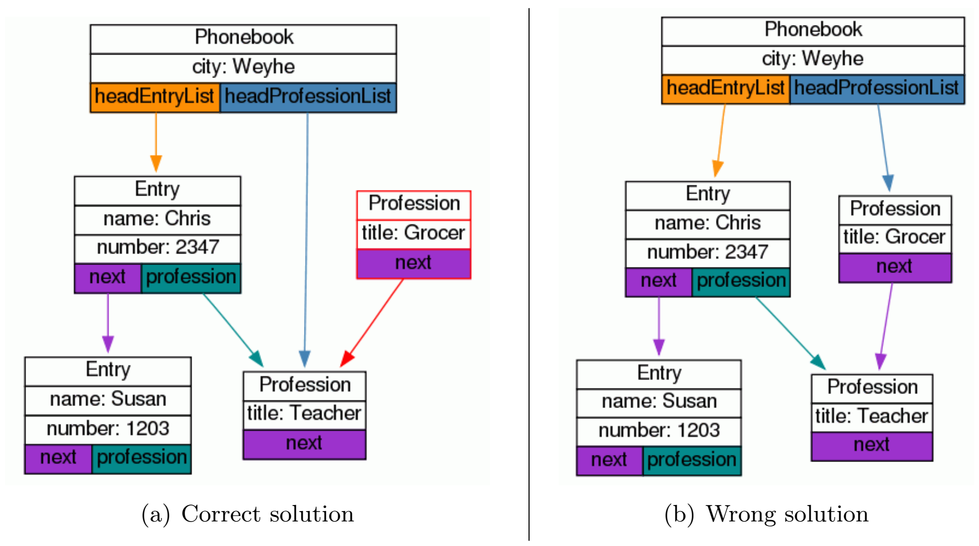


Figure 6.1.: Visualization of solutions for task 1 from the example. In the correct solution, the upper object of type `profession` marked with a red border has been created and removed correctly according to the given task and all pointers are set correctly, too. In the wrong solution, no object has been deleted, although it is clearly visible that the first object of type `profession` in the list is not referenced from any object of type `entry`.

6.1.3. Example

In this section, the use of the component is illustrated using a small example. Figures used in this section do just show the actual data structure visualization and ignore the visualization of the method call stack and local variables.

The exercises used in this example asks the student to implement a phone book. A phone book consists of a data structure composed of objects of type `entry`, containing attributes for name and phone number of the contacts stored in the phone book. In addition, each entry may have a reference to an object of type `profession`, which has an attribute for a title. Both a list of `entry` objects and a list of `profession` objects should be maintained in a phone book by pointing from one element in a list to its successor, while an object of type `phonebook` points to the heads of both lists. The exercise actual consists of several tasks that have to be solved step by step. Consequently, visualizations can be created after each step.

Java source code for the types `entry` and `profession` is provided to the students completely, so that they just have to implement algorithms for inserting objects into a phone book, sorting objects, removing objects and so on. The source code for the type `phonebook` is only provided as a stub that has to be filled by the students. See listing 6.2 for the given source code. The configuration file used on this exercise is the one already showed above in listing 6.1.

```
public class Entry {
   private String name;
   private int number;
   private Entry next;
   private Profession profession;

   public Entry(String name, int number, Entry next){
      this.name = name;
      this.number = number;
      this.profession = null;
      this.next = next;
   }

   public Entry(String name, int number,
        Profession profession, Entry next){
      this.name = name;
      this.number = number;
      this.profession = profession;
      this.next = next;
   }
}

public class Profession {
   private String title;
   private Profession next;

   public Profession(String title, Profession next){
      this.title = title;
      this.next = next;
   }
}

public class Phonebook {
   public String city;
   public Entry headEntryList;
   public Profession headProfessionList;

   // Add here the constructors and methods
    // necessary to solve the exercise.
}
```

Listing 6.2: Given Java source code for the tree classes used in the phone book data structure from the example. Predefined getter and setter methods are not shown here for brevity.

Task 1: Creating and deleting objects

The first task inside the exercise is to create a phone book, insert one entry with a reference to a profession, insert another entry without a reference to a profession, insert an additional profession and afterwards remove all professions that are not references by any entry. In a correct implementation, a visualization of the resulting data structure looks as shown in figure 6.1(a). In this case, the visualization module was configured to show all objects relevant for the data structure and to consider objects of type `entry` and `profession` as elements of a list, thus ordering them one below each other. Most interesting in this visualization is probably the upper object of type `profession`, which is marked with a red border. This is exactly that extra object that had to be created and deleted afterwards according to the given description of the task. Thus, a single visualization is enough to show that this object has been created and destroyed correctly and there is no need to run extra test cases or insert debug statements into the program code. Moreover, we can see that the "headProfessionList" pointer has been moved correctly to the successor of the deleted element.

As a comparison, figure 6.1(b) shows a visualization based on the same configuration for a wrong solution of the same exercise. Obviously there is no deleted object. Since the visualization is configured to show all objects relevant for the data structure, it is easy to check whether the first object in the professions list is referenced by any object of type `entry`. Thus the student may conclude that something is wrong with the algorithm for deleting objects, but everything is right with the algorithm for inserting objects.

Task 2: Sorting data structures

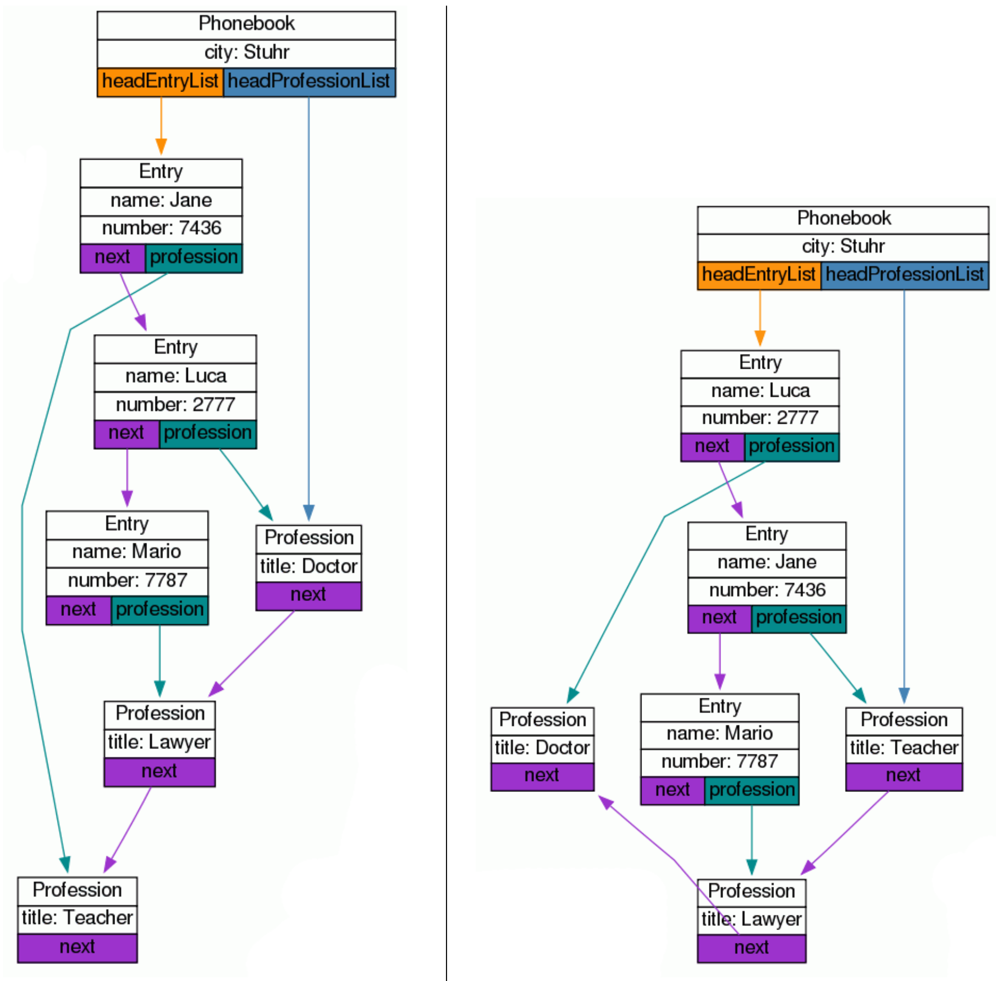
The second task inside the exercise is to create another phone book and insert several objects of the types `entry` and `profession` and some references between them. Afterwards both lists have to be sorted alphabetically. Figure 6.2(a) shows a visualization of a correct solution. The same configuration for the visualization module has been used again as for the visualizations of the first task. Obviously both lists are in alphabetical order as requested. In this simple case it may be sufficient to provide a textual output of both lists to check whether they are in the right order. However, it is easy to imagine, that these checks can be performed in an easier way with visualizations in more complex cases. This is already understandable from figure 6.2(b), which shows a visualization of a wrong solution. In a plain textual representation of the list produced by printing out the names it would be possible to notice that they are in the wrong order, but it would not be possible to see that the student might just have misunderstood the task by sorting the entries by number.

Task 3: Merging data structures

As already stated above, most benefits from visualizations can be expected if data structures tend to get complex. This is relevant in a third task in which students are asked to merge the both phone books created in the previous tasks. More detailed, they are asked to create a third object of type `phonebook`, containing copies of the complete content of the older phone books, i.e. copies of all objects of type `entry` in alphabetical order and copies of all objects of type `profession` in arbitrary order, but no duplicates. Thus pointers have to be corrected in case when two objects of type `profession` with the same title exist in both older phone books.

Figure 6.3 shows the visualization of a correct solution, again using the same visualization configuration as the previous ones. Obviously a third object of type `phonebook` has been created using copies of objects as requested. In textual output, students would be required to have much detailed knowledge about object identifiers to see whether they have used copies of objects or just added references, but the visualization makes this clear at once. All other requirements of the task can also be checked easily: The objects of type `entry` are in alphabetical order and the last object of type `profession` has two incoming references.

Figure 6.4 shows a visualization of a solution where something has gone completely wrong. A new object of type `phonebook` has been created, but no copies of older objects are used. In addition, only some pointers has been added or changed, so that the new phone book does not even contain all old objects in correct order. However, it can be observed that



(a) Correct solution

(b) Wrong solution

Figure 6.2.: Visualization of solutions for task 2 from the example. In the correct solution, it is easy to see that both lists are in the correct alphabetical order. In the wrong solution, the entries are obviously not sorted correctly. Note that the ugly layout is no bug. Compare to the subsequent visualization shown in figure 6.4 for explanation.

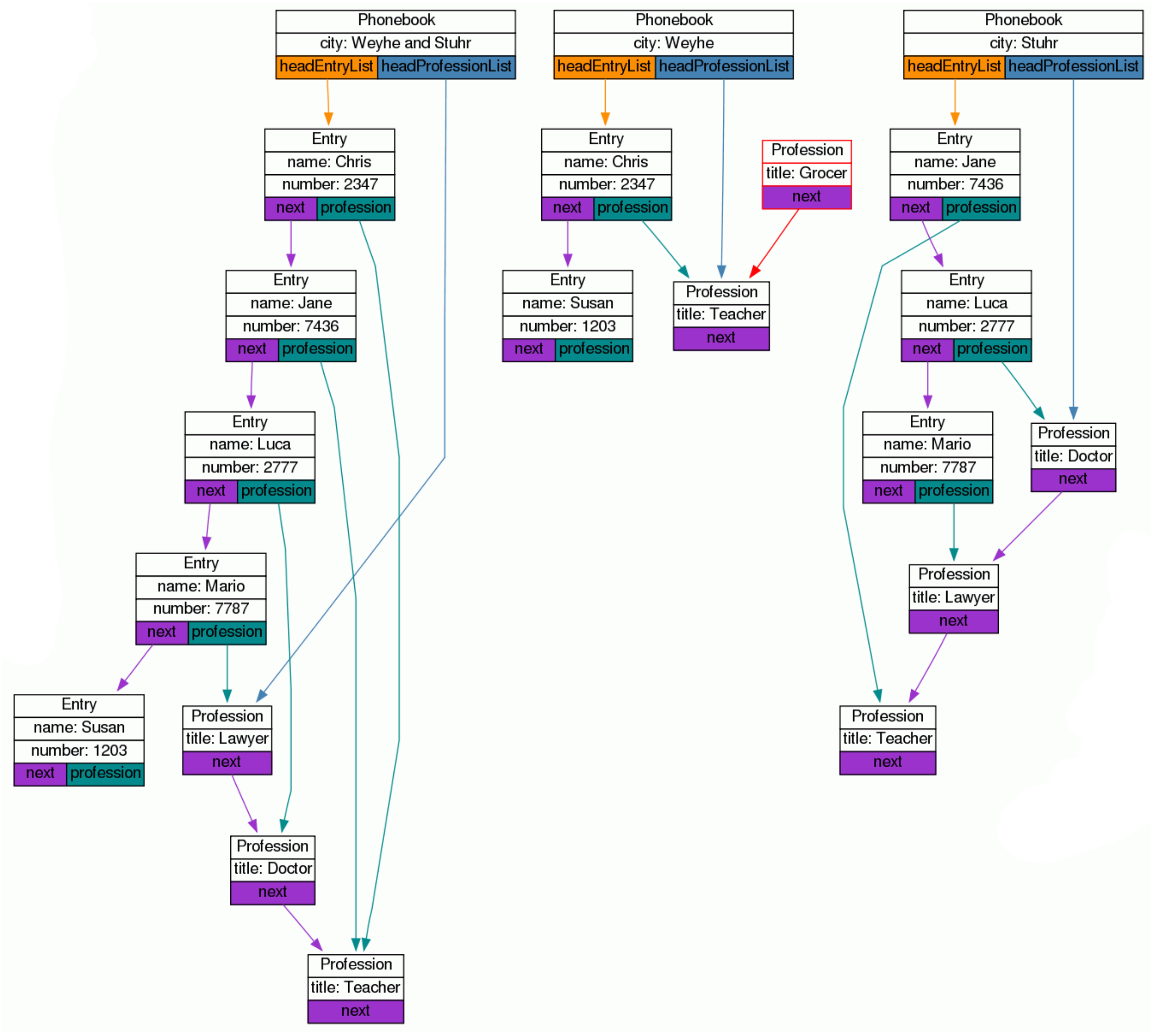


Figure 6.3.: Visualization of a correct solution for task 3 from the example. The newly created phone book is shown left, while the older phone books remain unchanged. Objects of type `entry` in the new phone book are in the correct order as requested and the last object of type `profession` has two incoming references.

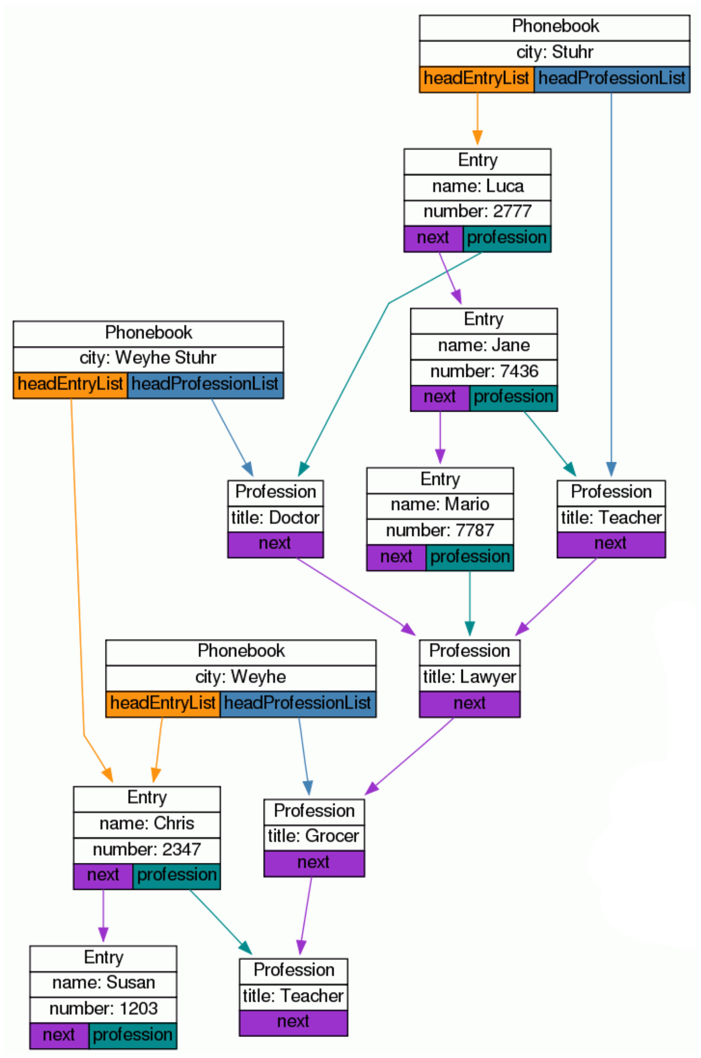


Figure 6.4.: Visualization of a wrong solution for task 3 from the example. A new phone book has been created, but objects from the older phone books are referenced instead of being copied. Moreover, only few pointers have been changed. The reuse of known objects in this task is the reason why the visualization module placed the object of type `profession` with title "doctor" at a position that did not seem to be reasonable in figure 6.2(b).

the reference between the professions “doctor” and “lawyer” has been turned and hence even the data structure of the older phone book has been destroyed. It is impossible to notice this from a textual output of the contents of the new phone book and it is also hard to notice this from a purely textual representation of the whole system state as delivered by a trace. Additionally, it would have been much more difficult to find this change if the layout algorithm does not preserve the mental map in sequences of visualizations. However, also the drawbacks of a foresighted algorithm can be seen in this example, as this change in the reference is the reason why this objects has already been placed in a non-optimal position in the previous visualization as shown in figure 6.2(b).

Nevertheless, it can be concluded that a component for automated generation of visualizations of data structures adds some additional feedback options, although it does not provide any additional explicit feedback on faults. Since it is based on the same data that is used for tracing program execution, it also includes not much overhead in analysis. Thus this component is a useful additional contribution in the context of software artefact analysis in e-assessment systems, almost similar to the post mortem performance and coverage analysis discussed in section 5.1.5.

6.1.4. Connecting Ideas

Although visualizations are so far intended as an additional means of feedback generation without contributing to software artefact analysis directly, some connections to the core topics of this thesis can be drawn. First, the static analysis techniques discussed in chapter 4 are suitable for general graph structures and can thus in principle be applied to object structures. In particular, they can be used to search for expected or unexpected arrangements of objects. Such structures can then be highlighted in the visualizations, thus adding additional feedback. In this case, the visualization is an artefact generated by analyzing a software artefact and at the same time is subject to artefact analysis itself.

Second, visualizations can be used as an alternate means to represent findings from dynamic analysis. In particular, some violations of temporal constraints on program execution may be easier to understand in visualizations than in textual traces, e.g. that fact that some object is never created or deleted. In turn, findings from trace analysis can also be used as triggers for visualization generation, so that visualization need not to be created for the whole program run, but just for parts that show some interesting properties. The same is true for static analysis of source code, that can also be used as a trigger.

6.2. Code Reading Exercises

It can be assumed that the amount of help students get from the traces generated by white-box testing programs depends on their ability to understand and interpret these traces. Consequently, it may be useful to increase their ability in reading these traces independent of actual programming exercises or faulty programs at hand. One setting that might improve student’s capability in reading and understanding traces is to ask students to create traces for small programs on their own. The assumption is that students are able to understand program behavior for their own programs from traces if they are able to create traces for programs that are new to them. In fact, exercises of this kind can be supposed to not only train student’s capabilities in creating traces, but also the necessary capability of understand program behavior from reading code. In case of the latter, writing traces is one possible way to document their understanding, where quizzes or short essays are equally valid options.

In particular, reading code is not only necessary to understand and debug own code, but also in other situations: Without the ability to read and understand code, it's almost impossible to learn from examples as they are presented in lectures or books. It is also impossible to contribute in collaborative exercises and team projects, because this also requires to read and understand code written by others.

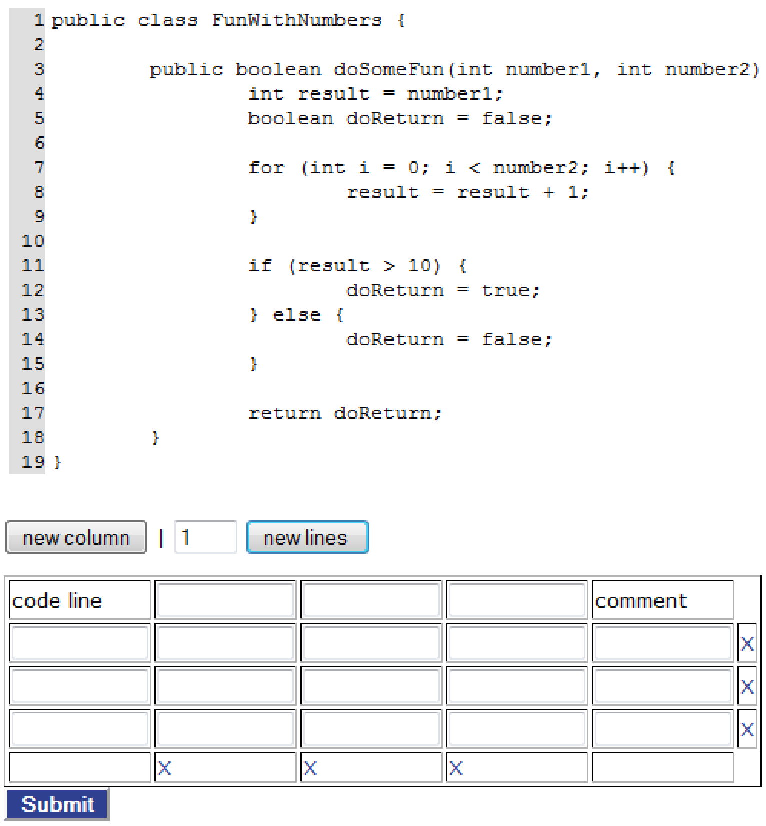
When talking about code writing exercises, it is quite obvious that there are several possible solutions to the very same exercise. The fact that these kinds of exercises are open with an unbounded number of possible solutions in the general case was one of the main motivating factors for using automated analysis of software artefacts. In contrast to that, code reading exercises are closed with exactly one correct answer. This in turn allows to detect errors in the answer more schematically: Students may fail to find all fields and variables used in the source code, which will result in traces with a wrong number of columns. Students may also miss to understand the control flow of the program, resulting in a wrong number or ordering of lines in the trace. Finally, students may also miss to understand data manipulation happening during the program execution, resulting in wrong content of trace cells or entire columns. Thus the type of error detected in the answer can be connected directly to a misunderstood concept and thus be used for a feedback message that refers to conceptual knowledge. This is in turn the knowledge required to read other traces as well and hence it can be assumed that this kind of feedback to code reading exercises will indeed help students to read other traces as well. Admittedly, no empirical studies on this have been conducted so far for this particular concept. At the time of writing, it is work in progress and has only been explored in one bachelor degree project.

Although code reading exercises are in principle closed with exactly one possible answer, there is space for variations: The same code can be invoked with different inputs, that usually result in different program behaviour and thus different traces. If some intelligent random generator for inputs is used when generating the task description, students can get motivated to attend the same exercise more than once. In particular, this can help them to train their ability to abstract from concrete program behaviour.

6.2.1. User Interface

Different to general programming or modelling exercises where students submit their solution as files uploaded to a server, the concept of code reading exercises as discussed above requires a specific user interface. In particular, it has to fulfill four requirements:

- It must be able to display the program that is to be read by the students, ideally including line numbers.
- It must allow students to create a trace table with an arbitrary number of columns and rows.
- It must allow students to fill in every cell of the trace table.
- It must be able to show feedback to a submitted trace table.



```
 1 public class FunWithNumbers {
 2
 3         public boolean doSomeFun(int number1, int number2) {
 4                 int result = number1;
 5                 boolean doReturn = false;
 6
 7                 for (int i = 0; i < number2; i++) {
 8                         result = result + 1;
 9                 }
10
11                 if (result > 10) {
12                         doReturn = true;
13                 } else {
14                         doReturn = false;
15                 }
16
17                 return doReturn;
18         }
19 }
```

code line				comment	
					X
					X
					X
	X	X	X		

Figure 6.5.: Prototypical user interface for code reading exercises. Students can extend the trace table to the necessary size by adding or removing rows and columns.

A prototypical realization of the first three requirements is shown in figure 6.5. It shows some source code in the upper part and an empty trace table in the lower part. A textual description of the exercise can be given above the code, but is not shown in the figure. Most importantly, the task description should include the parameter values that should be used when creating the trace.

Initially, the trace table shows just one row and two columns for code line and comment. Students may add rows and columns by using the buttons above the table. They can also remove rows and columns by using the X. Each cell including the head cell of every inserted column is an input field that can be filled in by the students. Filling in column heads and filling in line numbers is mandatory on each row and columns, respectively. Using the last column for comments is optional in any case and is primarily intended to help students in taking notes.

6.2.2. Component `TracAndCheckChecker`

A component implementing trace comparison between student's input and the actual correct trace has been realized in a bachelor degree project by Sebastian Meis and Dominik Schacht. Although the prototypical implementation does not allow for variable input values, the checking process is prepared to handle variable exercises. The process is divided into four steps:

- First, the program code is executed by some test driver, applying the input values used in the exercise. A trace is created during the execution as described in section 5.1 of this thesis. The trace is stored in an appropriate data structure.
- Second, the input provided by the students is converted to the same data structure as used for the trace.
- Third, some post-processing is applied to the trace, because there are some differences between automatically created traces and the user input. In particular, students may skip inputs for unchanged fields and may also skip entire lines that do not change variable values.
- Finally, both traces are handed over to a trace comparator class that applies several comparison methods.

The comparison is able to identify different kinds of errors: First, the number of rows and columns is checked and the variable names given in the column heads are compared. Possible errors that can be spotted in this step are "wrong line count", "missing variable", or "unknown variable". Second, the sequence of steps is checked by comparing the line numbers one by one. The only additional error possibly spotted by this analysis is "wrong line number" for any line number entry that does not match the expected sequence. Any errors detected in these two steps are collected for feedback report. If no errors are found in any of these steps, a third analysis is performed that compares the actual variable values in each of the steps. For each mismatch, an error of type "wrong variable value" is reported. Again, any errors reported in this step are included in the final feedback.

Most interestingly, the different errors that can be spotted during the analysis can be mapped to different kinds of misconceptions: Missing or wrong columns hint towards a general failure in identifying the variables used in a program. Missing lines or wrong ordering of lines hint towards failure in understanding the control flow of a program, while wrong variable values hint towards failure of understanding data flow or the semantics of variable manipulation operations. Hence it is not only possible to give evaluative feedback in terms of the number of mismatches or errors, but also descriptive improvement feedback that refers to the respective capabilities.

6.2.3. Connecting Ideas

As sketched above, analysis of the student's answer can happen automatically based on the correct trace and this in turn can be generated automatically from the source code used in the task description of the exercise. Thus preparation of the exercise already makes use of the dynamic analysis techniques discussed earlier in this thesis. However, this idea can be taken one step further for explicitly training on reading code written by others: Instead of using code samples generated by teachers, code snippets written by students as part of some coding exercise can be used to generate content for code reading exercises. In particular, this can also be used to demonstrate students the benefits of writing well-formatted code that is easy to read. Moreover, it can be used as introduction into peer-reviews of answers, where students do not just create traces for solutions written by others, but also generate general feedback to these solutions. However, since this does not involve automated analysis of software artefacts any longer, this idea clearly goes one step beyond the goals of this thesis. Nevertheless, it demonstrates the capabilities of automated analysis in automated preparation of other types of exercises.

6.3. Analysis and Feedback Generation based on Metrics

So far, the techniques discussed in this thesis focused on the analysis of a single solution to a given exercise, which is composed of a small number of artefacts or even just a single artefact. It was shown that a reasonable amount of feedback can be created from this kind of analysis. However, especially the context of e-assessment but also artefact analysis in general may be concerned with a large number of artefacts. This opens an additional opportunity to gain more insight and create more feedback: A single artefact can be understood and interpreted not only on its own, but in relation to the mass of similar artefacts. This analysis consequently is primarily concerned with descriptive feedback and of less value for evaluative feedback. A solution to a given exercise is right or wrong because of its own qualities, not because it has similar qualities than other solutions. In the same way, what is right and wrong has to be determined by the exercise directly, not by analyzing a larger set of solutions. Having said that, there is still much benefit that can be expected by inspecting the mass of solutions or at least by comparing solutions with each other.

In particular, achievement feedback has to be considered here that can be used to motivate students. If a solution for a programming exercise passes all dynamic checks, it is considered correct, but that does not necessarily mean that it is “good” in a wider meaning of this term. Instead, the code quality can be very bad and coding conventions can be violated. The latter can surely be checked by static checks to some extend, but there are still some properties that cannot be checked this way, e.g. the plain length of the program or the number of variables used. Obviously, there is very little meaning in talking about strict limits in these cases: A program is not bad because it introduces 16 variables and gets good by reducing the number of variables to 14. However, it can be motivating for a student to learn that his solution is correct with respect to functionality, but 50% longer than the average correct solution or the sample solution. In the same way, it can be an extra bonus beyond getting a good grade to know that the own solution managed to solve a problem with less nested loops than the sample solution. Note that the post mortem performance or coverage analysis on traces as discussed in section 5.1.5 can also be understood as some kind of metric-based comparison, where the metric is not based on the artefact directly, but on derived information.

A second goal is to create additional feedback on an incomplete solution, that considers the artefact as a whole rather than particular bits of it. Static checks on an UML diagram may inform a student that some particular association is missing. While this is a very concrete hint, it does not tell the student whether or not he is on the right track in general. A comparison with other solutions may tell the student that he uses much more different kinds of UML elements than other (possibly successful) students did. Although there might be nothing wrong about each particular element used, the solution as a whole may be overly complex, which in turn might be reason why the student didn’t notice the missing association found by the static check.

In order to do this kind of feedback generation, it is alluding to use metrics. Metrics have a long tradition in software engineering and have at all times been debatable: One the one hand it is known that you cannot judge what cannot be measured [34] and on the other hand it is also known that some metrics do not measure what they pretend to measure or that they influence to object that is measured [85]. This section contributes to this discussion by showing ways in which metrics can be used to generate feedback along the lines sketched above. In all cases, solutions to the same exercise are compared to each other based on the same metrics. No statement is made whether the metrics are useful to compare different solutions from different exercises and whether the metrics provide any abstract information that is useful beyond giving feedback. The use of metrics for feedback generation has also been explored at other places in literature, e.g. in [96] and [60].

As already mentioned above, solutions can be compared to each other. In particular, a student's solution can be compared to a sample solution for the same exercise. In this case, the metrics for the sample solution can be considered static, since the sample solution is not expected to change during the lifetime of an exercise in the general case. Thus for a student's solution the same metric-based feedback will be generated independent of the time of the submission and the characteristics of other solutions submitted in parallel. This is not true for metric-based checks that compare the student's solution to the mass of all submissions. For example, the average length of a program can change over time and consequently a solution that was shorter than the average at one point in time can be longer than the average at a different point in time. This fact has to be taken into account both when selecting the metrics to be used and when designing the feedback to be given based on these metrics.

The following subsections discuss the use of metrics for indirect feedback generation for programming exercises. A similar discussion is work in progress with some student's theses and thus not considered here.

6.3.1. Metrics for Programming Exercises

Typically, software product metrics are subdivided into metrics of size and metrics of complexity [27, 48]. While the former are concerned with measures like the number of lines or the number of statement, the latter are concerned with the structure of the program, e.g. in terms of different paths in the control flow. Studies show, that there is in general a strong correlation between size and complexity of code [78], but observations from real software projects need not to be valid for small programming exercises as well. Table 6.1 gives an overview of different metrics for Java programming exercises and some relations between them. It can be seen that some correlations vary much between different exercises. Investigations on reasons for that are beyond the scope of this thesis, but it motivates that metrics may contain information about a particular solution or the exercise in general.

When selecting actual metrics to be used for feedback generation, three aspects have to be taken into account: First, it has to be considered whether the metric can indeed motivate students to do something useful, in contrast to motivating them to do "bad tricks" just to get a better feedback. Second, it has to be considered whether the metric is suitable for comparison with the mass of solutions or just with a sample solutions as already mentioned above. Third, it has to be considered which impact predefined code templates have on the results of the metric, i.e. whether the use of predefined code templates in an exercise may

	Exercise 1	Exercise 2	Exercise 3
Number of Submissions	1309	820	600
Avg. Number of Statements (NoS)	272.57	464.40	238.04
Avg. Cyclomatic Complexity (CC)	84.69	129.49	130.37
CC / NoS	0.31	0.28	0.55
Correlation of CC and NoS	0.93	0.98	0.76

Table 6.1.: Sample metrics for three self-training exercises from winter term 2011/2012. While exercises 1 and 2 show much similarity with respect to ratio and correlation of CC and NoS, exercise 3 shows much different values.

render the results useless or invalid. The following subsections will discuss some metrics and reason about these three aspects for all of them. However, it is not the goal of this chapter to provide a detailed analysis on all available metrics and their usefulness for feedback generation. Instead, it is the goal to present some reasonably selected metrics that can be used to demonstrate how feedback can be given on metrics. So the focus is not on which metrics to use, but how to use metrics in general. It is assumed that the same techniques of feedback generation that will be discussed below can be used with other metrics as well. It is even considered possible that other metrics than the ones used in this thesis provide better results.

Metrics of Size

The simplest way of determining size is to use the file size of the source code file. However, this way of measuring size is usually not counted among the software product metrics. Instead, counting lines of code (LoC) is a widely known metric. Without digging into the details of various criticism of this metric, it can be noticed that it has a drawbacks with respect to e-assessment and automated tutoring: The result of this metric can easily be manipulated by adding or removing line breaks without changing any statement in the code. It thus seduces students to manipulate the source code just to please the metric. Hence it can be considered to be less helpful in analyzing student solutions.

Counting statements or expressions is another possible way of measuring the size of an artefact. In theory, these are almost as easy to manipulate as lines of code, as redundant or unreachable code can be inserted virtually everywhere in a source code document. However, two things have to be noticed: First, increasing the source size is typically not interesting for students, as positive feedback will usually be given for small and efficient code. Second, redundant or unreachable code can at least partially be identified by static checks and may thus trigger negative feedback messages or even reductions in the overall grade. In practice, it is thus less tempting for students to use this kind of manipulation and hence the number of statements (NoS) or number of expressions (NoE) can indeed be used as simple metrics of size. They can be used both for comparing to a single sample solution or to the mass of solutions. Moreover, they are stable in face of code templates used with the exercise, as the number of statements or expressions used in their can be subtracted from the overall value if necessary.

A special variation of NoS or NoE is counting subsets of statements or expressions, such as counting the number of loops or the number of references. The same considerations apply as above, but the particular metrics are more focused on certain aspects of the code.

Slightly more complex than counting elements of the source code is the use of Halstead metrics [66]. In these metrics, operands and operators in the program code are considered and the number of different operands and operators used is determined as the size of the vocabulary. All values gained this way can be set into ratio to each other for different analysis goals. In the context of teaching programming, the site of the vocabulary is a critical aspect for several reasons: First, code templates provided with the exercise may use a vocabulary that is larger than the one actually required from the students to solve the exercise. Consequently, any solution can have the same size of the vocabulary, making this metric useless. Second, no clear assumptions can be made whether using a larger or smaller vocabulary is better in a particular exercise. Giving positive feedback for using a large vocabulary may seduce students to create more complex solutions than necessary. In turn, giving positive feedback for small vocabularies may seduce students to create less readable or more inefficient solutions in order to avoid sophisticated statements. Hence Halstead metrics can be considered less helpful in this context.

Metrics of Complexity

The most prominent metric of complexity is probably the cyclomatic complexity [98]. It is based on the number of nodes and edges in the control flow graph of a program and thus measures the complexity of the decision structure. It effectively determines the number of independent linear execution paths in the program. It is hence independent of any changes that do not add or remove behaviour. Notably, the metric result can be calculated without computing the full control flow graph of a program by counting `if`-statements, conditional expressions, loop statements, `case`-statements, `catch`-blocks and logical operators that allow for shortcut evaluation.

The cyclomatic complexity metric can be considered useful in the context of e-assessment, as it is compatible with predefined code templates. The complexity of these templates can be computed and thus the complexity added by the student's solution is easy to determine. Moreover, it can be considered a usual didactical goal to teach students writing code that is not more complex than necessary. At least, it can be considered motivating for the student to see that their solution is less complex than an average solution. As the cyclomatic complexity cannot be reduced by simple tricks like reformatting code or twisting statements, there is little risk of negative consequences of this kind of extra motivation.

Specifically for object-oriented program, several other metrics have been defined that measure particular aspects of object-oriented code such as coupling of classes, cohesion, or visibility of methods [38, 23]. Most of these metrics are independent of the size of the code in the same way as the cyclomatic complexity and are thus suitable for measuring complexity or logical structure of a program. However, the suitability for e-assessment depends very much on the actual exercise: Introductory exercises often define method signatures and other crucial properties of object oriented code and students have little or no choice that can influence the metrics result. On the other hand, more advanced exercises can indeed make use of the metrics, as some of the aspects they measure may match exactly the topic of the exercise, such as designing code with a sophisticated inheritance structure.

6.3.2. Analysis Methods for n-dimensional Data

Two basic ideas can be considered for analysis of the data gained from metrics: Finding clusters, balance points, or core areas on the one hand, and finding outliers and corner cases on the other hand. Moreover, it can be considered interesting to do this kind of analysis for different subsets of all solutions, e.g. the subset of correct solutions and the subset of incorrect solutions.

For illustration, figure 6.6 shows three 2-dimensional plots of data using the same exercises as for table 6.1 above. Grey circles get larger the more solutions share the same metrics values for Cyclomatic Complexity (CC) and Number of Statements (NoS). The linear trend line is shown for the whole set of solutions. For calculating the averages, solutions that gained more than 50 out of 100 points as an overall grade are considered as correct, while all others are considered as incorrect. The size of the green and red auroras is equal in all cases and does not carry any additional meaning. Without applying any additional means of statistical analysis, several observations can be made just by manual visual inspection:

- For the first and the second exercise, the average values lay very close to the linear trend line, while they don't for the third exercise.
- Average values are close to each other in the first and third exercise, but less close together in the second.
- Average values are near the center of the area covered by solutions in the second and third exercise, while the average for incorrect solutions is close to the lower left end in the first one.
- The third exercise shows a large grey circle quite far away above the linear trend line, while large circles are close to the trend line in the other exercises.

Searching for reasons for this observations is beyond the intent of the illustration, but all these observations hint to the fact that there are clear differences between the exercises. Hence they also motivate to use metrics in the analysis of exercises. Moreover, some of the basic ideas from the introduction of this section can be realized based on this kind of data, e.g. identifying solutions that are correct but larger and much more complex than the average correct solution. However, manual visual inspection is only possible for data that can be plotted nicely in 2-dimensional figures. So more general techniques need to be discussed in order to advocate for automated analysis of metrics.

Outlier Detection

An obvious strategy for automated feedback generation is to detect solutions that deviate markedly from the other solutions submitted to the same exercise and hence fulfill a classical definition of an outlier [61]. It can be both interesting for the authoring student to know that the solution is different from all others, possibly hinting towards a rare misunderstanding, and for the teacher to draw special attention to this solution, possibly offering individual help to the student.

Several approaches exist to automated outlier detection, ranging from statistical methods to neural networks and machine learning [70]. While many of these approaches handle

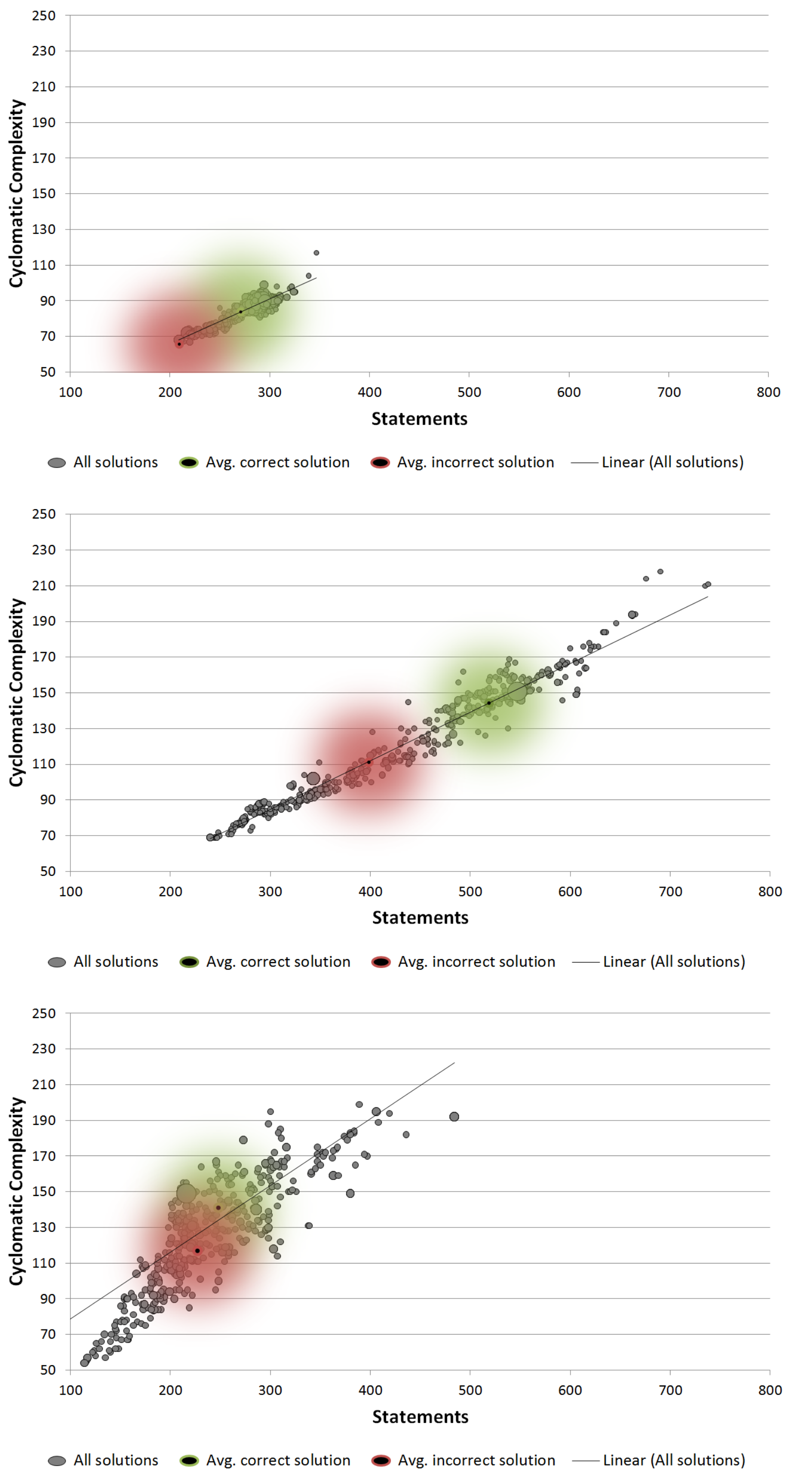


Figure 6.6.: Three 2-dimensional plots of data from three exercises using Cyclomatic Complexity (CC) and Number of Statements (NoS) as metrics. Raw data is the same as used in table 6.1.

outlier detection as a boolean decision, where each data point is either an outlier or not, there is also the approach of assigning a local outlier factor (LOF) to each data point, where higher factors represent a higher probability that the respective element is indeed an outlier [17]. This approach looks appropriate for didactical scenarios, where the borders between a "normal" solution and an "abnormal" solution are not strict. In particular, the local outlier factor can be used to track the development of an exercise solution over various submissions and create specific feedback if the LOF increases with each submission.

Looking at the upper diagram in figure 6.6, is is easy to see that outliers can occur and can clearly be identified in the data set for an exercise. However, in the other diagrams it is more vague which solutions still classify as an outlier, which stresses the possible benefits that can be achieved by not using boolean decisions but measures like LOF instead.

Cluster Detection

Many of the methods used for outlier detection are based on proximity or density of data elements, where both proximity and density are defined via the distance between data elements or the similarity of elements. These measures thus cannot only be used for outlier detection, but also for cluster analysis. Many different approaches to cluster analysis exist [46], with some notable differences: Hierarchical approaches like *nearest neighbour* are usually based some distance measure and thus can be parameterized with the minimum similarity or maximum distance between elements in the same cluster. In contrast to that, optimization approaches like *k-means* are parameterized with a predefined number of clusters and try to find the best partitioning of the data set, again based on distance measures. There are also density-based approaches that come close to the natural understanding of visual cluster detection by identifying clusters based on a high density of data elements.

Looking at figure 6.6 again, it can be noticed that finding good parameters may be a relevant problem in cluster detection. Neither hints towards a reasonable number of clusters nor hints towards reasonable distance or similarity limits are directly visible from these examples. Nevertheless, the obvious differences between average correct and wrong strongly solutions suggest that there are indeed remarkable differences in each data set. Hence it is future work to identify appropriate cluster detection methods or heuristics for for determining the parameters.

6.3.3. Other Analysis Methods

Besides cluster and outlier detection, other methods can also possibly be used, but have not yet been examined in the context of this thesis. In particular, methods for reducing the dimensions of data sets like principle component analysis (PCA) [31, 51] or t-SNE [151] can be expected to be helpful in cases where data from many different metrics exists. Using these methods, a multi-dimensional data set can be reduced to two or three dimensions, which allows graphical representation and thus manual inspection for cluster or outlier detection. However, this may lead to loosing the connection between the data elements and their actual metric values. It then would be possible to detect clusters of similar solutions, but it is no longer directly visible what are the common properties of these solutions. As this is probably important for feedback generation, additional effort is hence necessary before actual feedback can be generated from these clusters.

All techniques discussed so far handled the whole set of data elements uniformly. However, it has to be noticed that the individual data points are not independent, but that one student may submit several solutions to the same exercise that are incrementally eliminating errors. When these solutions are connected to each other, trajectories within the data space can possibly be found, that may reveal different approaches for incremental work on the exercises. As already mentioned above, a set of subsequent submissions may show an increasing local outlier factor, which means that it deviates away from the main clusters in the data space. Again, these ideas have not yet been explored systematically and offer room for future work on this topic.

Another aspect not discussed so far is the comparison of solution spaces for different exercises. Instead of searching for outliers or clusters with different properties inside the data set of one exercise, different data sets can be compared to derive interesting properties of the exercises themselves. Criteria for comparison can for example be the size of the data sets in various dimensions, the number of outliers found in each of these sets or the number of clusters generated with certain parameters. In doing so, analysis results for data sets based on metrics for individual solutions are used as metrics for exercises. This again stresses the potential benefit gained from using metrics for feedback generation and creates many ideas for future work far beyond the scope of this thesis.

6.3.4. Component `MetricsJavaChecker`

The implementation of a checker that generates metrics for Java program code follows that same design as the checkers for static analysis of Java program code as discussed in sections 4.3.1 and 4.3.2. However, there is no need to identify sub-components, because simple metrics can already be computed by applying the visitor pattern to the syntax tree generated by the parser.

The `MetricsJavaChecker` counts various syntax elements and thus provides the following metrics:

- **Cyclomatic complexity:** This metric is determined by counting relevant source code elements, i.e. loop statements, `if`-statements, `switch`-statements, conditional statements, `catch`-clauses, infix expressions that allow for shortcut evaluation, and method declarations in the source code.
- **Height of AST:** This metric is calculated as the maximum depth of any leaf in the tree.
- **Number of fields:** This metric is determined by counting all field declarations in the source code.
- **Number of methods:** This metric is determined by counting all method declarations in the source code.
- **Number of loops:** This metric is determined by counting all loop statements of any kind in the source code.
- **Number of statements:** This metric is determined by counting all statements in the source code.

- **Number of expressions:** This metric is determined by counting all expressions in the source code.
- **Number of references:** This metric is determined by counting all field accesses, constructor invocations, method invocations, and references by simple name (as they occur in access to local variables or class names) in the source code.

Metric values are then returned and attached to the solution as additional attributes. The component does not perform any further analysis. Instead, the metric values can be exported via the database and by used for further analysis in statistic tools. It has been used in several experimental evaluations. In particular, data for table 6.1 and figure 6.6 has been collected using the component.

A prototype implementation for data analysis based on VBA scripts for Microsoft Excel with some statistical methods for cluster and outlier detection has been implemented in parallel to this thesis in a bachelor thesis by René Lopez-Barillao.

6.3.5. Connecting Ideas

One implicit assumptions that connected metrics based analysis with other analysis techniques was already made in the examples used above: Any analysis that compares characteristics of correct solution with characteristics of incorrect solutions requires to use other techniques to divide the set of solutions appropriately. Besides using the overall grade, other analysis results can also be used to find partitions. For example, static analysis can be used to divide solutions into those using an iterative approach and those using a recursive approach. This is helpful, because comparing the complexity or size of an individual recursive solution to an average value largely based on iterative solutions can be considered less helpful.

In fact, using the overall grade of a solution means to include it in the statistical analysis as an additional metric beyond the ones discussed in this section. Of course, this can be done for more metrics gained from other techniques. As already mentioned at the beginning of this section, post mortem performance and coverage analysis as discussed in section 5.1.5 provides additional numerical data that can be included in a statistical analysis. This in turn strengthens the need for sophisticated statistical methods that allow to analyze large multi-dimensional data spaces.

Part III.

Review of Results

7. Evaluation

One of the goals of this thesis was to provide implementations that are ready to be used in the e-assessment system JACK. While this is mainly a technical contribution in the dimension of systems, evaluation can nevertheless be carried out in all four dimensions tackled by this thesis: In the dimension of systems, it can be evaluated whether JACK was able to run with the created plug-ins without problems and could fulfill its duties. In the dimension of techniques it can be evaluated whether the intended analysis worked well and robust under practical conditions and how the performance was in the production environment. In the dimension of inputs it can be evaluated how many rules or test cases were able to produce which amount of feedback, and whether this is a satisfying ratio. In the dimension of feedback it can be evaluated how many feedback messages could actually be generated and whether students were satisfied with the feedback generated by the system. This chapter will elaborate on all these aspects in the following subsections and will thus cover all four dimensions of automated feedback and grading. More detailed evaluations such as studies on the didactical effects of particular types of feedbacks are considered to be out of the scope of this thesis.

7.1. General Application Data and Scaling

As mentioned in chapter 3, the system JACK has been in use since 2006. The component `StaticJavaChecker` as presented in section 4.3.1 has been included in the system in 2008. The components `GReQLJavaChecker` and `GReQLUMLChecker` as presented in sections 4.3.2 and 4.3.3 have been in use since 2011. In especially, component `StaticJavaChecker` was replaced by component `GReQLJavaChecker` in this year. The component `TracingJavaChecker` as presented in section 5.1.6 has been completed in 2012, but trace generation was already available since 2010. Hence it was possible to observe all of them under practice conditions for at least one year. Component `DynamicUMLChecker` as presented in section 5.2.3 is the only one that has not been used under practice conditions so far.

Tables 7.1 and 7.2 show the usage statistics from winter term 2008/2009 until winter term 2013/2014 for an introductory course to programming in Java. In summary, more than 55'000 individual submissions (from about 12'000 students) have been graded automatically by JACK in the past seven years for that course. JACK has also been used on other courses, but these had much smaller participant numbers and are thus neglectable. Numbers are comparable to other systems, e.g. BOSS that handled 5'500 coursework submissions in the academic year 2004-05 [81] or PASS that handled 7'300 program submissions in four consecutive semesters starting from fall 2004 [162]. Based on the figures for JACK it can be said that all implementations for Java prove to be ready for practical use as they were able to handle a large amount of submissions without problems. The amount of data also qualifies for long-term studies as the one marking granularity presented in [47], which used data from eight terms that covered 7'010 submissions from 971 unique students. Notably, the average

	winter term 2008/09	winter term 2009/10	winter term 2010/11	summer term 2011	winter term 2011/12	summer term 2012	winter term 2012/13	winter term 2013/14
Trail	371	254	386	177	473	402	-	-
Exercise 1	811	936	1098	454	1536	949	1935	1105
Exercise 2	502	998	1024	598	1133	762	1816	1452
Exercise 3	519	1101	1311	343	1425	659	1740	1249
Exercise 4	530	844	1148	198	1363	430	1112	600
Exercise 5	296	512	645	166	983	265	978	349
Exercise 6	109	394	403	79	665	163	899	401
Total	3138	5039	6015	2015	7578	3630	8480	5156
Avg. submissions per student	2.67	2.39	3.03	3.01	3.04	2.78	3.32	3.32

Table 7.1.: Number of submissions to homework exercises in the introductory course to programming.

	winter term 2008/09	winter term 2009/10	winter term 2010/11	summer term 2011	winter term 2011/12	summer term 2012	winter term 2012/13	winter term 2013/14
Trail	258	287	299	48	293	125	339	268
Assignment 1	545	569	701	260	743	430	836	823
Assignment 2	352	588	608	224	833	373	825	766
Assignment 3	571	670	549	171	928	236	617	590
Assignment 4	512	552	495	90	814	170	443	357
Assignment 5	276	319	384	89	381	112	463	290
Assignment 6	150	254	233	32	352	55	237	285
Total	2664	3239	3269	914	4344	1503	3760	3379

Table 7.2.: Number of submissions to assignments under exam conditions in the introductory course to programming.

number of submissions per student in JACK seems to be constantly higher than the one observed for the system Bottlenose in a study across several courses [122]. However, this can well be triggered by different sizes of the exercises, as JACK also experienced a course with 4.52 submissions per student on significantly larger exercises.

Usage figures for component `GReQLUMLChecker` have already been discussed alongside the example presented in section 4.4.2.

To handle the amount of submission up to five worker servers have been used in parallel for one core server since 2011. Both before that time and afterwards surveys among students using JACK have been conducted that included questions regarding server load. In winter term 2010/2011 (thus before using grading servers in parallel), about 85% of 61 participants in the survey stated that they experienced an overloaded server. Moreover, 23% of the participants stated that they usually did not submit much solutions to JACK because they did not expect a timely response. In winter term 2011/2012 (thus after setting up parallel servers), only about 75% of 63 participants experienced an overloaded server and less then 10% stated that they did only submit few solutions because they didn't expect timely responses. Although 75% is still a quite high rate, it can be concluded that scaling worked well within the given architecture of JACK to adjust to increasing numbers of submissions. In fact, wait times for students could be reduced even more in the subsequent years. In winter term 2013/14, only about 26% of 61 survey participants reported an overloaded server. Moreover, nobody stated that he did not submit solutions because he didn't expect timely responses.

7.2. Performance

Besides reducing wait times for students by running grading in parallel, the absolute run time for feedback generation is also important. Thus for the solutions submitted to the six homework exercises in winter term 2012/2013, detailed performance statistics have been created. All grading activities were performed on virtual servers with 2 GHz CPU and 4 GB RAM. Results are shown in table 7.3. The figures do not include time consumed by standard compiler checks as these are performed prior to static and dynamic checks and cannot be configured. The performance impact of these checks is neglectable in comparison to other factors.

As can be seen, static checks consume less than one second in any case, although there are notable differences between different exercises. These variations are not necessarily induced by the number of files to be parsed or rules to be checked, but also by the size of the files (not shown in the table) and thus the complexity of the syntax graph. This is an expectable behavior. Since the performance impact is low in comparison to dynamic checks, there is no need to worry about the number of rules used in static checks from a performance point of view.

The time needed for test execution and trace generation does also vary, depending more directly on the number of test cases defined per exercise and also the content of the exercise itself (also not shown in the table). This is again the expected behavior. In particular, exercises including loops or recursion are likely to result in longer run times for tests than exercises without. Besides the expected relative performance behavior, the magnitude of run times is interesting: test execution time is better measured in minutes, rather than in seconds as for the static tests. It is thus the dominating factor influencing the total check time per exercise. From the didactic point of view on exercise design it is surely not desirable to reduce the number of test cases or simplify the exercise contents to increase grading performance. Instead, splitting the set of test cases into two or more that can be handled in parallel by several grading servers can be used to increase performance. While this does not increase the overall throughput of the grading system, it reduces the grading time for the individual submission. There is also a lower bound of three seconds for test case execution due to implementation details in component `TracingJavaChecker`. Reducing this obstacle may result in a slight increase in performance for some exercises. However, it can be assumed that a reduction by one or two seconds is almost irrelevant to the user, even if grading would be performed synchronously.

Although manipulating test cases or exercise design for the sake of performance is in general not recommended, it can nevertheless be helpful to have a closer look on the test case design. Unfortunately, exercise 6 in table 7.3 is an example of bad test case or exercise design from the performance point of view. In this exercise, some objects are stored in an

	Exercise size	Static Check			Dynamic Check	
	(files / methods)	Number of rules	Syntax graph generation	Pattern matching	Number of test cases	Total run time
Exercise 1	1 / 8	16	198 msec.	20 msec.	10	3103 msec.
Exercise 2	1 / 4	12	192 msec.	18 msec.	8	28673 msec.
Exercise 3	3 / 7	13	703 msec.	143 msec.	25	137466 msec.
Exercise 4	5 / 19	26	724 msec.	130 msec.	15	60206 msec.
Exercise 5	4 / 15	26	384 msec.	106 msec.	16	68856 msec.
Exercise 6	6 / 16	22	367 msec.	75 msec.	18	166262 msec.

Table 7.3.: Average performance results for various steps in feedback generation for self-training exercises in winter term 2012/2013. Time for standard compiler checks is not included in any of these numbers, but has to be added both for static and dynamic checks. Run time for dynamic check includes trace generation, but not trace alignment, because the latter was not used on all exercises.

array and are later searched by iterating over all array positions. Test cases include cases in which the searched object is not in the array and thus the full array has to be crawled. While this is good test strategy, size has gone wrong: The array is initialized with size 10000, although no test case adds more than 5 objects to it. Thus thousands of array indices are crawled in the test case without any benefit but a great loss in performance. While the sample solution consumes 189213 milliseconds for test execution, an alternate solution that uses an array of size 10 instead of 10000, just needs 17150 milliseconds without any loss in test quality.

Not included in the results discussed so far is the time needed for trace alignment, because it was not used throughout all exercises in winter term 2012/2013. Instead, performance measures where taken during the experiments discussed in section 5.1. In these experiments, a total amount of about 1'000 test cases has been executed on 115 solutions with 306 failed test cases that required trace analysis. On the average, this analysis took 81 milliseconds per trace with variations depending on the size of the trace. The contribution of trace alignment to the overall run time for feedback generation is thus also an inferior factor compared to the time used for trace generation and has hence only little impact on the overall performance.

7.3. Feedback Statistics

The evaluation so far covered primarily the dimensions of systems and techniques, but not the dimension of feedback. However, this dimension can be considered the most important one from a student's point of view. Thus this section deals both with the amount of feedback generated and the student's perception of the feedback messages.

Figure 7.1 shows the different categories of feedback messages that were generated by JACK for the homework exercises in winter term 2012/2013. As can be seen, about 70% of all feedback messages are generated because of dynamic checks executing the students' solution, while about 30% are generated by static checks. This may look odd on the first glance, as table 7.3 lists 115 static rules used in that term but less than 100 dynamic tests. However, it cannot be concluded that static checks are less efficient. Instead, a more detailed examination of the shares and some dependencies is necessary: In particular,

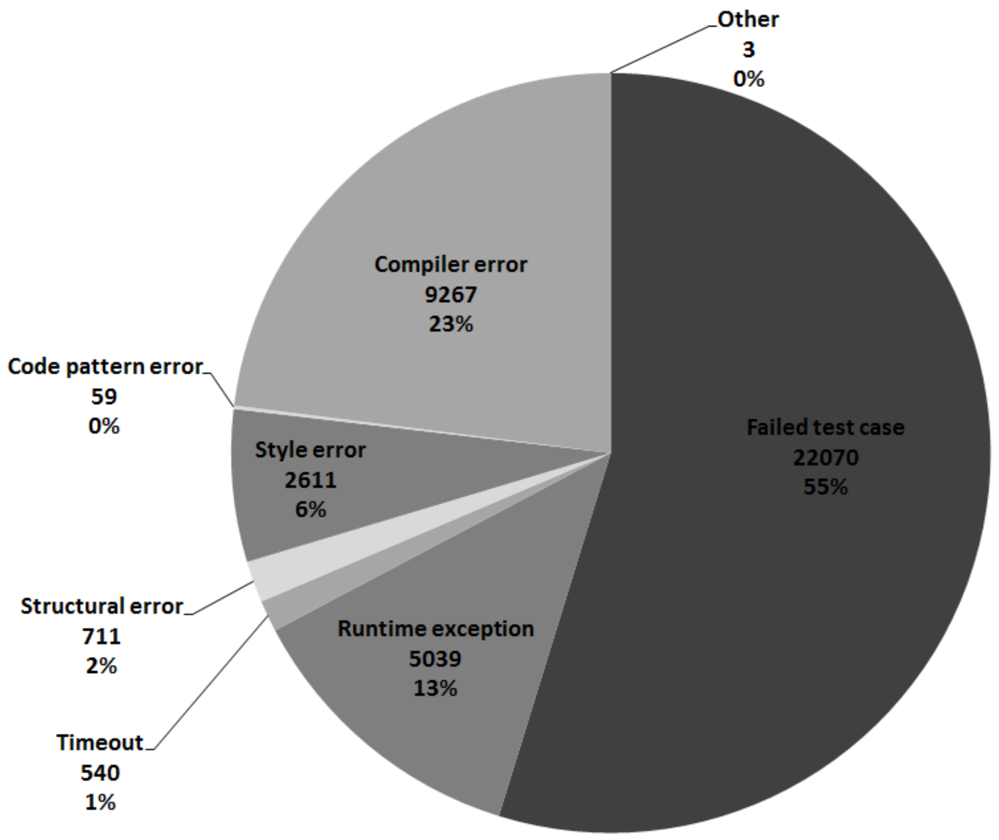


Figure 7.1.: Categories of different feedback messages generated by JACK in winter term 2012/2013.

feedback messages from static checks include compiler errors. Typically, a solution with compiler errors has many of them and in any case no additional feedback messages from dynamic checks are generated for them, as a solution with compiler errors is not runnable. Structural errors where students missed to use given code template correctly, and code pattern errors covering typical beginner flaws (such as breaking a `while`-statement by an extra semicolon right after the termination condition) are the smallest numbers. Almost any of these errors can be fixed independent of each other, hence a submission with many feedback messages from static checks is often followed by a submission with no errors in the static check. In turn, messages from dynamic checks relate to test cases that may have dependencies. Hence students that receive many feedback messages from test cases typically focus on a small subset of them and try to resolve these in the next submission. Consequently, the remaining feedback messages will be created again on this follow-up submission. Hence the great share of messages from dynamic tests does not necessary mean that static analysis techniques like the one discussed in chapter 4 are useless. On the contrary, these numbers show that they can add value even in very basic application scenarios and without creating complex combined feedback generation processes.

For some of these figures the rate of false positives has already been discussed in the respec-

		never				always	number of answers
JACK inadvertently reported mistakes.	2008/2009	53%	19%	19%	8%	2%	59
	2009/2010	30%	34%	28%	8%	0%	76
	2010/2011	40%	23%	20%	15%	2%	60
	2011/2012	37%	19%	18%	22%	5%	63
	2012/2013	No data available					
	2013/2014	18%	45%	27%	11%	0%	56
	Average	**35%**	**28%**	**23%**	**13%**	**2%**	
JACK inadvertently accepted incorrect solutions.	2008/2009	57%	33%	10%	5%	0%	58
	2009/2010	76%	12%	11%	1%	0%	75
	2010/2011	36%	16%	26%	18%	3%	61
	2011/2012	49%	29%	13%	6%	3%	63
	2012/2013	No data available					
	2013/2014	43%	28%	22%	7%	0%	54
	Average	**53%**	**23%**	**16%**	**7%**	**1%**	

Table 7.4.: Results from empirical surveys regarding user's perception of false positive and false negative results in JACK in self-training scenarios.

tive sections earlier in this thesis. A more complete picture can be gained by considering the user's perception directly. To get more insight into user's perception of automated feedback, surveys among students who used JACK in an introductory course to Java programming were conducted in several years. Their perception of feedback quality for self-training scenarios (homework exercises) in terms of false positives and overseen errors is shown in table 7.4. As can be seen, on the average 63% of the students encountered false positives never or just rarely, and 76% encountered false negatives never or just rarely. However, the latter rate does not help much in judging the quality of feedback, because students might just not know that their solution is in fact incorrect although JACK gives no feedback.

It would be interesting to compare these numbers to the performance of human teachers when grading exercises. Unfortunately, to the best of the authors knowledge, there is no statistical data published on how many mistakes are overseen by human teachers when marking programming exercises. Thus only indirect conclusions can be made from asking the students to compare JACK's performance to an average human performance. Results for this question are shown in table 7.5. On the average, about 49% of the participants agree or fully agree with the statement "JACK is able to detect mistakes at least as good as a human teacher", while only 24% reject this statement. This is in line with much older results for the system KASSANDRA, where 28% of students said that KASSANDRA would check exercises better than human teaching assistants and 26% saw no difference between the system and a human teaching assistant [157]. However, it stays an open question how much effort is necessary to close the gap and make automated systems really as good as human teachers in detecting mistakes. Because of the missing data about human performance, it cannot even be determined whether the gap is possibly just subjective. Some piece of an answer to this question is given by a case study for the feedback system PETCHA, which is based on MOOSHAK as a grading system: An experimental group showed 55% satisfaction with the automated feedback, while a control group showed 62% satisfaction with manual feedback [112]. This is a relatively small gap between the feedback quality from automated systems

		absolutely right				absolutely wrong	number of answers
JACK is able to *detect* mistakes at least as good as a human teacher	2008/2009	19%	37%	24%	11%	8%	62
	2009/2010	22%	26%	30%	13%	8%	76
	2010/2011	12%	36%	23%	20%	10%	61
	2011/2012	19%	19%	29%	22%	11%	63
	2012/2013	No data available					
	2013/2014	21%	36%	28%	12%	3%	58
	Average	**19%**	**30%**	**27%**	**16%**	**8%**	

Table 7.5.: Results from empirical surveys regarding JACK's capabilities of error detection.

		never				always	number of answers
JACK inadvertently reported mistakes.	2008/2009	72%	12%	12%	3%	2%	60
	2009/2010	62%	17%	13%	7%	1%	76
	2010/2011	66%	19%	9%	7%	0%	60
	2011/2012	56%	14%	16%	13%	2%	63
	2012/2013	No data available					
	2013/2014	82%	4%	11%	4%	0%	28
	Average	**66%**	**14%**	**12%**	**7%**	**1%**	

Table 7.6.: Results from empirical surveys regarding user's perception of false positive results in JACK in exam scenarios.

compared to human teaching assistants. However, the study also names a large amount of about 40% of exercises that got no feedback at all in both groups and it remains unclear how this has to be taken into account. Another piece of an answer can be found in a recent study on code comparison, where agreement between teachers and algorithms was as high as between different teachers [54]. Although checking code similarity is clearly not the same as finding errors, it suggests that 100% agreement between automated analysis and manual analysis is hard to reach, as 100% agreement between different teachers is hard to reach as well.

Independent of that, it also has to be taken into account for all automated feedback systems with similar mechanisms like JACK, that test cases for dynamic tests and rules for static tests have to be provided by humans and thus may miss mistakes for the same reason a human would miss them. This assumption is supported by the figures shown in table 7.6. It again asks for user's perception of false positives and false negatives, but now exclusively focusing on exam situations instead of self-training scenarios. Now 80% if the students did report that they never or just rarely had spotted a false positive result. Since the technical system was the same as in self-training scenarios, this difference must origin from a more careful exercise and test design by the teachers. This is also an argument in favor of using automated systems in general, because exercises defined for these systems can evolve over time and thus get more detailed, while human feedback capabilities may vary due to personal reasons or changing employments.

Nevertheless, the plain amount of feedback messages and their technical quality does not tell much about the feedback quality as perceived by the students. It can be assumed that

		very positive	positive	neutral	negative	very negative	number of answers
Attitude towards the statement "In a general sense, JACK is useful for *self-training*"	2008/2009	34%	44%	15%	7%	0%	61
	2009/2010	53%	29%	9%	5%	4%	77
	2010/2011	26%	43%	25%	5%	2%	61
	2011/2012	35%	27%	27%	10%	2%	63
	2012/2013	No data available					
	2013/2014	49%	33%	13%	5%	0%	61
	Average	**40%**	**35%**	**17%**	**6%**	**2%**	
Attitude towards statement "In a general sense, JACK is useful for *e-assessment*"	2008/2009	30%	36%	23%	8%	3%	61
	2009/2010	48%	16%	12%	13%	11%	77
	2010/2011	34%	38%	16%	10%	2%	61
	2011/2012	30%	35%	19%	10%	6%	63
	2012/2013	No data available					
	2013/2014	36%	38%	13%	11%	4%	56
	Average	**36%**	**32%**	**16%**	**11%**	**6%**	

Table 7.7.: Results from empirical surveys regarding the general attitude towards JACK in different scenarios.

a false alert or missing feedback message is not helpful, but on the other hand it cannot be said that a correct message is necessarily helpful in any case. Instead, students have to be asked directly. As the most general possible question the students were asked about their general attitude towards JACK both in self-training situations and as an automated grader for exams. Results are summarised in table 7.7. On the average, 75% of all participants showed a positive or very positive attitude towards JACK as a self-training tool and 68% showed a similar attitude towards JACK as an automated grader for e-assessments. This is somewhat surprising, because particularly in self-training it can be considered important to explain mistakes very well instead of just detecting them. Moreover, the students did report more false positives for self-training than for exams as discussed above. One possible interpretation is that students do not mind whether their assessment work is checked by a human teacher or by an automated system, so they see less benefits here. Another possible interpretation is that JACK's automated feedback is indeed helpful to the students in self-training situations even with a higher rate of false positive results and they miss it in exam situations. This is a strong argument in favor of the efforts made during this thesis, focusing on generating more feedback than plain right-or-wrong decisions.

The figures for JACK can be compared to results for other self-training or assessment systems to see whether this thesis has indeed contributed to the state-of-the-art for these systems. Unfortunately, detailed publications of empirical evaluations are rare. Table 2.2 in the introduction of the thesis shows results concerning the attitude of students towards different systems, as collected for different systems with different users. Results are comparable, because all systems include at least feedback on syntactical and semantical errors. According to these figures, the systems are accepted as helpful utilities. JACK is among the best results, being almost equal to Marmoset and even beating it for the very positive attitudes. While for most of the systems neutral and moderately positive attitudes dominate, the results for JACK clearly show a majority in positive and strong positive attitudes. Thus it can be deduced that from a user's perspective JACK is among the best available feedback systems and providing a notable improvement in user satisfaction. Notably, the satisfaction with JACK seems to be higher than the satisfaction with manual feedback in the study on PETCHA discussed above. However, it is very desirable to have more figures published for other systems in the future.

Instead of feedback messages informing students about right and wrong or giving actual feedback, some of the techniques discussed in this thesis do produce additional information. In particular, results from tracing can be used to compute test coverage, while the `KovidaChecker` component produced visualizations of object structures. Since the evaluation so far focused on direct feedback, some more insight into user's perception on these additional information can be considered useful. The survey in winter term 2013/14 thus included particular questions on test coverage and object structure visualization.

For object structure visualizations, 26 out of 50 participants who had worked on at least one exercise using this means of feedback, stated conformance with the statement "Visualizations were helpful to me". 10 participants thought that they needed more visualizations to consider them helpful. 9 participants stated that they did not pay much attention to the visualizations at all and just 5 participants thought that the visualizations were confusing and thus not helpful. These figures are a clear argument in favor of using this means of feedback.

For test coverage, results are even slightly better: 30 out of 55 participants stated conformance with the statement "Test coverage markers were helpful to me", while 8 participants ignored these markers. 2 participants considered the markers as confusing. Interestingly, 15 participants stated that they didn't even notice these markers. This can have two reasons: They might have not taken a look at their source code in the solution review or they might have had full test coverage so that there were simply no markers in their code. Nevertheless, these figures are again a clear argument for providing information on test coverage as an additional means of feedback.

8. Contribution Review

The goal set out by this thesis was to narrow the gap between the potential power of software engineering methods for automated artefact analysis and the methods actually used for feedback generation in recent e-assessment systems. The core part of this thesis tackled this goal in different dimensions and considering two different methods, that were both implemented as extensions for an existing e-assessment system. Additional contributions based on the same e-assessment system were also considered. The purpose of this chapter is to review all these contributions with respect to the goal. As the literature review in chapter 2 it is structured by the four dimensions of the discussed gap.

8.1. Dimension of Systems

Much work regarding a flexible system architecture for e-assessment systems has been done outside the scope of this thesis and thus been summarized with the overview on JACK in chapter 3 prior to the core chapters. The system JACK has consequently been used as a technological basis for implementations because of its extensibility. Following the ideas from the literature review, all techniques discussed in this thesis have been implemented as extensions to it instead of developing yet another e-assessment system. The efforts made in this thesis resulted in the creation of five new system components plus three additional components that were created in the same context. Three of the components discussed in this thesis are in practical use for several years now and two have been used in experiments with real student data, as summarized in table 8.1. Not only the functional properties of these implementations, but also performance and scaling has been evaluated at least for some of the components. The goal with respect to the dimension of system can thus be considered fulfilled without limitations, as there is no remaining work from a pure technical perspective of systems.

Sure enough, the goal in the dimension of systems was the one that was easiest to fulfill as the gap to be closed was mainly a gap in the other dimensions. However, the success of this thesis with respect to the dimension of systems shows that there is indeed little need for new systems, as systems exist that allow easy extensions.

This thesis contributes confirmation to the fact that there is no need to develop entirely new e-assessment systems. Instead, research should focus on individual techniques for feedback generation and their combination.

8.2. Dimension of Techniques

The core part of this thesis provided discussions of static and dynamic techniques for artefact analysis. For static analysis, a graph based approach to pattern matching has been developed that overcomes two limitations of recent e-assessment systems: It is not limited

Chapter	Component	In practical use	Experimental evaluation
4	`StaticJavaChecker`	yes	
	`GReQLJavaChecker`	yes	
	`GReQLUMLChecker`		yes
5	`TracingJavaChecker`		
	- trace generation	yes	
	- trace alignment		yes
	- coverage analysis	yes	
	`DynamicUMLChecker`		yes
6	`KovidaChecker`	yes	
	`TracAndCheckChecker`		yes
	`MetricsJavaChecker`		yes

Table 8.1.: Summary of the actually implemented components

to a particular type of artefact as professional tools usually are and it is not limited in the scope and detailedness of the analysis, e.g. by not being limited to single file analysis. As graph based representations of artefacts as well as pattern matching are no new approaches in general, it can be concluded that this portion of work done in this thesis indeed makes a bit more of the power of automated software artefact analysis methods available in actual e-assessment systems. As a decent part of the discussion, the approach has been realized using two different underlying techniques: A graph transformation system and a graph query language. The latter provides more flexibility and power in the design of pattern matching rules, while the former provides more graphical support. It was shown that both approaches are ready for use in a practical environment. Open problems with respect to tool support have also been identified and discussed to stimulate future work with focus on better tool support for teachers and exercise authors.

For dynamic analysis, a novel approach to trace analysis has been presented that makes use of sequence alignment algorithms and has not been discussed in literature on automated software engineering artefact analysis or e-assessment systems before. Different variations of the approach have been applied to different artefact types. It could be shown that the approach is useful both as primary and auxiliary means of artefact analysis. It can thus be used to refine the analysis methods used in recent e-assessment systems for common artefact types like programming exercises and also to extend the use of e-assessment systems to less commonly considered artefact types like diagrams with execution semantics. The approach is thus also a piece of work that fills the assumed gap. However, experimental results with respect to precision and recall showed that there is also space for further improvement and research. Results were satisfying for a prototypical and experimental implementation, but not excellent. This is in particular true for trace alignment in diagram analysis, where no experimentally validated heuristics for feedback generation based on match scores exist so far. From this point of view, this thesis also points out new gaps to be filled by further studies. Notably, the techniques for trace alignment have been defined independent of the techniques used for trace generation in this thesis, so that both techniques can be progressed and refined almost independent of each other.

Combinations of static and dynamic analysis have been touched by this thesis as an important aspect in various places, without providing implementations for fully integrated approaches. At the conceptional level, connections have been made e.g. by explaining how to prepare trace alignment for programming exercises by static analysis in order to find appropriate traces to compare to. The same is true for strong formal verification techniques like temporal logic, which has been mentioned as an open problem, while simpler verification of assertions has been considered both on the conceptual level and in the implementation. Thus with respect to combinations of static and dynamic analysis this thesis provides only minor contributions that hardly provide substantial filling for the assumed gap, but prepares the ground for further investigations.

The core part of this thesis also provided a chapter on additional contributions that are just loosely connected to static or dynamic analysis. The work on visualization shows that using white-box tests is not only helpful in the context of trace generation, but also allows for additional means of program behaviour explanation. While this does not add any value in terms of advanced artefact analysis capabilities, it shows the didactic value of the underlying techniques. The same goes for the work on code reading exercises, that use the same underlying techniques for a reversed exercise setting. Again this does not involve big technical challenges, but creates new opportunities in exercise design. Different to that, the use of software metrics does not offer any additional features for exercises, but allows easy comparison of solutions. This can be used both for feedback to students and for feedback to exercise authors. Hence this technique does not only contribute to the subject of feedback generation, but is a more general contribution to the whole process of using e-assessment systems.

This thesis contributes novel implementations that allow to apply the same flexible and powerful techniques for feedback generation to a broad range of artefacts. It thus overcomes current limitations of e-assessment systems that are limited to applying a particular technique to a particular type of artefact. The approaches to static and dynamic analysis presented in this thesis can be applied to virtually any software artefact independent of the language it is written in and the properties to assess.

8.3. Dimension of Inputs

During the literature review, the dimension of inputs was identified to be most important not on its own but in conjunction with the other dimensions. Therefore, all results produced in this thesis were judged with respect to this dimension. No particular focus was laid on creating progress in this dimension, but all results were well inside the borders of current state of research.

In particular, the approaches to static analysis require manual considerations by the exercise author as well as explicit design of checking rules. The same requirements have been reported for existing tools in the literature review. The approaches to dynamic analysis rely on comparisons to sample solutions, instead. This is also a typical technique that has been reported in the literature review as well. Hence it can be summarized that this thesis managed to provide new techniques to artefact analysis and feedback generation without requiring new or more complex ways of input generation than existing approaches. Moreover, ideas for tool support have at least been discussed as open problems in the respective chapters.

Similar observations can be made for the conjunction between the dimensions of input and the dimension of feedback: As shown by the evaluations, an improved user's perception of feedback quality could be reached without the need for more complex input to the checking techniques. This is an interesting observation, because it provided insight to the structure of the four dimensions: It is possible to achieve progress in some of the dimensions (e.g. the dimension of techniques or feedback), without the need for additional work in other dimensions. Consequently, work on improved analysis techniques or feedback generation can happen in parallel to work in improved input techniques, with each step in either of these dimensions narrowing the gap in the power of e-assessment systems.

Moreover, using metrics as described in the additional contributions of this thesis is in fact a way of using software analysis methods without the need for any additional input besides the actual solutions. Conclusions can be drawn just by comparing solutions to each other or to the average metric values gained from the set of solutions for an exercise. Of course, sample solutions can also be used as an additional input to this technique, adding some reference values to compare with.

This thesis contributes the insight that new techniques for feedback generation do not necessarily require more complex input than current techniques. On the contrary, although covering a broad range of artefacts and giving detailed feedback, a simple query based property specification and the provision of sample solutions turned out to be sufficient for the techniques explored in this thesis.

8.4. Dimension of Feedback

Extending the possibilities of feedback generation has been one of the motivating factors of this thesis and thus been considered in all work on analysis techniques. Both the work on static and dynamic analysis has been carried out with focus on the amount and kind of feedback that can be generated by these techniques. It could be shown that the feedback can be both evaluative and descriptive and that the discussed techniques are thus helpful both for automated grading and automated tutoring. It could also be shown that both improvement feedback and achievement feedback can be provided. The latter in particular involves comparing solutions to each other, using techniques like metrics.

The user's perspective on feedback generation has explicitly been taken into account in the evaluation and it could be shown that user satisfaction with the feedback generated in JACK is higher than user satisfaction with other recent e-assessment systems as reported in the literature. Although the evaluation explicitly stated that there is still a gap between the feedback capabilities of automated systems and the feedback capabilities of human teachers, it can be concluded that this thesis provides some progress over other existing systems because of the good evaluation results. Since it was not the goal of this thesis to make automated systems as good as human teachers, but to use more of the power of automated software artefact analysis for feedback generation, this goal can be considered fulfilled. However, this thesis also shows that there is still a gap to be closed and that there is a clear desire from a user's perspective for even better e-assessment systems. This in turn also proves the motivating assumptions of this thesis to be correct.

Applying the means for feedback generation developed in this thesis has resulted in a higher user satisfaction than reported for other systems before. So

this thesis not only contributes techniques for more feedback on more artefacts, but this gain in width and depth is also a gain in quality of feedback as perceived by the users.

9. Future Work

Possible extensions and open problems closely related to the particular concepts and implementations presented in this theses have already been discussed towards the end of the core chapters. Hence this chapter serves a more general purpose of pointing towards new directions of research and development that either extend or complement the work done in this thesis.

9.1. Techniques

As this thesis focused on static and dynamic analysis as two separate fields and just pointing towards possible combinations, any combination of both areas can be considered an interesting extension for future work. In particular, feedback generation techniques that generate messages based on two or even more different approaches instead of presenting distinct sets of messages and leaving the job of drawing conclusions to the users can be considered beneficial at least for e-assessment in self-training scenarios. As an alternative approach to combinations, one technique can be applied to the output of another technique, e.g. by generating sequence diagrams or object diagrams from execution traces and analyzing them with graph based pattern matching.

But also in each individual area, future work can complement the results of this thesis. Neither in the area of static analysis nor in the area of dynamic analysis, full coverage of all possible techniques was provided by this thesis. In particular, metric based checks were not considered for diagrams and traces were just analyzed by sequence alignment, but not by other means. There is again an option to apply additional techniques on their own, e.g. give feedback solely based on some software product metric, or to use these in combination, e.g. by deriving performance metrics from traces and comparing different solutions using different approaches by performance in order to give students feedback on more efficient implementations.

Finally, static and dynamic analysis cannot only be used for direct feedback generation but also for other purposes in e-assessment scenarios. For example, they can be used to identify students that are stuck in similar situations to bring them together so that they can solve their common problem in collaboration. As a variant of that approach, a system can identify students that have successfully solved this situation and can automatically derive helpful feedback from analysing their actions and changes without the need for explicit manual feedback authoring by teachers.

In general, the subjects of software analysis techniques in e-assessment systems can be considered to be split in three parts: First, a detailed analysis of the particular software artefact submitted as a solution to an exercises is the necessary basis. Second, findings from this basis can be used to provide feedback on this particular solution, helping the student to be more successful in the actual task at hand. Third, findings from the same basis can also be generalized either with respect to several solutions to different exercises by the same student or with respect to several solutions submitted to the same exercise by different students. This way, more insight on the general capabilities of a single student or general properties of an exercise can be gained.

9.2. Implementations

Although seven components for the e-assessment system JACK could successfully be developed during this thesis and some more ideas be discussed, this does not mean that there is no more work to do on the level of implementations. In particular, the features and architecture of JACK do imply some restrictions: All grading components working in this architecture work at exactly one solution at the same time and return results for just this solution. This mechanism can be enhanced with respect to different aspects:

- For checks that need to compare one solution to the mass of solutions, it is necessary to provide at least aggregated data about this mass of solutions to the grading component, e.g. average figures of some metrics. This can for example be used to provide feedback on the length of an execution trace in relation to the average length.
- For checks that need to compare two solutions with each other, the checker interface needs to be extended to take two sets of files as an input and to return two sets of results. This way, similarity checks between solutions can be implemented as jobs for the workers, where each submission generates one job for each comparison with existing solutions.
- If resources are returned by the grading components, they replace existing ones with the same name in the current implementation. However, in some cases it might be more useful to allow a grading component to append information in an existing resource. For example, a resource may list all similarities to other solutions and is extended by each new comparison made with an additional submission.

In addition, the current implementation does not allow for explicit dependencies between checks and also not for priorities. If several checks are configured for one exercise, each of these checks may be the first one to be performed. Explicit dependencies and priorization can provide two kinds of improvements in this area:

- Only with explicit dependencies, grading components can rely on the existence of results from other grading components. For some of the more complex techniques presented in this thesis, this is very important, e.g. identifying iterative and recursive approaches by static checks prior to trace comparison in dynamic checks.
- While some components are concerned with grading in the narrow sense of “right” and “wrong”, others may be concerned with providing additional information and feedback. This feedback may partly just be understandable in the context of the actual grade received, thus providing explicit grades has a higher priority than providing additional feedback. Moreover, students might be happy with some coarse grained feedback and already submit a new solution before more detailed feedback is computed. In these cases, skipping low priority checks for the outdated solution can help to improve the overall system performance.

But even without any changes in the JACK architecture there is surely enough future work that can be done on the level of implementations and that already has been discussed in the particular chapters of this thesis.

9.3. Evaluation

The evaluations carried out in this thesis were based on small experiments, surveys among students, and case studies from actual usage scenarios. Although this was sufficient to show that detailed automated feedback is considered helpful by the students, there is still much space for getting more detailed insight into the effects of enhanced feedback generation. In particular, the following questions are of interest:

- *Do students at different skill levels prefer different kinds of feedback?*
  If so, feedback generation needs to be coupled with user modeling in order to filter feedback presentation or generation. For the general research in formative feedback, there are indeed results that argue in favour of different feedbacks for different levels of proficiency [123].
- *Is there a dependency between the difficulty or size of the exercise and the kind of feedback preferred by the students?*
  If so, guidelines can be developed which kind of feedback generation techniques should be used on which kind of exercises, saving time for unnecessary feedback generation.
- *Is it necessary to limit the amount of feedback or the different kinds of feedback to preserve readability?*
  If so, it is not sufficient for a feedback generation technique to provide good precision and recall, but also a concise and intuitive representation and a clear scope where and when to be used.

As a generalization of these questions, it is also desirable to perform more studies on the quality of the feedback rather than on the pure fact that feedback can be generated. As shown by this thesis, various techniques exist that can produce lots of feedback messages, but closing the gap between potential power of automated systems and actual power of them even further is surely not only a question of mass, but also a question of quality.

10. Concluding Remarks

Software engineering and e-assessment do not share much common ground on the first glance. Both have developed their communities and made their research progress independently and there is no doubt that they have provided good findings. Nevertheless, e-assessment isn't limited to any specific domain, and thus can in particular be used in teaching software engineering. At the same time, software engineering is not limited to any specific purpose and thus can in particular be used to work with student's solutions to software engineering exercises. So narrowing the gap between both of them with a bridge that brings software engineering knowledge into e-assessment was the goal set out by this thesis.

To do so, several approaches have been discussed, implemented and validated in experiments and practice. The scope of the thesis was designed to cover both a reasonable broad selection of different artefacts, but also to provide detailed in depth studies of the selected techniques. This way, the core hypothesis of this thesis turned out to be true, providing some positive effects: Software engineering methods can indeed be used to gain more insight into solutions in an e-assessment system, resulting in more detailed and more precise feedback generation and thus finally improving student's learning experience.

Without any doubt, there is still a long way to go. While this thesis makes some useful contributions that have proven their value in practice, there is also a lot of open questions and remaining future work discovered at the same time. Although results from this thesis have already been used in practice, there is still space for improvements on some of the algorithms. There is also much space for more sophisticated editing support to help exercise authors in creating detailed analysis and feedback configurations more easily. And there are several concepts of artefact analysis in software engineering that have not been tackled by this thesis at all and that may provide equally beneficial results when used for analysis of software artefacts in e-assessment systems.

Besides all remaining work, a positive overall conclusion can be drawn: Software engineering research can help to improve e-assessment for complex exercise types, while e-assessment with complex exercise types can help teaching various aspects of software engineering. Research in areas connecting both fields provides interesting results for both communities and bears a chance for even more interesting findings in the future.

Bibliography

[1] Code Query Language 1.8 Specification. `http://www.ndepend.com/cql.htm`.

[2] Henshin Project. `http://www.eclipse.org/henshin/`.

[3] NDepend. `http://www.ndepend.com/`.

[4] The Java Language Specification. `http://docs.oracle.com/javase/specs/`.

[5] The Java Virtual Machine Specification. `http://docs.oracle.com/javase/specs/`.

[6] AGG website. `http://tfs.cs.tu-berlin.de/agg/`.

[7] Kirsti Ala-Mutka, Toni Uimonen, and Hannu-Matti Jävinen. Supporting students in C++ Programming Courses with Automatic Program Style Assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

[8] Kirsti M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer science education*, 15(2):83–102, 2005.

[9] Kristi Ala-Mutka. Problems in Learning and Teaching Programming. Technical report, Institute of Software Systems, Tampere University of Technology, 2003.

[10] Noraida Haji Ali, Zarina Shukur, and Sufian Idris. Assessment System For UML Class Diagram Using Notations Extraction. *IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.8, August 2007*, 7(8):181–187, 2007.

[11] Mario Amelung, Peter Forbrig, and Dietmar Rösner. Towards generic and flexible web services for e-assessment. In *ITiCSE '08: Proceedings of the 13^{th} annual conference on Innovation and technology in computer science education*, pages 219–224, New York, NY, USA, 2008. ACM.

[12] Cyrille Artho. Finding faults in multi-threaded programs. Master's thesis, Institute of Computer Systems, Federal Institute of Technology, Zurich/Austin, 2001.

[13] Daniel Baudisch, Manuel Gesell, and Klaus Schneider. Online Exercise System - A Web-based Tool for Administration and Automatic Correction of Exercises. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 1, pages 104–110. INSTICC, INSTICC Press, 2009.

[14] Steve Benford, Edmund Burke, and Eric Foxley. Courseware to support the teaching of programming. In *Developments in the teaching of computer science*, pages 158–166. University of Kent at Canterbury, 1992.

[15] Daniel Bildhauer and Jürgen Ebert. Querying Software Abstraction Graphs. In *Working Session on Query Technologies and Applications for Program Comprehension (QTAPC 2008), collocated with ICPC 2008*, 2008.

[16] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy. GAME: a Generic Automated Marking Environment for programming assessment. In *International Conference on Information Technology: Coding and Computing, 2004.*, volume 1, pages 212–216, 2004.

[17] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM.

[18] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. James Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[19] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight Detection of Infinite Loops at Runtime. In *24th IEEE/ACM International Conference on Automated Software Engineering, 2009.*, pages 161–169, 2009.

[20] Douglas Chalmers and W. D. M. McAusland. Computer-assisted Assessment. Technical report, Glasgow Caledonian University, 2002.

[21] CheckStyle Project. `http://checkstyle.sourceforge.net`.

[22] Peter M. Chen. An Automated Feedback System for Computer Organization Projects. *IEEE Transactions on Education*, 47(2):232–240, 2004.

[23] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[24] Marian Choy, Sam Lam, Chung Keung Poon, Fu Lee Wang, Yuen Tak Yu, and Leo Yuen. Design and Implementation of an Automated System for Assessment of Computer Programming Assignments. In *Advances in Web Based Learning – ICWL 2007*, volume 4823 of *LNCS*, pages 584–596, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Marian Choy, U Nazir, Chung Keung Poon, and Yuen Tak Yu. Experiences in Using an Automated System for Improving Students' Learning of Computer Programming. In *Proceesdings of Advances in Web-Based Learning – ICWL 2005*, volume 3583 of *LNCS*, pages 267–272, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] Gráinne Conole and Bill Warburton. A review of computer-assisted assessment. *Research in Learning Technology*, 13(1):17–31, 2005.

[27] Samuel Daniel Conte, Hubert Earl Dunsmore, and Vincent Y. Shen. *Software engineering metrics and models.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[28] Tom Copeland. *PMD applied.* Centennial Books, 2005.

[29] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[30] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 77–94, New York, NY, USA, 1977. ACM.

[31] Pádraig Cunningham. Dimension Reduction. Technical Report UCD-CSI-2007-7, University College Dublin, 2007.

[32] Charlie Daly and J.M. Horgan. An automated learning system for Java programming. *IEEE Transactions on Education*, 47(1):10–17, 2004.

[33] Will M Davies and Hugh C Davis. Designing Assessment Tools in a Service Oriented Architecture. In *Proceedings of 9th International CAA Conference*, 2005.

[34] Tom DeMarco. *Controlling Software Projects: Management, Measurement, and Estimates.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[35] Paul Denny, Andrew Luxton-Reilly, Ewan D. Tempero, and Jacob Hendrickx. Understanding the syntax barrier for novices. In Guido Rößling, Thomas L. Naps, and Christian Spannagel, editors, *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2011, Darmstadt, Germany, June 27-29*, pages 208–212. ACM, 2011.

[36] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3), September 2005.

[37] Christopher Douce, David Livingstone, James Orwell, Steve Grindle, and Justin Cobb. A technical perspective on ASAP - automated systems for assessment of programming. In *Proceedings of the 9th CAA Conference, Loughborough University*, 2005.

[38] Fernando Brito e Abreu and Rogério Carapuça. Object-Oriented Software Engineering: Measuring and Controlling the Development Process, 1994.

[39] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the Mental Map of a Diagram, August 1991. Research Report IIAS-RR-91-16E.

[40] Jürgen Ebert and Angelika Franzke. A Declarative Approach to Graph Based Modeling. In G. Tinhofer E. Mayr, G. Schmidt, editor, *Graphtheoretic Concepts in Computer Science*, number 903 in LNCS, pages 38–50, Berlin, 1995. Springer Verlag.

[41] Stephen H. Edwards. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of SIGCSE'04*, Norfolk, Virginia, USA, March 2004.

[42] Stephen H. Edwards, Zalia Shams, and Craig Estep. Adaptively Identifying Non-terminating Code when Testing Student Programs. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 15–20, New York, NY, USA, 2014. ACM.

[43] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformations.* Springer, 2006.

[44] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific, Singapore, 1999.

[45] E. Allen Emerson and Edmund M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, number 85 in LNCS, pages 169–181. Springer, 1980.

[46] Brian S. Everitt, Sabine Landau, Morven Leese, and Daniel Stahl. *Cluster Analysis.* John Wiley & Sons, 5 edition, 2011.

[47] Nickolas Falkner, Rebecca Vivian, David Piper, and Katrina Falkner. Increasing the Effectiveness of Automated Assessment by Increasing Marking Granularity and Feedback Units. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 9–14, New York, NY, USA, 2014. ACM.

[48] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[49] FindBugs Project. `http://findbugs.sourceforge.net/`.

[50] Melanie Fischotter, Michael Goedicke, Filiz Kurt-Karaoglu, Nils Schwinning, and Michael Striewe. Erster Jahresbericht zum Projekt 'Bildungsgerechtigkeit im Fokus' (Teilprojekt 1.2 - 'Blended Learning') an der Fakultät für Wirtschaftswissenschaften. Technical report, Universität Duisburg-Essen, Essen, 2013.

[51] Imola K. Fodor. A survey of dimension reduction techniques. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2002.

[52] George E. Forsythe and Niklaus Wirth. Automatic grading programs. *Communications of the ACM*, 8(5):275–278, May 1965.

[53] Davide Fossati, Barbara Di Eugenio, Christopher W. Brown, Stellan Ohlsson, David G. Cosejo, and Lin Chen. Supporting Computer Science Curriculum: Exploring and Learning Linked Lists with iList. *IEEE Trans. Learn. Technol.*, 2(2):107–120, 2009.

[54] Matheus Gaudencio, Ayla Dantas, and Dalton D. S. Guerrero. Can Computers Compare Student Code Solutions as Well as Teachers? In John Dougherty, Kris Nagel, Adrienne Decker, and Kurt Eiselt, editors, *Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE'14*, Atlanta, Georgia, USA, march 2014.

[55] Andrew Gaylard and Andreas Zeller. Data Displaying Debugger, 2009. `www.gnu.org/software/ddd/`.

[56] Marcela Genero, Mario Piattini, and Coral Calero. A survey of Metrics for UML Class Diagrams. *Journal of Object Technology*, 4:59–92, 2005.

[57] Alex Gerdes, Bastiaan Heeren, and Johan Jeuring. Constructing Strategies for Programming. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 1, pages 65–72. INSTICC, INSTICC Press, 2009.

[58] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of the 2nd Workshop on Computer Aided Verification, New Brunswick*, number 531 in LNCS, pages 176–185. Springer, 1991.

[59] GraphViz, 2009. `http://www.graphviz.org`.

[60] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. Feedback Provision Strategies in Intelligent Tutoring Systems Based on Clustered Solution Spaces. In Jörg Desel, Joerg M. Haake, and Christian Spannagel, editors, *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik*, pages 27–38, Hagen, Germany, 2012.

[61] Frank E. Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11(1):1–21, 1969.

[62] Susanne J. Gruttmann. *Formatives E-Assessment in der Hochschullehre.* MV-Wissenschaft, 2009.

[63] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Feedback Generation for Performance Problems in Introductory Programming Assignments. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[64] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 147–158, New York, NY, USA, 2008. ACM.

[65] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.

[66] Maurice Howard Halstead. *Elements of software science.* Elsevier, 1977.

[67] Colin Higgins, Tarek Hegazy, Pavlos Symeonidis, and Athanasios Tsintsifas. The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies*, 8(3):287–304, 2003.

[68] Colin A. Higgins and Brett Bligh. Formative computer based assessment in diagram based domains. In *Proceedings of the 11th annual SIGCSE conference on Innovation*

and technology in computer science education, ITICSE '06, pages 98–102, New York, NY, USA, 2006. ACM.

[69] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[70] Victoria Hodge and Jim Austin. A Survey of Outlier Detection Methodologies. *Artif. Intell. Rev.*, 22(2):85–126, October 2004.

[71] Andreas Hoffmann, Alexander Quast, and Roland Wismüller. Online-Übungssystem für die Programmierausbildung zur Einführung in die Informatik. In Silke Seehusen, Ulrike Lucke, and Stefan Fischer, editors, *DeLFI 2008, 6. e-Learning Fachtagung Informatik*, volume 132 of *LNI*, pages 173–184. GI, 2008.

[72] Jack Hollingsworth. Automatic graders for programming classes. *Communications of the ACM*, 3(10):528–529, October 1960.

[73] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[74] Petri Ihantola. Creating and Visualizing Test Data from Programming Exercises. *Informatics in Education*, 6(1):81–102, 2007.

[75] Petri Ihantola, Tuuka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.

[76] Petri Ihantola, Ville Karavirta, and Otto Seppälä. Automated visual feedback from programming assignments. In *Proceedings of the 6th Program Visualization Workshop (PVW '11)*, page 87–95, Darmstad, 2011. Technische Universität Darmstad.

[77] David Jackson and Michelle Usher. Grading student programs using ASSYST. *SIGCSE Bull.*, 29(1):335–339, 1997.

[78] Graylin Jay, Joanne E. Hale, Randy K. Smith, David Hale, Nicholas A. Kraft, and Charles Ward. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *Journal of Software Engineering and Applications*, 2(3):137–143, 2009.

[79] Java$^{\text{TM}}$Debug Interface API. `http://java.sun.com/javase/6/docs/jdk/api/jpda/jdi/index.html`.

[80] e-Learning Frameworks and Tools programme, 2007. `http://www.jisc.ac.uk/whatwedo/programmes/elearningframework.aspx`.

[81] Mike Joy, Nathan Griffiths, and Russell Boyatt. The BOSS Online Submission and Assessment System. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 2005.

[82] Java^{TM}Platform Debugging Architecture API. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[83] JUnit Project. `http://sourceforge.net/projects/junit/`.

[84] JVM^{TM}Tool Interface. `http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`.

[85] Cem Kaner and Walter P. Bond. Software Engineering Metrics: What Do They Measure and How Do We Know? In *In METRICS 2004. IEEE CS*. Press, 2004.

[86] Michael Kaufmann and Dorothea Wagner. *Drawing Graphs: Methods and Models.* Springer, 2001.

[87] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology*, 13(4), December 2003.

[88] Carsten Köllmann and Michael Goedicke. Automation of Java Code Analysis for Programming Exercises. In *Proceedings of the Third International Workshop on Graph Based Tools*, volume 1 of *Electronic Communications of the EASST*, 2006.

[89] Carsten Köllmann and Michael Goedicke. A Specification Language for Static Analysis of Student Exercises. In *Proceedings of the International Conference on Automated Software Engineering*, 2008.

[90] Mikko-Jussi Laakso. *Promoting Programming Learning.* PhD thesis, University of Turku, 2010.

[91] Demian Lessa, Jeffrey Czyz, and Bharat Jayaraman. JIVE: A Pedagogic Tool for Visualizing the Execution of Java Programs. Technical Report Technical Report 2010-13, Department of Computer Science and Engineering, University at Buffalo, 2010.

[92] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. In *14th IEEE International Conference on Automated Software Engineering, 1999.*, pages 255–258, 1999.

[93] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969.

[94] Tim A. Majchrzak. *Improving Software Testing*, chapter Testing and E-Assessment, pages 111–126. SpringerBriefs in Information Systems. Springer Berlin Heidelberg, 2012.

[95] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 499–504, New York, NY, USA, 2011. ACM.

[96] David Martín, Emilio Corchado, and Raúl Marticorena. A Code-comparison of Student Assignments based on Neural Visualisation Models. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the First International Conference on Computer Supported Eductation (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 1, pages 47–54. INSTICC, INSTICC Press, 2009.

[97] Mhairi McAlpine. Principles of Assessment. Technical report, Robert Clark Centre for Technological Education, University of Glasgow, February 2002.

[98] Thomas J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering,*, 2(4):308–320, 1976.

[99] Susan A. Mengel and Vinay Yerramilli. A case study of the static analysis of the quality of novice student programs. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, SIGCSE '99, pages 78–82, New York, NY, USA, 1999. ACM.

[100] Derek S. Morris. Automatic grading of student's programming assignments: an interactive process and suite of programs. In *33rd Annual Frontiers in Education*, volume 3, 2003.

[101] Thiemo Morth, Rainer Oechsle, Hermann Schloß, and Markus Schwinn. Automatische Bewertung studentischer Software. In *Workshop "Rechnerunterstütztes Selbststudium in der Informatik", Universität Siegen, 17. September 2007*, 2007.

[102] Edna Holland Mory. *Handbook of research on educational communications and technology*, chapter Feedback research revisited, pages 745–783. Erlbaum Associates, 2004.

[103] Peter Naur. Automatic grading of students' ALGOL programming. *BIT Numerical Mathematics*, 4(3):177–188, 1964.

[104] David J. Nicol and Debra Macfarlane-Dick. Formative assessment and self-regulated learning: a model and seven principles of good feedback practice. *Studies in Higher Education*, 31(2):199–218, 2006.

[105] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, corrected 2nd printing 2005 edition, 1999.

[106] Rainer Oechsle and Thomas Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In Stephan Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 176–190. Springer Berlin Heidelberg, 2002.

[107] OMG. Unified Modeling Language, Superstructure, version 2.4.1, 2012. OMG Document formal/2011-08-06.

[108] Abelardo Pardo. A multi-agent platform for automatic assignment management. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, ITiCSE '02, pages 60–64, New York, NY, USA, 2002. ACM.

[109] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[110] PMD Project. `http://pmd.sourceforge.net/`.

[111] Christian Queinnec. An infrastructure for mechanised grading. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the 2nd International Conference on Computer Supported Education, Valcenia, Spain*, volume 2, pages 37–45. INSTICC, INSTICC Press, 2010.

[112] Ricardo Alexandre Peixoto Queirós and José Paulo Leal. PETCHA: a programming exercises teaching assistant. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE '12, pages 192–197, New York, NY, USA, 2012. ACM.

[113] Khirulnizam Abd Rahman, Syarbaini Ahmad, and Md Jan Nordin. The Design of an Automated C Programming Assessment Using Pseudo-code Comparison Technique. In *National Conference on Software Engineering and Computer Systems*, 2007.

[114] Michael J. Rees. Automatic assessment aids for Pascal programs. *SIGPLAN Not.*, 17(10):33–42, 1982.

[115] Arend Rensink and Eduardo Zambon. A Type Graph Model for Java Programs. Technical report, Formal Methods and Tools Group, EWI-INF, University of Twente, 2009.

[116] Rohaida Romli, Shahida Sulaiman, and Kamal Zuhairi Zamli. Designing a Test Set for Structural Testing in Automatic Programming Assessment. *Int. J. Advance Soft Compu. Appl*, 5(3), 2013.

[117] Manuel Rubio-Sánchez, Päivi Kinnunen, Cristóbal Pareja-Flores, and Ángel Velázquez-Iturbide. Student perception and usage of an automated programming assessment tool. *Computers in Human Behavior*, 2013.

[118] Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 133–136, New York, NY, USA, 2001. ACM.

[119] Tom Schorsch. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, SIGCSE '95, pages 168–172, New York, NY, USA, 1995. ACM.

[120] Joachim Schramm, Sven Strickroth, Nguyen-Thinh Le, and Niels Pinkwart. Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System. In G. M. Youngblood and P. McCarthy, editors, *Proceedings of the 25th International Conference of the Florida Artificial Intelligence Research Society (FLAIRS)*, pages 472–477, Marco Island, FL, USA, 2012. AAAI.

[121] Anuj Shah. Web-CAT: A Web-based Center for Automated Testing. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[122] Mark Sherman, Sarita Bassil, Derrell Lipman, Nat Tuck, and Fred Martin. Impact of Auto-grading on an Introductory Computing Course. *J. Comput. Sci. Coll.*, 28(6):69–75, June 2013.

[123] Valerie J. Shute. Focus on Formative Feedback. *Review of Educational Research*, 78(1):153–189, 2008.

[124] J. Soler, I. Boada, F. Prados, J. Poch, and R. Fabregat. A web-based e-learning tool for UML class diagrams. In *Education Engineering (EDUCON), 2010 IEEE*, pages 973–979, 2010.

[125] Jaime Spacco, David Hovemeyer, William Pugh, Fawzi Emad, Jeffrey K. Hollingsworth, and Nelson Padua-Perez. Experiences with Marmoset: Designing and using an advanced submission and testing system for programming courses. *SIGCSE Bull.*, 38(3):13–17, 2006.

[126] Michael Striewe, Moritz Balz, and Michael Goedicke. A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In *Proceedings of the First International Conference on Computer Supported Education (CSEDU), 23 - 26 March 2009, Lisboa, Portugal*, volume 2, pages 54–61. INSTICC, 2009.

[127] Michael Striewe, Moritz Balz, and Michael Goedicke. Enabling Graph Transformations on Program Code. In *Proceedings of the 4th International Workshop on Graph Based Tools, Enschede, The Netherlands, 2010*, 2010.

[128] Michael Striewe and Michael Goedicke. Effekte automatischer Bewertungen für Programmieraufgaben in Übungs- und Prüfungssituationen. In *DeLFI 2009 - Die 7. E-Learning Fachtagung Informatik*, number 153 in LNI, pages 223–234. GI, 2009.

[129] Michael Striewe and Michael Goedicke. Feedback-Möglichkeiten in automatischen Prüfungssystemen. In *DeLFI 2010 - 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V.*, number 169 in LNI, pages 85–96. GI, 2010.

[130] Michael Striewe and Michael Goedicke. Visualizing Data Structures in an E-Learning System. In *Proceedings of the 2nd International Conference on Computer Supported Education (CSEDU), 07 - 10 April 2010, Valencia, Spain*, volume 1, pages 172–179. INSTICC, 2010.

[131] Michael Striewe, Michael Goedicke, and Moritz Balz. Computer Aided Assessments and Programming Exercises with JACK. Technical Report 28, ICB, University of Duisburg-Essen, 2008.

[132] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(2), February 1981.

[133] Hussein Suleman. Automatic marking with Sakai. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries*, SAICSIT '08, pages 229–236, New York, NY, USA, 2008. ACM.

[134] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In M. Nagel, A. Schürr, and M. Münch, editors, *Application of Graph Transformation with Industrial Relevance: International Workshop, AGTIVE'99*, volume 1779 of *Lecture Notes on Computer Science*, pages 481–488, Kerkrade, The Netherlands, 2000. Springer.

[135] Chung Man Tang, Vuen Tak Yu, and Chung Keung Poon. An experimental prototype for automatically testing student programs using token patterns. In José A. Moinhos Cordeiro, Boris Shishkov, Alexander Verbraeck, and Markus Helfert, editors, *Proceedings of the 2nd International Conference on Computer Supported Education, Valcenia, Spain*, volume 2, pages 144–149. INSTICC, INSTICC Press, 2010.

[136] Pete Thomas. The evaluation of electronic marking of examinations. In *ITiCSE '03: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, pages 50–54, New York, NY, USA, 2003. ACM.

[137] Pete Thomas, Neil Smith, and Kevin Waugh. An approach to the automatic grading of imprecise diagrams. Technical Report 2006/16, The Open University, Walton Hall, Milton Keynes, UK, 2006.

[138] Pete Thomas, Neil Smith, and Kevin Waugh. Automatically assessing graph-based diagrams. *Learning, Media and Technology*, 33(3):249–267, 2008.

[139] Pete Thomas, Kevin Waugh, and Neil Smith. Experiments in the automatic marking of ER-diagrams. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 158–162, New York, NY, USA, 2005. ACM.

[140] Nikolai Tillmann, Jonathan de Halleux, and Tao Xie. Pex for Fun: Engineering an Automated Testing Tool for Serious Games in Computer Science. Technical Report MSR-TR-2011-41, Microsoft Research, 2011.

[141] Nikolai Tillmann, Jonathan de Halleux, Tao Xie, and Judith Bishop. Pex4Fun: A Web-Based Environment for Educational Gaming via Automated Test Generation. In *Proc. 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Tool Demonstrations*, November 2013.

[142] Issa Traore. An Outline of PVS Semantics for UML Statecharts. *Journal of Universal Computer Science*, 6:2000, 2000.

[143] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA, 2007. ACM.

[144] Nghi Truong, Peter Bancroft, and Paul Roe. A Web Based Environment for Learning to Program. In *Proceedings of the 26th Annual Conference of ACSC*, pages 255–264, 2003.

[145] Nghi Truong, Paul Roe, and Peter Bancroft. Static Analysis of Students' Java Programs. In Raymond Lister and Alison L. Young, editors, *Sixth Australasian Computing Education Conference (ACE2004)*, pages 317–325, Dunedin, New Zealand, 2004.

[146] Vincent Tscherter. *Exorciser: Automatic Generation and Interactive Grading of Structured Excercises in the Theory of Computation.* PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland, 2004. Dissertation Nr. 15654.

[147] Athanasios Tsintsifas. *A Framework for the Computer Based Assessment of Diagram Based Coursework.* PhD thesis, University of Nottingham, School of Computer Science and Information Technology, 2002.

[148] Pat Tunstall and Caroline Gsipps. Teacher Feedback to Young Children in Formative Assessment: a typology. *British Educational Research Journal*, 22(4):389–404, 1996.

[149] Esko Ukkonen. Finding Approximate Patterns in Strings. *Journal of Algorithms*, (6):132–137, 1985.

[150] Antti Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, 1992.

[151] Laurens van der Maaten and Geoffrey Hinton. Visualizing Data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

[152] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM, April 2012.

[153] Dániel Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In *Proceedings of International Conference on Graph Transformation (ICGT 2002)*, pages 378–392. Springer, 2002.

[154] Anne Venables and Liz Haywood. Programming students NEED instant feedback! In *Proceedings of the fifth Australasian conference on Computing education - Volume 20*, ACE '03, pages 267–272, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[155] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering Journal*, 10(2), 2003.

[156] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

[157] Urs von Matt. Kassandra: The Automatic Grading System. Technical report, University of Maryland, 1994.

[158] Neil Walkinshaw, Marc Roper, and Murray Wood. The Java system dependence graph. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, pages 55–64, 2003.

[159] Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99–107, 2007.

[160] Colin H. West and Pitro Zafiropulo. Automated Validation of a Communications Protocol: The CCITT X.21 Recommendation. *IBM Journal of Research and Development*, 22(1):60–71, 1978.

[161] Klaus Wissing. Static Analysis of Dynamic Properties - Automatic Program Verification to Prove the Absence of Dynamic Runtime Errors. In *INFORMATIK 2007: Informatik trifft Logistik. Band 2. Beiträge der 37. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 24.-27. September 2007 in Bremen*, volume 110 of *LNI*, pages 275–279. GI, 2007.

[162] Y.T. Yu, C.K. Poon, and M. Choy. Experiences with PASS: Developing and Using a Programming Assignment aSsessment System. *International Conference on Quality Software*, 0:360–368, 2006.

[163] Andreas Zeller. Making Students Read and Review Code. In *Proceedings of the 5th ACM SIGCSE/SIGCUE Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '2000)*, pages 89–92, Helsinki, Finland, 2000.

[164] Albert Zündorf. eDOBS - Java Heap as UML Object Diagram, 2009. `www.se.eecs.uni-kassel.de/se/index.php?edobs`.

A. Materials from GReQL Implementation of Static Checks

A.1. TGraph Schema for Java 7 Syntax

```
TGraph 2;
Schema de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.Java7Schema;
GraphClass Java7;

VertexClass IJavaProject { name:String };
VertexClass IPackageFragment { name:String };

abstract VertexClass Element { line:Integer };
VertexClass AnonymousClassDeclaration:Element;
VertexClass CatchClause:Element;
VertexClass Comment:Element;
VertexClass CompilationUnit:Element { name:String };
VertexClass ImportDeclaration:Element { name:String, onDemand:Boolean, static:Boolean };
VertexClass Javadoc:Element;
VertexClass MemberRef:Element;
VertexClass MemberValuePair:Element;
VertexClass MethodRef:Element;
VertexClass Modifier:Element { name:String };
VertexClass PackageDeclaration:Element { name:String };
VertexClass TagElement:Element { name:String };
VertexClass TextElement:Element { content:String };
VertexClass TypeParameter:Element;

abstract VertexClass BodyDeclaration:Element;
VertexClass AnnotationTypeMemberDeclaration:BodyDeclaration;
VertexClass FieldDeclaration:BodyDeclaration;
VertexClass MethodDeclaration:BodyDeclaration { name:String, constructor:Boolean, dimension:Integer
     };

abstract VertexClass AbstractTypeDeclaration:BodyDeclaration;
VertexClass AnnotationTypeDeclaration:AbstractTypeDeclaration;
VertexClass TypeDeclaration:AbstractTypeDeclaration { name:String, interface:Boolean };

abstract VertexClass Type:Element;
VertexClass ArrayType:Type;
VertexClass ParameterizedType:Type { name:String };
VertexClass PrimitiveType:Type { name:String };
VertexClass SimpleType:Type { name:String };
VertexClass UnionType:Type;
VertexClass WildcardType:Type { upperBound:Boolean };

abstract VertexClass VariableDeclaration:Element;
VertexClass SingleVariableDeclaration:VariableDeclaration { name:String, varargs:Boolean, dimension:
     Integer };
VertexClass VariableDeclarationFragment:VariableDeclaration { name:String, dimension:Integer };

abstract VertexClass Statement:Element;
VertexClass AssertStatement:Statement;
VertexClass Block:Statement;
VertexClass BreakStatement:Statement { name:String };
VertexClass ContinueStatement:Statement { name:String };
VertexClass DoStatement:Statement;
VertexClass EmptyStatement:Statement;
VertexClass EnhancedForStatement:Statement;
VertexClass ExpressionStatement:Statement;
VertexClass ForStatement:Statement;
VertexClass IfStatement:Statement;
VertexClass LabeledStatement:Statement;
VertexClass ReturnStatement:Statement;
VertexClass SuperConstructorInvocation:Statement;
VertexClass SwitchCase:Statement;
VertexClass SwitchStatement:Statement;
VertexClass TryStatement:Statement;
VertexClass VariableDeclarationStatement:Statement;
VertexClass WhileStatement:Statement;

abstract VertexClass Expression:Element;
VertexClass ArrayAccess:Expression;
VertexClass ArrayCreation:Expression;
VertexClass ArrayInitializer:Expression;
VertexClass Assignment:Expression { name:String };
VertexClass BooleanLiteral:Expression { content:String };
VertexClass CastExpression:Expression;
VertexClass CharacterLiteral:Expression { content:String };
VertexClass ClassInstanceCreation:Expression;
VertexClass ConditionalExpression:Expression;
VertexClass FieldAccess:Expression;
VertexClass InfixExpression:Expression { name:String };
VertexClass InstanceofExpression:Expression;
VertexClass MethodInvocation:Expression { name:String };
VertexClass NullLiteral:Expression;
VertexClass NumberLiteral:Expression { content:String };
VertexClass ParenthesizedExpression:Expression;
VertexClass PostfixExpression:Expression { name:String };
VertexClass PrefixExpression:Expression { name:String };
VertexClass StringLiteral:Expression { content:String };
VertexClass SuperFieldAccess:Expression;
VertexClass SuperMethodInvocation:Expression { name:String };
VertexClass ThisExpression:Expression;
VertexClass TypeLiteral:Expression;
VertexClass VariableDeclarationExpression:Expression;

abstract VertexClass Name:Expression;
VertexClass SimpleName:Name { name:String };
VertexClass QualifiedName:Name;

abstract VertexClass Annotation:Expression;
VertexClass MarkerAnnotation:Annotation;
VertexClass NormalAnnotation:Annotation;
VertexClass SingleMemberAnnotation:Annotation;

abstract EdgeClass Child from Vertex (1,1) to Vertex (0,*) { position:Integer } ;
EdgeClass AnnotationTypeDeclarationBodyDeclarations:Child from AnnotationTypeDeclaration (1,1) to
     AnnotationTypeMemberDeclaration (0,*) role bodyDeclarations;
EdgeClass AnnotationTypeDeclarationModifiers1:Child from AnnotationTypeDeclaration (1,1) to Modifier
     (0,*) role modifiers1;
EdgeClass AnnotationTypeDeclarationModifiers2:Child from AnnotationTypeDeclaration (1,1) to
     SingleMemberAnnotation (0,*) role modifiers2;
EdgeClass AnnotationTypeDeclarationName:Child from AnnotationTypeDeclaration (1,1) to SimpleName
     (0,*) role name;
EdgeClass AnnotationTypeMemberDeclarationName:Child from AnnotationTypeMemberDeclaration (1,1) to
     SimpleName (0,*) role name;
EdgeClass AnnotationTypeMemberDeclarationType:Child from AnnotationTypeMemberDeclaration (1,1) to
     ParameterizedType (0,*) role type;
EdgeClass AnonymousClassDeclarationBodyDeclarations:Child from AnonymousClassDeclaration (1,1) to
```

```
    BodyDeclaration (0,*) role bodyDeclarations;
EdgeClass ArrayAccessArray:Child from ArrayAccess (1,1) to Expression (0,*) role array;
EdgeClass ArrayAccessIndex:Child from ArrayAccess (1,1) to Expression (0,*) role index;
EdgeClass ArrayCreationDimensions:Child from ArrayCreation (1,1) to Expression (0,*) role dimensions;
EdgeClass ArrayCreationInitializer:Child from ArrayCreation (1,1) to ArrayInitializer (0,*) role
    initializer;
EdgeClass ArrayCreationType:Child from ArrayCreation (1,1) to ArrayType (0,*) role type;
EdgeClass ArrayInitializerExpressions:Child from ArrayInitializer (1,1) to Expression (0,*) role
    expressions;
EdgeClass ArrayTypeComponentType:Child from ArrayType (1,1) to Type (0,*) role componentType;
EdgeClass AssertStatementMessage:Child from AssertStatement (1,1) to Expression (0,*) role message;
EdgeClass AssignmentLeftHandSide1:Child from Assignment (1,1) to ArrayAccess (0,*) role leftHandSide1
    ;
EdgeClass AssignmentLeftHandSide2:Child from Assignment (1,1) to FieldAccess (0,*) role leftHandSide2
    ;
EdgeClass AssignmentLeftHandSide3:Child from Assignment (1,1) to Name (0,*) role leftHandSide3;
EdgeClass AssignmentRightHandSide:Child from Assignment (1,1) to Expression (0,*) role rightHandSide;
EdgeClass BlockStatements:Child from Block (1,1) to Statement (0,*) role statements;
EdgeClass BodyDeclarationJavadoc from BodyDeclaration (1,1) to Javadoc (0,1) role javadoc;
EdgeClass CastExpressionType:Child from CastExpression (1,1) to Type (0,*) role type;
EdgeClass CatchClauseBody:Child from CatchClause (1,1) to Block (0,*) role body;
EdgeClass CatchClauseException:Child from CatchClause (1,1) to SingleVariableDeclaration (0,*) role
    exception;
EdgeClass ClassInstanceCreationArguments:Child from ClassInstanceCreation (1,1) to Expression (0,*)
    role arguments;
EdgeClass ClassInstanceCreationType:Child from ClassInstanceCreation (1,1) to Type (0,*) role type;
EdgeClass CompilationUnitImports:Child from CompilationUnit (1,1) to ImportDeclaration (0,*) role
    imports;
EdgeClass CompilationUnitPackage:Child from CompilationUnit (1,1) to PackageDeclaration (0,*) role
    package;
EdgeClass CompilationUnitTypes:Child from CompilationUnit (1,1) to AbstractTypeDeclaration (0,*) role
     types;
EdgeClass ConditionalExpressionElseExpression:Child from ConditionalExpression (1,1) to Expression
    (0,*) role elseExpression;
EdgeClass ConditionalExpressionThenExpression:Child from ConditionalExpression (1,1) to Expression
    (0,*) role thenExpression;
EdgeClass DoStatementBody:Child from DoStatement (1,1) to Block (0,*) role body;
EdgeClass ElementExpression:Child from Element (1,1) to Expression (0,*) role expression;
EdgeClass EnhancedForStatementBody:Child from EnhancedForStatement (1,1) to Statement (0,*) role body
    ;
EdgeClass EnhancedForStatementParameter:Child from EnhancedForStatement (1,1) to
    SingleVariableDeclaration (0,*) role parameter;
EdgeClass FieldAccessName:Child from FieldAccess (1,1) to SimpleName (0,*) role name;
EdgeClass FieldDeclarationFragments:Child from FieldDeclaration (1,1) to VariableDeclarationFragment
    (0,*) role fragments;
EdgeClass FieldDeclarationModifiers:Child from FieldDeclaration (1,1) to Modifier (0,*) role
    modifiers;
EdgeClass FieldDeclarationType:Child from FieldDeclaration (1,1) to Type (0,*) role type;
EdgeClass ForStatementBody:Child from ForStatement (1,1) to Statement (0,*) role body;
EdgeClass ForStatementInitializers:Child from ForStatement (1,1) to Expression (0,*) role
    initializers;
EdgeClass ForStatementUpdaters:Child from ForStatement (1,1) to Expression (0,*) role updaters;
EdgeClass IfStatementElseStatement:Child from IfStatement (1,1) to Statement (0,*) role elseStatement
    ;
EdgeClass IfStatementThenStatement:Child from IfStatement (1,1) to Statement (0,*) role thenStatement
    ;
EdgeClass IJavaProjectFragements:Child from IJavaProject (1,1) to IPackageFragment (0,*) role
    fragments;
EdgeClass IJavaProjectUnits:Child from IJavaProject (1,1) to CompilationUnit (0,*) role units;
EdgeClass InfixExpressionExtendedOperands:Child from InfixExpression (1,1) to Expression (0,*) role
    extendedOperands;
EdgeClass InfixExpressionLeftOperand:Child from InfixExpression (1,1) to Expression (0,*) role
    leftOperand;
EdgeClass InfixExpressionRightOperand:Child from InfixExpression (1,1) to Expression (0,*) role
    rightOperand;
```

```
EdgeClass InstanceofExpressionLeftOperand:Child from InstanceofExpression (1,1) to Type (0,*) role
     leftOperand;
EdgeClass InstanceofExpressionRightOperand:Child from InstanceofExpression (1,1) to Type (0,*) role
     rightOperand;
EdgeClass IPackageFragmentUnits:Child from IPackageFragment (1,1) to CompilationUnit (0,*) role units
     ;
EdgeClass JavadocTags:Child from Javadoc (1,1) to TagElement (0,*) role tags;
EdgeClass LabeledStatementBody:Child from LabeledStatement (1,1) to Statement (0,*) role body;
EdgeClass LabeledStatementLabel:Child from LabeledStatement (1,1) to SimpleName (0,*) role label;
EdgeClass MarkerAnnotationTypeName:Child from MarkerAnnotation (1,1) to SimpleName (0,*) role
     typeName;
EdgeClass MemberValuePairName:Child from MemberValuePair (1,1) to SimpleName (0,*) role name;
EdgeClass MemberValuePairValue:Child from MemberValuePair (1,1) to TypeLiteral (0,*) role value;
EdgeClass MethodDeclarationBody:Child from MethodDeclaration (1,1) to Block (0,*) role body;
EdgeClass MethodDeclarationModifiers1:Child from MethodDeclaration (1,1) to MarkerAnnotation (0,*)
     role modifiers1;
EdgeClass MethodDeclarationModifiers2:Child from MethodDeclaration (1,1) to Modifier (0,*) role
     modifiers2;
EdgeClass MethodDeclarationModifiers3:Child from MethodDeclaration (1,1) to NormalAnnotation (0,*)
     role modifiers3;
EdgeClass MethodDeclarationParameters:Child from MethodDeclaration (1,1) to SingleVariableDeclaration
      (0,*) role parameters;
EdgeClass MethodDeclarationReturnType:Child from MethodDeclaration (1,1) to Type (0,*) role
     returnType2;
EdgeClass MethodDeclarationThrownExceptions:Child from MethodDeclaration (1,1) to SimpleName (0,*)
     role thrownExceptions;
EdgeClass MethodInvocationArguments:Child from MethodInvocation (1,1) to Expression (0,*) role
     arguments;
EdgeClass NormalAnnotationTypeName:Child from NormalAnnotation (1,1) to SimpleName (0,*) role
     typeName;
EdgeClass NormalAnnotationValues:Child from NormalAnnotation (1,1) to MemberValuePair (0,*) role
     values;
EdgeClass ParameterizedTypeType:Child from ParameterizedType (1,1) to SimpleType (0,*) role type;
EdgeClass ParameterizedTypeTypeArguments1:Child from ParameterizedType (1,1) to SimpleType (0,*) role
      typeArguments1;
EdgeClass ParameterizedTypeTypeArguments2:Child from ParameterizedType (1,1) to WildcardType (0,*)
     role typeArguments2;
EdgeClass PostfixExpressionOperand:Child from PostfixExpression (1,1) to SimpleName (0,*) role
     operand;
EdgeClass PrefixExpressionOperand:Child from PrefixExpression (1,1) to Expression (0,*) role operand;
EdgeClass QualifiedNameName:Child from QualifiedName (1,1) to SimpleName (0,*) role name;
EdgeClass QualifiedNameQualifier:Child from QualifiedName (1,1) to Name (0,*) role qualifier;
EdgeClass SimpleNameValues:Child from SimpleName (1,1) to MethodDeclaration (0,*) role values;
EdgeClass SingleMemberAnnotationTypeName:Child from SingleMemberAnnotation (1,1) to SimpleName (0,*)
     role typeName;
EdgeClass SingleMemberAnnotationValue:Child from SingleMemberAnnotation (1,1) to QualifiedName (0,*)
     role value;
EdgeClass SingleVariableDeclarationType:Child from SingleVariableDeclaration (1,1) to Type (0,*) role
      type;
EdgeClass SuperConstructorInvocationArgument:Child from SuperConstructorInvocation (1,1) to
     Expression (0,*) role arguments;
EdgeClass SuperFieldAccessName:Child from SuperFieldAccess (1,1) to SimpleName (0,*) role name;
EdgeClass SuperMethodInvocationArguments:Child from SuperMethodInvocation (1,1) to Expression (0,*)
     role arguments;
EdgeClass SwitchStatementStatements:Child from SwitchStatement (1,1) to Statement (0,*) role
     statements;
EdgeClass TagElementFragments:Child from TagElement (1,1) to TextElement (0,*) role fragments;
EdgeClass TryStatementBody:Child from TryStatement (1,1) to Statement (0,*) role body;
EdgeClass TryStatementCatchClauses:Child from TryStatement (1,1) to CatchClause (0,*) role
     catchClauses;
EdgeClass TryStatementResources:Child from TryStatement (1,1) to VariableDeclarationExpression (0,*)
     role resources;
EdgeClass TypeDeclarationBodyDeclarations1:Child from TypeDeclaration (1,1) to FieldDeclaration (0,*)
      role bodyDeclarations1;
EdgeClass TypeDeclarationBodyDeclarations2:Child from TypeDeclaration (1,1) to MethodDeclaration
```

```
     (0,*) role bodyDeclarations2;
EdgeClass TypeDeclarationModifiers:Child from TypeDeclaration (1,1) to Modifier (0,*) role modifiers;
EdgeClass TypeDeclarationSuperclassType1:Child from TypeDeclaration (1,1) to ParameterizedType (0,1)
     role superclassType1;
EdgeClass TypeDeclarationSuperclassType2:Child from TypeDeclaration (1,1) to SimpleType (0,1) role
     superclassType2;
EdgeClass TypeDeclarationSuperInterfaceTypes1:Child from TypeDeclaration (1,1) to ParameterizedType
     (0,*) role superInterfaceTypes1;
EdgeClass TypeDeclarationSuperInterfaceTypes2:Child from TypeDeclaration (1,1) to SimpleType (0,*)
     role superInterfaceTypes2;
EdgeClass TypeDeclarationTypeParameters:Child from TypeDeclaration (1,1) to TypeParameter (0,*) role
     typeParameters;
EdgeClass TypeLiteralType:Child from TypeLiteral (1,1) to Type (0,*) role type;
EdgeClass TypeParameterName:Child from TypeParameter (1,1) to SimpleName (0,*) role name;
EdgeClass UnionTypeTypes:Child from UnionType (1,1) to Type (0,*) role types;
EdgeClass VariableDeclarationExpressionFragments:Child from VariableDeclarationExpression (1,1) to
     VariableDeclarationFragment (0,*) role fragments;
EdgeClass VariableDeclarationExpressionType:Child from VariableDeclarationExpression (1,1) to Type
     (0,*) role type;
EdgeClass VariableDeclarationFragmentInitializer:Child from VariableDeclarationFragment (1,1) to
     Expression (0,*) role initializer;
EdgeClass VariableDeclarationStatementFragments:Child from VariableDeclarationStatement (1,1) to
     VariableDeclarationFragment (0,*) role fragments;
EdgeClass VariableDeclarationStatementType:Child from VariableDeclarationStatement (1,1) to Type
     (0,*) role type;
EdgeClass WhileStatementBody:Child from WhileStatement (1,1) to Statement (0,*) role body;
EdgeClass WildcardTypeBound1:Child from WildcardType (1,1) to ParameterizedType (0,*) role bound1;
EdgeClass WildcardTypeBound2:Child from WildcardType (1,1) to SimpleType (0,*) role bound2;

abstract EdgeClass Access from Vertex (1,1) to Vertex (0,*);
EdgeClass ExpressionAccess1:Access from Expression (1,1) to BodyDeclaration (0,*) role access1;
EdgeClass ExpressionAccess2:Access from Expression (1,1) to VariableDeclaration (0,*) role access2;
EdgeClass TypeAccess:Access from Type (1,1) to TypeDeclaration (0,*) role access;
```

A.2. Additional Function for Signature Checks of Java Methods

```
/*
 * JGraLab - The Java graph laboratory
 * (c) 2006-2010 Institute for Software Technology
 * University of Koblenz-Landau, Germany
 *
 * ist@uni-koblenz.de
 *
 * Please report bugs to http://serres.uni-koblenz.de/bugzilla
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */
/**
 *
 */
package de.uni_due.s3.jack2.backend.checkers.greqljavachecker.morefunctions;

import java.util.ArrayList;

import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     ArrayTypeComponentType;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.MethodDeclaration;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     MethodDeclarationModifiers2;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     MethodDeclarationParameters;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     MethodDeclarationReturnType;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     MethodDeclarationThrownExceptions;
import de.uni_due.s3.jack2.backend.checkers.greqljavachecker.schema.java7.schema.
     SingleVariableDeclarationType;
import de.uni_koblenz.jgralab.AttributedElement;
import de.uni_koblenz.jgralab.Edge;
import de.uni_koblenz.jgralab.EdgeDirection;
import de.uni_koblenz.jgralab.Graph;
import de.uni_koblenz.jgralab.NoSuchAttributeException;
import de.uni_koblenz.jgralab.Vertex;
import de.uni_koblenz.jgralab.graphmarker.AbstractGraphMarker;
import de.uni_koblenz.jgralab.greql2.exception.EvaluateException;
import de.uni_koblenz.jgralab.greql2.exception.WrongFunctionParameterException;
import de.uni_koblenz.jgralab.greql2.funlib.Greql2Function;
import de.uni_koblenz.jgralab.greql2.jvalue.JValue;
import de.uni_koblenz.jgralab.greql2.jvalue.JValueImpl;
import de.uni_koblenz.jgralab.greql2.jvalue.JValueType;

/**
 * Checks whether a method has a given signature.
 *
 * <dl>
 * <dt><b>GReQL-signature</b></dt>
 * <dd><code>BOOL hasSignature(node:VERTEX, sig:STRING)</code></dd>
 * <dd>&nbsp;</dd>
 * </dl>
 * <dl>
 * <dt></dt>
 * <dd>
 * <dl>
 * <dt><b>Parameters:</b></dt>
 * <dd><code>node</code> - the method declaration to be evaluated</dd>
 * <dd><code>sig</code> - the expected signature</dd>
 * <dt><b>Returns:</b></dt>
 * <dd>True iff node defines a method with signature sig, false otherwise
 * </dl>
 * </dd>
 * </dl>
 *
 * @author michael.striewe@paluno.uni-due.de
 *
 */
public class HasSignature extends Greql2Function {
    {
        JValueType[][] x = { { JValueType.VERTEX, JValueType.STRING, JValueType.BOOL } };
        signatures = x;

        description = "Checks whether a method has a given signature.";

        Category[] c = { Category.COMPARISONS };
        categories = c;
    }

    @Override
    public JValue evaluate(Graph graph,
            AbstractGraphMarker<AttributedElement> subgraph, JValue[] arguments)
            throws EvaluateException {
        if (checkArguments(arguments) < 0) {
            throw new WrongFunctionParameterException(this, arguments);
        }

        Vertex v = arguments[0].toVertex();

        if (!v.getM1Class().equals(MethodDeclaration.class))
            return new JValueImpl(false); // Vertex is no method declaration

        String signature = arguments[1].toString();
        String sigName[] = signature.substring(0, signature.indexOf("(")).split(" ");
        String sigArguments[] = signature.substring(signature.indexOf("(")+1, signature.indexOf(")"))
             .split(",");
        String sigExceptions[] = new String[0];

        if (signature.indexOf("throws") > -1)
            sigExceptions = signature.substring(signature.indexOf("throws")+6).split(",");

        if (!v.getAttribute("name").equals(sigName[sigName.length-1].trim()))
            return new JValueImpl(false); // Method declaration has wrong name

        if (v.getAttribute("constructor").equals("false") && sigName.length==1)
            return new JValueImpl(false); // Method declaration is no constructor, but should be

        if (v.getAttribute("constructor").equals("false") && sigName.length==2 && (sigName[0].trim().
             equals("public") || sigName[0].trim().equals("private") || sigName[0].trim().equals("
             protected")))
            return new JValueImpl(false); // Method declaration is no constructor, but should be
```

```
if (v.getAttribute("constructor").equals("false") && sigName.length>=2 && !hasTypeName(v.
     getFirstEdgeOfClass(MethodDeclarationReturnType.class).getOmega(), sigName[sigName.length
     -2].trim()))
    return new JValueImpl(false); // Method declaration has wrong return type

if (v.getAttribute("constructor").equals("false")) {
    modifiers: for (int i = sigName.length-3; i >=0; i--) {
        Edge e = v.getFirstEdgeOfClass(MethodDeclarationModifiers2.class);
        while (e != null) {
            if (e.getOmega().getAttribute("name").equals(sigName[i].trim()))
                continue modifiers;
            e = e.getNextEdgeOfClass(MethodDeclarationModifiers2.class);
        }
        return new JValueImpl(false); // Method declaration misses a modifier
    }
}

if (sigArguments.length > 0 && sigArguments[0].trim().length() > 0) {
    if (v.getDegree(MethodDeclarationParameters.class) != sigArguments.length)
        return new JValueImpl(false); // Method declaration has wrong number of parameters

    for (int i = 0; i < sigArguments.length; i++) {
        String[] parameterParts = sigArguments[i].trim().split(" ");
        Edge e = v.getFirstEdgeOfClass(MethodDeclarationParameters.class);
        boolean found = false;
        while (e != null) {
            if (Integer.parseInt(e.getAttribute("position").toString()) == i) {
                found = true;

                if (parameterParts.length == 2 && !e.getOmega().getAttribute("name").equals(
                     parameterParts[1]))
                    return new JValueImpl(false); // Method declaration misses a parameter by
                         name
                if (!hasTypeName(e.getOmega().getFirstEdgeOfClass(SingleVariableDeclarationType
                     .class).getOmega(), parameterParts[0]))
                    return new JValueImpl(false); // Method declaration misses a parameter by
                         type
            }
            e = e.getNextEdgeOfClass(MethodDeclarationParameters.class);
        }
        if (!found)
            return new JValueImpl(false); // Method declaration misses a parameter
    }
}

if (sigExceptions.length > 0 && sigExceptions[0].trim().length() > 0) {
    if (v.getDegree(MethodDeclarationThrownExceptions.class) != sigArguments.length) {
        System.out.println("Exception check 1 failed!");
        return new JValueImpl(false); // Method declaration has wrong number of declared
             exceptions
    }
    for (int i = 0; i < sigExceptions.length; i++) {
        String exception = sigExceptions[i].trim();
        Edge e = v.getFirstEdgeOfClass(MethodDeclarationThrownExceptions.class);
        boolean found = false;
        while (e != null) {
            if (e.getOmega().getAttribute("name").equals(exception)) {
                found = true;
                break;
            }
        }

        if (!found)
            return new JValueImpl(false); // Method declaration misses a declared exception
    }
```

```
        }

        return new JValueImpl(true);
    }

    private boolean hasTypeName(Vertex v, String name) {
        String aName = "";

        try {
            aName = (String)v.getAttribute("name");
        } catch (NoSuchAttributeException nsae) {
            // OK, no attribute name
        }

        if (aName.equals(name))
            return true;
        if (v.getFirstEdgeOfClass(ArrayTypeComponentType.class, EdgeDirection.OUT) != null && name.
             endsWith("[]"))
            return hasTypeName(v.getFirstEdgeOfClass(ArrayTypeComponentType.class, EdgeDirection.OUT).
                 getOmega(), name.substring(0, name.length()-2));
        return false;
    }

    @Override
    public long getEstimatedCardinality(int inElements) {
        return 1;
    }

    @Override
    public long getEstimatedCosts(ArrayList<Long> inElements) {
        return 2;
    }

    @Override
    public double getSelectivity() {
        return 1;
    }

}
```

A.3. TGraph Schema for parts of UML2

```
TGraph 2;
Schema de.uni_due.s3.jack2.backend.checkers.greqlumlchecker.schema.uml2.schema.UML2Schema;
GraphClass UML2;

abstract VertexClass Element;
abstract VertexClass Feature { isStatic:Boolean };


abstract VertexClass TemplateableElement:Element;
abstract VertexClass NamedElement:TemplateableElement { name:String, visibility:String };
abstract VertexClass MultiplicityElement:Element { isOrdered:Boolean, isUnique:Boolean, lower:String,
      upper:String };
abstract VertexClass TemplateParameter:Element;
abstract VertexClass ActivityGroup:Element;
VertexClass PackageableElement:NamedElement;
abstract VertexClass BehavioralFeature:NamedElement {isLeaf:Boolean, isStatic:Boolean, isAbstract:
     Boolean, concurrency:String };
VertexClass Generalization:Element;
VertexClass TemplateSignature:Element;
VertexClass Pseudostate:NamedElement {outgoing:String, incoming:String, kind:String};
VertexClass ExceptionHandler:Element {handlerBody:String, exceptionInput:String, exceptionType:String
     };
VertexClass Comment:TemplateableElement {body:String, annotatedElement:String};

VertexClass Clause:Element {test:String, body:String, predecessorClause:String, successorClause:
     String, decider:String, bodyOutput:String};

abstract VertexClass Type:PackageableElement;
abstract VertexClass TypedElement:NamedElement;

VertexClass InstanceSpecification:PackageableElement {classifier:String};

VertexClass Dependency:PackageableElement;
VertexClass Enumeration:DataType;
VertexClass PrimitiveType:DataType;
VertexClass Usage:Dependency;
VertexClass Package:PackageableElement;
VertexClass Constraint:PackageableElement {constrainedElement:String};
VertexClass EnumerationLiteral:PackageableElement;
VertexClass ClassifierTemplateParameter:TemplateParameter;

abstract VertexClass Classifier:Type { isAbstract:Boolean, isLeaf:Boolean, templateParameter:String,
     packageableElement_visibility:String, redefinedClassifier:String, powertypeExtent:String,
     useCase:String, representation:String};
VertexClass DataType:Classifier;
VertexClass Abstraction:Dependency;
VertexClass BehavioredClassifier:Classifier {classifierBehavior:String};
VertexClass Class:BehavioredClassifier { isActive:Boolean };
VertexClass Interface:Classifier;
VertexClass AssociationClass:Class;
VertexClass Realization:Abstraction;
VertexClass Stereotype:Class;

abstract VertexClass StructuralFeature:TypedElement,MultiplicityElement,Feature;
VertexClass Property:StructuralFeature { isReadOnly:Boolean, defaultValue:String, isComposite:Boolean
     , isDerived:Boolean, isDerivedUnion:Boolean, aggregation:String };

VertexClass Port:Property {isBehavior:String, isService:String, redefinedPort:String, protocol:String
     };
VertexClass ExtensionEnd;

abstract VertexClass Relationship;
```

```
VertexClass Association:Relationship,Classifier { isDerived:Boolean };

VertexClass Operation:BehavioralFeature { isQuery:Boolean };
VertexClass Parameter:TypedElement,MultiplicityElement { defaultValue:String, direction:String,
     isStream:Boolean, isException:Boolean };
VertexClass OpaqueExpression:TypedElement {body:String, language:String };

VertexClass Extension:Association;

VertexClass Trigger:NamedElement {port:String};
VertexClass ChangeTrigger:Trigger;
VertexClass MessageTrigger:Trigger;
VertexClass TimeTrigger:Trigger {isRelative:Boolean};
VertexClass AnyTrigger:MessageTrigger;
VertexClass CallTrigger:MessageTrigger {operation:String};
VertexClass SignalTrigger:MessageTrigger {signal:String};

VertexClass Behavior:Class {isReentrant:Boolean, redefinedBehavior:String, specification:String,
     precondition:String, postcondition:String};
VertexClass Activity:Behavior { isSingleExecution:Boolean, isReadOnly:Boolean, body:String, language:
     String, action:String};
VertexClass StateMachine:Behavior {extendedStateMachine:String};
VertexClass RedefinableElement:NamedElement {isLeaf:Boolean};
VertexClass RedefinableTemplateSignature:RedefinableElement;
VertexClass ActivityEdge:RedefinableElement {guard:String, weight:String, source:String, target:
     String, redefinedElement:String, inPartition:String, interrupts:String};
VertexClass ActivityNode:RedefinableElement {outgoing:String, incoming:Boolean, redefinedElement:
     String, inPartition:String, inInterruptibleRegion:String};
VertexClass ControlNode:ActivityNode;
VertexClass ExecutableNode:ActivityNode;
VertexClass ObjectNode:ActivityNode {type:String, ordering:String, inState:String, selection:String};

VertexClass ProtocolStateMachine:StateMachine;

VertexClass FinalNode:ControlNode;
VertexClass InitialNode:ControlNode;
VertexClass DecisionNode:ControlNode {decisionInput:String};
VertexClass ForkNode:ControlNode;
VertexClass MergeNode:ControlNode;
VertexClass JoinNode:ControlNode {isCombineDuplicate:Boolean};
VertexClass Action:ExecutableNode {effect:String};
VertexClass Pin:ObjectNode {isOrdered:String, isUnique:Boolean};
VertexClass ExpansionNode:ObjectNode {regionAsOutput:String, regionAsInput:String};
VertexClass CentralBufferNode:ObjectNode;
VertexClass ActivityParameterNode:ObjectNode {parameter:String};

VertexClass ActivityFinalNode:FinalNode;
VertexClass FlowFinalNode:FinalNode;
VertexClass DataStoreNode:CentralBufferNode;
VertexClass InputPin:Pin;
VertexClass OutputPin:Pin;

VertexClass AcceptEventAction:Action {trigger:String, result:String};
VertexClass ApplyFunktionAction:Action {function:String};
VertexClass ClearAssociationAction:Action {association:String};
VertexClass CreateObjectAction:Action {classifier:String};
VertexClass DestroyObjectAction:Action {isDestroyLink:Boolean, isDestroyOwnedObjects:Boolean};
VertexClass InvocationAction:Action {onPort:String};
VertexClass LinkAction:Action;
VertexClass RaiseExceptionAction:Action {exception:String};
VertexClass ReadExtentAction:Action {classifier:String};
VertexClass ReadIsClassifiedObjectAction:Action {isDirect:Boolean, classifier:String};
VertexClass ReadLinkObjectEndAction:Action {end:String};
VertexClass ReadLinkObjectEndQualifierAction:Action {qualifier:String};
VertexClass ReadSelfAction:Action {isReplaceAll:Boolean, oldClassifier:String, newClassifier:String};
VertexClass ReclassifyObjectAction:Action {isReplaceAll:Boolean, oldClassifier:String, newClassifier:
     String};
VertexClass ReplyAction:Action {replyToCall:String, replyValue:String, returnInformation:String};
VertexClass StartOwnedBehaviorAction:Action;
VertexClass StructuralFeatureAction:Action {structuralFeature:String};
VertexClass StructuredActivityNode:Action {mustIsolate:Boolean};
VertexClass TestIdentityAction:Action;
VertexClass VariableAction:Action {variable:String};

VertexClass AcceptCallAction:AcceptEventAction {returnInformation:String};
VertexClass BroadcastSignalAction:InvocationAction {signal:String};
VertexClass CallAction:InvocationAction {isSynchronous:Boolean};
VertexClass SendObjectAction:InvocationAction;
VertexClass SendSignalAction:InvocationAction {signal:String};
VertexClass ReadLinkAction:LinkAction;
VertexClass WriteLinkAction:LinkAction;
VertexClass ClearStructuralFeatureAction:StructuralFeatureAction;
VertexClass ReadStructuralFeatureAction:StructuralFeatureAction;
VertexClass WriteStructuralFeatureAction:StructuralFeatureAction;
VertexClass ConditionalNode:StructuredActivityNode {isDeterminate:Boolean, isAssured:Boolean};
VertexClass ExpansionRegion:StructuredActivityNode {mode:String, outputElement:String, inputElement:
     String};
VertexClass LoopNode:StructuredActivityNode {isTestedFirst:Boolean, bodyPart:String, setupPart:String
     , decider:String, test:String, bodyOutput:String};
VertexClass ClearVariableAction:VariableAction;
VertexClass ReadVariableAction:VariableAction;
VertexClass WriteVariableAction:VariableAction;

VertexClass AddVariableValueAction:WriteVariableAction {isReplaceAll:Boolean};
VertexClass RemoveVariableValueAction:WriteVariableAction;
VertexClass AddStructuralFeatureValueAction:WriteStructuralFeatureAction {isReplaceAll:Boolean};
VertexClass RemoveStructuralFeatureValueAction:WriteStructuralFeatureAction;
VertexClass TimeObservationAction:WriteStructuralFeatureAction;
VertexClass DurationObservationAction:WriteStructuralFeatureAction;
VertexClass CreateLinkAction:WriteLinkAction;
VertexClass DestroyLinkAction:WriteLinkAction;
VertexClass CallBehaviorAction:CallAction {behavior:String};
VertexClass CallOperationAction:CallAction {operation:String};

VertexClass CreateLinkObjectAction:CreateLinkAction;
VertexClass InterruptibleActivityRegion:ActivityGroup {interruptingEdge:String, containedNode:String
     };

VertexClass ControlFlow:ActivityEdge;
VertexClass ObjectFlow:ActivityEdge {isMulticast:Boolean, isMultireceive:Boolean, transformation:
     String, selection:String};

VertexClass ActivityPartition:NamedElement {isDimension:Boolean, isExternal:Boolean, containedEdge:
     String, containedNode:String, represents:String};

EdgeClass ElementExceptionEdge from Element (0,1) to ExceptionHandler (0,*);
EdgeClass OwnedOperationEdge from Classifier (0,1) role classifier to Operation (0,*) role
     ownedOperation;
EdgeClass SuperClassEdge from Class (0,*) to Class (0,*) role superClass;
EdgeClass NestedClassifierEdge from Classifier (1,1) to Classifier (0,*) role nestedClassifier;
EdgeClass OwnedAttributeEdge from Classifier (0,1) role owningClassifier to Property (0,*) role
     ownedAttribute;
EdgeClass TypeEdge from TypedElement (0,*) role typedElement to Type (0,1) role type;
EdgeClass OwnedParameterEdge from Operation (0,1) role operation to Parameter (0,*) role
     ownedParameter;
EdgeClass AssociationEdge from Property (2,*) role property to Association (0,1) role association;
EdgeClass OwnedEndEdge from Classifier (0,1) role ownedEndClassifier to Property (0,*) role ownedEnd;
EdgeClass MemberEndEdge from Classifier (0,1) role memberEndClassifier to Element (0,*) role
     memberEnd;
EdgeClass AssociationClassEdge from Property (1,*) role associationProperty to AssociationClass (1,1)
```

```
      role associationClass;
EdgeClass PackagedElementEdge from Package (0,1) role owningPackage to PackageableElement (0,*) role
     packagedElement;
EdgeClass OwnedRuleEdge from Package (0,1) role ruleOwningPackage to Constraint (0,*) role ownedRule;
EdgeClass ConstrainedElementEdge from Constraint (0,*) role constraint to Element (0,*) role
     constrainedElement2;
EdgeClass SpecificationEdge from Constraint (0,1) role specifiedConstraint to OpaqueExpression (0,*)
     role specificationExpression;
EdgeClass ActivityEdgeOpaqueEdge from ActivityEdge (0,1) to OpaqueExpression (0,*);
EdgeClass NestedPackageEdge from Package (0,1) role nestingPackage to Package (0,*) role
     nestedPackage;
EdgeClass GeneralizationEdge from Classifier (1,1) role specific to Generalization (0,*) role
     generalization;
EdgeClass GeneralEdge from Generalization (0,*) to Classifier (1,1) role general;
EdgeClass ClientEdge from Dependency (1,1) to Classifier (1,1) role client;
EdgeClass ClientActivityNodeEdge from Dependency (1,1) to ActivityNode (1,1);
EdgeClass SupplierEdge from Dependency (1,1) to Classifier (1,1) role supplier;
EdgeClass SupplierActivityNodeEdge from Dependency (1,1) to ActivityNode (1,1);
EdgeClass ActivityEdgeSourceEdge from ActivityEdge (1,1) to NamedElement (1,1) role source1;
EdgeClass ActivityEdgeTargetEdge from ActivityEdge (1,1) to NamedElement (1,1) role target2;
EdgeClass ActivityPartitionContainmentEdge from ActivityPartition (1,1) to Element (1,1);
EdgeClass InterruptibleActivityRegionContainmentEdge from InterruptibleActivityRegion (1,1) to
     Element (1,1);
EdgeClass InterruptingEdgeEdge from InterruptibleActivityRegion (1,1) to ActivityEdge (1,1);
EdgeClass ExpansionRegionContainmentEdge from ExpansionRegion (1,1) to Element (1,1);
EdgeClass ConditionalNodeContainmentEdge from ConditionalNode (1,1) to Element (1,1);
EdgeClass LoopNodeContainmentEdge from LoopNode (1,1) to Element (1,1);
EdgeClass StructuredActivityNodeContainmentEdge from StructuredActivityNode (1,1) to Element (1,1);
EdgeClass ActivityContainmentNodeEdge from Activity (0,1) to ActivityNode (1,1) role node;
EdgeClass ActivityContainmentRegionEdge from Activity (0,1) to InterruptibleActivityRegion (1,1);
EdgeClass ActivityContainmentTriggerEdge from Activity (0,1) to Trigger (1,1);
EdgeClass ActivityContainmentActivityPartitionEdge from Activity (0,1) to ActivityPartition (1,1);
EdgeClass ActivityContainmentExceptionHandlerEdge from Activity (0,1) to ExceptionHandler (1,1);
EdgeClass ActivityContainmentPseudoEdge from Activity (0,1) to Pseudostate (1,1);
EdgeClass ActivityContainmentActivityEdge from Activity (0,1) to Activity (1,1);
EdgeClass ActivityContainmentPortEdge from Activity (0,1) to Port(1,1);
EdgeClass ActivityContainmentSpecificationEdge from Activity (0,1) to InstanceSpecification(1,1);
EdgeClass ActivityContainmentBehaviorEdge from Activity (0,1) to Behavior(1,1);
EdgeClass AcceptEventActionTriggerEdge from AcceptEventAction (0,1) to Trigger (1,1);
EdgeClass InterruptibleActivityRegionContainmentNodeEdge from InterruptibleActivityRegion (0,1) to
     ActivityNode (1,1) role node;
EdgeClass ActivityNodeContainmentNodeEdge from ActivityNode (0,1) to ActivityNode (1,1) role node;
EdgeClass ActivityNodeContainmentPinEdge from ActivityNode (0,1) to Pin (1,1);
EdgeClass ActivityNodeContainmentTriggerEdge from ActivityNode (0,1) to Trigger (1,1);
EdgeClass ActivityNodeContainmentExceptionHandlerEdge from ActivityNode (0,1) to ExceptionHandler
     (1,1);
EdgeClass ActivityNodeContainmentPseudoEdge from ActivityNode (0,1) to Pseudostate (1,1);
EdgeClass ActivityNodeContainmentPortEdge from ActivityNode (0,1) to Port(1,1);
EdgeClass ActivityNodeContainmentSpecificationEdge from ActivityNode (0,1) to InstanceSpecification
     (1,1);
EdgeClass ActivityNodeContainmentBehaviorEdge from ActivityNode (0,1) to Behavior(1,1);
EdgeClass PackageContainmentPortEdge from PackageableElement (0,1) to Port (1,1);
EdgeClass PackageContainmentEdgeEdge from Package (0,1) to ActivityEdge (1,1) role edge;
EdgeClass ActivityContainmentEdgeEdge from Activity (0,1) to ActivityEdge (1,1) role edge;
EdgeClass ActivityPartitionContainmentEdgeEdge from ActivityPartition (0,1) to ActivityEdge (1,1);
EdgeClass ExceptionHandlerContainmentActivityEdge from ExceptionHandler (0,1) to Activity (1,1);
EdgeClass ExceptionHandlerContainmentNodeEdge from ExceptionHandler (0,1) to ActivityNode (1,1);
EdgeClass OwnedRuleRedefinableEdge from RedefinableElement (0,1) to Constraint (0,*);
EdgeClass OwnedLiteralEdge from Enumeration (0,1) role owningEnumeration to EnumerationLiteral (0,*)
     role enumerationLiteral;
EdgeClass OwnedTemplateSignatureEdge from Classifier (0,1) role templateClassifier to Element (0,*)
     role ownedTemplateSignature;
EdgeClass OwnedClassifierTemplateParameterEdge from Element (0,1) role templateSignature to
     ClassifierTemplateParameter (0,*) role ownedClassifierTemplateParameter;
EdgeClass OwnedParameterElementEdge from ClassifierTemplateParameter (0,1) role
```

```
     classifierTemplateParameter to Type (0,*) role ownedParameterElement;
EdgeClass OwnedDataTypeMemberEdge from Package (0,1) to DataType (0,*) role ownedDataTypeMember;
EdgeClass ElementImportEdge from PackageableElement (0,1) to PackageableElement (0,*) role
     importedElement;
```

A.4. Ruleset for UML2 Diagram Exercise

```
<umlcheckerrules>
    <rule type="presence" points="4">
        <query>from x : V{Class}, y : V{Property} with x --> y and x.name="Tarif" and (
             capitalizeFirst(y.name)="Name" or capitalizeFirst(y.name)="Bezeichnung") report x.name, y
             .name end</query>
        <feedback>Das Diagramm enthaelt keine Klasse "Tarif" mit der Eigenschaft "Name" oder "
             Bezeichnung".</feedback>
    </rule>
    <rule type="presence" points="4">
        <query>from x : V{Class}, y : V{Property} with x --> y and (x.name="Kunde" or x.name="
             Einwohner") and capitalizeFirst(y.name)="Name" report x.name as "start", y.name as "end"
             end</query>
        <feedback>Das Diagramm enthaelt keine Klasse "Kunde" oder "Einwohner" mit der Eigenschaft "
             Name".</feedback>
    </rule>
    <rule type="presence" points="5">
        <query>from x : V{Classifier} with x.name="Rechnung" report x.name end</query>
        <feedback>Das Diagramm enthaelt keine Klasse "Rechung".</feedback>
    </rule>
    <rule type="presence" points="5">
        <query>from x : V{Class} with x.name="Telefongesellschaft" report x.name end</query>
        <feedback>Das Diagramm enthaelt keine Klasse "Telefongesellschaft".</feedback>
    </rule>
    <rule type="presence" points="5">
        <query>from x : V{Class} with x.name="Teilrechnung" report x.name end</query>
        <feedback>Das Diagramm enthaelt keine Klasse "Teilrechnung".</feedback>
    </rule>

    <rule type="presence" points="11">
        <query>from x,y : V{Class} with x &lt;-- V{Property} &lt;-- V{Association} --> V{Property}
             --> y and (x.name="Einwohner" or x.name="Kunde") and y.name="Rechnung" report x.name as "
             start", y.name as "end" end</query>
        <feedback>Das Diagramm enthaelt keine Beziehung zwischen einem Einwohner/Kunde und einer
             Rechnung oder keine Klassen mit diesen Namen.</feedback>
    </rule>
    <rule type="presence" points="11">
        <query>from x,y : V{Class} with x &lt;-- V{Property} &lt;-- V{Association} --> V{Property}
             --> y and (x.name="Einwohner" or x.name="Kunde") and y.name="Tarif" report x.name as "
             start", y.name as "end" end</query>
        <feedback>Das Diagramm enthaelt keine Beziehung zwischen einem Einwohner/Kunde und einem
             Tarif oder keine Klassen mit diesen Namen.</feedback>
    </rule>
    <rule type="presence" points="11">
        <query>from x,y : V{Class} with x &lt;-- V{Property} &lt;-- V{Association} --> V{Property}
             --> y and x.name="Telefongesellschaft" and y.name="Tarif" report x.name as "start", y.
             name as "end" end</query>
        <feedback>Das Diagramm enthaelt keine Beziehung zwischen einer Telefongesellschaft und einem
             Tarif oder keine Klassen mit diesen Namen.</feedback>
    </rule>
    <rule type="presence" points="11">
        <query>from x,y : V{Class} with x &lt;-- V{Property} &lt;-- V{Association} --> V{Property}
             --> y and x.name="Telefongesellschaft" and y.name="Teilrechnung" report x.name as "start
             ", y.name as "end" end</query>
        <feedback>Das Diagramm enthaelt keine Beziehung zwischen einer Telefongesellschaft und einer
             Teilrechnung oder keine Klassen mit diesen Namen.</feedback>
    </rule>
    <rule type="absence" points="11">
        <query>from x,y : V{Class} with x &lt;-- V{Property} &lt;-- V{Association} --> V{Property}
             --> y and x.name="Teilrechnung" and y.name="Tarif" report x.name as "start", y.name as "
             end" end</query>
        <feedback>Das Diagramm enthaelt eine direkte Assoziation zwischen einer Teilrechung und einem
              Tarif. Damit kann jedoch nicht die vertelefonierte Menge abgebildet werden.</feedback>
    </rule>

    <rule type="absence" points="10">
       <query>from x : V{Class} with x.name="PinkPanther" or x.name="Pink Panther" or x.name="
            Pink_Panther" report x.name end</query>
       <feedback>Das Diagramm enthaelt eine Klasse, die die Telefongesellschaft "Pink Panther"
            modelliert. Diese ist jedoch nur eine Instanz einer Telefongesellschaft und sollte daher
            in einem Klassendiagramm nicht erscheinen.</feedback>
    </rule>

    <rule type="absence" points="3">
       <query>from x : V{Association} with isNull(x.name) or x.name="" report x.name end</query>
       <feedback>Das Diagramm enthaelt unbenannte Assoziationen.</feedback>
    </rule>
    <rule type="absence" points="3">
       <query>from x,y : V{Property} with not (x=y) and x &lt;--{OwnedEndEdge} V{Association} -->{
            OwnedEndEdge} y report x.name, y.name end</query>
       <feedback>Das Diagramm enthaelt Assoziationen ohne Leserichtung.</feedback>
    </rule>
    <rule type="presence" points="3">
       <query>from x : V{Association}, y : V{Property} with x --> y and not isNull(y.name) and not (
            y.name="") report x.name as "start", y.name as "end" end</query>
       <feedback>Das Diagramm enthaelt keine Rollen an Assoziationen.</feedback>
    </rule>
    <rule type="absence" points="3">
       <query>from x : V{Association}, y : V{Property} with x --> y and isNull(y.lower) and isNull(y
            .upper) report x.name as "start", y.name as "end" end</query>
       <feedback>Das Diagramm enthaelt Assoziationen ohne Kardinalitaeten.</feedback>
    </rule>

    <rule type="absence" points="0">
       <query>from x,y : V{Property} with not isNull(x.name) and not isNull(y.name) and x.name=
            capitalizeFirst(x.name) and not (y.name=capitalizeFirst(y.name)) report x.name, y.name
            end</query>
       <feedback>Hinweis (ohne Punktabzug): Ein Diagramm sollte eine konsistente Schreibweise
            enthalten, in der entweder alle Attribute mit einem Grossbuchstaben oder alle Attribute
            mit einem Kleinbuchstaben beginnen.</feedback>
    </rule>
    <rule type="absence" points="0">
       <query>from x : V{Property} with x.aggregation="composite" report x.name end</query>
       <feedback>Hinweis (ohne Punktabzug): Die Verwendung der Komposition statt der Aggregation ist
             in dieser Aufgabe nicht unbedingt notwendig.</feedback>
    </rule>
</umlcheckerrules>
```

B. Materials from Trace Alignment Implementation

B.1. Functions for Alignment of Program Trace Columns

```
public int align_global(String[] x, String[] y, String[] xt, String[] yt, int gap) {
    int m = x.length;
    int n = y.length;

    int[] S0 = new int[n+1];
    int[] S1 = new int[n+1];
    for (int j = 0; j < n+1; j++) S1[j]=j*gap;

    for (int i = 1; i < m+1; i++) {
        S0 = S1;
        S1 = new int[n+1];
        S1[0] = i*gap;

        for (int j = 1; j < n+1; j++) {
            int a = S0[j-1]+scoreValues(x[i-1],y[j-1])+scoreTypes(xt[i-1],yt[j-1]);
            int b = S1[j-1]+gap;
            int c = S0[j]+gap;
            int max = Math.max(Math.max(a, b), c);
            S1[j] = max;
        }
    }

    return S1[n];
}

private int scoreValues(String a, String b) {
    if ((a == null || a.isEmpty()) && (b == null || b.isEmpty()))
        return 0;

    if (a.equals(b))
        return 1;

    try {
        if (!a.isEmpty() && !b.isEmpty() && Double.parseDouble(a) != Double.NaN && Double.
             parseDouble(a) == Double.parseDouble(b))
            return 1;
    } catch (Exception ex) {
        // ignore
    }

    return -1;
}

private int scoreTypes(String a, String b) {
    if ((a == null || a.isEmpty()) && (b == null || b.isEmpty()))
        return 0;

    if (a.equals(b))
        return 1;
    while (a.endsWith("[]") && b.endsWith("[]")) {
        a = a.substring(0, a.length()-2);
        b = b.substring(0, b.length()-2);
    }

    if ((a.equals("double") || a.equals("float")) && (b.equals("double") || b.equals("float")))
        return 1;
    if ((a.equals("int") || a.equals("long") || a.equals("short") || a.equals("byte")) && (b.
         equals("int") || b.equals("long") || b.equals("short") || b.equals("byte")))
        return 1;

    return -1;
}
```

B.2. Functions for Alignment of Complete Program Traces

```
public double[] align_global(String[][] x, String[][] y, int gap) {
    int m = x.length;
    int n = y.length;

    int[] S0 = new int[n+1];
    int[] S1 = new int[n+1];
    for (int j = 0; j < n+1; j++) S1[j]=j*gap;

    Tuple[][] T = new Tuple[m+1][n+1];
    T[0][0] = new Tuple(0,0);
    for (int j = 1; j < n+1; j++) T[0][j]=new Tuple(0,j);

    for (int i = 1; i < m+1; i++) {
        S0 = S1;
        S1 = new int[n+1];
        S1[0] = i*gap;
        T[i][0] = new Tuple(i-1,0);

        for (int j = 1; j < n+1; j++) {
            int a = S0[j-1]+score(x[i-1],y[j-1]);
            int b = S1[j-1]+score(null,y[j-1]);
            int c = S0[j]+score(x[i-1],null);
            int max = Math.max(Math.max(a, b), c);
            S1[j] = max;

            if (a == max) {
                T[i][j] = new Tuple(i-1, j-1);
            } else if (b == max) {
                T[i][j] = new Tuple(i, j-1);
            } else if (c == max) {
                T[i][j] = new Tuple(i-1, j);
            }
        }
    }

    return traceback(T, m, n, x, y);
}

private int score(String[] a, String[] b) {
    int s = 0;

    if (a == null)
        return (b.length / 3);

    if (b == null)
        return (a.length / 3);

    for (int i = 0; i < a.length && i < b.length; i++)
        if (scoreValues(a[i], b[i]) > 0) {
            s = s+2;
        } else if (a[i].isEmpty() && b[i].isEmpty()) {
            s = s+1;
        } else if (a[i].isEmpty() || b[i].isEmpty()) {
            s = s-1;
        } else {
            s = s-2;
        }

    return s;
}

private double[] traceback(Tuple[][] t, int i, int j, String[][] x, String[][] y) {
    LinkedList<String[]> s1 = new LinkedList<String[]>();
    LinkedList<String[]> s2 = new LinkedList<String[]>();

    while (true) {
        Tuple current = t[i][j];
        if (current.a == i && current.b == j) break;

        for (int k = current.a; k < i; k++) {
            s1.addFirst(x[k]);
        }
        for (int k = current.b; k < j; k++) {
            s2.addFirst(y[k]);
        }

        if (i-current.a < j-current.b) {
            for (int k = 0; k < j-current.b-(i-current.a); k++) {
                s1.addFirst(null);
            }
        } else {
            for (int k = 0; k < i-current.a-(j-current.b); k++) {
                s2.addFirst(null);
            }
        }

        i = current.a;
        j = current.b;
    }

    ArrayList<Double> ratings = new ArrayList<Double>();

    while(!s1.isEmpty()) {
        String[] a = s1.pop();
        String[] b = s2.pop();
        String[] next_a = null;
        String[] next_b = null;

        int k = 0;
        while ((next_a == null || next_b == null) && k < s1.size() && k < s2.size()) {
            next_a = s1.get(k);
            next_b = s2.get(k);
            k++;
        }

        double rating = rateMatch(a, b, next_a, next_b);
        if (a != null)
            ratings.add(rating);
    }

    double[] lineRate = new double[ratings.size()];
    int index = 0;
    for (Double r : ratings) {
        lineRate[index] = r;
        index++;
    }

    return lineRate;
}

private double rateMatch(String[] a, String[] b, String[] next_a, String[] next_b) {
    int globalCount = 0;
    int globalMatch = 0;
    int globalCountNext = 0;
    int globalMatchNext = 0;

    if (a != null && b != null) {
```

```
        double match = 0.0d, count = 0.0d;
        for (int k = 0; k < a.length && k < b.length; k++) {
            if (!a[k].isEmpty() && scoreValues(a[k], b[k]) > 0) match++;
            if (!a[k].isEmpty()) count++;
        }
        globalCount = (int)count;
        globalMatch = (int)match;
    }

    if (next_a != null && next_b != null) {
        double match = 0.0d, count = 0.0d;
        for (int k = 0; k < next_a.length && k < next_b.length; k++) {
            if (!next_a[k].isEmpty() && scoreValues(next_a[k], next_b[k]) > 0) match++;
            if (!next_a[k].isEmpty()) count++;
        }
        globalCountNext = (int)count;
        globalMatchNext = (int)match;
    }

    int diff = 0;
    if (a != null && b != null && next_a != null && next_b != null)
        diff = (globalMatchNext - globalMatch) - (globalCountNext - globalCount);

    return diff;
}
```

B.3. Functions for Alignment of Activity Sequences

```
public double check(List<String[]> sequences, String[] sampleSequence) {
    double result = 0.0;
    for (String[] pathSequence : sequences) {
        double temp = alignSequence(pathSequence, sampleSequence, -1);
        double ratio = temp/sampleSequence.length;
        if (ratio > result)
            result = ratio;
    }

return result;
}

private int alignSequence(String[] x, String[] y, int gap) {
    int m = x.length;
    int n = y.length;

    int[] S0 = new int[n+1];
    int[] S1 = new int[n+1];
    for (int j = 0; j < n+1; j++) S1[j]=j*gap;

    for (int i = 1; i < m+1; i++) {
        S0 = S1;
        S1 = new int[n+1];
        S1[0] = i*gap;

        for (int j = 1; j < n+1; j++) {
            int a = S0[j-1]+score(x[i-1],y[j-1]);
            int b = S1[j-1]+gapScore(x[i-1]);
            int c = S0[j]+gapScore(y[j-1]);
            int max = Math.max(Math.max(a, b), c);
            S1[j] = max;
        }
    }

    return S1[n];
}

private int stringDistance(char[] x, char[] y, int gap) {
    int m = x.length;
    int n = y.length;

    int[] S0 = new int[n+1];
    int[] S1 = new int[n+1];
    for (int j = 0; j < n+1; j++) S1[j]=j*gap;

    for (int i = 1; i < m+1; i++) {
        S0 = S1;
        S1 = new int[n+1];
        S1[0] = i*gap;

        for (int j = 1; j < n+1; j++) {
            int a = S0[j-1]+(x[i-1]==y[j-1]?0:1);
            int b = S1[j-1]+gap;
            int c = S0[j]+gap;
            int min = Math.min(Math.min(a, b), c);
            S1[j] = min;
        }
    }

    return S1[n];
}
```

```
private int score(String a, String b) {
    if (a == null || b == null)
        return 1;

    if (a.equals(b))
        return 1;

    if (stringDistance(a.toCharArray(), b.toCharArray(), 1) <= (a.length()+b.length())/6)
        return 1;

    return -1;
}

private int gapScore(String a) {
    if (a.equals("->") || a.equals("n/a"))
        return 0;

    return -1;
}
```