Efficiency Improvements in the Quality Assurance Process for Data Races

Vom Fachbereich Informatik der
Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Wirtsch.-Inf. Alexander Klaus

Datum der wissenschaftlichen Aussprache: 08.04.2022

Dekan:	Prof. Dr. Jens Schmitt
Berichterstatter:	Prof. Dr. Dr. h. c. Dieter Rombach
Berichterstatter:	Prof. Dr. Reinhard Gotzhein

DE-386

Acknowledgements

It was a long journey that eventually has led to finalizing this thesis. Throughout this journey, many different people have supported me.

First, I would like to thank my parents, who raised my interest in computers, and who enabled me to study computer science.

I feel deep thankfulness to my wife Maryna and my children Julius and Johann, who had to be very patient and enduring during too many long weekends and evenings spent for conducting the work presented in this thesis.

I would like to thank Prof. Dr. Dr. h.c. Dieter Rombach and Prof. Dr.-Ing. Jörg Dörr for their valuable advice and their patience during the many years of working on this thesis. I would also like to thank Prof. Dr. Dr. h.c. Dieter Rombach and Prof. Dr. Reinhard Gotzhein for their commitment to evaluate this thesis, and Prof. Dr. Hans Hagen for taking over the chair of the dissertation committee.

Many thanks go to my former colleague from Fraunhofer IESE, Dr. Frank Elberzhager, who supported me in the early steps of my thesis. Many thanks also go to Gero Zimmer, who conducted a Master Thesis in the context of my thesis, and with whom I had many intensive discussions about UML Activities.

I would like to thank Prof. Dr. Katharina Anna Zweig and Dr. Marsha Kleinbauer from the Technische Universität Kaiserslautern for their support and the collaboration on the graph theoretic aspects in the analysis of UML Activities.

Many thanks go to Prof. Dr.-Ing. Jörg Dörr and Dr. Christian Jung from Fraunhofer IESE and to Prof. Dr. Volodymyr O. Liubchak from Sumy State University for their support in conducting the controlled experiment.

I would also like to thank Caroline Frey for her organizational support.

Abstract

As the usage of concurrency in software has gained importance in the last years, and is still rising, new types of defects increasingly appeared in software. One of the most prominent and critical types of such new defect types are data races. Although research resulted in an increased effectiveness of dynamic quality assurance regarding data races, the efficiency in the quality assurance process still is a factor preventing widespread practical application. First, dynamic quality assurance techniques used for the detection of data races are inefficient. Too much effort is needed for conducting dynamic quality assurance. Second, dynamic quality assurance techniques used for the analysis of reported data races are inefficient. Too much effort is needed for analyzing reported data races and identifying issues in the source code.

The goal of this thesis is to enable efficiency improvements in the process of quality assurance for data races by: (1) analyzing the representation of the dynamic behavior of a system under test. The results are used to focus instrumentation of this system, resulting in a lower runtime overhead during test execution compared to a full instrumentation of this system. (2) Analyzing characteristics and preprocessing of reported data races. The results of the preprocessing are then provided to developers and quality assurance personnel, enabling an analysis and debugging process, which is more efficient than traditional analysis of data race reports. Besides dynamic data race detection, which is complemented by the solution, all steps in the process of dynamic quality assurance for data races are discussed in this thesis.

The solution for analyzing UML Activities for nodes possibly executing in parallel to other nodes or themselves is based on a formal foundation using graph theory. A major problem that has been solved in this thesis was the handling of cycles within UML Activities. This thesis provides a dynamic limit for the number of cycle traversals, based on the elements of each UML Activity to be analyzed and their semantics. Formal proofs are provided with regard to the creation of directed acyclic graphs and with regard to their analysis concerning the identification of elements that may be executed in parallel to other elements. Based on an examination of the characteristics of data races and data race reports, the results of dynamic data race detection are preprocessed and the outcome of this preprocessing is presented to users for further analysis.

This thesis further provides an exemplary application of the solution idea, of the results of analyzing UML Activities, and an exemplary examination of the efficiency improvement of the dynamic data race detection, which showed a reduction in the runtime overhead of 44% when using the focused instrumentation compared to full instrumentation. Finally, a controlled experiment has been set up and conducted to examine the effects of the preprocessing of reported data races on the efficiency of analyzing data race reports. The results show that the solution presented in this thesis enables efficiency improvements in the analysis of data race reports between 190% and 660% compared to using traditional approaches.

Finally, opportunities for future work are shown, which may enable a broader usage of the results of this thesis and further improvements in the efficiency of quality assurance for data races.

Table of Contents

List of Figures

List of Tables

1 Introduction

This chapter elaborates on the context and topic of this thesis and explains the contributions of this thesis. The chapter presents the research approach and closes with an outline of the following chapters.

1.1 Context

Distributed systems are an established area in computer science since many decades [La78]. With the rise of multicore CPUs and multiprocessor systems in the past years [Ch09], [JT14], problems, which were specific to distributed systems became part of common programming [La78], [WS06].

In processor assembly, there was a shift from creating more processing power in terms of CPU clock rates towards creating multi core processors [La10]. To benefit from these developments, software needs to be developed using multithreading techniques, so that computations may be executed in parallel on different cores [La10], [Pa19b]. Nowadays, multithreading techniques are widely used in software development [B019], [OGH20].

With multithreading techniques being used in software development, parallelism and concurrency are part of modern software applications [Me15], [R019]. Parallelism itself does not necessarily lead to concurrency. Instead, software using concurrent computation regularly involves the usage of shared memory [OGH20].

The following Figure 1 demonstrates this distinction. While the computations in the upper part of this figure are executed in parallel, the lower part shows resource 2 as shared resource. This shared resource leads to the different computations not only running in parallel, but concurrently.

However, with concurrent computation, new types of defects evolved, which are perceived as being hard to detect and more time consuming than defects not related to concurrency [PLZ09], [Er10], [Ka17], [Li19].

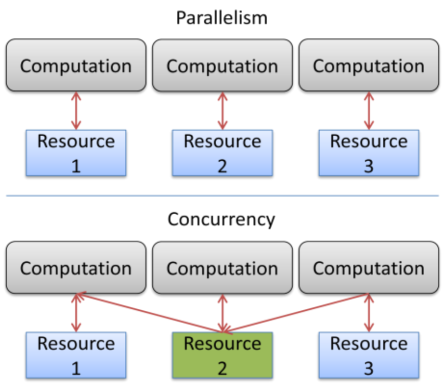


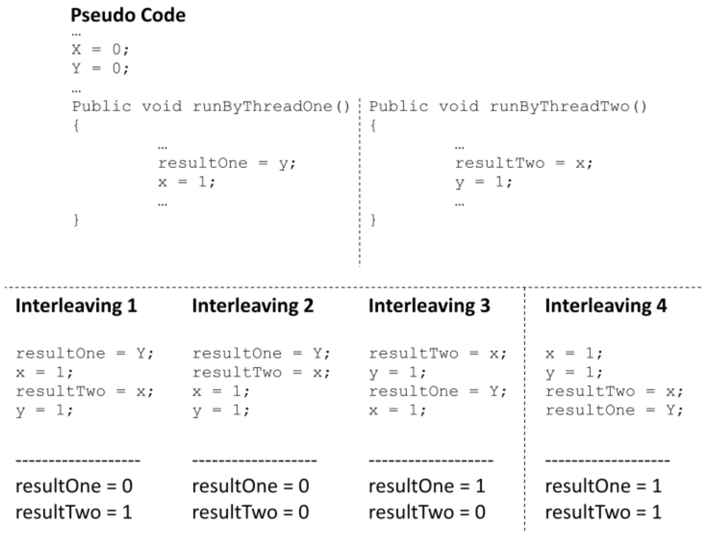
Figure 1: Parallelism and Concurrency

Concurrency is related to non-determinism in the execution of different concurrently executed computations [Ba06b]. This means that it is not clear, and not predictable, in which order the computations, and the instructions involved in these computations, are executed. It cannot be determined upfront. With non-deterministic execution of concurrent instructions, access to shared memory may happen in different interleavings [Ab17], [O'05], [Kl12], [Ab17]. Some of those interleavings may be unexpected and may lead to problems [FQ03], [PS08], [KZC13], [Kl13].

Unexpected interleavings may, for example, be reached due to misunderstandings of program and compiler behavior. Often, the assumption of sequential consistency is made, it is assumed that instructions are executed in the order, in which they appear in the source code [KZC15]. However, in many modern programming languages, such as `Java` or `C++`, this sequential consistency is not guaranteed in the case of multithreaded software [B012a], [R019]. In addition, changes in variables executed in one thread may not always be immediately visible to other threads [B012a], [KZC15].

In the following Figure 2, an example for a concurrent computation is given and four exemplary out of 24 possible interleavings and the consequences of these four interleavings are demonstrated.

The methods `runByThreadOne` and `runByThreadTwo` are executed in parallel by two different threads. Both methods access both variables `x` and `y`, and thus, a concurrent computation is given. The first three interleavings show typical examples, of how the instructions in the two methods may be executed. The fourth, separated, interleaving is possible due to instruction reordering. This instruction reordering leads to the instructions in each method not being executed in the order written in the application. Sequential execution of the instructions in any of these methods is not guaranteed. As a result, the two variables `resultOne` and `resultTwo` can have any of the depicted value combinations after the execution of the code shown.



Pseudo Code

```
…
X = 0;
Y = 0;
…
Public void runByThreadOne() | Public void runByThreadTwo()
{                            | {
        …                    |         …
        resultOne = y;       |         resultTwo = x;
        x = 1;               |         y = 1;
        …                    |         …
}                            | }
```

```
Interleaving 1     Interleaving 2     Interleaving 3     | Interleaving 4

resultOne = Y;     resultOne = Y;     resultTwo = x;     | x = 1;
x = 1;             resultTwo = x;     y = 1;             | y = 1;
resultTwo = x;     x = 1;             resultOne = Y;     | resultTwo = x;
y = 1;             y = 1;             x = 1;             | resultOne = Y;

------------------ ------------------ ------------------ | ------------------
resultOne = 0      resultOne = 0      resultOne = 1      | resultOne = 1
resultTwo = 1      resultTwo = 0      resultTwo = 0      | resultTwo = 1
```

Figure 2: Pseudo Code and Exemplary Interleavings

Various categorizations exist for concurrency defects [Lu08], [As15], [Lo17], [WLW17]. Most categorizations mention the following types:

- Data Race – a data race is defined as a concurrent access of at least two threads to the same memory location, of which at least one access is a write, without proper synchronization [Ba06a], [La10].

  Often, literature focuses on data races consisting of two threads, i.e., two concurrent accesses, as in [NA07][La10].

- Deadlock – a deadlock "is the situation in which one or more processes in a system are blocked forever because of requirements that can never be satisfied" [Ho72]. An example for such a blocking situation in concurrent systems is a thread waiting for a resource held by another thread, which cannot be released by this thread. Deadlocks are not necessarily related to concurrency [Ho72], but new types of deadlock situations emerged due to concurrency [Lu08], [As15].

- Livelock – a livelock has similarities with a deadlock, except that threads are executing, i.e., they are not blocked, and change state, but cannot progress [As15], [Lo17].

- Starvation – starvation occurs when a thread is delayed forever because other threads are given priority, so that this thread cannot proceed [St18].

- Order violation – an order violation occurs, whenever the interleaving expected by the developer of operations does not happen, e.g., when the access to shared memory does follow the intended order [Lu08], [As15].

- Atomicity violation – an atomicity violation occurs, when a section of code was intended by the developer to be executed atomically, but was not executed atomically, i.e., whenever another code sections was executed in parallel [Lu08], [As15].

- Unintended sharing – a resource meant to be used in isolation is shared between different threads [Er10].

These categories for concurrency defects are not orthogonal, e.g., a data race can be related to an atomicity violation, order violation or an unintended sharing of resources [KZC13].

Detecting concurrency defects is more challenging than detecting defects in sequential software [Ab17]. Concurrency defects may not occur in every execution of the software, but may be dependent on certain interleavings. Due to the huge interleaving space of concurrent software [ZSL10] and the non-deterministic behavior [AS15], detecting and debugging such defects is very time consuming [Sa97], [Bl18].

Out of these concurrency defects, data races are especially critical [HMR14], [Xu20]. A data race may not only lead to directly observable behavior, such as a crash of the software, but may also result in data corruption [Kl13], which may not be directly observable [RGB20], or inconsistent program states [JT14]. Data races can lead to critical defects. Examples are `Therac-25` [Ka17], [Bi17], which resulted in deaths of multiple patients, or the Northeast Blackout of 2003 [ZSL10], [Bi17], which resulted in 55 Million people being without electricity, a financial loss of estimated six billion dollars [Mi08], and which contributed to the deaths of almost 100 people [Re12].

Precisely detecting and debugging data races is said to be NP-hard [Ba06a] [Sl09]. Since the amount of parallel and concurrent software is still growing, the number of data races in software is expected to further grow [KZC15], [Bi17], [Ge19]. Thus, although data race detection is an active research area since many years, there is still a growing need for efficient data race detection [ZSL10], [Ge19], [Li19].

1.2 Problem Statement

Both static and dynamic quality assurance techniques have been developed with a focus on data race detection. Sophocleous and Kapitsaki conducted a survey with 252 individuals and came to the conclusion, that 96.8% of these individuals are using static and dynamic quality assurance techniques [SK20]. While this is a sign that the usage of both static and dynamic quality assurance techniques is widespread in general, the situation is different when focusing on concurrency. Vasallo et al. conducted a survey with 56 developers, interviewed eleven experts from industry and investigated 176 open source systems with regard to the usage of automated static analysis tools [Va20]. They came to the conclusion, that when "using static analysis tools, only 8% use it for concurrency" [Va20]. This means that when targeting data races or concurrency in general, dynamic quality assurance techniques are most prevalent. A reason might be that "for object-oriented software, static analysis does not suffice" [ABF04], [SH20][1]. Static data race detectors are "prone to excessive false warnings" [Er10], may miss data races, or require annotations [Er10]. In addition, Kasikci et al. mention that static data race detection techniques report more false positives than dynamic techniques [KZC15].

Nevertheless, static quality assurance techniques are a valuable and necessary technique, as they are not as limited as dynamic quality assurance techniques. Dynamic quality assurance techniques rely on actual program execution, and can only observe those parts of a program, which are executed. As such, these techniques depend on, e.g., the depth of source code coverage reached with test cases used for executing different paths through a program. Parts that are not executed are not observed, and data races may be missed. In contrast, static quality assurance techniques are not limited to any execution, and analyze complete programs. Thus, both techniques have value and should be used together.

Dynamic testing techniques usually rely on instrumentation [PG08], i.e., code is added to the system under test, e.g., "to observe the runtime behavior of each thread in the program" [Ba06b]. Besides affecting system behavior and the timing in the software itself, one of the main problems related to instrumentation is runtime overhead. Erickson et al. report about dynamic data race detection tools that it "is not uncommon for such tools to incur up to $200x$ slowdowns" [Er10], which means a runtime overhead of up to $20{,}000\%$. Bodden and Havelund report about runtime overheads of up to 230% [BH08]. Chew and Lie report about runtime overheads of up to 72% [CL10]. Flanagan and Freund report slowdowns of up to $8.5x$, which results in a runtime overhead of 850% [FF09]. Besides the improvements achieved in research, the overhead in current techniques is considered to be significant [Xu20]. Roemer et al., e.g., present an optimization for a dynamic quality assurance approach, that still induces a runtime overhead of more than $6x$ [RGB20].

Based on these observations, it can be concluded that dynamic quality assurance techniques suffer from a huge runtime overhead and as a result,

[1] The techniques for static and dynamic data race detection will be examined in more detail in Chapter 2.4.

that too much time is needed for conducting quality assurance. This leads to the first practical problem to be solved by this thesis:

Practical Problem 1

Dynamic quality assurance techniques used for the detection of data races are inefficient – too much effort is needed for conducting dynamic quality assurance.

Besides this runtime overhead, mainly the effectiveness has been improved in recent years. In 2008, Bodden and Havelund reported a number of 70 data races found [BH08], and Sen reported 547 data races detected in the application `jigsaw` [Se08]. Nowadays, however, modern approaches are able to detect thousands of data races in a single program. As an example, Roemer et al. present statistics for different systems under test [RGB20]. Even for small applications, such as `pmd` with 61,000 lines of code, up to 10,000 data races have been reported by the data race detector [RGB20]. For others, such as `xalan` with 176,000 lines of code, the data race detector even reports more than 12 million data races [RGB20].

These huge numbers of data races detected lead to difficulties in the practical application. Raychev et al. warned for their own approach, that "the race detector produces too many races to be practically useful" [RVS13]. For analyzing all these reported data races, "developers have to manually check all the reports, which is tedious and very time consuming. As a result, the wide use of data race detectors has been limited in practice." [Zh11]

The increased effectiveness of dynamic quality assurance techniques, i.e., the increased number of reported data races, has also worsened these drawbacks, as modern techniques, as shown above, result in huge log information, i.e., data race reports to be analyzed. However, only few publications mention techniques to ease analysis of the reported data races besides possibilities to reproduce detected data races [HMR14] or record and replay techniques [LD19]. Such replay techniques allow replaying the situations, during which a data race occurred. Still, all data race reports found need to be analyzed manually for identifying and resolving problems in the source code, resulting in a huge effort necessary. Godefroid and Nagappan conducted a survey with 684 employees from Microsoft [GN08] and observed that analyzing data races is very time consuming; it "often takes days of work to analyze a single concurrency bug" [GN08]. Remaining work focuses on visualization of data races [Tr14], [Ko15], [Pa19b], [Wa20] [2].

[2] The techniques for the analysis of data race reports will be examined in more detail in chapter 3.5.

To make it worse, many of the reported data races are duplicates. As an example, out of the 12 million reported data races for `xalan`, there are only 162 distinct data races, and out of the 10,000 data races reported for `pmd`, only 12 data races are distinct (with distinct meaning affecting "distinct program locations") [RGB20].

These observations lead to the second practical problem to be solved by this thesis:

Practical Problem 2

Dynamic quality assurance techniques used for the analysis of data races are inefficient – too much effort is needed for analyzing reported data races and identifying issues in the source code.

Summarized, the process of dynamic quality assurance for data races lacks efficiency in terms of time and effort needed. A simplified depiction of this process for dynamic quality assurance technique can be found in Figure 3.

On the left side, and shown as out of scope, is the creation of test cases. Dynamic data race detectors rely on existing test cases, and do not include specific techniques for their creation. Instead, test cases are created using standard test case creation techniques, e.g., during unit testing. On the right side of this figure, the process of dynamic quality assurance for data races is shown: the source code is instrumented and test cases are executed under control of the dynamic data race detector. This instrumentation and the techniques used by the dynamic data race detectors to detect data races result in the runtime overhead, and thus, in the inefficiency described in the first practical problem. After the execution of the test cases, the reported data races need to be analyzed to identify problems in the source code, and finally for debugging the system under test. Since this is a manual process without much guidance besides the log information in the data race reports, as described above, this is a time consuming task.

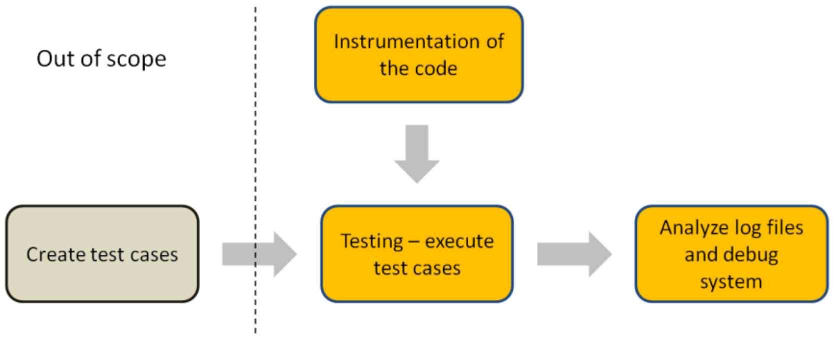

Out of scope
Instrumentation of the code
Create test cases
Testing – execute test cases
Analyze log files and debug system

Figure 3: Simplified Process of Dynamic Quality Assurance for Data Races

Running example – Original approach

The system `Apache Tomcat` in Version 8.0.26 is to be tested using the dynamic data race detection tool `RV-Predict`[3] [HMR14]. The test cases to be executed are the unit test cases shipped together with `Apache Tomcat` 8.0.26 without any modification. Testing is conducted under `Ubuntu` 18.0.4. Executing the test cases without any instrumentation takes 46 minutes and two seconds. Now, the same test cases are executed with the goal to detect data races, and the code is instrumented by the data race detection tool. This instrumentation is conducted automatically, and executing the test cases takes 345 minutes and ten seconds. The runtime overhead for testing with a focus on data races is $6.5x$, i.e., test execution takes 650% the time it took without instrumentation and analysis. Testing results in 771 folders containing the results. It is unknown, how many reported data races are duplicates, and which variables and which source code classes are affected.

Thus, the developers need to manually analyze all reports to identify problematic accesses to variables in the source code. Since the number of reports is too high for one developer to be able to complete the analysis in a reasonable amount of time, the reports are split up between the developers. However, there are overlaps in the reports each developer analyzes, and thus, effort is spent on communication and coordination, and on the analysis of data races already under investigation of other developers.

1.3 Contribution

The goal of this thesis is improve the efficiency in the process of dynamic quality assurance for data races.

To solve the practical problems described above, the idea is to optimize the runtime overhead during test execution by using knowledge about the dynamic behavior of the system under test and by providing guidance and knowledge for the analysis of data race reports.

The instrumentation of source code, usually conducted as instrumentation of the complete source code, shall be optimized by focusing instrumentation on those source code locations that can be prone to data races, i.e., classes in the source code, which can be executed in parallel to other classes in the source code. This leads to a reduction in the amount of instrumentation necessary, and thus, reduces runtime overhead.

[3] This tool can be downloaded online: https://runtimeverification.com/predict/ (last visited: 30.01.2021)

Guidance for the analysis of data race reports shall optimize the process step of analysis by providing information on duplicate entries and on the number of data races reported, and by providing possibilities to focus analysis on specific variables and source code classes. With the possibility to ignore duplicates, time spent on navigating through data race reports and reading reports for data races already analyzed can be saved. Furthermore, an analysis can be conducted more focused, and communication and coordination effort can be reduced in a team of developers analyzing data race reports.

These efficiency improvements shall be illustrated with the example described above:

Running example – Optimized approach

The system `Apache Tomcat` in Version 8.0.26 is to be tested using the dynamic data race detection tool `RV-Predict` [HMR14]. The test cases to be executed are the unit test cases shipped together with `Apache Tomcat` 8.0.26 without any modification. Testing is conducted under `Ubuntu` 18.0.4. Executing the test cases without any instrumentation takes 46 minutes and two seconds. Now, the same test cases are executed with the goal to detect data races, and the code is instrumented based on the knowledge, which source code classes can be executed in parallel to other source code classes, and which source code classes can be ignored. This instrumentation is conducted automatically, and executing the test cases takes 212 minutes and two seconds. The runtime overhead for testing with a focus on data races is $3.6x$, i.e., test execution takes 360% the time it took without instrumentation and analysis. Testing results in 771 folders containing the results, with 489 empty result files and 282 result files with data race reports.

These reports contain 2,079 data races, of which 2,002 are duplicates. The remaining 77 data races affect 28 variables and 21 different source code classes. The developers split the reports between each other, so that a distinct set of variables is analyzed by each developer.

This example demonstrates that quality assurance can profit from the results of this thesis. With a reduced instrumentation, runtime overhead is reduced, and faster quality assurance is possible. As a side-effect, as instrumentation, i.e., code injected in the original source code, also alters the runtime behavior of a system under test, the reduction in the amount of instrumentation also leads to a system under test, which is more close to the original system compared to a fully instrumented system.

Developers can benefit from more information, and thus, a more goal-oriented debugging. With the ability to sort out duplicate reports, and to focus on specific variables and source code classes, analysis and debugging can be conducted more efficiently, as no effort needs to be spent to analyze data races, which already have been analyzed or are currently under analysis. In addition, the effort for communication and coordination of the debugging activities can be reduced.

The **scientific problems** to be solved in this thesis are connected to the challenge of using knowledge about the dynamic behavior of the system to improve the efficiency for dynamic quality assurance for the detection of data races, and to the challenge of improving efficiency in the analysis of data race reports by using knowledge about data race characteristics.

This thesis therefore is targeted at answering the following research questions:

Research Question 1

How can knowledge about the dynamic behavior of the system be used to achieve a reduction in the runtime overhead during dynamic quality assurance for data races?

Research Question 2

How can knowledge about the characteristics of data races and data race reports be used to improve the efficiency in analyzing data race reports?

To answer these research questions, the following solution is proposed in this thesis, targeting two challenges:

1) providing a tool-supported technique for the systematic and complete usage of knowledge about the dynamic behavior of a system for the instrumentation of a system under test, and

2) providing a tool-supported technique for the incorporation of knowledge of characteristics of data race reports into guidance for the analysis of these data race reports.

To solve these challenges, the dynamic behavior of a system must be analyzed, and all classes in the source code, which may run in parallel to other classes, must be identified. To achieve this, models of the dynamic behavior of a system shall be analyzed. Goal of the analysis is the identification of elements in these models, which represent parts of the source code that can be executed in parallel to other parts of the source code. The results of the analysis shall then be used to focus the instrumentation for dynamic quality assurance for data races, so that only classes in the source code, which are subject to data races, are instrumented.

After execution of the test cases for the detection of data races, the data race reports shall be analyzed, and the results shall be provided to the development team. This preprocessing allows the development team to sort out duplicates, and to group the data races according to the source code classes affected and according to the variables affected.

As described, this approach is not meant to substitute existing approaches to dynamic data race detection. Instead, the process of dynamic data race detection for data races is optimized. This means, the approach provides input to the dynamic data race detection tool, and processes the output of this data race detection tool.

The solution idea is depicted in the following Figure 4.

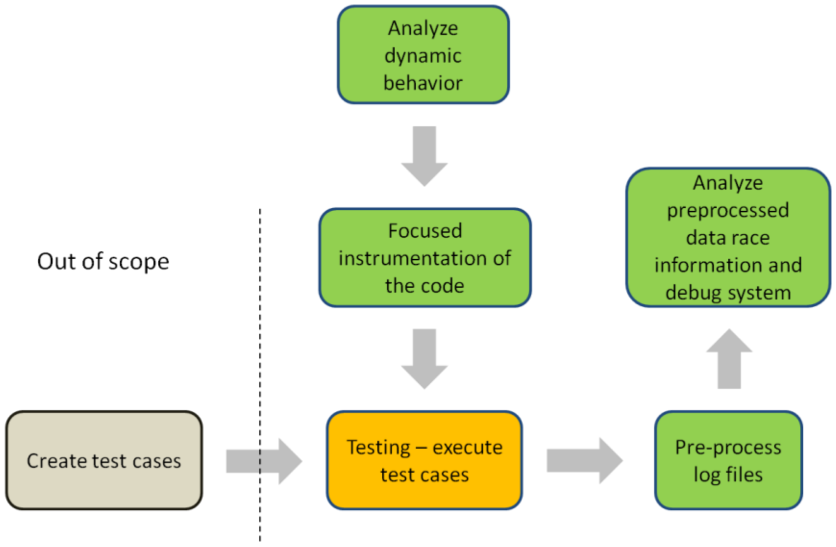

Analyze dynamic behavior
Out of scope
Focused instrumentation of the code
Analyze preprocessed data race information and debug system
Create test cases
Testing – execute test cases
Pre-process log files

Figure 4: Proposed Solution Idea - Optimized Process

To solve the problems mentioned above and to establish the solution idea, several research objectives are to be reached. These research objectives form the remaining outline of this thesis.

Research Objectives

1) Assess the representations of dynamic behavior of a system with the goal to select one representation for the analysis.

2) Analyze the characteristics of the chosen representation of the dynamic behavior with the goal to understand the specifics and how these specifics influence the interpretation of this representation.

3) Create a technique to analyze the chosen representation of the dynamic behavior of a system for parallel elements, with the goal to obtain an algorithmic approach for the analysis.

4) Show the completeness and correctness of the obtained algorithmic approach, with the goal to establish confidence in this approach.

5) Evaluate the effects on efficiency of the test execution of the algorithmic approach compared to not using the algorithmic approach to focus instrumentation.

6) Enable the practical application of the algorithmic approach, with the goal to obtain a fully automated tool implementing this algorithmic approach.

7) Analyze characteristics of data race reports, with the goal to obtain knowledge about how to preprocess data races reports so that the analysis is more efficient.

8) Enable the practical application of the knowledge obtained on characteristics of data race reports, with the goal to obtain tool support.

9) Empirically evaluate the effects of the preprocessing of the data race reports and of the tool support on the efficiency on the analysis of data race reports compared to analyzing unprocessed data race reports.

While research objectives 1 to 6 are concerned with the first practical problem, research objectives 7 to 9 are concerned with the second practical problem.

Upon realization and usage of the solution idea, several benefits are expected. From a scientific perspective, the benefit is related to the analysis of the representation of the dynamic behavior of a system.

Hypothesis 1 – Complete and Correct Analysis

H_1. The analysis of the representation of the dynamic behavior of a system for parallel elements is complete and correct, .i.e., there are no false positives and no false negatives.

From a practical point of view, the benefit can be perceived as efficiency improvement in the process of dynamic quality assurance for data races in terms of a reduced effort without impact on the outcome.

No precise hypotheses can be given regarding the efficiency improvement concerning the dynamic detection of data races, in terms of the runtime overhead when executing test cases, as this depends on the amount of parallelism in the system under test. Assuming a direct relation of the amount of parallelism in a system and the efficiency improvement in the runtime overhead, the hypothesis is formulated as:

Hypothesis 2 – Efficiency Improvement in the Dynamic Detection of Data Races (Runtime Overhead)

H_2. Using the focused approach for the instrumentation, the runtime overhead of test execution of a system under test is reduced compared to the runtime overhead using the same test execution technique on the same system under test without the focused approach. The reduction in the runtime overhead is at least inversely proportional to the amount of parallelism in the representation of that system under test.

For illustration, if 10% of the elements in the representation of a system represent parallel behavior, a reduction of the runtime overhead of 90% can be achieved, and if 80% represent parallel behavior, a reduction of 20% can be achieved.

The efficiency improvement in the analysis of data race reports is defined using one main hypothesis, which can be split up into three different hypotheses.

Hypothesis 3 – Efficiency Improvement in Analyzing Data Races

H_3. Using the log preprocessing, the effort for results analysis is at least 40% less with at least the same effectiveness compared to using the unprocessed log files.

Hypothesis 3.1 – Efficiency Improvement in Analyzing Data Race Reports for Data Races

$H_{3.1}$. Using the log preprocessing, the number of analyzed reported data races in a given amount of time is at least 40% higher compared to using the unprocessed log files.

Hypothesis 3.2 – Efficiency Improvement in Analyzing Data Race Reports for Data Races affecting a Variable in the Source Code

$H_{3.2}$. Using the log preprocessing, the number of analyzed reported data races related to specific variables in a given amount of time is at least 40% higher compared to using the unprocessed log files.

Hypothesis 3.3 – Efficiency Improvement in Analyzing Data Race Reports for Data Races affecting a Source Code Class

$H_{3.3}$. Using the log preprocessing, the number of analyzed reported data races related to specific source code locations in a given amount of time is at least **40%** higher compared to using the unprocessed log files.

Analysis in this context means identifying the location of a reported access. Specific in this context means that in the experimental evaluation, a variable or a source code location (source code class) is predefined and only these predefined items are of interest. The term effectiveness refers to an amount (of analyzed data races), while the term efficiency refers to an amount in relation to the time needed.

As can be seen, no hypotheses are stated regarding the effectiveness of testing. The test case creation, the execution of test cases and the analysis of the instrumented parts of the source code and the information gained during testing are not affected by this thesis. The solution is not meant to replace or improve the effectiveness of test case execution, but instead focuses on efficiency improvements as mentioned above. As such, the solution proposed in this thesis complements existing approaches for dynamic data race detection.

The following Figure 5 shows the relations of the practical and underlying scientific problems, the goals and research objectives of this thesis, and the related hypotheses.

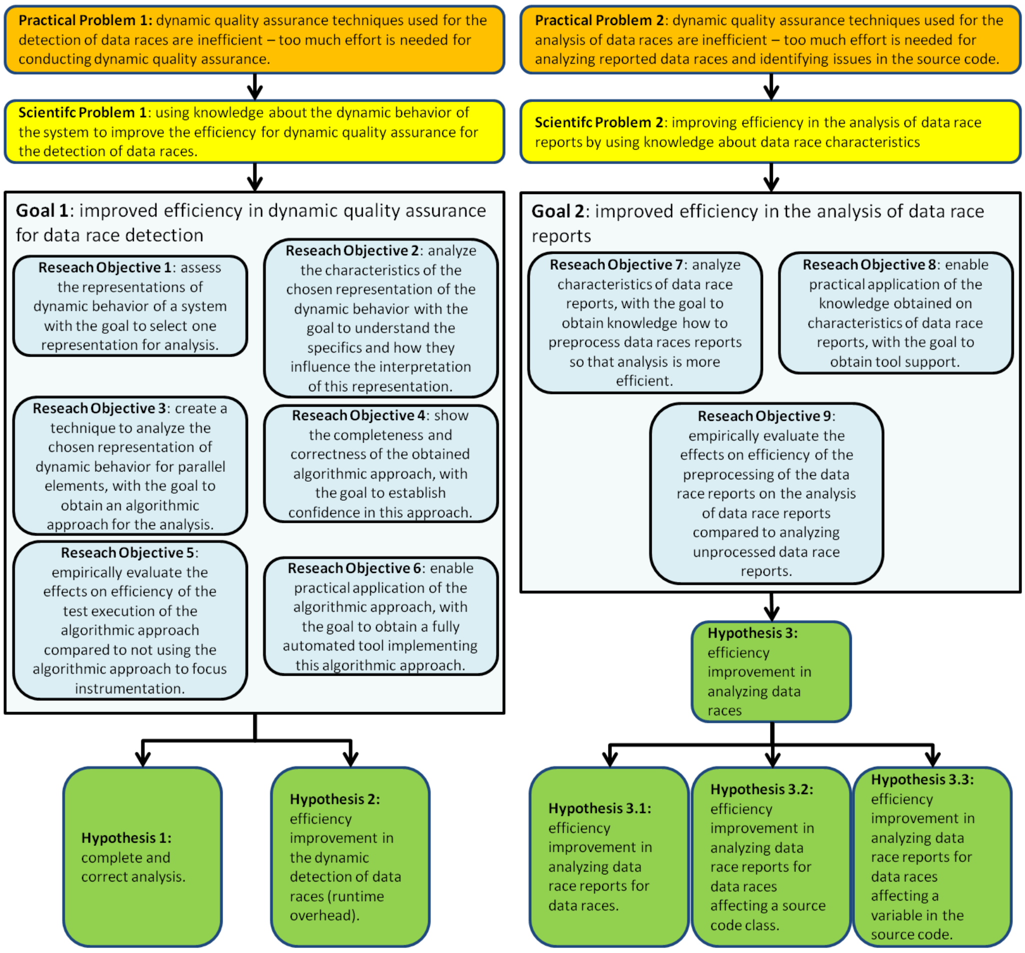

Practical Problem 1: dynamic quality assurance techniques used for the detection of data races are inefficient – too much effort is needed for conducting dynamic quality assurance.
Practical Problem 2: dynamic quality assurance techniques used for the analysis of data races are inefficient – too much effort is needed for analyzing reported data races and identifying issues in the source code.
Scientifc Problem 1: using knowledge about the dynamic behavior of the system to improve the efficiency for dynamic quality assurance for the detection of data races.
Scientifc Problem 2: improving efficiency in the analysis of data race reports by using knowledge about data race characteristics
Goal 1: improved efficiency in dynamic quality assurance for data race detection
Reseach Objective 1: assess the representations of dynamic behavior of a system with the goal to select one representation for analysis.
Reseach Objective 2: analyze the characteristics of the chosen representation of the dynamic behavior with the goal to understand the specifics and how they influence the interpretation of this representation.
Reseach Objective 3: create a technique to analyze the chosen representation of dynamic behavior for parallel elements, with the goal to obtain an algorithmic approach for the analysis.
Reseach Objective 4: show the completeness and correctness of the obtained algorithmic approach, with the goal to establish confidence in this approach.
Reseach Objective 5: empirically evaluate the effects on efficiency of the test execution of the algorithmic approach compared to not using the algorithmic approach to focus instrumentation.
Reseach Objective 6: enable practical application of the algorithmic approach, with the goal to obtain a fully automated tool implementing this algorithmic approach.
Goal 2: improved efficiency in the analysis of data race reports
Reseach Objective 7: analyze characteristics of data race reports, with the goal to obtain knowledge how to preprocess data races reports so that analysis is more efficient.
Reseach Objective 8: enable practical application of the knowledge obtained on characteristics of data race reports, with the goal to obtain tool support.
Reseach Objective 9: empirically evaluate the effects on efficiency of the preprocessing of the data race reports on the analysis of data race reports compared to analyzing unprocessed data race reports.
Hypothesis 3: efficiency improvement in analyzing data races
Hypothesis 1: complete and correct analysis.
Hypothesis 2: efficiency improvement in the dynamic detection of data races (runtime overhead).
Hypothesis 3.1: efficiency improvement in analyzing data race reports for data races.
Hypothesis 3.2: efficiency improvement in analyzing data race reports for data races affecting a source code class.
Hypothesis 3.3: efficiency improvement in analyzing data race reports for data races affecting a variable in the source code.

Figure 5: Problems, Goals, Research Objectives, and Hypotheses

1.4 Assumptions and Limitations

The solution idea and earning the benefits when applying this solution idea cannot be realized in every case. Instead, this thesis builds upon the following assumptions and accepts some limitations:

Existence of a representation of the dynamic behavior: the approach presented in thesis assumes that a representation of the dynamic behavior of a system already exists in the representation format needed or can be created or generated.

However, the solution idea includes a transformation of the elements of the chosen transformation to a limited subset of elements, so that other representations can be used by adjusting the transformation or adding another transformation part. This can be seen as an interface, which allows connecting other representation formats without a need to change the algorithm for analysis. Admittedly, semantics exclusive to such a representation may not be considered by the analysis; in such a case, the algorithm would need to be adapted.

Completeness of the representation of the dynamic behavior: furthermore, it is assumed that the representation is complete, in the sense that the system under test is correctly and completely included.

Granularity of the representation of the dynamic behavior: finally, it is assumed that the representation is on a granularity level, which allows a direct connection between source code classes and elements in the representation. Since such a direct connection can then be broken down into single source code classes and elements using standard methods, it is assumed for simplicity, that there is a one-to-one relationship between elements in the representation and classes in the source code (and not a one-to-many, many-to-one, or many-to-many relationship).

Compatibility of the dynamic data race detection tool: since the solution idea includes focusing the instrumentation, the dynamic data race detection tool needs to be configurable in terms of controlling the instrumentation.

Data Race Reports: the solution idea relies on the dynamic data race detection tool to report the data races found in a format, which can be processed in an automated manner.

As can be seen later, the solution idea relies on basic information regarding data race reports (access type (read or write), variable affected, and location in the source code) for preprocessing, so that no further assumptions or limitations are given. Adjusting the preprocessing to different representations (e.g., a different ordering of the information in a report) is an engineering task and the assumption of a specific format for the tool support is a necessity, which does not limit the approach.

1.5 Research Approach

The research approach followed in this thesis can be broken down as follows:

State of the Practice analysis: the current state of the practice was captured by a literature review with respect to quality assurance in the field of concurrency. As data races are considered as a prominent and critical issue in practice, research was focused on those. Current problems in quality assurance for data races were identified. Based on these problems, requirements for a solution to mitigate these problems could be derived.

For the analysis of representations of dynamic behavior, the first step was to identify a representation format, which is of practical value. Thus, a literature survey was conducted to obtain information regarding the suitability of representation formats for the dynamic behavior of a system, with consideration of the usage in practice.

State of the Art analysis: the identified problems drove the literature review regarding research approaches. The requirements derived on basis of those problems were used to assess the approaches from research, and gaps were identified regarding the analysis of representations of the dynamic behavior of a system.

Regarding the analysis of data race reports and efficiency improvements in the analysis of data race reports, a general lack of research was detected, as not much literature exists in this regard. Literature mostly focuses on detecting or visualizing data races, and less on the analysis of found data races with the goal to debug a system under test. Still, the existing literature was valuable, as it helped identify directions for the own research.

Development of the solution idea: based on these research gaps, the solution idea was developed. For the analysis of representations of the dynamic behavior, two major gaps were identified: first, a lack of a formal basis for the analysis, and second a lack in the systematic and complete analysis of all elements and element combinations in the chosen representation.

In addition, mitigating the limitation to only one possible representation was another driver for the solution idea. Thus, a subset of all elements of the chosen representation format (called "basic elements") was selected. A transformation of all other elements to one or a combination of basic elements under consideration of the semantics of the elements was then created. The transformed set of elements served as basis for the formal model and the systematic algorithmic approach to analyzing the representation format.

The data race reports, on the other hand, were analyzed, and characteristics were derived, which then formed the basis for the preprocessing of those reports.

Examination of Efficacy: the approach for analyzing representations of the dynamic behavior was formally proven to be correct and complete (hypothesis 1). The effect of the results of the analysis, i.e., the focus for the instrumentation, in terms of efficiency improvements, was tested on an exemplary system. As described above, a general statement regarding the improvement of the runtime overhead cannot be given. However, it could be tested for a system, if the improvement of the runtime overhead is at least inversely proportional to the amount of parallelism in the representation, as stated in hypothesis 2.

Finally, a controlled experiment was performed to assess efficiency improvements in the analysis of data race reports with regard to the preprocessing of data race reports (hypothesis 3).

1.6 Outline

In chapter 2, the foundations for the contents of this thesis are presented. These foundations enable a deep understanding of this thesis. Furthermore, the concepts, formal definitions, and results of this chapter form the basis for the decisions taken and the solution presented in this thesis.

In chapter 3, related work is discussed with regard to analyzing behavioral models, and with regard to analyzing data race reports. The approaches found in the literature are discussed in terms of strengths and gaps regarding the mitigation of the practical problems and the scientific problem, and requirements for a solution are identified.

In addition, the possibilities and limitations for the classification of data races with the goal to minimize the effort for data race analysis and debugging are discussed. Multiple research approaches to classify data races in harmless (benign) data races and harmful data races have been developed. However, such classifications are controversially discussed in research. It is shown why these classifications cannot be used without misclassifications. As the reliability of such classifications is low, such a classification is not part of the solution presented in this thesis.

In chapter 4, the main contribution of this thesis is presented. This chapter contains the formal foundation for the analysis of the representation of the dynamic behavior of a system and the approach for the analysis. The analysis itself is conducted in three steps: the transformation of the elements of the representation format to the set of basic elements, the analysis of behavioral models based on this set of basic elements, and a post-analysis handling of special cases. Formal proofs regarding the analysis of the representation of the dynamic behavior show the completeness and correctness of the analysis. The theoretical and formal approach for analysis is implemented in a tool for practical application. Finally, this solution is assessed based on the identified requirements for a solution.

In chapter 5, the approach for the analysis and preprocessing of data race reports, i.e., the results of the dynamic data race detection, is presented. Characteristics of data races and data race reports form the basis for the preprocessing, which is then implemented in a tool. This tool not only presents the results of the preprocessing but also provides guidance in terms of knowledge about the reported data races and in terms of their distribution to variables and source code classes.

In chapter 6, the examinations of efficacy for each step in the process of dynamic quality assurance are presented. Besides an exemplary application of the approach for focusing the instrumentation, the results of the analysis are discussed, and the impact of applying this analysis and its results on the efficiency of dynamic data race detection are demonstrated exemplarily. Finally, a controlled experiment regarding efficiency improvements for the analysis of data race reports is presented and the results and implications are discussed.

The thesis closes with a summary and an outlook on future work in chapter 7.

1.7 Summary

As the usage of concurrency in software has gained importance in the last years, and is still rising, new types of defects increasingly appeared in software. Quality assurance is still struggling with such concurrency-related defects. One of the most prominent and critical types of such defects are data races.

Although research resulted in an increased effectiveness of dynamic quality assurance regarding data races, the efficiency in the quality assurance process still is a factor preventing widespread practical application of these techniques.

The contributions presented in this thesis enable efficiency improvements in the process of dynamic quality assurance for data races. The main scientific contribution in the area of the analysis of representations of the dynamic behavior of a system is presented. The results of the analysis are used as input to steer the instrumentation of a system under test, resulting in a lower runtime overhead during test execution compared to a full instrumentation of this system. The results of the test execution are then preprocessed, enabling an analysis and debugging process, which is more efficient than the traditional analysis of data race reports.

This thesis describes the concepts, the formal basis, and the realization of the solution idea as well as the examinations of the efficacy. Besides dynamic data race detection (i.e., test execution), which is complemented by the solution idea, all steps in the process of dynamic quality assurance for data races, as depicted in Figure 3, are discussed in the subsequent chapters.

2 Foundations

As described in chapter 1, this thesis provides a solution for efficiency improvements in the process of quality assurance for data races. This chapter gives an overview on the foundations in the areas discussed in this thesis. The goal of this chapter is to enable a deeper understanding of topics, as the concepts, formal definitions, and results of this chapter form the basis for the decisions taken as part of the solution idea.

2.1 Research Approach

The areas discussed in this thesis comprise different steps in the process of quality assurance of data races. As such, different topics need to be considered. The existing literature was thus reviewed with the goal to identify the central concepts of those topics.

As data races are in the focus of this thesis, the questions to be answered are:

1) What is a data race?

2) What can be done in software engineering, i.e., programming, to prevent data races?

3) How can data races be detected?

4) What are advantages and disadvantages of different approaches to detect data races?

In addition, as behavioral models of software are part of this thesis, further questions arise:

5) What types of behavioral models exist?

6) What can be represented by these models?

While reviewing the literature with the goal to answer these questions, further questions came up with regard to the background of the topic of those questions. As an example, a formal definition of data races requires an understanding of ordering relations. This understanding is also required for an informed insight into the advantages and disadvantages of quality assurance techniques for data races. The topics discussed in questions 1 to 4 are interrelated and cannot be viewed in isolation. As such, the literature review was an iterative process, leading to a deeper understanding in those areas. The result of this research provides a common foundation of the topics of this thesis.

2.2 Orderings, Consistency Models and Data Races

The following definitions partially follow the contents of the publication "Time, clocks, and the ordering of events in a distributed system" by Leslie Lamport [La78].

When executing a sequential program, all events in that execution take place one after the other. Assuming two events a and b, either a is executed and afterwards b is executed, or b is executed first and then a is executed. With defining an ordering relation $<$ between two events, expressing a relation of two events to the order in time they took place, the relation $a < b$ can be understood as "a took place at an earlier point in time than b". Since the execution is sequential, no two events can take place at the same point in time. All events in such a sequential execution can be compared. The relation is transitive, i.e., if a third event c exists and $a < b$ and $b < c$, then $a < c$. The events of an execution of a sequential program form a sequence, a total order [La78].

When executing a parallel or concurrent program, this does not hold. With two processes p and q, with the events p_a and p_b within process p, and the events q_a and q_b within process q, and those two processes executing in parallel, the following can be observed:

1) Either $p_a < p_b$, or $p_b < p_a$, and

2) Either $q_a < q_b$, or $q_b < q_a$.

For simplicity, it is assumed that $p_a < p_b$ and $q_a < q_b$. Since for each process, the events in that process are executed sequentially, the set of elements in each process is totally ordered. However, the set of all elements (in this case: the events in the two processes) is not necessarily totally ordered. Assuming that p_b involves notifying process q, and process q cannot start to execute without notification (following the definitions of [La78]) , meaning that $p_b < q_a$, then with $p_a < p_b$, and $q_a < q_b$ these events can be ordered: $p_a < p_b < q_a < q_b$. However, assuming an event p_c with $p_b < p_c$, this event p_c cannot be compared to the events q_a and q_b. Thus, there is no total order, but a partial order [La78].

As mentioned, an analogous definition has been published by Leslie Lamport [La78]. However, the author mentioned the necessity of precise and global clocks to be able to use the notion of time, and speaks of "happened before" using the relation $\rightarrow$ [La78] instead of using a relation $<$ meaning "took place at an earlier point in time". With this definition, "two distinct events $\boldsymbol{a}$ and $\boldsymbol{b}$ are said to be concurrent, if $\boldsymbol{a} \nrightarrow \boldsymbol{b}$ and $\boldsymbol{b} \nrightarrow \boldsymbol{a}$" [La78], with $\nrightarrow$ meaning "not $\rightarrow$". The *happened before* relation, in the literature often referred to as *happens before*, is one of the two central concepts used by most quality assurance techniques for data race detection.

Note that the definitions above apply to executions of software, and cannot be compared to the order, in which instructions are written in that software without assuming sequential consistency [La79]. However, sequential consistency is not guaranteed in all cases in various modern programming languages, e.g., it is not guaranteed in all cases in `Java` [Ma04] or `c++` [BA08].

Based on such a relation, Netzer and Miller developed a formalization of data races [NM92]. The authors use a program execution $\boldsymbol{P}$ containing events, a temporal ordering relation and a shared-data dependence relation as starting point. A shared-data dependence relation describes a relation of two events, in which one event accesses a shared variable, which another event later accesses, with at least one access modifying that variable [NM92]. In a situation, in which one event accesses a shared memory location with a write access and another event accesses this location with either a read or write access, there exists a *data conflict* between those two events.

The authors formulate three different sets of program execution prefixes of $\boldsymbol{P}$. Such a prefix contains the same events as an initial part of $\boldsymbol{P}$, operating on the same input. Netzer and Miller further use the term *feasible* to denote program executions, which can actually happen based on the program semantics. This includes *explicit* synchronization operations, i.e., synchronization using operations provided by the respective programming language exactly for this purpose, but also self-constructed operations to force some order of execution or control flow. An example for such an *implicit* operation is the usage of a flag, which may trigger a certain event $\boldsymbol{e}$, but which needs to be set upfront by another event $\boldsymbol{f}$ in another process. If event $\boldsymbol{f}$ is executed before the flag is computed, the event $\boldsymbol{e}$ is executed. If event $\boldsymbol{f}$ is not executed before the flag is computed, the event $\boldsymbol{e}$ is not executed. In such a case, there is a shared-data dependence [NM92].

The set of program executions $\boldsymbol{F_{SAME}}$ contains all feasible program executions with the same events as a prefix of $\boldsymbol{P}$ and the same shared-data dependences as $\boldsymbol{P}$. The set of program executions $\boldsymbol{F_{DIFF}}$ contains feasible program executions with the same events as a prefix of $\boldsymbol{P}$, but the shared-data dependences may differ. Finally, the set of program executions $\boldsymbol{F_{SYNC}}$ contains the same events as a prefix of $\boldsymbol{P}$, but the relations in terms of ordering and shared-data dependence only need to adhere to explicit synchronization constructs.

This means that F_{SYNC} may contain executions, which are not feasible, i.e., the program cannot be executed in this way [NM92]. The authors point out that many quality assurance techniques only consider explicit synchronization, and thus implicitly use the set F_{SYNC} [NM92].

Netzer and Miller define a data race between two events a and b over a (here unspecified) set of program executions F as

1) A data conflict exists in P between a and b, and

2) There exists a program execution $P' \in F$, containing events a' and b', such that $a' \nrightarrow b'$ and $b' \nrightarrow a'$ [NM92].

With these definitions, two types of data races can be defined:

1) A *feasible data race* between events a and b exists, iff a data race between events a and b exists over F_{DIFF} or F_{SAME} [NM92].

2) An *apparent data race* between events a and b exists, iff a data race between events a and b exists over F_{SYNC} [NM92].

Due to the usage of F_{SYNC}, not all *apparent data races* can occur in a program execution. However, as mentioned above, many quality assurance techniques imply F_{SYNC}, and thus, report data races, which may not be possible. This is especially an issue in static quality assurance, as the code is not executed. But depending on the analysis technique used, also dynamic quality assurance may be prone to reporting *apparent data races*.

The existence of *apparent data races* indicates the existence of at least one *feasible data race*, but it is not clear, where this *feasible data race* is located and what variable is involved [NM92]. As there may be many *apparent data races* indicating one *feasible data race*, developers and quality assurance personnel "can be overwhelmed with large amounts of misleading information, irrelevant for debugging, that masks the location of actual failures." [NM92] This issue has already been mentioned in chapter 1.2.

2.3 Process Synchronization in Software

Modern programming languages provide possibilities to implement different concepts to cope with concurrency. As this thesis is not concerned with programming details, but with quality assurance for data races, the goal of this chapter is not to provide a complete overview of all techniques, but to provide an overview of the most prominent techniques.

A fundamental concept in concurrency is the concept of mutual exclusion, first defined by Dijkstra in 1965 [Di65b]. Since then, many concepts have been defined to control access to shared resources and to realize mutual exclusion.

Semaphores have been published by Dijkstra in 1965 [Di65a]. These are data structures used to control access to shared resources or to synchronize concurrent events to enforce an order of operations. *Semaphores* can be binary, used for implementing *locks* (explained below), or counting. A counting *semaphore* controls the number of free resources or waiting threads. A thread requesting a resource, which is not free, will be sent to a wait state, and notified upon availability of that resource, so that this thread can continue to operate on the now free resource. With this mechanism, a queue for waiting threads can be realized. Java offers a class *semaphore* for the implementation of this concept.

Monitors have been described by Brinch Hansen [Br73] and Hoare [Ho74]. A *monitor* is a programming language construct with a set of operations. Only one thread can use a *monitor* at a certain point in time. Thus, *monitors* can be used for mutual exclusion. Often, *monitors* use the concept of *locks*.

A *lock*, also called *mutex*, is a mechanism for controlling access to a resource, thus enforcing mutual exclusion. A thread trying to access such a resource has to acquire this lock and then releases it when the access is finished. A *lock* can be blocking, which means a thread accessing a resource with a *lock*, has to wait passively if that resource is not free, or a *spinlock*, which means a thread waits and tries repeatedly to acquire the *lock*. A problem with *locks* is that the operation to check availability and to acquire the *lock* needs to be atomic to prevent synchronization problems. This is not guaranteed in software, and among the most prominent solutions to this problem are the first known accepted algorithm for this problem, Dekker's Algorithm (published by Dijkstra [Di65a]), Peterson's Algorithm [Pe81], the Eisenberg & McGuire algorithm [EM72], Lamport's bakery algorithm [La74], and Szymański's algorithm [Sz88].

If a thread accesses such a shared resource secured with a lock, it may have acquired one or more locks (due to former operations). The accumulation of the different *locks* held by a thread at this point in time is called *lockset* [Sa97]. *Locksets* are the second of the two central concepts used by most quality assurance techniques for data race detection.

The keyword *volatile* for variables in Java is used to guarantee that accesses to this variable are immediately visible to all threads and that there exists a happened-before relation between different operations on volatile variables [Ma04]. Actions on *volatile* happen in a total order. The usage of *volatile* does not induce the overhead of ensuring mutual exclusion [Ma04].

In general, there is a total order over all synchronization actions. Manson describes this order as follows: "volatile writes are ordered before subsequent volatile reads of the same variable. Unlocks are ordered before subsequent locks of the same monitor." [Ma04]

As mentioned before, operations in different threads cannot be totally ordered due to parallelism. However, synchronization mechanisms create dependencies between threads, which generally enable partial order relations between events in different threads.

Operations or keywords provided by programming languages that enable implementing and using the techniques or concepts are called *explicit* synchronization operations (as mentioned in chapter 2.2).

Synchronization, or mutual exclusion, may also be achieved using other programming constructs, such as the usage of a flag described in chapter 2.2, and is referred to as *implicit* synchronization.

2.4 Static and Dynamic Quality Assurance for Data Races

As mentioned in chapter 1.2, both static and dynamic quality assurance techniques should be used for data race detection. Both types of techniques, however, bring along drawbacks.

Static quality assurance analyzes applications without executing them. As such, programming constructs and their implications need to be considered during analysis. Static quality assurance suffers from three different drawbacks:

1) Code constructs that cannot be analyzed without ambiguity

2) Analysis techniques that have to trade off between precision and scalability

3) Code constructs that cannot be analyzed with static methods

In object-oriented programming, inheritance, polymorphism and dynamic binding bear problems in identifying correct paths, as it is often unclear before execution, which methods are called [ABF04], [SH20]. Thus, analysis may consider paths in a program, which cannot be executed. This may result in findings, which are not *feasible*, i.e., the program cannot be executed in a way that would enable such a data race to appear.

In addition, as mentioned before, often only *explicit* synchronization constructs are considered, but not *implicit* synchronization [NM92] (see chapter 2.3). As explained above, techniques only considering *explicit* synchronization report *apparent* data races, and not *feasible* data races [NM92] (see chapter 2.2).

Static analysis often relies on *lockset analysis* [NA07] (see chapter 2.3), i.e., for each access to a variable, the set of locks held is examined. If a variable may be accessed at the same time by different threads, the sets of locks held in these concurrently executing threads are compared. If the intersection of these sets is empty, this is interpreted as an access without mutual exclusion. Whenever such pairs of accesses are found, and one of these accesses is a write access, a data race is reported.

However, *lockset analysis* requires further techniques to be employed, since it is not always clear, if two references refer to the same element, or not. Due to this, pointer analyses in terms of *alias analyses* or *points-to analyses* are used [Sp16]. A *points-to analysis* is used to "determine the set of objects pointed to by a variable or field carrying a reference" [OM08]. An *alias analysis* [DMM98] determines accesses to memory locations. If several accesses exist to the same memory location, the references, or *pointers*, are said to *alias*.

Different algorithms exist for such analyses, and different sensitivity levels, which influence the precision and the general outcome of the analysis, can be distinguished. Widely used are context- and flow-sensitivity [SAB19], [HS09] as well as field- [SAB19] or object- [HS09] sensitivity. As an example, a flow-sensitive analysis considers the order of statements in an application, and a context-sensitive analysis distinguishes between different calls to the same method [HS09]. While sensitivity improves the precision, the analysis will become more complex, and due to this, scalability is affected negatively: "precise analyses need to encode a drastically larger - even infinite - data-flow domain that leads to analyses that are difficult to scale." [Sp19] Gharat, Khedker and Mycroft, as an example, presented a "fully flow-and context-sensitive exhaustive points-to analysis to C programs as large as 158 kLoC" (Kilo Lines of Code) in 2020 [GKM20]. However, modern applications can be much larger than this.

Due to this, a tradeoff has to be made in static quality assurance for data races: different algorithms for analysis with varying precision and scalability have been developed. Two of the most prominent algorithms used, Steensgaard's Algorithm [St96] and Andersen's Algorithm [An94], are flow-insensitive and context-insensitive. In practice, an over-approximation is used for analysis, i.e., the analysis is scalable and less precise, but does not contain false negatives. The drawback is that such analyses "produce an unacceptable amount of false positives". [Sp19]

Finally, code constructs exist that cannot be analyzed by static quality assurance techniques. Among those are code that is dynamically generated or loaded, binary or native code parts, which cannot be accessed by the algorithms used during static quality assurance, and programming techniques or code, such as reflection or invoke statements (e.g., for invoking a method that is defined by a value in a variable), for which no possibility exists to analyze them without executing the program [SK18].

Summarized, static quality assurance techniques imply a tradeoff between scalability and precision, leading to either a limitation in the size of the programs to be analyzed, or a high amount of false positives. These techniques do not produce false negatives, but false positives, with the exception of code that cannot be analyzed without execution of the program. Such code may also contain data races, which could then not be detected, thus resulting in false negatives.

Static quality assurance is thus a valuable means for data race detection, but should not be used alone, but in conjunction with dynamic quality assurance techniques.

Dynamic quality assurance techniques execute applications, usually using test cases, and use instrumentation. Code is injected into those parts of the application, which shall be analyzed. This injected code then enables to extract information relevant for the analysis. As explained in chapter 1.2, such instrumentation may lead to a massive runtime overhead and results in a changed runtime behavior of the application under analysis.

Dynamic quality assurance often relies on the *happened-before relation* (see chapter 2.2) to detect data races. The necessity of using global clocks mentioned in chapter 2.2 is replaced by using *vector clocks* [Fi88], [Ma88] or variants of those. The underlying concept is to establish causal relationships between events of different processes, so that the existence or non-existence of ordering relations can be used to assess events for possible data races.

The concept of *vector clocks* introduces the usage of several process-specific clocks in absence of a globally available clock. These clocks are combined and used as a *vector* [Ma88]. Considering three different processes, such a vector is three-dimensional (and n-dimensional for n different processes). Each dimension in each vector starts with the number zero, and can only be increased. Each process is assigned to one dimension. Processes may influence each other by sending messages and receiving messages, and process-internal events may exist. Each of these three types of events in such a process increases the value in this specific dimension. Every message sent contains a copy of the vector specific to the sending process. The vector in the receiving process is then updated, i.e., the vectors are combined, and for each dimension, the higher number is used. This way, an approximation of the global time is calculated based on the information available to this process. Considering two processes, the process-specific vectors used for a combination may be $(2, 0)$ and $(0, 3)$, resulting in the vector $(2, 3)$. Such vector clocks establish a causal relationship. If there is no causal relationship between two events, a possibility for a data race exists [Ma88]. The following

Figure 6, adapted from [Ma88], shows an exemplary computation including three different processes.

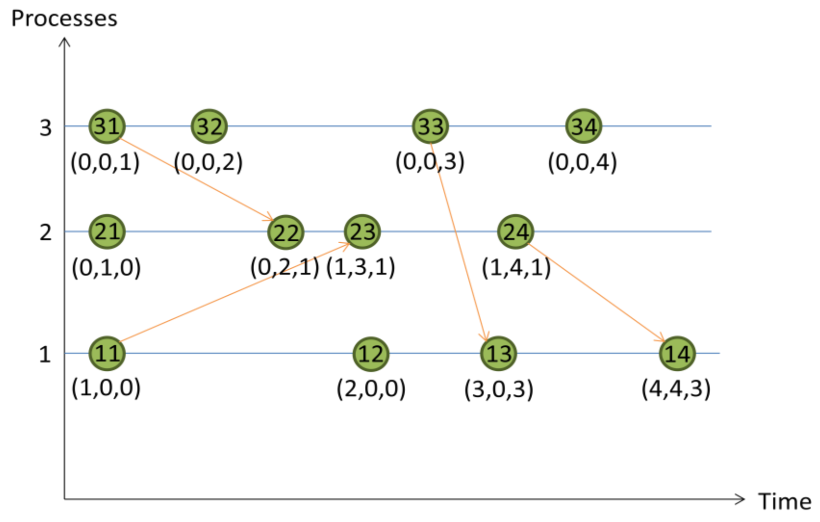


Figure 6: Event Diagram, Adapted from [Ma88]

In this figure, events are depicted with a green dot, containing a number indicating the process number ($1, 2,$ or 3), followed by the number of the event in that process. Arrows represent messages sent and received by processes. These messages, or arrows, establish causal relationships. As an example, event 33 sends a message to process 1, which is received in event 13. The vector of process 1 is then updated and changes from $(2,0,0)$ to $(3,0,3)$. For events in process 1, if the first number of the vector is equal to or greater than 3, an order is then evident related to all events of process 3 with the process-related number in this vector smaller or equal to that number (3). Event 14, with vector $(4,4,3)$ is in an order compared to event 32 with vector $(0,0,2)$. This order has been established by the message sent in event 33 and received in event 13. But there is no order established between event 14 with vector $(4,4,3)$ and event 34 with vector $(0,0,4)$. Thus, even if events 14 and 32 access the same variable and there is at least one write access, there is no possibility for a data race. However, a data race may, e.g., exist between events 14 and 34, or between events 12 and 32.

Vector Clocks are an early and fundamental concept. However, as systems have become dynamic, other concepts with more flexibility were needed and introduced. It may not be clear from beginning of the execution, how many processes may exist, as this may, e.g., depend on the input to the execution. In addition, processes may be created and finish dynamically during runtime of an application.

A concept considering this variability was created by Fidge [Fi91] a few years after the V*ector Clock* concept was introduced. Since then, various mechanisms have been developed, such as *Clock Trees* [Au97], improved *Matrix Clocks* [DB03] (originally introduced by [FM82]), *Tree Clocks* [La07], and *Interval Tree Clocks* [ABF08].

Vector Clock-based concepts focus on events in general, and this "generality of concepts if unnecessary in most cases" [FF09], when using such concepts for data race detection. As a result, optimizations exist, which reduce the complexity of such approaches. Among the most popular approaches is using an *epoch* to capture many of the events in a lightweight format, and then to order these *epochs* to the remaining *vector clock*-monitored events [FF09].

In addition to those concepts, many other mechanisms exist in other areas of computer science, such as *Version Vectors* [Pa83], used with regard to mutual (in)consistency in distributed systems.

Conducting dynamic quality assurance using the *happened-before relation* relies on an execution of the application to be checked. Thus, the quality of the results of such approaches depends on the proportion of executed different paths through an application compared to all possible paths through this application. Parts of the application that are not executed cannot be analyzed for data races using dynamic data race detection.

However, even parts of the program, which are executed, may contain data races, which are not detected by approaches using the *happened-before relation*. The order of events may differ in other executions, and such a different order may lead to data races not present in the observed execution of an application. Thus, in dynamic data race detection, techniques have been developed to reorder events in execution traces, with the goal to detect more data races. Such techniques are called *predictive data race detection* techniques.

However, the *happened-before relation* itself implies limitations in the possibilities to reorder events. These limitations prevent creating some orders of events, although these different orders of events are *feasible* [KMV17]. Therefore, other relations as *happened-before* have been explored in research, which allow more data races to be detected.

The *happened-before relation* was introduced within the context of message passing [Sm12]. Such message passing is strict in the sense, that sending and receiving messages is only possible in one ordering. However, mutual exclusion, e.g., introduced by using *locks*, is not limited to only one order of executing sections protected by locks [Sm12]. This means when two different parts of a program can only be executed one after the other due to a *lock* used, there is no strict order implied by this *lock*, which of these sections has to be executed first. Referring to Figure 6, if the arrows would not represent messages, but synchronized accesses, their direction could be reversed, leading to a different order of execution.

Research thus tried to create new ordering relations, which more flexibly capture the nature of issues related to data races. Smaragdakis et al. introduced the *causally-precedes relation (CP)* [Sm12], weakening the *happened-before relation*. The *CP relation* is targeted at causality of events [Sm12]. Due to this, it allows a more flexible reordering of events, and thus enables to detect more data races. However, this relation was criticized and another relation was introduced by Kini et al. [KMV17]. This new relation, *Weak Causal Precedence (WCP)* slightly relaxes the rules enforced by *CP*, to enable the detection of additional data races. However, the authors show that whenever their algorithm detects a so-called *WCP-race*, it could not only be data race, but also a deadlock [KMV17]. Although such a *WCP*-race is a true positive regarding concurrency, in the strict sense of data race detection, one could argue that such a finding might be a false positive.

Roemer et al. mention that *WCP* is "the weakest known" relation "that is also sound" (sound means in this context that any reported *issue* is a true *issue*), but still misses data races [RGB18]. The authors introduce a *doesn´t commute analysis (DC)*, which is weaker than WCP but may contain false positives [RGB18]. Thus, data race detection using only *DC* is not sound. Due to this, an additional component is used by the authors to analyze each reported issue [RGB18]. The authors present a proof that the detection using *DC* is complete, however, the analysis, used to filter out false positives by searching for a reordering that confirms a reported issue, is not complete [RGB18]. For their proof of completeness, the authors use the assumption that conflicting accesses cannot be reordered [RGB18].

Pavlogiannis shows that this assumption is not true in all cases, and presents examples, for which both the *DC* and the accompanying analysis of the results fail, thus producing false negatives [Pa19a]. The author introduced *Trace-close Partial Orders* and a decision-solving algorithm to decide if two events are conflicting, i.e., if a data race might exist. Although the author reports to find more data races than the approaches mentioned before, the algorithm is limited to input traces of two processes [Pa19a].

Summarized, there has been a development in using partial ordering techniques for data race detection towards weaker and more flexible orderings compared to the traditional *happened-before relation*, which allow finding more data races than stricter ordering techniques. However, this flexibility leads to more event reorderings, which have to be analyzed, thus increasing runtime overhead of the approaches. This problem of increased runtime overhead has been captured in Practical Problem 1 (chapter 1.2).

This increased runtime overhead is the reason, why Pavlogiannis has limited the data race detection to two processes [Pa19a]. Other approaches operate with a limitation in the reordering of events, the so-called *windowing*. When reordering events, such approaches do not examine the full set of events, but limit the set of events to a certain number of events, a *window*. As an example, Huang et al. limit their approach to windows of 10,000 events each, and these windows are examined separately [HMR14]. The downside of such an approach is, that events, which are in different windows, are analyzed separately, and thus, data races involving events in different windows cannot be detected. However, the tool implementing their approach can be configured to compute smaller or bigger windows, with reduced or increased runtime overhead.

A slightly different approach has been used by Mathur et al., who introduced the *schedulable happens-before* (SHB) ordering [MKV18]. This ordering builds upon the *happened-before relation*. The authors tackle a different problem, not considered by other approaches. Whenever a data race in an application exists, the behavior of this application after this data race is affected. This means that data races, which are detected after the first data race in an execution happened, might not reflect the behavior an application would show when this first data race would have been corrected [MKV18]. *SHB* strengthens the *happened-before relation*, and with using reordering of events following the SHB, aims to detect data races [MKV18].

Finally, an optimization for approaches using predictive data race detection has been presented by Roemer et al. [RGB20], using *conflicting critical section* optimizations for reducing the general runtime overhead of approaches. Using this optimization, critical sections are compared, i.e., sections protected, e.g., by locks, and *conflicting critical sections* are identified, which are used as basis for further analysis. Two critical sections conflict, when the same variable is accessed, and at least one of the accesses is a write access. The authors also use an optimization for the vector clocks used, and report an improvement in the runtime overhead, which allows predictive analysis techniques "to perform nearly as well as state-of-the art non-predictive race detectors" [RGB20]. While non-optimized predictive data race analysis usually can cause a runtime overhead of $30x$, the optimizations lead to a runtime overhead of $6x$ compared to a non-instrumented execution of an application [RGB20].

A completely different research direction is established by SAT (Satisfiability) - or SMT- (Satisfiability Modulo Theory) solving approaches. Such approaches are based on the same execution traces and events as the partial order based and predictive approaches described above. SAT- or SMT-solving approaches rely on execution traces and the characteristics of data races. A data race is then detected using a constraint solving algorithm. Feasible trace reorderings are examined using an SMT-solver, with consideration of certain constraints. Such reorderings are examined to detect data races.

A basis for this research direction was set by Serbănută et al., who introduced a *maximal causal model*, which opposes the partial order relations by focusing on consistency of shared memory accesses and formulating constraints for shared memory consistency [SCR08].

However, with regard to specific programming languages and their concurrency-related constructs (see Chapter 2.3), the *maximal causal model* is not encompassing, as not all such constructs are considered. Said et al. have thus extended the *maximal causal model* to include "a wide range of synchronization primitives in Java" [Sa11]. The authors also introduced the usage of an SMT-solver instead of enumerating all feasible trace reorderings [Sa11].

Based on this work, Huang et al. presented an approach with "maximal detection capability for any sound race detector given the same execution trace under sequential consistency" [HMR14]. The authors minimize the constraints for trace reordering to obtain maximal flexibility for such reorderings, thus enabling a higher detection capability [HMR14]. The authors formulate a data race as a property over a model of execution traces, representing the set of feasible traces following the *maximal causal model*. This model is created using a formula for specifying all feasible traces based on the observed execution trace, while adhering to the *maximal causal model.* Constraints are then formulated for *must happen-before* relations, locking, and for data races. The *must happen-before* constraints differ from the *happened-before relations* already mentioned, as the constraints are not used with regard to read and write events for different threads, and acquire and release events. This is because, as explained above, such an ordering would unnecessarily limit the flexibility and thus, prevent detecting possible data races. Instead, the *must happen-before* constraint considers the creation (i.e., fork) and start events of a thread, and the end and possible join events. Lock constraints are used for formulating mutual exclusion consistency with regard to acquire and release events. Race constraints consist of arithmetic constraints with regard to read and write events to the same variable, and control flow constraints [HMR14].

An SMT-solver is then used to examine possible feasible traces for the constraints formulated with regard to a pair of variables. If the SMT-solver reports a feasible trace, a data race is detected [HMR14].

Besides their usage for detecting data races in multithreaded programs, such SMT-solver based approaches are also used in the area of distributed systems [PMS20].

Finally, a variety of other approaches for quality assurance with regard to data races exist, such as explicitly specifying mechanisms for synchronization and then verifying the adherence of an application to those specifications [FF20].

Hybrid approaches combine one or more techniques, such as lockset analysis and happened-before analysis.

As runtime overhead is one of the prevalent problems in quality assurance for data races, as shown above, due to the inherent complexity [MPV20], research also focused on optimization approaches that complement existing approaches for data race detection. Besides optimizations regarding the vector clock concept [FF09], or regarding conflicting critical sections as basis for the analysis in predictive data race detection [RGB20], the number of events to consider during an analysis has been targeted.

Huang et al. introduced an approach to filter out events that are redundant. Redundancy in this context means, that those events cannot be used to detect new data races, e.g., multiple read or write events to the same variable in the same critical section [HZZ13], [RH15].

Other approaches use a sampling strategy. Using such a strategy, not all read or write events are examined. Instead, a proportion of those events is examined, leading to a reduced runtime overhead [BCM10], [MMN09], [BCM10]. However, this leads to the possibility of missing data races, which could have been detected examining those events, which are ignored.

Besides sampling instrumentation, another strategy is to reduce the level of detail of the instrumentation. Von Praun and Gross used instrumentation on the object level instead of instrumenting single variables [PG01]. The authors use this concept for object race detection, and not data race detection [PG01].

The concept of object level instrumentation has been adopted for data race detection by Yu et al., who relied on an adaptive analysis [YRC05]. The authors dynamically adapt the tracked granularity (e.g., object granularity and field granularity) and history (lockset and set of threads) information of an application, with the goal to reduce the runtime overhead [YRC05]. As such a reduced amount of details may lead to missed data races, the algorithm also presents warnings to the users [YRC05].

Effinger et al. introduced *interference-free regions* to reduce instrumentation [Ef12]. Instead of instrumenting single accesses to a variable, regions are identified for instrumentation, and multiple accesses to the same variable are combined during the instrumentation. This leads to a lower runtime overhead, but this approach may miss data races [Ef12].

The approach presented in this thesis can be classified into the optimization approaches. However, instead of targeting the events in an execution trace, the instrumentation itself is in the focus for optimization. Based on an analysis of the dynamic behavior of an application, classes to instrument are identified, and instrumentation is conducted only for those places, which may be executed in parallel to other places or to themselves. For such an analysis, an abstract model of the dynamic behavior is used, leading to reduced complexity compared to analyzing whole applications.

2.5 Behavioral Models of Software

Various languages and notations are employed in the area of model-driven development. Seven languages and notations can be identified as being the most prominent ones [BK20], [St20], [GG21], [Sc21]:

1. BPMN (Business Process Model and Notation)
2. WS-BPEL (Web Services Business Process Execution Language)
3. ER Diagrams (Entity-Relationship Diagrams)
4. EPC (Event-Driven Process Chains)
5. Petri Nets
6. YAWL (Yet Another Workflow Language).
7. UML (Unified Modeling Language)

BPMN [OM13] provides a notation for business users for creating business processes. Developers may implement the software supporting those processes using the created model. *BPMN* is used to create "a standardized bridge for the gap between the business process design and process implementation" [OM13]. A *BPMN* model may consist of three types of sub models: processes, choreographies, and collaborations. A process is a "sequence or flow of activities". A collaboration is used to model interactions between business entities [OM13]. A choreography defines the expected behavior between different processes. The notation provides, among others, elements for events, activities, message exchange, data objects, sequence flows, and gateways (used for decisions, merging, forks, and joins) [OM13]. The *BPMN* contains compliance points, and software can claim compliance with *BPMN* if those compliance points are fulfilled [OM13]. Software may be modeled using this notation, but the intent of *BPMN* is not to provide a notation for modeling software. Instead, "Inter-operation of Business Processes at the human level, rather than the software engine level, can be solved with standardization of the Business Process Model and Notation (BPMN)" [OM13].

WS-BPEL is directed at specifying the behavior of business processes based on web services in the form of abstract and executable business processes [OA07]. The intention of this language is to "achieve interoperability between applications by using Web standards" [OA07]. The language defines, among others, elements for activities, message exchange, conditional behavior, repetitive behavior, concurrency and synchronization (both as special forms of activities), as well as events [OA07]. The *WS-BPEL* standard does not provide a modeling notation, but only a language in a style similar to XML [OA07]. As with *BPMN*, this language is targeted at business processes, and not software. The *BPMN* standard provides a mapping from *BPMN* to *WS-BPEL* [OM13].

The *Entity Relationship Model*, introduced by Chen [Ch76], serves to model data and associations between data. It consists of entities and entity sets, relationships, roles and relationship sets, and attributes, values and value sets. An entity, which can be classified into an entity set, is described as "a thing which can be distinctly identified" [Ch76]. A relationship is used to describe associations between such entities. Relations can be formulated as a relationship set, a mathematical relation. An entity can have a role associated with a relationship, which is used to define "the function that it performs in the relationship" [Ch76]. An entity or a relationship can have attribute-value pairs, to express information about such an entity or relationship. Values can be classified into value sets. An attribute is a function mapping from a relationship set or an entity set into a value set [Ch76]. An *Entity Relationship Model* is a data model, originally used for database design [Ch76], and does not contain any elements to describe the behavior of a system.

Event-Driven Process Chains are process models used to describe a dynamic view on the execution of functions driven by events [KNS92], the control flow. An event is a passive state representing the occurrence of values of attributes, which triggers a function. A function describes the execution of an operating process contributing to reaching a corporate objective [KNS92]. In the model, events may trigger functions, which may lead to triggering an event. Events and functions may be connected directly or using connection operators. Such connection operators may describe a conjunctive connection, a disjunctive connection, or an adjunctive connection. To model the data view and for meta modeling, originally *Entity Relationship Models* were used [KNS92]. However, using these models for meta modeling was discarded in favor of the *UML* [Sc02]. *Event-Driven Process Chains* are targeted at organizational workflows, and are separated from implementation specific considerations related to software [NZ98].

Petri Nets, introduced by Petri, consist of a formal model and graphical notation for describing distributed systems [Pe62], thus providing support for concurrency. *Petri Nets* are used in many different areas, including software engineering and workflow specification. A *Petri Net* consists of places, representing states, and transitions, representing state changes. Graphically, places are represented as circles, and transitions are represented as rectangles. Different definitions of *Petri Nets* exist, which are mostly equivalent [Wi08]. One definition states that *Petri Nets* are a triple (P, T, F), with P being a finite set of places, T being a finite set of transitions, and F being a flow relation ($F: P \times T \cup T \times P$). Flow relations are graphically represented as directed arcs leading from a place to a transition or leading from a transition to a place.

The state space (the set of all states) of a *Petri Net* is defined as $\mathbb{N}^P$, with the relation $s: P \rightarrow \mathbb{N}$ being a state or marking of a *Petri Net* [Wi08]. If $s(p_i) = k$, then the *place* p_i holds k tokens. Graphically, a token is shown as a dot in the place. A transition in state s is enabled, i.e., the *Petri Net* may change its state, if $s \geq F(\cdot, t)$, meaning $\forall p \in P: s(p) \geq F(p, t)$. In other words, a transition is enabled, if there is a token in each place connected to this transition, with this connection representing a flow from this place to the transition.

The state s changes to s' if $s \geq F(\cdot, t)$ and $s' = s - F(\cdot, t) + F(t, \cdot)$ [Wi08]. A *Petri Net* starts at an initial state s_0 and is called dead when no transition is enabled. Different variations of *Petri Nets* exist, such as *Colored Petri Nets* or *High-Level Petri Nets* [Je82].

Yet Another Workflow Language has been created with consideration of *Petri Nets*, but with a different semantics [vH05]. Van der Aalst and Hofstede examined different workflow patterns, and the suitability of *Petri Nets* and several workflow management systems to model those patterns [vH05]. On the basis of their results, the authors extended *workflow nets*, which are themselves extensions of *Petri Nets* [va98], to support constructs, such as multiple instances with different levels of synchronization or cancellation patterns. Although *Yet Another Workflow Language* is formally defined, with similarities to *Petri Nets*, the token concept, e.g., is not formally defined [vH05]. *Yet Another Workflow Language* may be used for modeling software and its behavior, but the language is focused on workflows.

The *Unified Modeling Language* (*UML*) consists of a language and specifications for a variety of models and is accompanied by graphical representations of such models. The *UML* is currently available in Version 2.5.1 [OM17]. The *UML* originated from different concepts for object-oriented development [BRJ96] and is targeted at analysis, design and implementation of software systems, although processes and workflows also may be modeled using the *UML*. The specification consists of two main areas: structural modeling (e.g., using the graphical representations *class diagram* or *component diagram*), and behavioral modeling (e.g., using the graphical representations *activity diagram* or *sequence diagram*). The focus in this thesis is on *UML Activities* instead of other representations of the *UML*, because the focus is on representing the behavior of an application and because *UML Activities* are more commonly used for representing complete applications. Just as *UML Activities*, *UML Sequences* may also be used for modeling concurrent behavior, but *UML Sequences* are more likely to be used for modeling the interactions between objects in specific cases, and not the application as a whole [Fo10]. In recent versions, especially *UML Activities* have received changes in the form of extensions and formalizations. Basically, an *activity* captures dynamic behavior and consists of nodes and edges. Nodes may be executable, hold data, or may be nodes "specifying the sequencing of executable nodes" [OM17]. Edges may be control flow edges or data flow edges. As such, activities represent both the control flow and the data flow. *Activities* describe "models of computation" [OM17], which are "inherently concurrent" [OM17]. *Activities* may form hierarchies, and special constructs exist for, e.g., timing, events, signals, or exceptions [OM17].

2.6 Summary

Research in the area of concurrency has been conducted both to understand and define concurrency and related issues, such as data races, and to cope with these issues in a constructive and analytical manner.

Based on the *happened-before relation*, describing a partial order over events in parallel or concurrent processes, a formal definition for *data races* has been defined. In addition, data races were separated into *feasible data races* and *apparent data races*.

To cope with concurrency, the concept of mutual exclusion was defined. In development, both implicit and explicit synchronization can be implemented using different programming language constructs. Explicit synchronization is implemented using specific programming language elements, such as *semaphores*, *monitors*, or *locks*. Besides the *happened-before relation*, the concept of *locksets* is used as central concept for quality assurance for data races.

Static quality assurance often relies on *lockset analysis* as central concept. As the source code of an application is not executed, not all code constructs can be analyzed correctly or without ambiguity. In addition, static quality assurance techniques have to balance between precision and scalability. Although static quality assurance in general may be able to detect all data races, with the limitations mentioned, the techniques usually suffer from a huge amount of false positives. One reason is that static quality assurance techniques often not only report *feasible data races*, but also *apparent data races*.

Dynamic quality assurance, on the other hand, classically only reported *feasible data races*, but was limited to data races related to executions observed. Based on the *happened-before relation*, other relations were defined, which were less strict and enabled data race detection techniques to find a higher number of data races. Additionally, techniques to reorder events in the observed execution traces were developed. The development of less strict ordering relations has lead to techniques, which report a high number of data races, but also report false positives.

As a less strict ordering relation implies more possibilities to reorder events in execution traces, more data races could be detected, but at the cost of an increased runtime overhead. This has lead to a research area focused on optimizations of existing approaches in terms of the runtime overhead. Optimizations were introduced regarding the underlying *vector clock* concept, the analysis phase in predictive data race detection, the events of an execution trace to consider for analysis, and, as in this thesis, the instrumentation itself.

Behavioral models form the basis for the solution presented in this thesis. In model-driven development, several different approaches for modeling exist, and seven languages and models have been identified as being the most prominent ones. These approaches vary in their main application domain, targeting mainly business processes or workflows, or mainly software development, have different levels of underlying formalisms, and vary in their support for concurrency and software related constructs.

3 Related Work

This chapter shows an overview on the related work with regard to the practical and scientific problems of this thesis, and the research questions.

The purpose of this chapter is on the hand to discuss the decisions made with regard to the solution presented in this thesis, and on the other hand to assess similar approaches found in the literature regarding their ability to fulfill the goals of this thesis.

3.1 Research Approach

As shown in chapter 2.4, a variety of static and dynamic techniques exist for quality assurance focused on data races. Research has led to a substantial improvement in the effectiveness of quality assurance approaches, but at the cost of increased runtime overhead. Although research has identified approaches to reduce this runtime overhead, it is still perceived as a problem. Current solution ideas focus on different aspects with regard to the usage of the data collected by instrumenting the code, or on different depths of instrumentations. Research on the instrumentation itself has focused on *how* to instrument code, but not on *what* to instrument. The solution presented in this thesis is new in terms of providing a systematic approach regarding *what* to instrument.

Existing approaches rely on a sampling strategy to steer the examination of read or write accesses, which may lead to missed data races, as explained in chapter 2.4. The solution idea presented in this thesis instead steers instrumentation based on places in the source code, i.e., source code classes. The definition of a data race (as stated in chapter 2.2) shows that several requirements need to be fulfilled for a data race to exist: multiple accesses to the same data with at least one of the accesses being a write access, and concurrency without proper synchronization. The approaches discussed in chapter 2.4 are based on these requirements to detect a data race. The solution presented in this thesis reduces the runtime overhead by focusing on one of these requirements, the concurrency. If there is no concurrency, i.e., no parallel execution of threads, there cannot be a data race, independent of what data is accessed and how it is accessed. If source code classes in the code exist, which cannot be executed in parallel to other source code classes, they do not need to be analyzed regarding possible data races, as there cannot be data races. Thus, there is no need to instrument those source code classes.

To obtain the basis for such a decision on *what* to instrument, a model of the dynamic behavior of a system is analyzed. As multiple models are used in the area of model-driven development (see chapter 2.5), an assessment is made in this chapter, which model to use in the solution.

Having identified the model, requirements for an approach to analyze such models can be set up. Related work is then analyzed with regard to their ability to fulfill these requirements.

The related work described in this chapter has been identified and analyzed following the guidelines for systematic literature reviews defined by Kitchenham et al. [Ki07b], [KE14]. The following figure shows the steps conducted for the literature review.

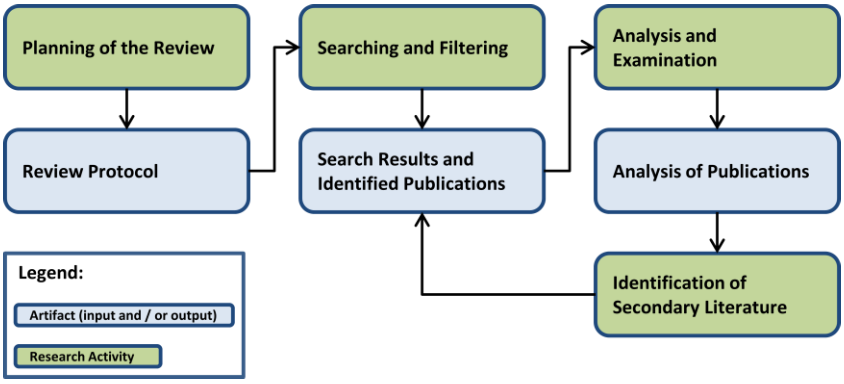


Figure 7: Literature Review of Related Work

As such a systematic literature review, its goal and the research question to be answered, as well as the requirements for a solution, depend on a clear definition of the type of model in the focus of research, the decision on the type of model to focus on has to be made before starting the systematic literature review. This decision is presented next.

A detailed explanation of all activities conducted during the literature review, following the guidelines provided by Kitchenham et al. [Ki07b], is given in chapter 3.3.

3.2 Behavioral Models of Software as Basis for the Analysis of the Dynamic Behavior

The models presented in chapter 2.5 target different areas and can thus be classified into the following categories:

1) Workflow and process models: BPMN, WS-BPEL, EPC, YAWL

2) Software models: ER Diagrams, Petri Nets, UML

As workflow and process models target different areas, and are not meant for modeling software, they have been discarded. Of the software models, ER Diagrams serve to model data and relations between entities. Such diagrams are not able to describe the dynamic behavior of software.

Of the models mainly used in model-driven engineering, only two types are suitable for representing the dynamic behavior of software, and have thus been examined in detail. The assessment resulted in *UML Activities* being selected as the type of behavioral model used as basis for the analysis of the dynamic behavior of software.

Both *UML Activities* and *Petri Nets* share a token concept. While for *UML Activities*, the tokens are related to the execution of the activities (regarding control tokens), the tokens in *Petri Nets* are related to their state. The token concept in *Petri Nets* is embedded in formalisms as foundation for the model. *Petri Nets* are based on formal semantics, and analysis techniques exist for different properties, such as invariants or deadlocks [va98]. In contrast, *UML Activities* and the related token concept are not formally defined.

However, van der Aalst and ter Hofstede mention three "serious limitations" regarding Petri Nets [vH05]:

1) A process or thread may instantiate several sub threads. The number of such threads may be flexible, and, e.g., depend on the input to an application. In addition, some of those threads running in parallel may need to be synchronized at various occasions. In such a situation, creating a *Petri Net* may require keeping track of the identities of those sub threads and the number of threads active using a counter [vH05]. Van der Aalst and ter Hofstede judge the tasks to create such elements manually by designers as being "not acceptable" [vH05].

2) Additional challenges arise when considering threads that are optional, i.e., in some cases, a number of threads need to be executed, and in other cases, a different number of threads need to be executed in the context of the same main task. After all those threads have been completed, the main task may be continued. Such a situation may, e.g., arise in computations, for which different sub tasks of this computation are spread to different threads. Thus, different forms of synchronization occur: no synchronization, partial synchronization and full synchronization [vH05]. In such cases, substantial effort has to be made to create appropriate Petri Nets capturing this flexibility [vH05].

3) By definition, enabling a transition in *Petri Nets* is based on tokens being present on incoming arcs; this transition is always local [vH05]. Thus, to capture non-local events, such as a timer or a cancellation of an action, additional effort is necessary to capture the nature of removing tokens based on non-local events. This often results in "spaghetti-like" diagrams [vH05].

Using *UML Activities*, such situations may be modeled using standard modeling elements provided by the language. *ControlNodes*[4], such as *ForkNodes* and *JoinNodes*, *DecisionNodes* and *MergeNodes*, are used to manage the control flow. Additional nodes may, among others, be used for referencing other activities, i.e., to establish calling hierarchies, to send and receive signals, to raise and react to events or exceptions, to model loops, or to define groups of nodes, which may be interrupted during execution [OM17].

These differences show the different foci and strengths of the languages. Although *Petri Nets* enable automated analyses, modeling programming constructs as those mentioned above, are only possible with additional effort. This shifts the effort of modelers from modeling the actual program behavior towards modeling administrative constructs [vH05]. Compared to using *UML Activities*, more effort is thus needed for modeling a concurrent application using *Petri Nets*. The analysis capabilities regarding data races using *Petri Nets* are limited in terms of the state space or the variables to be analyzed and often require additional manual steps [BHO20], [KO20], [XZL21].

Thus, the existing advantages of *Petri Nets* regarding analysis cannot be fully used in the context of this thesis. The disadvantage in terms of modeling the behavior of concurrent applications however, is present. With regard to this and considering the possibilities of transforming UML constructs to source code for further usage of the results of modeling, i.e., a more efficient software engineering process, *UML Activities* have been selected as representative for behavioral models used in the solution presented in this thesis.

3.3 Analyzing UML Activities

To retrieve related work in terms of using *UML Activities* in the detection of data races, a systematic literature review was conducted as mentioned in chapter 3.1.

When examining the UML superstructure [OM17] regarding *UML Activities*, several requirements can be defined for an approach to analyze such *UML Activities*:

[4] In the following, the elements of *UML Activities* are written according to the notation in the UML superstructure [OM17]. As an example, a "control node" is written as "*ControlNode*".

- Requirement 1 – Multiple *UML Activities*: using the node type *CallBehaviorAction*, it is possible to connect *UML Activities*. Situations can be created, in which an activity $\boldsymbol{A}$ contains a *CallBehaviorAction*, which leads to activity $\boldsymbol{B}$, which in turn contains a *CallBehaviorAction* leading back to activity $\boldsymbol{A}$. This would lead to an activity spanning cycle. Any approach capable of fully analyzing *UML Activities* must not only consider *CallBehaviorActions*, and thus a set of connected *UML Activities*, but also provide solutions for such an activity spanning cycle.

- Requirement 2 – Multiple *InitialNodes*: as defined in the UML superstructure, an *InitialNode* is a starting point for the execution of an activity. However, any activity may contain several *InitialNodes*. When execution of such an activity begins, all *InitialNodes* are executed, thus forming a parallel execution without a *ForkNode*. An approach capable of handling concurrency must be able to consider multiple *InitialNodes* as a source of parallelism.

- Requirement 3 – Multiple *FinalNodes*: according to the UML superstructure, there is no limit in the amount of *FinalNodes* included in an activity [OM17]. While *FlowFinalNodes* end one of possibly multiple executions (i.e., a flow of tokens) in an activity, an *ActivityFinalNode* ends the execution of the whole activity (i.e., all token flows). An approach must consider that multiple *FinalNodes* with different semantics may exist within the same activity.

- Requirement 4 – Cycle Traversals: an activity may contain node combinations that form cycles. Depending on the nodes within that cycle, such a cycle may have different exit conditions, which may require multiple cycle traversals to be fulfilled. The exact amount of necessary cycle traversals depends on the specific cycle characteristics. Thus, any approach properly handling cycles must take into account the characteristics of such cycles.

- Requirement 5 – *ForkNodes* and *JoinNodes*: according to the UML superstructure, a *ForkNode* has multiple outgoing edges, which are all followed concurrently [OM17]. A *JoinNode* has one outgoing and multiple incoming edges and may have a *joinSpec*, which specifies the condition, which must hold for the *JoinNode* to offer a token at the outgoing edge. The UML superstructure mentions no dependency between *ForkNodes* and *JoinNodes* [OM17]. An activity may contain *ForkNodes* without *JoinNodes*. In case of multiple *InitialNodes*, *JoinNodes* may also be present without any *ForkNodes*. An approach handling concurrency therefore must account for both *ForkNodes* and *JoinNodes* independently.

- Requirement 6 – *UML Activity* Elements: besides well known control node types, such as *InitialNodes*, *FinalNodes*, *ActionNodes*, *DecisionNodes*, *MergeNodes*, *ForkNodes* and *JoinNodes*, and object node types, many other node types or elements exist. Examples for additional elements are *AcceptEventActions*, *StructuredActions*, *InvocationActions*, *RaiseExceptionActions*, *InterruptibleActivityRegions*, *InterruptingEdges*, or different types of *Pins*. Any approach fully capable of analyzing *UML Activities* must consider the complete set of elements related to *UML Activities* specified in the UML superstructure [OM17].

3.3.1 Process of the Systematic Literature Review

The first step was the planning of the literature review. In this phase, the rationale for the survey and the research question to be answered were specified. Based on those, the search strategy including selection and assessment criteria could be defined. The search was limited to publications from 2005 on, as in this year the UML was released in Version 2.0, which included several additions to *UML Activities*.

The research question to be answered by the literature review was: *What approaches exist to detect places with parallel access to shared data in UML Activities?*

This question targets the existing literature regarding approaches analyzing *UML Activities* with the goal to identify either data races directly or modeling elements involved in data races.

The second step (searching and filtering) included testing and then applying the search string developed as part of the search strategy to the identified sources. The search string was applied at SCOPUS, IEEE Xplore Digital Library and ScienceDirect. Those libraries include most of the relevant work in the area of this thesis. The libraries also cover contents of other databases, such as the Proceedings hosted by Springer. Thus, a search using other databases was not necessary.

The first part of the analysis consisted of filtering out duplicates and invalid entries, such as entries referencing tables of contents of proceedings. As many publications were found using the search string, which were not related to the context of this literature review, a filtering was applied by reading and analyzing the titles and the abstracts (step two). Publications, which were found to not being relevant in the context of this research, were sorted out. Publications selected as being relevant were then analyzed in detail (step three). The references of those publications have been examined to identify additional publications not yet identified (step four). For those additional findings, the filtering and analysis approach steps have been repeated.

The systematic literature review and the process described have been repeated several times, with the last time of the review being at the time of writing the thesis. Although results published in the last years could not be considered when creating the solution presented in this thesis, they serve for comparison regarding the current state regarding approaches to analyze *UML Activities*.

Although a systematic approach was applied, this chapter is not entitled to provide a complete overview on all publications related to the topic of this thesis.

The majority of the publications found are concerned with methods for test generation. The remaining publications focus on other model-related quality assurance techniques, such as model checking, formal modeling and proof, product line tests, methods, which do not focus on test generation, and analysis techniques. The literature review also revealed two methods for model conversion, i.e., one technique for model reduction, and one technique for the specification of Event-B models based on *UML Activities*. One publication is concerned with a comparison of two methods for test generation based on *UML Activities*, but does not present an own method.

13 publications contained an approach for the analysis of *UML Activities* considering parallel access to shared data or nodes, which may be executed in parallel to other nodes or to themselves. The analysis revealed several limitations and inabilities in the approaches.

3.3.2 Sun

The approach from Sun et al. [Su08], [SZL09], [Su15] targets concurrent applications and derives test scenarios based on coverage criteria. The approach transforms *UML Activities* to extended binary trees. Then, the nodes are traversed to generate test scenarios. Finally, test cases are derived. The approach is limited to *UML Activities*, which include specific combinations of *ForkNodes* and *JoinNodes* to detect concurrency, for which all outgoing edges of a *ForkNode* eventually lead to incoming edges of one *JoinNode*. Such combinations form clearly limited areas, concurrent regions, in which parallel execution of nodes is possible. This results in a defined amount and combination of nodes between a pair of *ForkNodes* and *JoinNodes*. The authors describe three coverage criteria for concurrent regions to steer the amount of test cases in terms of the coverage of paths achieved. "Weak concurrency coverage" is used for generating one sequence of nodes for a concurrent region without considering interleavings between parallel nodes. "Moderate concurrency coverage" results in generating all sequences regarding parallel branches without considering different interleavings of parallel nodes. "Strong concurrency coverage" is used for generating all sequences of nodes, with consideration of interleavings. As *UML Activities* may contain cycles, the solution to the problem of how to control and limit possibly infinite cycle traversals is critical to any approach. Sun et al. limit the cycle traversal to one cycle traversal. The approach only supports *UML Activities* with one *InitialNode* and one *FinalNode*.

3.3.3 Sapna

Sapna and Mohanty [SM08] present an approach to generate test cases in terms of paths through a *UML Activity*. The authors rely on user input to apply criteria for constraints. These constraints result in a reduction of the interleaving possibilities of concurrent operations defined in *UML Activities*, and thus, in a reduction of the test cases to generate. In the article, two types of criteria are defined for test scenario generation: priority-based criteria, in which the interleavings are based on priorities added by users, and level-based selection, in which dependencies are used to reduce the number of possible test paths. In this context, dependency means that one node has to be executed before another node can start. To steer the amount of paths generated for testing, three coverage criteria are supported in addition: node, transition and path coverage are supported. The approach also relies on combinations of *ForkNodes* and *JoinNodes* to detect concurrency. There is no proposal to solve the problem of cycle traversals and infinite paths. Instead, cycles are only traversed at most twice. The approach supports *UML Activities* with one *InitialNode* and one *FinalNode*.

3.3.4 Xu

Xu et al. present an algorithm to generate abstract test cases from *UML Activities* [XLL05], [Xu08]. The authors propose using adaptive agents to find all paths in activities. The authors mention that the tool also relies on pairs of *ForkNodes* and *JoinNodes* to detect concurrency. The authors present four patterns for such pairs, which describe different scenarios, such as a nested fork-join, or a branch inside a fork-join-pair. Test cases are derived based on all interleavings of nodes within such concurrent regions. The approach supports one *InitialNode* and multiple *FinalNodes*. Cycle traversals are limited to a fixed number of traversals, which can be defined by users. *ExpansionRegions*, which can be executed multiple times depending on the input provided to that region, are limited to a single execution. *ExceptionHandlers* are considered in the test case generation similar to a *DecisionNode*, i.e., two test cases are created. One test case assumes that an exception has been raised, and the second test case assumes that the exception has not been raised. For *InterruptibleActivityRegions*, all possibilities for an event that interrupts execution of the nodes in that *InterruptibleActivityRegion* are listed and multiple test cases are created.

3.3.5 Chandler

Chandler et al. present a method to generate usage scenarios for testing of programs based on *UML Activities* [CLL07]. For creating such usage scenarios, each edge of a *UML Activity* is visited at most once during a cycle traversal, and twice overall. The authors enlarge the patterns for detecting concurrency presented by Xu et al. [XLL05], and present six categories for classification of situations describing concurrency, which are based on the aforementioned approach, and extend these. The patterns include nested fork-join-pairs and cycles inside a fork-join-pair. These patterns also describe situations, in which the processing of nodes inside such a fork-join-pair is dependent on certain guard conditions. For these patterns, possible test sequences are generated. However, more complex situations are not supported, e.g., activities, which do not contain pairs of *ForkNodes* and *JoinNodes*, or branches and cycles, which span across the borders of *ForkNodes* and *JoinNodes*. The approach considers multiple *InitialNodes* and *FinalNodes*, and relies on combinations of *ForkNodes* and *JoinNodes* to detect concurrency. The algorithm processes *UML Activities* and generates paths through the diagrams, which can be used for testing.

3.3.6 Lei

Lei et al. present an approach and a tool for testing Java programs based on *UML Activities* for data races [LWL08]. Data races are identified by checking state transitions of the shared variables. To enable the analysis of state transitions for shared variables, *UML Activities* are extended with so-called "data operation tags" and the system under test is instrumented accordingly. Test cases are generated randomly by using a path analysis based on the *UML Activity*. Several restrictions apply with regard to the *UML Activities*: swim lanes are mandatory and are named after the class in the source code. Any method that implements a certain node of the *UML Activity* must be known and annotated to this node. The swim lanes and the references to source code in the *UML Activities* have to be annotated manually. There is no statement regarding handling of cycles in *UML Activities* for path analysis. The generated test cases can be executed using the provided tool. The information added to *UML Activities* is used for instrumentation of the source code. In addition, users are required to specify the input and the expected output regarding the test cases. During execution of the system under test, a trace file is written, and monitored state transitions of the shared variables are analyzed for data races. Access times are recorded by wrapping the first and last lines of getter and setter methods for shared variables. If time overlaps are found, a data race is reported. However, the authors mention, that due to the inherent non-determinism related to concurrency, the system under test has to be executed several times to reveal specific interleavings. The approach supports activities with one *InitialNode* and multiple *FinalNodes*.

3.3.7 Boghdady

Boghdady et al. transform *UML Activities* stored in an XML format to a table containing information related to the nodes, input and output, and predecessor and successor information [B011b]. The table is then used to create an activity dependency graph. Paths through this directed graph form test cases. The approach relies on a single *InitialNode* and *FinalNode*. Cycles are traversed at most once. Parallel nodes are summarized to a single node in the activity dependency graph. Thus, clearly separated concurrent regions are required by the approach.

3.3.8 Kundu

Kundu and Samanta use directed cyclic graphs as intermediate representation of *UML Activities* [KS09]. Those graphs are used for generating test cases. The approach only considers isolated activities and relies on a single *InitialNode*, but multiple *FinalNodes* per activity are considered. Cycles are traversed at most once. The approach also requires concurrent regions, and parallel nodes are serialized, i.e., one sequence of node execution is considered.

3.3.9 Kim

Kim et al. present an approach for black box testing, and focus on inputs and outputs from and to users or testers. The authors propose to transform *UML Activities* to I/O explicit activity diagrams [Ki07a] (I/O refers to Input/Output). Then, a directed graph is generated and test cases are extracted as paths through this graph. It is unclear, how the information whether an action node involves data visible to users is obtained. Activities may only include a single *InitialNode* and a single *FinalNode*. Cycles are traversed once. The approach relies on concurrent regions. Nodes in the concurrent region are classified as being related to either input or output data. Only nodes related to input are considered regarding interleavings of concurrently executing nodes. It is assumed that "if a tester waits long enough", then nodes providing output "will eventually execute" [Ki07a]. This assumption is used for reducing the amount interleavings considered for test cases by always using nodes providing output as starting points.

3.3.10 Verma

Verma and Arora present an approach to generate test cases based on isolated *UML Activities* [VA14]. Based on an XML representation, in a first step all incoming and outgoing edges for each node are identified. Assuming only one *InitialNode*, all test cases start with this node. The identified outgoing edges of this node are then used to identify the successor node. If a *DecisionNode* is reached, the path up to this node is copied, and each outgoing edge of the *DecisionNode* is appended to one of the copies, i.e., additional test cases are created. In case of *ForkNodes*, all paths are followed "simultaneously" in the same test case [VA14]. The pseudo code algorithm does not contain information, how exactly test cases are generated in such a case. After all test cases are generated, all test cases, which do not contain pairs of *ForkNodes* and *JoinNodes*, are discarded. Cycles are traversed once.

3.3.11 Lima

Lima et al. propose a framework to detect non-determinism in *UML Activities* [LTN19]. The authors use CSP, process algebra, as representation. *UML Activities* are transformed to CSP and then analyzed using an existing model checker. The approach considers *SendSignalActions*, *AcceptEventActions*, and *CallBehaviorActions*. No information is given related to the handling of cycles of activities calling each other. *UML Activities* compatible with the approach may have multiple *InitialNodes* and *FinalNodes*. There are also no restrictions regarding *ForkNodes* and *JoinNodes*. The model checker, however, only reports that non-determinism is present, and gives an example of such a situation, i.e., a path leading to a point in an activity, where the non-determinism occurs. Such a non-deterministic situation may not necessarily be related to parallelism, but may also be related, e.g., to a *DecisionNode* with outgoing edges, for which it is not defined which edge to follow [LTN19].

3.3.12 Summary and Assessment

Summarized, none of the approaches contain a formal and theoretic basis for the included algorithms, which is shown to be correct. Several of these approaches also rely on pairs of *ForkNodes* and *JoinNodes* to detect concurrency [Su08], [SZL09], [Su15], [SM08], [XLL05], [Xu08], [CLL07], [B011b], [KS09], [Ki07a], [VA14]. However, there is no mandatory relation of *ForkNodes* and *JoinNodes*, and *UML Activities* adhering to the UML specification can include *ForkNodes*, but no *JoinNodes*, and may also contain multiple *InitialNodes*, which then start in parallel. Most approaches restrict the number of *InitialNodes* and / or *FinalNodes* to be contained in *UML Activities* [Su08], [SZL09], [Su15], [SM08], [XLL05], [Xu08], [LWL08], [B011b], [KS09], [Ki07a], [VA14].

Additionally, two of the approaches are only able to recognize certain concurrency patterns, for example the "looping-nested-fork-join" pattern [XLL05], [Xu08], [CLL07], as mentioned above. In case models include combinations of concurrent nodes, which do not match these patterns, the combinations will not be analyzed appropriately or the *UML Activity* will even be rejected.

Support for concurrency requires full coverage of the implications of concurrency. With relying on certain coverage criteria, an approach can only capture some of the interleavings possible in concurrent regions. In some cases, interleavings are even enforced, while other interleavings are neglected [SM08]; parallel executions are serialized to obtain a certain execution order. Non-determinism, respectively the problem of data races, is not appropriately covered using such an approach.

None of the articles discussed explicitly mentions the problems of cycle traversals, which is however, a critical and non-trivial problem with regard to nodes, which may run in parallel to other nodes or to themselves. Instead, cycles are often always traversed to a limited or hard-coded extent. The most flexible approach is published by Xu et al., and includes a hard limit, which can be set by users [Xu08]. As can be seen in chapter 4.6.5, such a limit is insufficient, as *UML Activities* can be created, for which any fixed limit (regardless of the hard-coded number) is not enough to obtain complete results. To fully consider the implications of such *UML Activities*, users would be required to manually analyze each activity to obtain the correct number for a limit.

Finally, although all approaches discussed are published after the publication date of the UML V2.0, only two of the publications discuss modeling constructs newly introduced with Version 2.0 [Xu08], [LTN19]. One of those articles also contains a discussion regarding possible relationships and hierarchies between *UML Activities [LTN19]*.

The following Table 1 summarizes the results of the assessment.

	Require-ment 1	Require-ment 2	Require-ment 3	Require-ment 4	Require-ment 5	Require-ment 6
Approach	Multiple UML Activities	Multiple Initial Nodes	Multiple Final Nodes	Cycle Traversals	Forks and Joins	UML Elements
Sun	-	-	-	-	-	-
Sapna	-	-	-	-	-	-
Xu	-	-	+	0	-	0
Chandler	-	+	+	-	-	-
Lei	-	-	+	???	+	-
Boghdady	-	-	-	-	-	-
Kundu	-	-	+	-	-	-
Kim	-	-	-	-	-	-
Verma	-	-	+	-	-	-
Lima	0	+	+	-	+	0

(-: not fulfilled, 0: partially fulfilled, +: fulfilled, ???: unclear)

Table 1: Assessment of Existing Approaches

As can be seen, many of the requirements are not completely fulfilled by existing approaches. The areas the least fulfilled are handling of multiple *UML*

Activities (requirement 1), Cycle Traversals (requirement 4), and support for a complete set of *UML Activity* Elements (requirement 6). The solution presented in this thesis is targeted at closing these gaps.

3.4 Classifying Data Races

As the practical problems defined in chapter 1.2 are strongly related to the effort spent in quality assurance for data races, it was initially intended that the solution presented in this thesis also contains an approach for the classification of data races. However, a review of the literature and existing approaches revealed that classifying data races into harmful and harmless data races cannot be conducted with the reliability required for a solution to be applicable in practice.

Based on the high number of reported data races (see chapter 1.2), several approaches to classify data races have been published. Researchers stated that 80% to over 90% of all reported data races are harmless and *benign* [Er10], [KZC12], [Na07], [Zh11], [YRC05]. It is assumed that such data races do not affect the correctness of an application, and may also be intentionally included in an application [KZC12].

Due to this, eliminating such types of data races from the amount of reported data races would lead to effort reduction, as no time would be spent on data races, which do not affect the correctness of an application. The main approaches for classification of data races into harmful or benign data races include heuristic classification, replay-based classification, and identification of ad-hoc synchronization.

Heuristic classification uses patterns to identify data races to be considered harmless. Such patterns may be, among others, updates of statistics counters, usage of special variables, which include intentional and harmless data races (such as for the current time, which is constantly updated), and concurrent read and write operations to different flag bits in the same memory location [Er10].

In replay-based classification, the execution of an application is replayed for a specific data race. In the replay, the order of accesses to the shared variable is switched with regard to the original execution. If both executions yield the same result, the data race is considered as being potentially benign [Na07].

Identifying ad-hoc synchronization is concerned with finding custom synchronization operations in an application. If such operations exist, and a data race is involved in that ad-hoc synchronization, the data race is considered as being benign [Zh11].

However, all these approaches suffer from drawbacks. Heuristic classification is dependent on the context of an application. If, e.g., such a statistics counter is critical to that application, then such a data race could not be classified as being benign. In general, heuristic classification can produce false positives and false negatives [KZC15].

Replay-based classification also can produce false negatives and false positives. If a replay of two concurrent operations produces the same result as the original execution, this may be due to the input parameters. A different input to the test case may produce different results. Different results in the original execution and the replay with switched accesses may also be intentional and correct, e.g., when printing the order of accesses. Another problem arises when a different ordering of two concurrent operations is not possible due to programming constructs, leading to a classification of the data race being harmful, which may or may not be correct.

Finally, the identification of ad-hoc synchronization may also produce false positives and false negatives. It may not be possible for such an approach to identify all code constructs leading to synchronization. In addition, if ad-hoc synchronization is detected, there is no analysis whether such a custom synchronization operation is correctly implemented.

The problems mentioned can lead to high misclassification rates, and Kasikci et al. report misclassification rates of 50% for ad-hoc synchronization, and of 74% for replay-based classification [KZC15].

Finally, it is disputed whether benign data races in an application, independent of how they are identified, are indeed harmless [Bo11a], [Ad10], [Bo12b]. For C and C++, the semantics for an application with data races are undefined, and the reason is that due to compiler optimizations, an application with data races may lead to unforeseen behavior [Ad10], [Bo12b]. For Java, only weak semantics are given for applications with data races [Ad10], [Bo12b]. It can be argued that the language specifications do not consider that memory accesses will happen concurrently, i.e., without proper synchronization with regard to a defined behavior of an application [Ad10], [Bo12b]. An example for unforeseen behavior of an application due to instruction reordering has been presented in chapter 1.1 (see interleaving 4).

Often, classification approaches make assumptions regarding memory models, which are not always true, i.e., assuming single processor environments and sequential consistency (see chapters 1.1 and 2.4) [KZC15]. However, in multiprocessor environments, updates to a memory location may not always be immediately visible to other threads on other processors. When assuming only a single processor, schedulers only execute one thread at a time. In multiprocessor environments, multiple threads may be executed concurrently on different processors. Together with delayed visibility of updates on different processors, it is not guaranteed that threads on different processors see updates to memory locations in the same order than they were originally processed [KZC15]. Thus, those classifications are conducted under wrong assumptions.

Finally, Adve and Boehm argue that although it is known that compilers can perform optimizations, an instruction reordering may not only appear due to such optimizations, but also due to store buffers, which enable performance optimizations [AB10]. Such an instruction reordering may have unforeseen consequences, if concurrent accesses are not synchronized. The authors state that "even with sequential consistency, such simultaneous accesses can remain dangerous" [AB10].

Based on such doubts regarding data races being benign, Boehm has stated that "non-determinism is unavoidable, but data races are pure evil" [Bo12b]. Automated approaches for classification of data races often include wrong assumptions and may lead to false positives and false negatives with a misclassification rate of up to 74%, as shown above. In addition, even if a data race is classified as being benign, it is unclear, whether this classification may only be correct for the observed case, and perhaps the specific input for execution, or if such a classification would be valid in all execution scenarios. Finally, unforeseen behavior due to instruction reordering, as discussed above, may lead to unknown consequences.

The consequence for this thesis is, that a classification of data races into harmful and harmless data races, being unreliable, has not been included as part of the solution.

3.5 Analysis of Data Races

Improvements in the efficiency of quality assurance related to data races thus cannot be made reliably by sorting out some of the findings, raising the question how efficiency improvements may be reached in analyzing the high amount of data race reports.

As explained in chapter 1.2, several techniques exist in this regard, such as reproducing detected data races and record and replay techniques. However, it is still necessary to analyze all data race reports manually to identify and resolve problems in the source code, and those techniques also offer no support in identifying duplicate data race reports. The only information given with regard to the analysis of reports is contained in those reports, and includes information such as stack traces, information regarding the class and line of concurrent accesses, the variable or memory location, and locks held by threads involved, as e.g., provided by approaches presented in [Ba06b], [CL10], [Bl18], [HMR14], [Li19]. However, it is up to users of those data race detectors to analyze the information given. Thus, the problem of too much effort needed for analyzing data race reports is not solved by such solutions.

Considering the problem regarding the effort necessary for analyzing data race reports and the reasons for this problem (see chapter 1.2), as well as the characteristics of data race reports, several requirements can be defined for an approach to enable efficiency improvements in the analysis of data race reports:

1) Duplicates: many of the data race reports by data race detection tools are duplicates. A data race consists of a pair of accesses to the same shared variable. Assuming two data races drA and drB, a duplicate may exist in that both data races contain exactly the same pair of accesses. However, a duplicate may also exist in terms of single accesses, when both data races are related to the same shared variable. Assuming that drA contains accesses drA_1 and drA_2, and drB contains accesses drB_1 and drB_2, duplicate single accesses exist, iff $\{drA_1, drA_2\} \cap \{drB_1, drB_2\} \neq \emptyset$. Thus, duplicate entries may not only exist on data race level, but also on access level. A solution to improve the efficiency of data race analysis shall consider such duplicate entries on both levels.

2) Focus on shared variables: any data race is related to one shared variable, and multiple different data races may exist related to the same shared variable. When multiple data races related to the same shared variable exist, eliminating one of the data races from the source code may not impact other data races. Data races may still exist, and may even exist in the same source code classes and lines that have just been rewritten for eliminating a data race regarding the same variable. Thus, to effectively eliminate data races related to the same shared variable, all those related data races need to be considered. Any approach for the analysis of data race reports shall thus enable an overview on all accesses to the same shared variable.

3) Focus on source code classes: any access to a shared variable is related to a specific source code class and line. There may be different source code classes involved in data races related to the same shared variable, and there may be multiple accesses to different shared variables related to data races located in the same source code class. For analyzing the source code, an approach shall thus be able to not only focus on shared variables, but also on source code classes. Such a focus can especially be valuable for risk-based approaches, with different risk levels assigned to components and source code classes of an application.

4) Status tracking: since data race detection often results in hundreds or more data race reports, it is useful to keep track of the status of data race reports and the entries included. Using a status, such as "closed" or "fixed",

it is possible to monitor progress in the analysis of data race detection, and also to concentrate on entries not yet analyzed. Any approach shall thus allow monitoring the current status of an entry regarding the analysis of data race reports.

5) Information preservation: approaches to improve the efficiency in the analysis of data race reports always include a processing of data race reports (with a minimum of reading the information provided to identify duplicate entries according to requirement 1, see above). However, as such approaches have to present the results of this processing, users have a different view on the results compared to just analyzing the unprocessed data race reports. This view shall nevertheless allow seeing all information contained in the original data race reports, so that no possibly important information is hidden from and inaccessible to users. Especially when identifying duplicates, those duplicates shall not be deleted, as the information is then lost. Thus, any approach shall preserve all information contained in data race reports and allow users to access this information.

Only a few publications cover the problem of how to use the information provided to efficiently analyze data race reports.

Pande has published an approach to visualize modified execution traces using AspectJ, additional instrumentation of the code, and refactoring [Pa19b]. However, the tool only visualizes one execution trace by highlighting source code entries, and does not provide further information regarding data race analysis. The purpose of the tool is to visualize thread-based events of an execution trace. However, when focusing on data race reports, such information is usually provided in clear text, and thus, highlighting these places in the source code would not need additional instrumentation, but a parser and static access to the source code (i.e., without the need to execute the source code again). In addition, the scalability of the approach is limited and further decreases not only with the size of an application, but also with the number of active threads. The author mentions that the usability of the visualization is limited if many interleavings happen. Finally, the instrumentation used adds additional thread switches to the execution, thus producing incorrect interleavings [Pa19b].

Trümper also presented an approach for visualization of execution traces for multithreaded applications [Tr14]. The approach focuses on visualizing sets of hierarchical event sequences and can be used for analyzing dependencies between threads. Although different views are provided, there is no focus or support for the analysis of data races or a possibility to integrate data race reports [Tr14].

Walker et al. focus on static analysis tools and mention the need to "show results in a concise, comprehensive way" [Wa20]. The authors present a dashboard providing an overview on the results of static analysis reports [Wa20]. The approach presented only covers results of static analyses, and is not related to concurrency issues, such as data races. Due to this, there is no specific assistance related to the characteristics of data races.

Koutsopoulos et al. identified the need for an efficiency improvement of "the manual investigation and classification process of the data race warnings through improved usability of the available information" [Ko15]. The authors focus on visualization and a reduction of information. Reduction is achieved by merging access locations, which have "the same full path, location (line and column), thread id, shared variable id, and action (read/write)" [Ko15]. The visualization of access pairs is thus lost, as "the typical representation of race pairs no longer exists" [Ko15]. Accesses are presented graphically on a file-based graph representation. Shared variables are shown in a row on top of the screen. All files with accesses to a shared variable are presented below and are connected to this shared variable with lines. Using such an approach, not only pairs of accesses, but all accesses to a shared variable can be observed. The authors acknowledge the complexity of such a view, and provide further functionality, including a filter function for code conditions, a zoom function to view details of accesses within a file, and an abstracted view of the call graph [Ko15].

Summarized, the need for efficiency improvements in the analysis of data race reports has been identified in research, but only a small amount of approaches have been developed. Only one approach focuses on data race reports [Ko15].[5] This approach tackles the problem of duplicate entries by merging access locations, which share characteristics (as described above); however, the information on duplicates is lost by such a merge, and cannot be recovered by users. The approach focuses on shared variables, and lists all of them at once, i.e., in the same view. The authors mention the complexity of this view, and considering large applications with thousands of data race reports, it remains unclear how this amount of information can be managed within the same view. The listing of different files enables an overview on all files, within which a shared variable is accessed, but the information, how many different accesses exist, is only visible by inspecting all information in those related file listings, as there may be multiple accesses within the same file. Additionally, by presenting all accesses within a file in a long list, and several connections to other files and to multiple shared variables at the same time, readability is impacted. It is not clear, which part of the list is related to other file list parts, i.e., other source code classes, or to which variable. As the approach relies on static analysis, it is not possible to provide lock ids for comparison or an execution trace. Finally, although the tool supports analysis, the article does not mention possibilities to track the progress of the analysis, i.e., to set a

[5] Due to this, the assessment of fulfillment of the requirements defined in this chapter is only given textually, and not within a table.

status for an entry, and to filter out data races, which have already been fixed [Ko15]. There is no quantitative evaluation of the approach. Although "a qualitative evaluation suggested a definite speedup in the investigation process of data race warnings" [Ko15], there is no additional information given on this evaluation besides that it has been applied to real-time systems with more than 100,000 lines of code. Thus, it is not possible to assess this evaluation and the efficiency improvements achieved by using this tool.

3.6 Summary

In this chapter, the focus of the thesis has been narrowed to *UML Activities* as basis for the analysis of representations of the dynamic behavior of an application. An overview on related work in this area has been presented. Although approaches for analyzing *UML Activities* have been identified, many of those approaches reveal gaps with regard to the identified requirements for a solution. The largest gaps in the related work are concerned with analyzing sets of connected *UML Activities* (requirement 1), handling *Cycle Traversals* within a *UML Activity* (requirement 4), and support for a complete set of *UML Activity* elements (requirement 6).

This chapter also presented an overview on the literature regarding the classification of data races into harmful and harmless data races. Based on the discussions in research, it has been shown why such classifications cannot provide reliable results. Thus, it has been decided that such a classification, although initially planned, will not be part of this thesis.

Finally, this chapter has provided an overview on related work regarding analyzing data race reports. In this area, little research has been conducted, and although improvements have been reached by the sole identified existing approach, which focuses on efficiency improvements in the analysis of data race reports, gaps still exist. The ideas underlying this approach serve as valuable input for the solution presented in this thesis.

Profound knowledge of the current state of the art and existing research gaps is the basis for scientific work. This chapter has thus provided the basis for the solution presented in the following chapters of this thesis.

Chapter 4 will present the solution for the analysis of *UML Activities*, which closes the identified research gaps. The result of the analysis, a set of nodes representing source code classes, can be used to steer the instrumentation for dynamic data race detection. Support for the analysis of data race reports is not yet sufficiently provided, and research gaps exist in this area. In chapter 5, this thesis presents a solution for the analysis of data race reports, tackling those identified research gaps. Chapter 6 is concerned with assessments and evaluations of the solution.

4 Analysis of UML Activities as Basis for Focusing Quality Assurance

In this chapter, the solution for the analysis of *UML Activities* is presented. This solution is targeted at the first practical problem (as defined in chapter 1.2).

The solution consists of three parts: in a first step, *UML Activities* are read in and the UML elements are transformed into a limited set of *basic elements*, with preserving the semantics of the elements as necessary for the goal of the analysis: identifying nodes that can be executed in parallel to other nodes.

This transformation on the one hand allows limiting the analysis to a reduced set of UML elements. On the other hand, such a transformation enables creating transformation approaches for other modeling languages besides UML, so that the further steps of this approach can be reused.

The transformed set of *UML Activities* is then analyzed using *Directed Acyclic Graphs*, in short *DAGs*, forming the second step.

As the analysis shall obtain complete results, the longest path possible is required, i.e., paths that may be shortened are not shortened. Such a shortening could lead to loosing information, e.g., in the form of missing nodes that may run in parallel to other nodes, thus possibly leading to false negatives.

The third step consists of a post-analysis handling of elements, for which further measures are necessary besides those of the transformation and the analysis phase.

Technically, the solution processes files created by `Enterprise Architect` from Sparx Systems Ltd. The transformation tool reads such files, processes the transformation, and outputs an XML file. This XML file is then read in for the analysis. Output of the analysis is a list of nodes. This list can then be used to limit the instrumentation for the dynamic analysis related to data races.

4.1 Research Approach

The first part of the research for creating a solution for the analysis of *UML Activities* consisted of an analysis of the structure of *UML Activities*. These activities are treated as graphs, but they do not have a formal foundation (see chapter 3.2). Furthermore, the tokens used in *UML Activities* are only able to be transported in one direction.

For the analysis of *UML Activities*, the intent is to employ a formally defined representation, thus allowing certain properties to be formally proven. As *UML Activities* already are specified as graphs according to the UML [OM17], but with semantics specific to the UML, the direct formal representation is a graph. Due to the token flow within *UML Activities*, directed graphs are selected. Finally, since *UML Activities* may contain cycles, and those have to be analyzed, the idea is to break up these cycles for the formal representation to be analyzed. Thus, directed acyclic graphs are used for the analysis of *UML Activities*.

Since such a directed acyclic graph only contains nodes and edges, and not a multitude of different node types and edge types, as *UML Activities*, the next step was to analyze the elements of *UML Activities* according to the UML. Since all those elements have different and specific semantics, a mapping of the semantics represented by the elements of UML Activities to graphs containing nodes and edges had to be created. As a direct mapping was not possible in all cases, the approach for the analysis consists of several phases: the transformation, the analysis, and the post-analysis phase.

Presenting the results of this research approach resulted in the following structure of the chapter. As the solution presented in this thesis is related to the field of graph theory, a short introduction into important concepts relevant for this solution is given in chapter 4.2.

This chapter then provides a concise description of the set of UML elements provided the UML superstructure (chapter 4.3). As this superstructure consists of more than 700 pages, of which more than 200 pages are related to *UML Activities*, only a short description can be provided. Those UML elements are split up into basic elements and additional elements, since these elements require different strategies for handling them during the analysis (chapter 4.4).

With the knowledge provided in the former chapters, the transformation part of the solution can be presented (chapter 4.5). Chapter 4.6 then presents the solution for the analysis of UML Activities for nodes that may run in parallel to other nodes. The analysis is accompanied with a post-analysis phase, targeted at elements, which cannot be completely handled during the analysis phase (chapter 4.7).

In chapter 4.8, the solution is then discussed and assessed with regard to the solutions presented in chapter 3.3.

4.2 Graph Theoretic Concepts

All graphs $G = (V, E)$ in this chapter are simple and directed with node set V and edge set $E \subseteq V \, x \, V$, except where explicitly noted. For a given graph G, the sets V and E may be referred to as V_G and E_G, for clarity. In edge $e = (u, v) \in E$, node u is its *tail* and node v is its *head*. In that case, e is an incoming edge of v, and an outgoing edge of u.

Since G is simple, it contains no self-loops: $\forall \, v \in V, \; (v, v) \notin E$, and no parallel edges: $|\{e \in E \,|e = (u, v) \wedge u, v \in V\}| \leq 1$.

Given a vertex (i.e., node) v in a simple and directed graph $G = (V, E)$, the neighborhood of v is the subgraph of G built by all incoming and outgoing edges of v and all nodes connected to these edges except for v itself (since G is simple). Any node in the neighborhood of v is called *adjacent to* v or *neighbor of* v. The *degree* of a vertex v is the number of neighbors of v. The *degree* of v in G can be split up into the *indegree* and the *outdegree*. The indegree of v is the sum of all nodes connected to v with an edge, of which v is its head. The outdegree is the sum of all nodes connected to v with an edge, of which v is its tail. Thus, the *indegree* of v in G $d_G^-(v) := |\{u \in V \,|(u, v) \in E\}|$ and the *outdegree* of v in $d_G^+(v) := |\{u \in V \,|(v, u) \in E\}|$.

A *walk* $w = v_1 v_2 \dots v_i$ in a graph G is a sequence of nodes such that $v_j \in V$ for $j = 1, \dots, i$ and $(v_k, v_{k+1}) \in E$ for $k = 1, \dots, i - 1$ with \$$\alpha(w) := v_1$ and $\omega(w) := v_i$. A walk with $v_1 = v_i$ is called a *cycle*. A directed graph without any cycle is called a *Directed Acyclic Graph* or *DAG*, for short.

Given the walks $w_1 = v_1 v_2 \dots v_i$ and $w_2 = u_1 u_2 \dots u_j$, let $cat(w_1, w_2)$ denote a new walk $w' = v_1 v_2 \dots v_i u_2 \dots u_j$ whenever $\omega(w_1) = \alpha(w_2)$, otherwise $cat(w_1, w_2)$ is undefined. The walk w' is called the concatenation of w_1 and w_2.

Given the walks w_1 and w_2, w_2 is a *direct successor* of w_1, iff $\omega(w_1) = \alpha(w_2)$. The walk w_1 is then called a *direct predecessor* of w_2.

Given the walks $w_1, w_2, \dots, w_n$ with w_{i+1} being a direct successor of w_i $\forall \, i \leq n - 1$, then w_j is called an *indirect successor* of w_i, if $i + 1 < j \leq n$. In such a case, w_i is called *indirect predecessor* of w_j.

The following Figure 8 shows a directed graph H illustrating these concepts. Nodes are depicted as dots and identified with letters, and edges are depicted with solid and with dashed arrows and are identified with numbers.

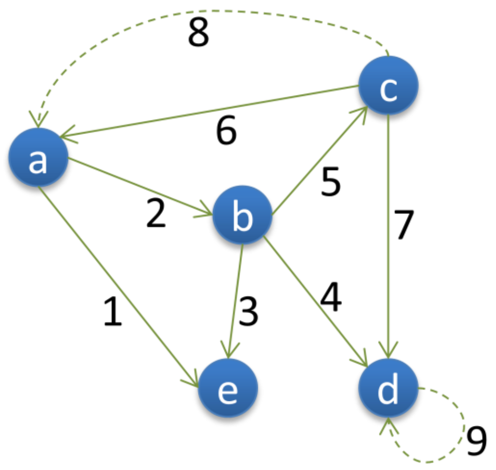


Figure 8: Directed Graph

As can be seen, this graph is cyclic (e.g., walk $w_1 = abca$) and is thus not a DAG. This graph is also not simple, as it contains parallel edges (edges 6 and 8), and a loop (edge 9), and is thus not adhering to the definition of graph G above. Eliminating the red edges 8 and 9 results in a simple graph. Eliminating the edges 6, 8, and 9 results in a DAG.

The head of edge 1 is e, and the tail of edge 1 is a. The neighborhood of b consists of the nodes a, c, d, e and the edges $2, 3, 4, 5$. The degree of b is thus 4, which can be split up into the indegree $d_H^-(b) = 1$ and the outdegree $d_H^+(b) = 3$. The walks w_1 and $w_2 = ae$ can be concatenated and $cat(w_1, w_2) = abcae$. w_1 is the direct predecessor of w_2, and w_2 is the direct successor of w_1. Given the walks $w_3 = ab$, $w_4 = bc$ and $w_5 = cd$, w_5 is an indirect successor of w_3, and w_3 is an indirect predecessor of w_5. As $cat(cat(cat(w_1, w_3), w_4), w_5)$ is defined, w_5 is also an indirect successor of w_1.

4.3 UML Elements

In this chapter, *UML Activities* and elements within activities are discussed. The semantic implications of different elements and combinations of elements are analyzed. This chapter provides the knowledge required to understand the decisions made for the solution to analyze *UML Activities*. This chapter does not intend to provide a complete overview on *UML Activities*. Such an overview can be found in the UML superstructure [OM17].

According to the UML superstructure, “a Behavior is a specification of events that may occur dynamically over time” [OM17]. Thus, such a Behavior can be used to describe what happens during the runtime of a system. A *UML Activity* is a “kind of Behavior [...] that is specified as a graph of nodes interconnected by edges” [OM17]. The graphical description of a *UML Activity* is called UML Activity Diagram.

A *UML Activity* may have input and output parameters, so-called *ActivityParameterNodes*. An example for such an input parameter is a postal address for a *UML Activity* specifying a delivery.

As already described in chapter 3.2, *UML Activities* use a token concept: nodes offer tokens to edges. Edges can transport tokens to nodes connected to their head. Whether a token is transported or not depends on two requirements: first, an edge may have a guard condition. If this guard condition evaluates to false, then the token is rejected. Second, the token type must match the type of the edge: both object and control tokens exist, and both object edges (*ObjectFlow*) and control edges (*ControlFlow*) exist.

This token concept not only defines how an activity is executed, but also how the execution can be stopped. The execution of an activity ends as soon as an *ActivityFinalNode* is reached by a token, or when no more tokens can be transported by edges and no nodes are currently executing.

UML Activities may be modeled implicitly and explicitly. In the first case, certain nodes or combinations of nodes and edges are not modeled, but implied. In the latter case, all nodes and edges are modeled. The interpretation of implicit modeling constructs is given in the UML superstructure [OM17].

UML Activities may contain the following types of nodes:

1) *ControlNodes*: these nodes steer flows between nodes within an activity. Seven types of *ControlNodes* exist [OM17]:

    a) *InitialNodes*: these nodes are automatically executed as soon as the execution of a surrounding activity is started. *InitialNodes* have an indegree of 0 and an outdegree > 0, and place a control token to outgoing edges. If an *InitialNode* has multiple outgoing edges, a token is placed on all these outgoing edges, thus starting a parallel execution. This implies a single edge from an *InitialNode* to a *ForkNode* (explained below), which then has multiple outgoing edges. An activity can have multiple *InitialNodes*, which are executed simultaneously. This implies a single *InitialNode*, which is connected by a single edge to a *ForkNode*, which is then connected to these former *InitialNodes*. If an activity has no *InitialNode*, all nodes in that activity, which have an indegree of 0 start executing. This again implies a single *InitialNode*, which is connected by a single edge to a

ForkNode, which is connected to those nodes implicitly modeled with an indegree of 0.

b) *FlowFinalNodes*: a *FlowFinalNode* has an indegree of 1 (assuming explicit modeling) and an outdegree of 0. Any token reaching such a node is destroyed. This assumes explicit modeling. Implicitly modeled, an *FlowFinalNode* may have an indegree > 1, which is interpreted as having a *MergeNode* (see below) connected to those incoming edges, which itself is connected by a single edge to that *FlowFinalNode*.

c) *ActivityFinalNodes*: an *ActivityFinalNode* has an indegree of 1 (assuming explicit modeling) and an outdegree of 0. If a token reaches an *ActivityFinalNode*, all tokens within the surrounding activity are destroyed, thus ending the execution of this activity. Furthermore, this also impacts other activities synchronously invoked from within the activity. If an *ActivityFinalNode* is modeled with an indegree > 1, again a *MergeNode* is implied, similar to *FlowFinalNodes*.

d) *DecisionNodes*: a *DecisionNode* has one incoming *ControlFlow* and may have an additional *ObjectFlow* representing the input to that decision. Such a node has at least two outgoing *ControlFlows*, which represent the flows followed based upon the decision made. Each of these outgoing *ControlFlows* may have a guard condition. If the condition is satisfied, the edge may accept the token offered. If conditions of multiple outgoing edges are satisfied, the token is offered to only one *ControlFlow*, and the choice of this *ControlFlow* is made non-deterministic according to the UML [OM17]. Thus, a *DecisionNode* has an indegree of 1 or 2, and an outdegree > 1.

e) *MergeNodes*: a *MergeNode* unites several incoming flows into one flow without synchronization. It has multiple incoming edges and one outgoing edge. The edge may either be all *ObjectFlows* or all *ControlFlows*. Thus, a *MergeNode* has an indegree > 1 and an outdegree of 1.

f) *ForkNodes*: a *ForkNode* is used to start parallel flows and has one incoming edge. The token received through the incoming edge is copied and one of these tokens is offered to each of the outgoing edges in parallel. Since an edge may have a guard condition attached, it depends on whether such a condition is attached, and whether this guard condition is evaluated to true, if the token is accepted by the edge. Assuming that at least two outgoing edges are present, a *ForkNode* has an indegree of 1 and an outdegree of > 1. Having more than one outgoing edge is not required by the UML for

ForkNodes, however, as the intent of using a *ForkNode* is to start parallel flows, it can be argued that such a construction is a mistake.

g) *JoinNodes*: a *JoinNode* is used to synchronize parallel flows and continue with one flow. A *JoinNode* has a *ValueSpecification*, which defines the conditions for the node to offer a token to the outgoing edge. If no explicit specification is provided, then tokens need to be offered on all incoming edges, to be consumed by the node. Using a *ValueSpecification*, it may also be sufficient to offer tokens on some of the incoming edges. The evaluation of this *ValueSpecification* (to true of false) is started whenever a new token is offered on an incoming edge, and cannot be interrupted by newly arriving tokens. It is also not possible to start multiple evaluations in parallel. However, the format of such a *ValueSpecification* is not exactly defined in the UML [OM17], and a *ValueSpecification* may also be given using textual expressions. Thus, it cannot be guaranteed that the *ValueSpecification* can be processed by an algorithm, and thus it is not considered during the analysis. Assuming explicit modeling, a *JoinNode* has an indegree > 1 and an outdegree of 1.

Both *JoinNodes* and *ForkNodes* can implicitly be modeled with an indegree and an outdegree > 1. In these cases, a combination of a *JoinNode* (with an indegree > 1) connected by a single edge to a *ForkNode* (with an outdegree > 1) is implied.

2) *ObjectNodes*: such nodes hold object tokens, representing objects, i.e., some kind of data. An *ObjectNode* may hold several object tokens, independent of the value represented by the individual token. All *ObjectNodes* except *ActionPins* (see below) have an indegree and an outdegree of 1, assuming explicit modeling. Four different types of *ObjectNodes* are defined in the UML [OM17]:

a) *ActivityParameterNodes*: These nodes are attached to an activity, and represent input and output parameters for objects, thus forming sources and sinks of objects used in an activity.

b) *CentralBufferNodes*: such nodes act as buffers between incoming and outgoing *ObjectFlows*. All object tokens offered to such nodes are immediately accepted and held until they can be passed to an outgoing *ObjectFlow*.

c) *DataStoreNodes*: these nodes are similar to *CentralBufferNodes*, but in addition act as a store. All object tokens passed to outgoing *ObjectFlows* are copied, and this copy is stored in the *DataStoreNode* as long as the surrounding activity is executing.

d) *ActionPins*: an *ActionPin* holds object tokens and represents an input to or an output from an action. As such, an *ActionPin* is associated with a node, and therefore either has an indegree of 0 and an outdegree of 1, or vice versa, assuming explicit modeling.

3) *ExecutableNodes*: these are nodes that represent an execution as a step within the overall desired behavior specified by the surrounding activity [OM17]. *ControlNodes* steer the flow of execution, i.e., they are used to influence the sequences of *ExecutableNodes* computed (i.e., the control flow). *ObjectFlows* are used to manage the data flow between these *ExecutableNodes* [OM17].

   All *ExecutableNodes* are *ActionNodes*. All incoming and outgoing edges are *ControlFlows*. For processing data, *ActionPins* are used (as described above). An *ExecutableNode* has an implicit *JoinNode* or *ForkNode* attached in case of multiple incoming or outgoing edges, thus with explicit modeling, such a node has an indegree and an outdegree of 1.

   When an *ExecutableNode* has finished its execution, a control token is offered to the outgoing edge.

   An *ExecutableNode* may raise an exception (*RaiseExceptionNode*, see below), and if this exception is not handled during the execution of this node, this exception is propagated to the outside of the *ExecutableNode*. Any *ExecutableNode* may be associated with one or more *ExceptionHandlers*, i.e., *ExecutableNodes*, which match certain types of exceptions. If multiple *ExceptionHandlers* match an exception, it is not defined in the UML, which *ExceptionHandler* will be executed [OM17].

   The UML defines 42 different *ActionNodes*, which can be categorized into ten categories [OM17]:

   1) *InvocationActions*: actions related to the invocation of behaviors or operations in behaviors, sending signals or objects.

   2) *ObjectActions*: actions related to operations on objects.

   3) *LinkActions*: actions related to operations on links, i.e., associations and their instances.

   4) *LinkObjectActions*: actions related to operations on link objects, i.e., instances of *AssociationClasses* (associations with class properties).

   5) *StructuralFeatureActions*: abstract class for all actions concerning structural features (e.g., attributes).

6) *VariableActions*: actions related to operations on variables.

7) *AcceptEventActions*: actions waiting for certain events to be triggered before executing.

8) *StructuredActions*: actions containing nodes and edges themselves to realize more complex behaviors (in comparison to single actions).

Some of the action types are not clustered into categories by the UML itself, and have been put into categories as shown below for the purpose of further processing:

9) *None:* actions, which are not specified by the UML (functionality not defined by the UML), so-called *OpaqueActions*.

10) *Other actions: ReduceActions* that reduce a collection of values to a single value, and *RaiseExceptionActions*, which throw an exception.

The *ActionNode* category *StructuredActions* bears a special complexity and is thus further examined in this chapter. Details for all other *ActionNode* types can be found in the UML superstructure [OM17].

StructuredActions can contain nodes and edges themselves, and may thus serve as containers for other nodes. The following *StructuredActions* exist:

StructuredActivityNodes: this type acts as container in the form of a sub-activity, without further semantics. It is also called *simple.*

ConditionalNodes: such a node type consists of at least one *clause*, which represents a branch of the conditional executions. Each clause contains *body* and *test* sections, which contain disjoint subsets of the *ExecutableNodes* contained in the *ConditionalNode*. When a *ConditionalNode* is executed, all *test* sections are executed, and if the evaluate to true, the respective *body* section is executed. *Test* sections (and related *body* sections) may also be executed in parallel to each other.

LoopNodes: these node types represent iterations in the computation. All *ExecutableNodes* within a *LoopNode* are either part of the *setup*, the *test*, or the *body*. The setup contains the initialization of the *LoopNode*, the test decides whether the body is executed, and the body contains *ExecutableNodes* to be executed during the iterations. Different types of loops can be created using *LoopNodes*, e.g., the body part may be executed before the test part, or after the test part.

SequenceNodes: a *SequenceNode* contains nodes, which are executed in sequence, i.e., one after the other. It defines a total order of *ExecutableNodes* within itself.

ExpansionRegions: such types process collections of values or objects and contain *ExecutableNodes* and edges. For each value or object in the collection, the *ExpansionRegion* is executed once. Collections are defined by the execution engine executing the activity, and could be sets, bags or other collection types. If results are produced, these can be stored in output collections. There may be different numbers of input and output collections. The flow across boundaries of an *ExpansionRegion* is specified by *ExpansionNodes* (which are *ObjectNodes*). The nodes and edges contained in an ExpansionRegion are executed once per element of the input collections. The *mode*, i.e., how the execution is conducted, is defined by a parameter. Tokens offered to *InputPins* of an *ExpansionRegion* and tokens offered by edges crossing the boundaries of an *ExpansionRegion* (in this case from outside to inside the region), are copied for each execution of an *ExpansionRegion*, so that an execution does not influence other executions of that same region. For edges crossing a boundary of an *ExpansionRegion* from inside to outside the region, the semantics are not defined [OM17]. If an *ExpansionRegion* contains an *ActivityFinalNode*, then all executions of this *ExpansionRegion* are stopped, and the results are offered at the output *ExpansionNodes*, i.e., the activity containing such and *ExpansionRegion* continues to execute. The mode of execution may be parallel, in which case all executions are processed concurrently, iterative, in which case one execution has to finish before the next execution can start, or stream. In stream mode, there is only one execution, but multiple tokens are offered on the outgoing edges [OM17].

Finally, activities may contain *ActivityGroups*, in the form of *ActivityPartitions* and *InterruptibleActivityRegions*. *StructuredActivityNodes* belong to both *ActivityGroups* and Actions [OM17], and have thus been included in the action types above. Nodes and edges may belong to multiple of such *ActivityGroups*, and an *ActivityGroup* may itself contain multiple nodes and edges.

ActivityPartitions "do not affect the token flow of the model" [OM17] and are used to allocate characteristics or resources among the nodes of an activity.

InterruptibleActivityRegions form a group of nodes, which may be interrupted without interrupting the whole surrounding activity. Such regions contain *InterruptingEdges*, which lead from within to outside of the region. Not all such edges must be *InterruptingEdges*. When a token flows through such an *InterruptingEdge*, all nodes within the region stop executing and all tokens within the region are deleted, except tokens traversing an edge leading from within to outside of the region. This means that although an execution may be interrupted, it depends on the timing of the interruption whether another token is transported to nodes outside of the region as a result of the interrupted execution [OM17]. An *AcceptEventAction* inside such a region with an indegree of $\mathbf{0}$ is only reacting to events if a token exists within this region [OM17].

4.4 Handling of UML Elements in the Analysis

UML Activities can be modeled using implicit and explicit notation. As Schattkowsky and Förster show, implicit modeling, which consists, e.g., of the combination of certain nodes into one node or omitting nodes (which are then implied during the interpretation, as shown above), can lead to ambiguities and wrong interpretations of *UML Activities* [SF07]. It is thus assumed for the analysis that explicit modeling has been used. Implicit notation constructs, if found, are transformed into explicit modeling constructs during the transformation from the source file.

A *UML Activity* $\boldsymbol{D}$ is a graph and is created to capture the essential steps in computation (assuming a *UML Activity* is used in the context of representing dynamic behavior of software). A *UML Activity* may contain several types of nodes and edges. The solution presented in this thesis includes a preprocessing for *UML Activities* to reduce the types of nodes to consider in the analysis, i.e., certain types of nodes are transformed to constructs of other types of nodes (see chapter 4.5). For the analysis, the elements in *UML Activities*, discussed above, are split up into *basic elements* and *additional elements*. *Basic elements* are directly considered in the analysis, i.e., they are not changed during the preprocessing.

Additional elements can either be transformed to a combination of *basic elements*, which form an equivalent in terms of the goal of the solution, or need to be considered after the analysis, e.g., to filter out false positives.

In this thesis, a concrete occurrence of a node in a *UML Activity* during traversal, i.e., execution, is called an *instance* of a node. If a *UML Activity* is executed, and a certain node is reached and traversed, this is referred to as an *instance*. Since nodes can be traversed multiple times, given that cycles are possible in $\boldsymbol{D}$, each traversed node is considered as a separate instance of this node in $\boldsymbol{D}$. There may be several instances of the same node. An edge $\boldsymbol{e}$ of a *UML Activity* $\boldsymbol{D}$ is also referred to as $\boldsymbol{e_D}$ whenever it is suitable for clarity reasons.

4.4.1 Basic Elements

As mentioned, only basic elements need to be considered in the analysis, as for all other elements of *UML Activities*, a dedicated handling is used. These basic elements are as follows:

1) *InitialNodes*.

2) *ActionNodes*. There exist various types of *ActionNodes*, which have to be handled differently. Thus, *ActionNodes* form an exception in that they are also included in chapter 4.4.2.

3) *DecisionNodes*. For simplicity, it is assumed that a decision can directly lead to one (but only one) of multiple cases (represented by the outdegree of this *DecisionNode* in $\boldsymbol{D}$) without nesting multiple binary if-then-decisions into each other; it is also assumed that there are at least two different cases.

4) *MergeNodes*. *MergeNodes* can appear independent of case distinctions, i.e., there is no rule in the UML, which restricts their usage to the context of case dependent decisions (by former *DecisionNodes*).

5) *Final nodes*. For the analysis, only *FlowFinalNodes* are considered, as *ActivityFinalNodes* are transformed (see chapter 4.4.2).

A *UML Activity* containing these five types of nodes represents all possible executions of a single computation thread, which computes a single result on a single processor by a single sequence of computation events (computations and decisions). A finishing execution results in a finite sequence, a non-finishing execution results in an infinite sequence. To represent a single execution instance by a *DAG*, all cycles are *rolled out* by numbering multiple instances of the same node in the sequence consecutively. Any such single execution instance starts with an *InitialNode*, followed by a sequence of action and control nodes that ends with a *FinalNode* if and only if the execution instance finishes its computation.

To represent parallel computations in a *UML Activity*, two more types of nodes are required:

6) *ForkNodes.* A *ForkNode* represents a single incoming computation thread and $\boldsymbol{d_D^+}$ outgoing computation threads that run in parallel. It is always assumed that a *ForkNode* is followed by at least two different threads.

7) *JoinNodes*. A *JoinNode* represents one outgoing and $\boldsymbol{d_D^-}$ incoming computation threads. Semantically, it represents a situation where threads are synchronized.

Parallel computations may also emerge directly at the start of the computation of a *UML Activity*, as explained above. In such a case, computation starts in parallel at each of the *InitialNodes* in that *UML Activity*. During the preprocessing, such nodes are substituted with a single initial node, which is directly connected to a *ForkNode*, after which the parallel computation starts.

An example of a fragment of a UML Activity Diagram can be seen in Figure 9. This diagram consists of five different *ActionNodes* and six *ControlNodes*, i.e., one *InitialNode* and one *FinalNode*, one *ForkNode* and one *JoinNode*, one *DecisionNode* and one *MergeNode*.

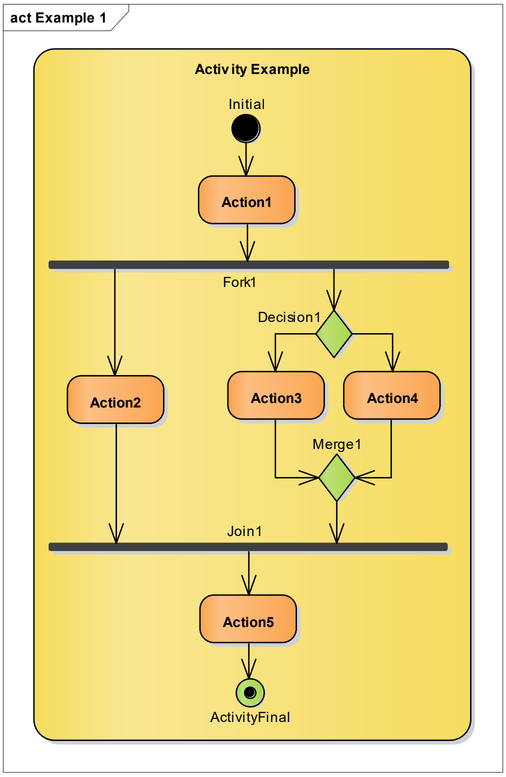


Figure 9: Exemplary UML Activity Diagram 1

The *UML Activity Diagram* shows parallel computations, which are started at the node "Fork1" and which end at node "Join1". Due to the *DecisionNode* "Decision1", two different computations and thus, two different execution instances are possible, one leading through node "Action3", the other leading through "Action4". Both instances lead to "Merge1", after which there is no further difference in the computations.

4.4.2 Additional Elements

Besides the nodes mentioned in chapter 4.4.1, further elements exist, for which a special handling was necessary during the transformation and/or after the analysis.

The biggest group, which had to be handled separately, was the group of action types (i.e., the different *ActionNodes*). The following Table 2 shows the different action types and their according categories. The categories of action types are independent of those groups. Note that during the transformation, it is not necessary for the purpose of the solution presented in this thesis to retain the exact structure of an activity, but to obtain all possible executions, with the statements made in chapter 4 in mind.

The action types are divided into four groups and different strategies are applied for handling those action type groups:

- *Group Reduction*: all action types, for which no special handling was necessary, are transformed into an *ActionNode* (see chapter 4.4.1); this transformation is called a *reduction*. All node types belonging to this group are marked with a "Y (a)" in Table 2, in the transformation column.

- *Group Specific*: All action types, which are marked with a "Y" in the transformation column and no marking in the post-analysis column of Table 2, are treated with a specific handling during the transformation. This may result in those elements being transformed to other elements, which belong to other groups. An example is the *BroadcastSignalAction*, which is transformed to a *SendSignalAction*, which itself is handled in the post-analysis phase.

- *Group Post-Analysis*: the action types marked with a "-" in the transformation column of the table are treated according to the semantics of *ActionNodes*. This means they are treated as *ActionNodes*, but their special type and semantics are preserved. They are handled after the analysis, to reflect their specialized meanings and semantics (marked with a "Y" in the post-analysis column in Table 2).

- *Group Structured*: special cases are the *StructuredActivityNode* and the *ExpansionRegion*, which needed a special handling both during the transformation and after the analysis (marked with a "Y" in the transformation and in the post-analysis column).

The specific strategies and rationales are explained in chapter 4.5 for the transformation and in chapter 4.7 for post-analysis handling.

Category	Name	Trans-formation	Post-Analysis
(none)	OpaqueAction	Y(a)	-
InvocationActions	StartObjectBehaviorAction	Y(a)	-
	CallBehaviorAction	-	Y
	CallOperationAction	Y(a)	-
	SendObjectAction	Y(a)	-
	SendSignalAction	-	Y
	BroadcastSignalAction	Y	-
ObjectActions	CreateObjectAction	Y(a)	-
	DestroyObjectAction	Y(a)	-
	TestIdentityAction	Y(a)	-
	ReadSelfAction	Y(a)	-
	ValueSpecificationAction	Y(a)	-
	ReadExtentAction	Y(a)	-
	ReclassifyObjectAction	Y(a)	-
	ReadIsClassifiedObjectAction	Y(a)	-
	StartClassifierBehaviorAction	Y(a)	-
LinkActions	ReadLinkAction	Y(a)	-
	CreateLinkAction	Y(a)	-
	DestroyLinkAction	Y(a)	-
	ClearAssociationAction	Y(a)	-
LinkObjectActions	ReadLinkObjectEndAction	Y(a)	-
	ReadLinkObjectEndQualifierAction	Y(a)	-
	CreateLinkObjectAction	Y(a)	-
StructuralFeatureActions	ReadStructuralFeatureAction	Y(a)	-
	AddStructuralFeatureValueAction	Y(a)	-
	RemoveStructuralFeatureValueAction	Y(a)	-
	ClearStructuralFeatureAction	Y(a)	-
VariableActions	ReadVariableAction	Y(a)	-
	AddVariableValueAction	Y(a)	-
	RemoveVariableValueAction	Y(a)	-
	ClearVariableAction	Y(a)	-
AcceptEventActions	AcceptEventAction	-	Y
	AcceptCallAction	Y(a)	-
	ReplyAction	Y(a)	-
	UnmarshallAction	Y(a)	-

Category	Name	Trans-formation	Post-Analysis
StructuredActions	StructuredActivityNode	Y	Y
	ConditionalNode	Y	-
	LoopNode	Y	-
	SequenceNode	Y	-
	ExpansionRegion	Y	Y
Other actions	ReduceAction	Y(a)	-
	RaiseExceptionAction	Y(a)	-

Table 2: Handling of Action Types

Besides the action types mentioned above, further elements exist, which need to be considered individually. The following Table 3 contains all other elements, which need to be considered. Note that elements in the same groups in this table are not necessarily in the same category according to the UML. In some cases, they are put together because certain relations between them are considered, e.g., between *ObjectNodes* and *ObjectFlows*.

The element "*ActionNode*" in the group *ExecutableNodes* is in parentheses, because all *ActionNodes* are handled in Table 2.

Group	Name	Trans-formation	Post-Analysis
FinalNodes	ActivityFinalNodes	Y	-
ObjectElements	ActivityParameterNodes	Y	-
	ActionPins	Y	-
	CentralBufferNodes	Y	-
	DataStoreNodes	Y	-
	ObjectFlows	Y	-
ExecutableNodes	"ActionNodes"	see Table 2	see Table 2
	ExceptionHandler	Y	-
	InputPin	Y	-
	OutputPin	Y	-
ActivityGroups	ActivityPartitions	Y	-
	InterruptibleActivityRegions	Y	Y
	InterruptingEdges	Y	Y

Table 3: Handling of Further Elements

All elements marked with a "Y" in the transformation column in this Table are handled during the transformation. Details on the individual handling and rationales for each handling can be found in chapter 4.5.

All elements marked with a "Y" in the post-analysis column in this Table are handled after the analysis. Details on the individual post-analysis handling can be found in chapter 4.7.

As a short summary in this section, elements marked with a "Y" in the transformation column, but no "Y" in the post-analysis column are either transformed to other elements or *basic elements*, or deleted (other transformations may be necessary for deletion). All other elements are handled according to the semantics of *ActionNodes*, but their special semantics are retained for post-analysis consideration.

The purpose of the analysis is identifying nodes that may run in parallel to other nodes, but with the goal to steer instrumentation for detecting data races. Due to this and since parallelism itself is not a problem, the computation of pairs of nodes that can possibly run in parallel is limited to nodes that actually may be related to source code parts accessing data.

By definition *InitialNodes*, *FinalNodes*, *MergeNodes*, *ForkNodes*, and *JoinNodes* are not related to data accesses. Thus, the focus of the analysis can be set to *DecisionNodes* and *ActionNodes*. In the following, these two types of nodes are called *data accessing nodes*, short *DANs*. Note that the UML actually defines many other node types, which are related to possible data accesses, but that the limitation to these two node types is possible due to the preprocessing, which transforms those other node types.

The goal of the analysis of *UML Activities* can thus be reformulated to identifying all DANs that may be executed in parallel to other DANs or to themselves. For this, an intermediary representation of a *superset containing all possible execution instances* as *DAGs*, based on *UML Activities*, is used.

4.5 Transformation of UML Activities

The transformation of *UML Activities* serves to limit UML elements to be considered for the analysis to *basic elements*. The transformation of UML elements, together with a formal reasoning for the decisions regarding the transformation are extensively documented by Zimmer in [Zi16], a master thesis conducted in the context of this thesis. In this chapter 4.5, only a short summary is given.

This transformation is conducted using eight steps:

1) Reducing *ActivityFinalNodes*

2) Reducing *ExpansionRegions*

3) Reducing non-structured actions

4) Reducing *ObjectNodes* and *ObjectFlows*

5) Reducing *ConditionalNodes*, *LoopNodes* and *SequenceNodes*

6) Reducing *ExceptionHandlers*

7) Remodeling regarding *InitialNodes* and *FlowFinalNodes* (implicit to explicit)

8) Remodeling regarding *ForkNodes* and *JoinNodes* (implicit to explicit)

First, *ActivityFinalNodes* are reduced to *FlowFinalNodes*, *ExpansionRegions* to simple *StructuredActivityNodes* and most non-structured actions to generic actions. *ActionNodes* related to signals and *CallBehaviorActions* require special considerations, since they might have an impact on the execution order. In the next step, *ObjectNodes* and *ObjectFlows* are reduced. All pairs of *ObjectNodes* ordered by *ObjectFlows* are then expressed via *ControlFlows* between *ExecutableNodes.*

The transformation of *SequenceNodes*, *LoopNodes* and *ConditionalNodes* follows the idea of expressing the implied *ControlFlow* ordering of these nodes explicitly by means of new *DecisionNodes* and *MergeNodes*. Next, *ExceptionHandlers* are reduced.

Finally, additional actions are performed to make implicit constructs in *UML Activities* explicit. This involves two steps: first, the *InitialNodes* and *FlowFinalNodes* of each container are combined, and implicitly modeled constructs are explicitly modeled. Second, implicit *ForkNodes* and *JoinNodes* are explicitly modeled by the introduction of new *ForkNodes* and *JoinNodes*.

As the transformation of elements may require a former transformation of other elements, the sequence of the steps of the transformation is critical to the success of the transformation. Elements related to signals and *CallBehaviorActions* may impact the order of the execution of *ExecutableNodes* in *UML Activities*. The transformation of *ObjectNodes* and *ObjectFlows* may have an impact on all other transformations following this step. The transformation of *ExceptionHandlers* may require the introduction of new *CallBehaviorActions*.

The elements requiring a dedicated handling during the transformation, presented in chapter 4.4.2, are analyzed in the following. It is shown how these elements are handled and what assumptions, if any, have been made.

4.5.1 Group Reduction

All elements in the "Group Reduction" are transformed to "plain *ActionNodes*" (reduced), i.e., the information that data may be read or written is retained.

ObjectActions, *LinkActions*, *LinkObjectActions*, *StructuralFeatureActions*, *VariableActions*, and *ReduceActions* deal with data manipulation and are thus contained in this group.

During the execution of *RaiseExceptionActions*, an exception is thrown. If the current container has an *ExceptionHandler*, it is executed. Otherwise the exception propagates to the outside. If no *ExceptionHandler* matches, the exception terminates the current activity [OM17]. As will be explained in chapter 4.5.7 concerning *ExceptionHandlers*, it is not relevant, where an exception is exactly thrown, but where and how this exception is handled. Thus, *RaiseExceptionActions* can be reduced. However, the exception raised still needs to be taken into account, i.e., the respective *ExceptionHandlers* are considered during transformation and re-arrangement of the *UML Activity*.

OpaqueActions are actions whose impact is described in a textual modeling language. Since the functionality of such actions is not defined by the UML and thus cannot be evaluated, they are reduced.

As the focus of the solution is set to *UML Activities*, the analysis does not support elements, which refer to other UML elements outside of activities, such as *UML Class Diagrams*. Because of this, the following elements of activities are also reduced: *StartObjectBehaviorActions*, *CallOperationActions*, *SendObjectActions*, *AcceptCallActions*, *ReplyActions*, and *UnmarshallActions*.

For *SendObjectActions*, the UML mentions that objects sent may also be signals. In such a case, it is assumed that the respective element is modeled using a *SendSignalAction*.

4.5.2 Group Structured

StructuredActivityNodes are handled both during the transformation and post-analysis. These elements are containers for nodes and edges. Edges are allowed, which cross the borders of these containers, and which directly connect an element outside of the container with an element inside the container (*CrossingEdges*).

CrossingEdges require a special handling after the analysis due to the special semantics for such edges and the target nodes (see [OM17]).

Thus, *StructuredActivityNodes* are left intact during the transformation. In the output of the transformation, there is an element "*StructuredActivityNode*", which itself contains other elements.

Both the *StructuredActivityNodes* and the contained elements are subject to the analysis: there can be (1) parallel executions of elements outside a *StructuredActivityNode* and the structured activity itself (and thus, the elements inside, with consideration of *CrossingEdges*), and (2) parallel executions of elements inside a *StructuredActivityNode*.

ExpansionRegions also are containers for other elements, and also may contain *CrossingEdges*. With *ExpansionRegions*, collections of values or objects can be processed in three different processing modes: iterative, streaming and parallel (see chapter 4.3). The collections of elements are realized by *ExpansionNodes*. *ExpansionRegions* can have *InputPins*, whose tokens are copied for each execution of an *ExpansionRegion*, and *OutputPins*.

The UML states that the "semantics is undefined for offering tokens to such *OutputPins*" [OM17]and for "*ActivityEdges* from within the expansion executions" [OM17] to outside of the ExpansionRegion.

ExpansionRegions in iterative mode are transformed to *StructuredActivityNodes*, and these are handled as described above. *ExpansionNodes* are transformed into pins.

In the iterative mode, values are processed one by one, i.e., each value is processed in an execution of the *ExpansionRegion*, and such executions can never run in parallel in iterative mode. Thus, no data races between executions of an *ExpansionRegion* can exist.

For the streaming and the parallel mode, executions may be parallel to each other [OM17]. If a collection of elements to process contains references to the same object, this can lead to a data race. However, whether a data race is actually possible, depends on the modeling, and on whether a modeler has taken measures, so that duplicate references to the same object cannot exist.

ExpansionRegions in parallel and in streaming mode are also transformed to *StructuredActivityNodes*, and the mode can be attached to this *StructuredActivityNode* for further analysis. That way, possible false positives, introduced by ignoring whether duplicate references may exist in a collection, could be resolved after the analysis, if modelers use assertions regarding duplicate object references in a collection. However, such assertions are not standardized.

4.5.3 Group Specific

For each of the elements in the "Group Specific", an individual handling during the transformation is necessary.

A *BroadcastSignalAction* sends a signal similar to *SendSignalActions* (which are handled post-analysis and left unchanged during the transformation), but to multiple targets. The UML states, that the "manner of identifying the exact set of objects that are broadcast targets is not defined in this specification, however, and may be limited to some subset of all the objects that exist." [OM17] For being able to at least partially support such types of actions, it is assumed that there is exactly one target. Since then, the behavior is exactly the same as for *SendSignalActions*, *BroadcastSignalActions* are transformed to *SendSignalActions*. Checking whether there is an element receiving this signal is done in the post-analysis phase.

ConditionalNodes, *LoopNodes* and *SequenceNodes* are all containers for elements, and order the execution of contained nodes implicitly with regards to the *ControlFlow*. In general, such loops, branches and sequences can be expressed by means of *DecisionNodes*, *MergeNodes* and *ControlFlows*.

Because of this, such nodes are transformed and *StructuredActivityNodes* are used to retain the distinct parts contained in those nodes, e.g., setup and test parts. These parts are ordered using *DecisionNodes*, *MergeNodes* and *ControlFlows*.

4.5.4 Group Post-Analysis

Elements in the "Group Post-Analysis" are *CallBehaviorActions*, *AcceptEventActions* and *SendSignalActions*.

A *CallBehaviorAction* is used to invoke other Behaviors, in the context of this thesis UML Activities (the UML also describes other types of Behaviors). Such invocations may be synchronous or asynchronous.

A *SendSignalAction* itself only sends a signal and then execution is continued without waiting for the reception of this signal. *AcceptEventActions*, in contrast, wait for a signal before execution can continue. This means that for each *AcceptEventAction*, a respective signal had to be sent before the *AcceptEventAction* can execute. Whether this is the case depends on the concrete execution. Since possible executions are analyzed after the transformation, these nodes are preserved during the transformation and then handled in the post-analysis phase. During the analysis, these nodes are treated as if they were *ActionNodes*. Information on callers and callees for *SendSignalActions* and *AcceptEventActions* is attached to the respective nodes.

4.5.5 ActivityFinalNodes

ActivityFinalNodes end all flows in a *UML Activity*. In case of a synchronous call of other *UML Activities* within such a *UML Activity*, these other activities are also aborted. Since only the worst case is relevant, i.e., the longest execution possible, all flows, which are currently executing are further executed until these flows also end at a *FinalNode*. This is effectively the same as interpreting *ActivityFinalNodes* as *FlowFinalNodes*. Because of this, all *ActivityFinalNodes* are transformed into *FlowFinalNodes*.

4.5.6 ObjectElements

Two different *ObjectElements* can be distinguished: *ObjectNodes* and *ObjectFlows*.

Different types exist for *ObjectNodes*: *ActivityParameterNodes*, *ActionPins*, *CentralBufferNodes*, and *DataStoreNodes*.

ObjectNodes represent objects, which are produced, manipulated or read in actions. *ObjectNodes* have *ObjectFlows* as incoming and outgoing edges. Instead of using *ObjectNodes*, the flow of objects can also be represented by using pins attached to *ActionNodes*.

ActivityParameterNodes are used for representing input and output parameters of *UML Activities*, and are used for holding objects. Both *CentralBufferNodes* and *DataStoreNodes* represent buffers between *ObjectNodes*, with slight differences (chapter 4.3).

ObjectFlows, although transporting objects, define an ordering of actions due to the necessity of the objects being transported to nodes before these nodes can and will be executed.

ObjectNodes are replaced during the transformation, and *ObjectFlows* are replaced by *ControlFlows* and re-connected (see [Zi16]).

ActivityParameterNodes, as they influence how and when activities are started and ended (objects must have reached the respective *ActivityParameterNodes*), are replaced by *InitialNodes* and *FinalNodes*.

DataStoreNodes and *CentralBufferNodes* are also replaced analogous to *ObjectNodes*. During these transformations, it is necessary to consider and reflect the special semantics of *ObjectFlows* and *ObjectNodes*, which differs from that of *ControlNodes* and *ControlFlows*. See chapter 4.3 and [OM17] for the semantics and [Zi16] for details on the transformations.

ActionPins are reduced and transformed depending on whether an *ActionPin* is related to an *ActionNode* or to a container. In the former case they are transformed to a combination of *MergeNodes* and *JoinNodes*, or to a combination of *DecisionNodes* and *ForkNodes* (adhering to the semantics of *ObjectNodes* and *ActionNodes*). In the latter case, *ActionPins* are additionally handled similar to *ActivityParameterNodes* for the edges leading to or coming from nodes inside such a container. See [Zi16] for additional information.

4.5.7 ExecutableNodes

ExecutableNodes are abstract and the UML states, that all concrete kinds of them are actions. All *ExecutableNodes* can be associated with an *ExceptionHandler*. *ExecutableNodes* can have input pins and output pins attached to them, associated to *ObjectFlows*, as explained above.

ExceptionHandlers in turn also contain *ExecutableNodes*. *ExceptionHandlers* are always associated with a certain type of exception, which can occur during the execution of an *ExecutableNode*.

If during an execution, an exception occurs, an appropriate handler is searched for along the hierarchies, i.e., from inside the current container, where the exception was thrown, to the enclosing containers. If a handler was found, the nodes contained in this *ExceptionHandler* are executed, i.e., control flow is transferred from the node throwing the exception to the appropriate handler. Afterwards, the control flow is returned, and the execution continues from the point after the node throwing this exception.

An exception may be thrown from a region containing several nodes, and in that case, the control flow returns to the point reached when the region has finished execution.

For *ExceptionHandlers*, considering the overall goal of the analysis, the longest execution path possible needs to be considered, as otherwise some possible parallel executions of nodes could be missed.

Thus, *ExceptionHandlers* are replaced with *DecisionNodes*, and the nodes inside the handlers are combined with the *DecisionNode* using a *ControlFlow*, i.e., an exception is thrown, or not. This *DecisionNode* is placed at the latest point possible, i.e., if an exception is thrown, it is always assumed that all nodes, which can be executed before the exception occurs, are executed.

4.5.8 ActivityGroups

An *ActivityGroup* is a collection of *ActivityEdges*, *ActivityNodes* and other, possibly nested, *ActivityGroups*. *ActivityGroups* are abstract, and the UML describes two concrete types: *ActivityPartitions* and *InterruptibleActivityRegions* [OM17].

As *ActivityPartitions* do not have any impact on the execution semantics and are meant for structuring purpose only, they are not relevant for the analysis and are deleted.

InterruptibleActivityRegions may contain *InterruptingEdges*, which may either be *ControlFlows* or *ObjectFlows*. As soon as an *InterruptingEdge* is traversed, the execution of all contained nodes in the region terminates.

An additional element, which needs to be considered in combination with *InterruptibleActivityRegions*, is the *AcceptEventAction* (see chapter 4.3).

In contrast to *StructuredActivityNodes*, the region itself is not connected to other elements with edges, and is not a region, for which encapsulation holds. This means, there always exist *CrossingEdges*, which connect an element outside of the region to an element inside of the region. *InterruptingEdges* are *CrossingEdges* by definition.

Thus, during the transformation, the nodes inside such *InterruptibleActivityRegions* are preserved (and possibly transformed because of their type). The connections between the nodes inside the region, and to and from the nodes outside the region are also preserved. Finally, the information regarding which nodes are located in which *InterruptibleActivityRegion*, is preserved for post-analysis handling.

4.6 Analysis of UML Activities

The input for the analysis of *UML Activities* is the XML file produced by the transformation tool. This file may contain multiple *UML Activities*, which are processed one after the other. Possible relations from one activity to another activity are considered during the analysis, thus enabling the analysis of hierarchies of *UML Activities*. The analysis is conducted using a formally defined approach, and consists of creating and analyzing *DAGs*.

4.6.1 Representing Execution Instances of Computations by DAGs

A *UML Activity* $\boldsymbol{D}$ represents the way to compute the result given any input. It thus represents multiple, possible execution instances where the input is known. For the analysis, the exact timing of the computation steps is irrelevant, and thus abstracted. For each execution instance, the single computation threads can be represented by directed walks (as defined in chapter 4.2).

Single sequential threads occur in the following situations:

- between an *InitialNode* and the first *ForkNode* or a *FinalNode*,
- between a *ForkNode* or *JoinNode* and the next *ForkNode* or *JoinNode*, and
- between a *ForkNode* or *JoinNode* node and a *FinalNode*.

Given a *ForkNode* u, $d_D^+(u)$ walks start, and given a *JoinNode* v, $d_D^-(v)$ walks end. This can be easily represented by a DAG where the vertices are the walks and where the edges connect instances of *JoinNodes* or *ForkNodes*.

The following procedure summarizes the creation of all possible *DAGs* based on a given *UML Activity* such that each possible execution instance is represented by exactly one *DAG*. The set of all *DAGs* produced by the following procedure is a superset of all possible *DAGs* representing an execution instance since some decisions along the computation might not be independent of each other such that some combinations of walks might never occur.

The superset S of *DAGs* is created by the following procedure. Each $DAG = (W, E) \in S$ has a vertex set W, a set of walks (where exactly one walk begins with an *InitialNode*), and an edge set $E \subseteq W \times W$, the connections between walks described by the procedure, also referred to as E_{DAG} whenever suitable for clarity reasons.

Consider a *UML Activity* D with *InitialNode* u.

1. Create a DAG with single vertex $w_0 = u$ and add the *DAG* to S. Set the current edge e of D to the outgoing edge of u. Set the current vertex w of the *DAG* to w_0.

2. For current edge $e = (u, v) \in E_D$, create a new instance v' of v and update w to wv'.

    a. If v is an *ActionNode* or *MergeNode*, set the current edge to be the outgoing edge of v in D and repeat step 2.

    b. If v is a *DecisionNode*, copy this *DAG* $d_D^+ - 1$ times. For each copy, set the current walk to w and add the *DAG* to S. For each outgoing edge $(v, x) \in E_D$ of v, repeat step 2 for a separate *DAG* (with current walk w) with the current edge in D updated to (v, x).

    c. If v is a *ForkNode*, mark vertex $w \in W$ as complete. Let $n = |W|$. Create a vertex $w_i = v' \in W$ and edge $(w, w_i) \in E_{DAG}$ for each i where $n \leq i \leq n + d_D^+(v) - 1.$ For each outgoing edge $(v, x) \in E_D$ of v, repeat step 2 with the current vertex w set to $w_i \in W$ with the current edge e in D set to (v, x).

    d. If v is a *JoinNode*, mark vertex $w \in W$ with current edge e. Search for vertices z in the DAG such that $\omega(z)$ is an instance of the *JoinNode* v. Let $n = |W|$. If there is a set $\{w_{i_1}, w_{i_2}, \dots, w_{i_{d_D^-(v)-1}}\}$ of such vertices which have marks, the set E' such that $|E' \cup \{e\}| = d_D^-(v)$,

i. Create vertex $w_n = v' \in W$, edge $(w, w_n) \in E_{DAG}$, and edges $(w_{i_h}, w_n) \in E_{DAG}$ for each h where $1 \leq h \leq d_D^-(v) - 1$.

ii. Set current edge e to the outgoing edge of v in D, set the current vertex to $w_n \in W$, and repeat step 2.

e. If v is a *FinalNode*, then do nothing. The current vertex is complete.

An example for created *DAGs* can be seen in Figure 10. It shows two *DAGs* created based on the *UML Activity* depicted in Figure 9. Each of the *DAGs* contains four walks and two marks, each containing an edge. The *DAGs* differ in one walk, more precisely in the node following the node "Decision1": *DAG 1* covers "Action3", and *DAG 2* contains "Action4".

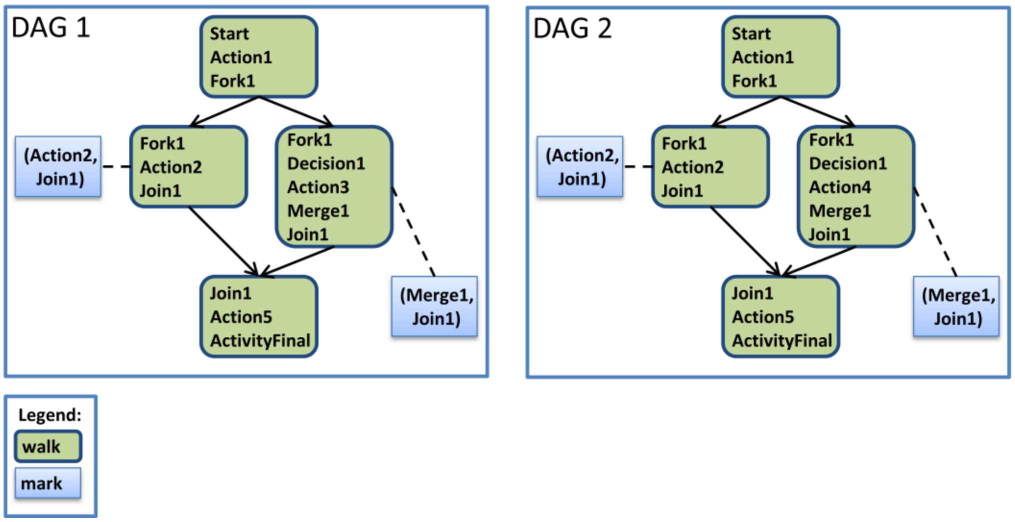


Figure 10: DAGs Created for the Exemplary UML Activity Diagram 1

As can be seen, each instance of a node is only contained once in the *DAG*, with the exception of *ForkNodes* and *JoinNodes*. Although one instance of these nodes is always contained in directly succeeding walks, it only belongs to one computation step, as every other single instance of a node. However, as *ForkNodes* and *JoinNodes* represent the transition from one walk to another, they are included more than once.

Only *valid UML Activities* are considered. To be valid, a *UML Activity* must adhere to the definitions for the *UML Activity* itself and the nodes and edges given above. *UML Activities* can be implicitly and explicitly modeled, and the preprocessing during the transformation creates explicitly modeled activities out of implicitly modeled ones.

Concerning execution instances, worst case execution instances are considered. This means that, if due to parallel execution, one of the parallel computations reaches a *FinalNode*, it is always assumed that other parallel computations have proceeded as far as possible. With this assumption, the number of considered execution instances is reduced without losing information, as shorter execution instances are subsumed under the longest path possible. Since the goal of the solution is to find possible data races, the worst case is the relevant case, as otherwise possible data races (in the analysis: nodes that may be executed in parallel to other nodes) could be missed.

Every execution instance leads to a single *DAG* by construction. However, two sets of input values may result in the same traversal of a *UML Activity*, and thus in the same *DAG*. Execution instances resulting from such sets of input values are *equivalent*, they are the same.

Lemma 1

Every pair of non-equivalent execution instances through a UML Activity $\boldsymbol{D}$ results in a unique DAG.

Proof

Assumption: two non-equivalent execution instances $\boldsymbol{ei_1}$ and $\boldsymbol{ei_2}$ result in the same *DAG*.

By definition, non-equivalent execution instances result in different outgoing edges of a *DecisionNode* $\boldsymbol{u}$ being followed at least once. Thus, different *DAGs* are created.

For each of the *DAGs*, a different outgoing edge of $\boldsymbol{u}$ is followed, thus leading to a difference in the resulting *DAGs*. As each instance can only be processed once by definition, one *DAG* is created for each single outgoing edge of an instance of a *DecisionNode*.

If two different execution instances result in the same *DAG*, this thus means that for each *DecisionNode* encountered in $\boldsymbol{D}$, the same outgoing edge has been followed for the construction of the *DAG*. However, if for each *DecisionNode* all outgoing edges followed are the same, then $\boldsymbol{ei_1}$ and $\boldsymbol{ei_2}$

represent the same single execution instance. Thus, there cannot be different execution instances that lead to the same *DAG*, and thus, every execution instance results in a unique *DAG*. ■

It is also possible that there is no set of input values that leads to a given combination of edge traversals of a *UML Activity*, due to *DecisionNodes* possibly being dependent, as explained above. For this reason, a *DAG* in the set S of *DAGs* might not reflect any execution instance.

Since an execution instance may be infinite, a *DAG* can have an infinite size. This will be discussed in chapter 4.6.5.

Two different *DAGs* based on the same *UML Activity* differ in at least one decision made at one of the *DecisionNodes* because that is the only way to create different *DAGs*. Every directed path in any *DAG* describes a sequence of computational threads that need to be executed sequentially and that never run in parallel. The following fundamental corollary is now proven, which highlights the relation of paths in a *DAG* and parallelism:

Corollary 1

Two computation threads A, B can run in parallel if an only if there is no path in the *DAG* in which A and B are contained, i.e., no path from A to B or B to A.

Proof

By construction, all computations in a computation thread are finished before any successor threads as represented in the *DAG* can start. Let there now be a directed path from A to B (without loss of generality). Then, the computations from computation thread A are, by construction of the *DAG*, finished before B's results are computed and the two threads cannot run in parallel. Thus, if two threads run in parallel, there cannot be a directed path between them in the *DAG*.

If there is no single path containing both A and B in the *DAG*, let $P(u, A)$ and $P(u, B)$ denote the set of all directed paths from the *InitialNode* u to A and u to B, respectively. For any given paths $p_A \in P(u, A)$ and $p_B \in P(u, B)$ there needs to be some common vertex w to both paths as both start at u. The last common vertex w cannot be either A or B, as otherwise the longer path would contain both A and B in contrast to the assumption. After w, there exist no further common vertices along p_A and p_B. Thus $\omega(w)$ needs to be a *ForkNode*, because a *ForkNode* is the only node of the *UML Activity* that creates more than one outgoing edge attached to its corresponding node in the *DAG*. Since A and B are thus in different computation threads regarding any pair of paths, they can run in parallel. ■

4.6.2 Parallelism and Data Races

As mentioned above, nodes in walks, which are executed in parallel and access data are in general prone to possible data races. Not all parallel executions lead to a data race, but each data race is related to parallel executions. In the following, the link between walks in *DAGs* and *DANs* in UML Activities is discussed.

Theorem 1

Let w_1 and w_2 be two walks in some *DAG*.

If two walks w_1 and w_2 are executed in parallel then so are all pairs of instances of *DANs* from the *UML Activity* D (u_1, u_2) with $u_1 \in w_1$ and $u_2 \in w_2$.

Proof

Assumption: Let w_1 and w_2 be two walks in some *DAG*, which are executed in parallel. Let u_1 and u_2 be a pair of instances of *DANs* in a *UML Activity* with $u_1 \in w_1$ and $u_2 \in w_2$.

As defined above, each walk contains a sequence of nodes, which are executed sequentially. Let w_3 be a walk with n elements $v_1 \dots v_n$. Then ∀ pairs of nodes (v_i, v_{i+1}) with $v_i \in w_3$ and $v_{i+1} \in w_3$ and $1 \leq i < n$, ∃ an edge e with $e = (v_i, v_{i+1})$.

A walk w_4, which is a direct successor of w_3 is defined to start executing after w_3 has finished. This means that $\omega(w_3) = \alpha(w_4)$. This means that all *DANs* that are contained in w_3 have been executed when the *DANs* contained in w_4 start.

If the two walks w_1 and w_2 are executed in parallel, neither does w_1 start after w_2 has been completely executed, nor does w_2 start after w_1 has been completely executed. Thus, no edges exist in the *DAG*, which lead from u_1 to

u_2. If u_1 and u_2 are not executed in parallel, then they either run sequentially, or they represent the same instance of a certain *DAN*. However, if they represent the same instance, they cannot belong to different walks by definition, as each instance of a *DAN* is contained only in one walk.

Thus, if u_1 and u_2 are not executed in parallel, then they run sequentially. If they run sequentially, then edges exist, which lead from u_1 to u_2 or from u_2 to u_1 in the *DAG*. However, since the walks w_1 and w_2 are executed in parallel, no edges exist in D between nodes in w_1 and nodes in w_2. Thus, the nodes u_1 and u_2 cannot run sequentially, they are executed in parallel. ■

Due to this, if two walks are executed in parallel, all *DANs* in one walk are prone to possible data races with all the *DANs* in the other walk. The remaining questions are whether such two *DANs* actually access the same data, if at least one of the accesses is a write access, and if there is a correct synchronization of these accesses. Answering these questions is part of the dynamic data race detection.

4.6.3 On the Number of DAGs as a Result of Transforming UML Activities

After elaborating on walks in a single *DAG*, parallelism and data races, the different possibilities for results of the construction of *DAGs* out of a *UML Activity* are discussed.

Since only the traversal of *DecisionNodes* influences the number of *DAGs* created, there are three different situations to consider:

1) A *UML Activity* contains no cycles and no *DecisionNodes*.

2) A *UML Activity* contains *DecisionNodes*, which can be traversed once each. This is the case when there are *DecisionNodes*, but no cycle, or when there are cycles, but the *DecisionNodes* cannot be traversed within a cycle traversal.

3) A *UML Activity* contains *DecisionNodes*, which can be traversed more than once. This is the case when there are cycles, and within a cycle traversal, a *DecisionNode* can be traversed.

In the following, the effects of these situations on the number of *DAGs* created for a *UML Activity* are examined.

Lemma 2

If all nodes in a valid *UML Activity* D are executed sequentially and D contains no decisions, then ∃ a walk through every node in a single *DAG* of D.

Proof

Assumption: all nodes in a valid *UML Activity* D are executed sequentially and D contains no decisions, and there is no walk through every node in a single *DAG* of D.

Let u_i be the nodes in D, with $i = 1, \dots n$. If all nodes in D are executed sequentially, then $(u_j, u_{j+1}) \in E \; \forall \, j = 1, \dots \, n-1$.

Since all nodes are executed sequentially, i.e., there are no *ForkNodes*, and there are no *DecisionNodes*, each node has $d_D^+ \leq 1$. In such a case, during the construction of the walks for *DAGs*, no new walks are created and no new *DAGs* are created. Thus, there is a single *DAG* of D with exactly one walk.

If ∃ $v \in V$ and v is not in the walk, and since this walk is the only walk in the *DAG*, then no edge $\boldsymbol{e}$ exists between v and any node u. However, since all nodes in D are executed sequentially and an edge exists for each node in D either to its direct successor or its direct predecessor or both in a sequence of nodes, v has to be included in the walk. ■

Lemma 3

If all nodes in a valid *UML Activity* D are sequential and D contains *DecisionNodes* but no cycles, within which a *DecisionNode* can be traversed, then ∃ a walk through every node in a set of *DAGs* of D. For a *UML Activity* D with a set of *DecisionNodes* $\{u_1, u_2, \dots, u_n\}$, the set of *DAGs* has a size of $\sum_{i=1}^{n}(d_D^+(u_i) - 1) + 1$.

Proof

Assumption: All nodes of D are sequential and D contains *DecisionNodes* but no cycles, within which a *DecisionNode* can be traversed. Then ∃ a node v which is not covered in any walk.

By Lemma 2, if all nodes in D are executed sequentially, a single walk is created. However, due to *DecisionNodes* multiple *DAGs* are created. Each of these *DAGs* contains one walk, covering one execution instance. Since, given a *DecisionNode* $\boldsymbol{u}$ with $d_D^+(u) > 1$, $d_D^+(u) - 1$ new *DAGs* are created and each of the *DAGs* containing $\boldsymbol{u}$ represents the traversal of a different outgoing edge of $\boldsymbol{u}$, all outgoing edges of $\boldsymbol{u}$ are covered. Thus, all possible differences in execution instances are represented in different *DAGs*.

Since all nodes in $\boldsymbol{D}$ are sequential and $\boldsymbol{D}$ is valid, there is an edge $\boldsymbol{e}$, which connects $\boldsymbol{v}$ either to its direct predecessor or to its direct successor, or both. Since all nodes are sequential, there exists exactly one initial node $\boldsymbol{s}$. Without loss of generality, we can thus assume that $\boldsymbol{v}$ is a (direct or indirect) successor of $\boldsymbol{s}$ or that $\boldsymbol{v}$ is $\boldsymbol{s}$. If $v = s$, it is included in every *DAG* and thus the assumption is wrong.

If $v \neq s$, then in the sequence of nodes from $\boldsymbol{s}$ to $\boldsymbol{v}$, there is either no *DecisionNode* or there is at least one *DecisionNode*. If there is no *DecisionNode* in the sequence, then up to $\boldsymbol{v}$, only one execution instance exists, and this execution instance includes $\boldsymbol{v}$, as shown above.

If there is at least one *DecisionNode* in the sequence of nodes from $\boldsymbol{s}$ to $\boldsymbol{v}$, then, as shown, all different execution instances resulting from the outgoing edges of *DecisionNodes* are covered in different *DAGs*. Thus, ∃ a *DAG*, which includes the sequence of nodes from $\boldsymbol{s}$ to $\boldsymbol{v}$, and thus, ∃ a *DAG*, which includes $\boldsymbol{v}$. Since all nodes in *D* are sequential, this *DAG* contains one walk. Thus, ∃ a walk in a *DAG* that covers $\boldsymbol{v}$.

Given a *DecisionNode* $\boldsymbol{u}$, $d_D^+(u) - 1$ different *DAGs* are created. Since no cycles exist in $\boldsymbol{D}$, within which a *DecisionNode* can be traversed, each *DecisionNode* in an execution instance can only be instantiated once. Since all execution instances cover all decisions and the set of *DAGs* covers all execution instances, during the construction of the set of *DAGs*, $\sum_{i=1}^{n}(d_D^+(u_i) - 1)$ new *DAGs* are created. Since the construction of *DAGs* starts with an empty *DAG*, a *UML Activity*, which contains at least one *DecisionNode*, results in $\sum_{i=1}^{n}(d_D^+(u_i) - 1) + 1$ *DAGs*. ■

Lemma 4

If all nodes in a valid *UML Activity* $\boldsymbol{D}$ are sequential and $\boldsymbol{D}$ contains at least one cycle, within which *DecisionNodes* can be traversed, then ∃ a walk through every node in a set of *DAGs* of $\boldsymbol{D}$. There are infinitely many resulting *DAGs*.

Proof

By Lemma 3, if all nodes of D are sequential and D contains decisions, every node is included in at least one walk of a *DAG* out of the set of *DAGs* created.

D contains at least one cycle, and this cycle contains at least one *DecisionNode* u. Due to the cyclic structure, this u may be instantiated more than once, up to infinitely many times. Since each instance of u is independent of other instances, for each of these instances $d_D^+(u) - 1$ DAGs are created. Since a cycle can be traversed infinitely many times, infinitely many decisions have to be made. Thus, a *UML Activity* D, which contains at least one cycle, within which a *DecisionNode* can be traversed, results in infinitely many *DAGs*. ■

4.6.4 Token Concept for Walks in a DAG

It is now clear, and proven, how many *DAGs* will be created using the solution shown above in the different situations that exist using the basic elements of *UML Activities*. It has also been shown, that if two walks are executed in parallel, all *DANs* in one walk may run in parallel with all *DANs* in the other walk. The remaining question is how such parallel walks may be identified.

For identifying parallel walks, and for symbolizing the execution path of nodes contained in walks in a *DAG*, tokens are used. Each token is modeled by a set of sequences of numbers for each walk in each *DAG* in the set of *DAGs*. Each walk has exactly one token, which contains one or more sequences of numbers. The numbers are separated by a dot.

T_a is defined as the token of a walk a, containing a set of sequences. $|T_a|$ is defined as the cardinality of T_a, i.e., the number of sequences in Token T_a.

s_{aj} is defined as being the j-th sequence in the token T_a. $P(s_{aj})$ is defined as the prefixes of sequence s_{aj}. A prefix is defined as a proper prefix, i.e., the length of a prefix of a sequence is always smaller than the sequence itself. $\forall\ p\ \in\ P(s_{aj})$, with the length of $p\ =\ k$, the j-th position with $1\ \leq j\ \leq k$ is identical to the j-th position of s_{aj}. The sequences in tokens are constructed as follows:

1. There is exactly one token T at the root walk of the *DAG*, i.e., the walk containing the *InitialNode* of a *UML Activity*. This token is modeled by the sequence $\mathbf{0}$.

2. The sequences and the set of sequences in a token T follow the outgoing edges of a walk. They will be changed according to the following case distinction:

a. The current walk e has k outgoing edges, with $k > 1$. Copy T $k - 1$ times such that in total there are k copies, $T_1, T_2, \dots, T_k$. Append a dot and one of the numbers 1 to k to all sequences in each token, corresponding to the index of this token. The token T_i is then assigned to the i-th successor of e, with $1 \leq i \leq k$.

b. The current walk e has 1 outgoing edge and the successor walk v has j incoming edges. Let $T_1, T_2, \dots, T_j$ be the tokens of the incoming neighbors of v. The sequences in the tokens of the incoming neighbors of v are copied and attached to the token of v. The walk v then has one token with $\sum_{n=1}^{j} |T_n|$ sequences.

The intuition behind the sequences in a token is to memorize the walks on which this token builds on, i.e., the token contains only those sequences of the tokens of walks that are definitely finished, when the walk related to the token is executed. The sequences may either be unchanged and combined out of sequences of other tokens, or extended, according to the rules for the construction of the sequences in tokens described above. Actions that are in a sequence are combined into one walk, and new walks only emerge when parallel threads are started or synchronized. As an example, if the *InitialNode* in a *UML Activity* leads to four actions in a sequence, these would all be contained in the root walk and covered by the token 0. Note that tokens in different *DAGs* cannot be set into relation to each other, as different *DAGs* for the same *UML Activity* represent alternative executions.

Theorem 2

Let a, b be two walks in some *DAG*. The walks a, b can be executed in parallel, iff $\forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$ it holds that $s_{ai} \neq s_{bj}$, $s_{ai} \notin P(s_{bj})$, and $s_{bj} \notin P(s_{ai})$.

Proof

($\Leftarrow$) Walks a and b run in parallel $\Leftarrow \forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$ it holds that (1) $s_{ai} \neq s_{bj}$, (2) $s_{ai} \notin P(s_{bj})$, and (3) $s_{bj} \notin P(s_{ai})$.

Assuming $\forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$ it holds that (1) $s_{ai} \neq s_{bj}$, (2) $s_{ai} \notin P(s_{bj})$, and (3) $s_{bj} \notin P(s_{ai})$ and walks a and b run not in parallel. If a and b do not run in parallel, they are identical or are successors of each other.

If (1), (2) and (3), then s_{ai} and s_{bj} are different at at least one position n, with $n \leq$ length of s_{ai} and $n \leq$ length of s_{bj} $\forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$ by construction. Since all sequences start with a $\mathbf{0}$, representing the root walk, s_{ai} and s_{bj} are identical at the first position by construction. A given sequence is only changed, if during the construction of the *DAG*, a *ForkNode* is reached; in such a case, a dot and a number are added to each sequence in the token, with each succeeding walk being represented by a different number (see the construction of the sequences and the tokens above). Thus, the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode*.

Since (2) and (3) $\forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$, walks a and b are not successors of each other, which are separated by a *ForkNode* in the sequence of walks between a and b or directly between a and b.

Since (1) $\forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$, walks a and b are not identical and walks a and b are not successors of each other, which are directly separated by a *JoinNode*. Since two succeeding walks can only be separated by *ForkNodes* or by *JoinNodes* by construction, walks a and b can thus not be successors of each other.

However, if the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode*, a and b are not identical to each other and are not direct or indirect successors of each other, a and b are different successors or successors of different successors of the common *ForkNode*. Thus, they run in parallel. This is a contradiction to the assumption.

$(\Rightarrow)$ Walks a and b run in parallel $\Rightarrow \forall s_{ai} \in T_a$ and $\forall s_{bj} \in T_b$ it holds that (1) $s_{ai} \neq s_{bj}$, (2) $s_{ai} \notin P(s_{bj})$, and (3) $s_{bj} \notin P(s_{ai})$.

1. Assumption: walks a and b run in parallel, and $\exists s_{ai} \in T_a$ and $\exists s_{bj} \in T_b$, so that $s_{ai} = s_{bj}$.

   Then, the tokens T_a and T_b contain the same sequence s, with $s = s_{ai} = s_{bj}$. This can only happen, if T_a and T_b are identical, or if a sequence from token T_a is copied to token T_b, or vice versa.

   If T_a and T_b are identical, then a and b are identical, as, by construction, the same walk can only be contained once in each *DAG* with identical tokens. However, a walk cannot run in parallel to itself.

   If a sequence from token T_a is copied to token T_b or vice versa, then a *JoinNode* has been encountered between walks a and b. Without loss of generality, it can be assumed that a *JoinNode* has been encountered between a and b (and not between b and a). It can be excluded that a

ForkNode is encountered between $\boldsymbol{a}$ and $\boldsymbol{b}$, as otherwise, the sequence would have been changed due to that *ForkNode*.

If the *JoinNode* between $\boldsymbol{a}$ and $\boldsymbol{b}$ is not the same, then a sequence is not copied from one token to another token. Thus, $s_{ai} \neq s_{bj}$. If the *JoinNode* between $\boldsymbol{a}$ and $\boldsymbol{b}$ is the same, then $\boldsymbol{b}$ can only be executed after $\boldsymbol{a}$ has been finished. Thus, they cannot run in parallel.

2. Assumption: walks $\boldsymbol{a}$ and $\boldsymbol{b}$ run in parallel, and $\exists\, s_{ai} \in T_a$ and $\exists\, s_{bj} \in T_b$, so that $s_{ai} \in P(s_{bj})$.

   Then, s_{ai} is a prefix of s_{bj} and the length of s_{ai} is smaller than the length of s_{bj}. A number is by construction only attached to a sequence, if a *ForkNode* is encountered during the construction of the *DAG*. Since $\boldsymbol{a}$ and $\boldsymbol{b}$ run in parallel, the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode*.

   Let the length of s_{ai} be $\boldsymbol{k}$ and let the length of s_{bj} be $\boldsymbol{n}$. Then, $\boldsymbol{k} < \boldsymbol{n}$, since s_{ai} is a prefix of s_{bj}. Since s_{ai} being identical to s_{bj} up to the $\boldsymbol{k}$-th position, each sequence of a walk directly following a *ForkNode* being extended by a number, and the number being separate for each different direct successor of a *ForkNode*, the sequences of walks represented by s_{ai} and s_{bj} cannot both contain the same *ForkNode*, or both share the same successor walk following a *ForkNode*, up to the $\boldsymbol{k-1}$-th position of the sequence. If the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode* and the same direct successor walk of this *ForkNode*, they cannot run in parallel.

   However, they can share a *ForkNode* in the $\boldsymbol{k}$-th position of the sequence. This means that $\boldsymbol{a}$ ends with the same *ForkNode* that $\boldsymbol{b}$ starts with and that $\boldsymbol{b}$ is a successor of $\boldsymbol{a}$. Thus, $\boldsymbol{b}$ can only run after $\boldsymbol{a}$ has been executed and is finished. Thus, they cannot run in parallel.

3. Assumption: walks $\boldsymbol{a}$ and $\boldsymbol{b}$ run in parallel, and $\exists\, s_{ai} \in T_a$ and $\exists\, s_{bj} \in T_b$, so that $s_{bj} \in P(s_{ai})$.

   Then, s_{bj} is a prefix of s_{ai} and the length of s_{bj} is smaller than the length of s_{ai}. A number is by construction only attached to a sequence, if a *ForkNode* is encountered during the construction of the *DAG*. Since $\boldsymbol{a}$ and $\boldsymbol{b}$ run in parallel, the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode*.

   Let the length of s_{bj} be $\boldsymbol{k}$ and let the length of s_{ai} be $\boldsymbol{n}$. Then, $\boldsymbol{k} < \boldsymbol{n}$, since s_{bj} is a prefix of s_{ai}. Since s_{bj} being identical to s_{ai} up to the $\boldsymbol{k}$-th position, each sequence of a walk directly following a *ForkNode* being extended by a number, and the number being separate for each different direct successor of a *ForkNode*, the sequences of walks represented by s_{ai} and s_{bj} cannot both contain the same *ForkNode*, or both share the same successor walk following a *ForkNode*, up to the $k-1$-th position of the sequence. If the sequences of walks represented by s_{ai} and s_{bj} both contain the same *ForkNode* and the same direct successor walk of this *ForkNode*, they cannot run in parallel.

However, they can share a *ForkNode* in the k-th position of the sequence. This means that b ends with the same *ForkNode* that a starts with and that a is a successor of b. Thus, a can only run after b has been executed and is finished. Thus, they cannot run in parallel. ∎

4.6.5 Complete Analyses of UML Activities with Limited Cycle and Edge Traversals

It has been shown how to create *DAGs* based on *UML Activities* using basic elements. The token concept is used for identifying parallel walks. *DANs* in such parallel walks may be executed in parallel.

In this chapter, the challenges related to using *UML Activities* to determine pairs of nodes that may run in parallel with regard to cycles are discussed. Obviously, to determine all possible pairs of *DANs*, all such nodes must be contained in at least one of the considered execution instances of the *UML Activity*.

It is desirable to determine such pairs in a reasonable amount of time and one common solution is to limit the number of traversals of a cycle in the *UML Activity* (see chapter 3.3).

However, it is possible that a data accessing node, u, runs in parallel with itself. Thus, if it is possible to execute u twice or more often in the same execution instance, then it is also possible that there exists an execution instance where one instance of u is running in parallel with a different instance of u. For this reason, some cycles must be traversed at least once (compare the definition of cycles in chapter 4.2) if such a pair is to be determined as possibly running in parallel.

Figure 11 illustrates a *UML Activity Diagram* exhibiting this situation with node “Action2”.

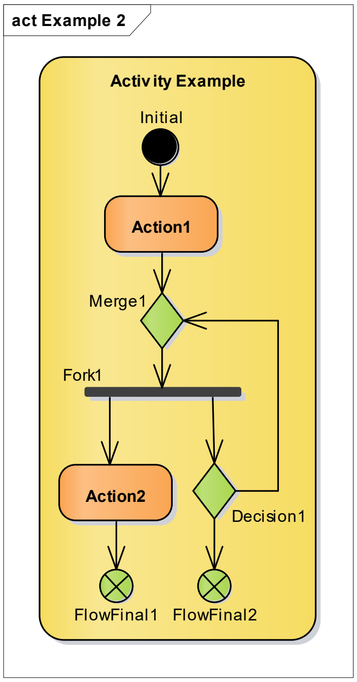


Figure 11: Exemplary UML Activity Diagram 2

Additionally, considering nodes and edges that are contained in at least one execution instance, then the following conditions together are not enough to ensure that all possible pairs of data accessing nodes, which may run in parallel, are identified:

- all nodes of the *UML Activity* have been traversed at least once,
- all edges of the *UML Activity* have been traversed at least once, and
- all cycles of the *UML Activity* have been traversed at least once.

Figure 12 shows an example of a *UML Activity*, for which there is no way to determine that the node "Action2" is in parallel with itself given adherence to the above conditions. Note that the number of incoming edges to the *JoinNode* "Join1", k, need only be 2 and these conditions will already be too limiting. It can be seen that requiring $2 * d_D^-(Join1)$ traversals of each cycle are necessary to ensure that all possible data races are determined for this figure. Since k can be varied by including additional edges, there is no fixed limit in the number of cycle traversals that might be necessary for computing all pairs of data accessing nodes possibly running in parallel with this requirement.

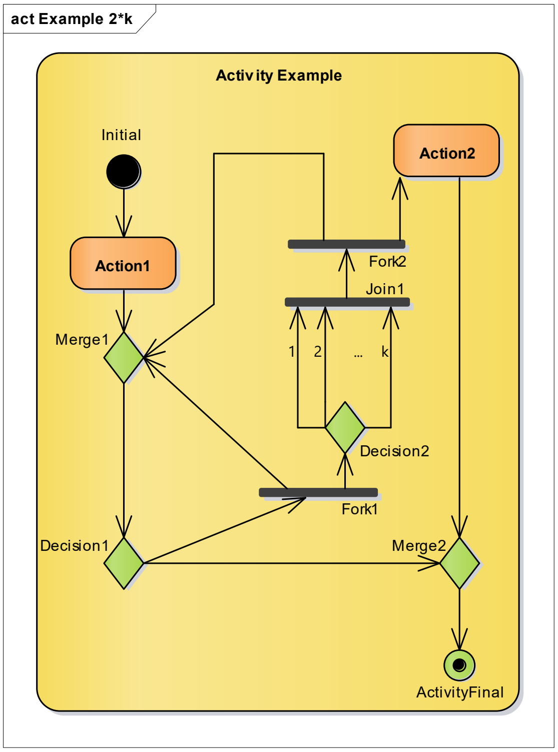


Figure 12: Exemplary UML Activity Diagram 3 (2*k Cycle Traversals)

This figure can be edited slightly (see Figure 13) leading to requirements that the number of traversals of cycles be quadratic in the indegree (more precise: for the two *JoinNodes* "Join1" and "Join2", it is $2 * d_D^-(Join1) * d_D^-(Join2)$.

This means there is no fixed limit in the number of cycle traversals that might be necessary for computing all pairs of data accessing nodes possibly running in parallel with this requirement. It has been shown that for all approaches computing nodes that may run in parallel, which use hard-defined limits in the number of cycle traversals, an example demonstrating the incompleteness of the analysis can easily be created. This shows the limitations of the approaches for the analysis of *UML Activities* discussed in chapter 3.3.

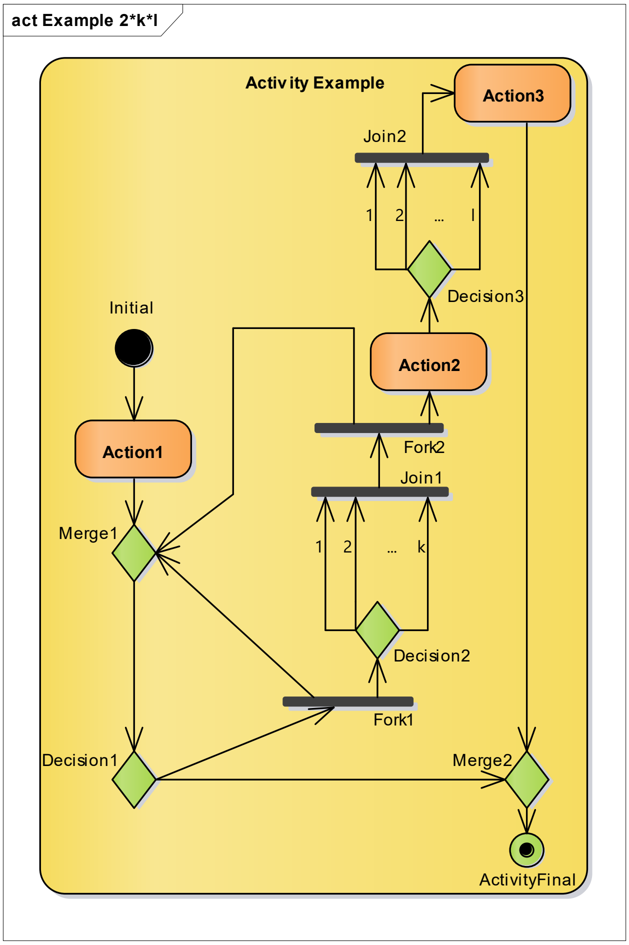


Figure 13: Exemplary UML Activity Diagram 4 (2*k*l Cycle Traversals)

By definition, the number of *DAGs* created depends on the number of traversals of *DecisionNodes* in a *UML Activity* (see chapter 4.6.1 and chapter 4.6.3). If there are cycles in the activity, this results in an infinite number of resulting *DAGs*. If there are no cycles, but a set of *DecisionNodes* $\{u_1, u_2, \dots, u_n\}$ then the number of *DAGs* created is $\sum_{i=1}^{n}(d_D^+(u_i) - 1) + 1$. If there are no *DecisionNodes*, then one *DAG* is created that captures all nodes in the *UML Activity*. Parallelism in a *UML Activity* does not affect the number of *DAGs* created, but instead the number of computation threads, i.e., walks in a *DAG*.

Cycles are insofar related to *DecisionNodes* as without such a *DecisionNode* inside a cycle, it would not be possible to exit a cycle, resulting in an infinite *DAG*. *UML Activities* with a possibility to exit cycles instead lead to an infinite number of resulting *DAGs*.

Without a stopping criterion, cycles would thus not be analyzable. It has been shown that any fixed limit in the number of cycle traversals is not enough for *UML Activities* in general to be completely analyzed for nodes that may run in parallel.

Thus, a dynamic stopping criterion is required, which is adapted for each *UML Activity*.

In addition to the challenge of exiting cycles, nodes possibly running in parallel to themselves must be considered in this stopping criterion.

In this regard, the criterion forming the lower bound for cycle traversals is not related to a single *DAG* or to a number of nodes or edges traversed within a *DAG*. Instead, all combinations of two traversals of edges must be contained in the sum of the *DAGs*. A *DAG*, which contains all edges of a *UML Activity* traversed at least twice (where possible), is called *DAG**.

The addition "(where possible)" refers to the fact that some edges cannot be traversed twice or more often. This is, e.g., true for the first edges in a *UML Activity*, following the *InitialNode*, when there is no cycle at this point. There is only one *InitialNode* as the analysis follows the transformation and preprocessing, as shown in chapter 4.5. As there is no cycle at that point, these edges cannot be traversed more than once. Thus, all edges until a *MergeNode* is reached are ignored for this stopping criterion (but traversals are still counted).

The following Figure 14 shows a *UML Activity Diagram*, for which it is not enough to create a random *DAG* containing two traversals of each edge. Instead, a certain combination of edge traversals is required.

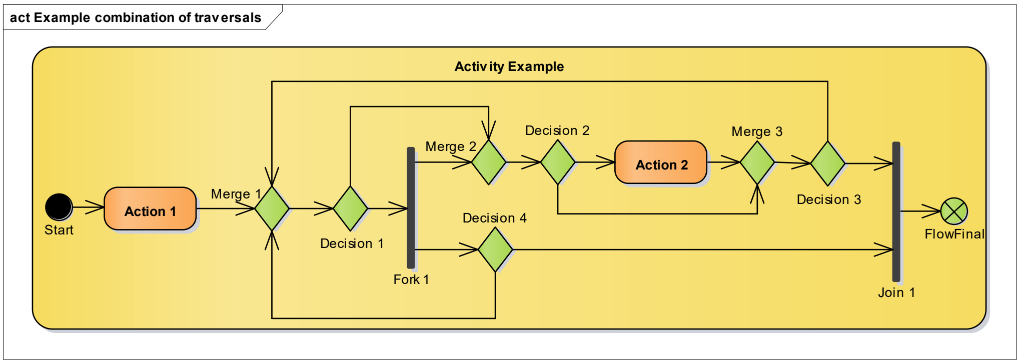


Figure 14: Exemplary UML Activity Diagram 5 (Combination of Edge Traversals)

In the example above, the node "Action 2" may be executed in parallel to itself. However, the edge (Decision 2, Action 2) must be traversed twice, and the walks containing these two traversals must not be successor or predecessor of each other. The following two paths contain two traversals of the edge (Decision 2, Action 2), but do not exhibit the parallel execution of node "Action 2":

Path 1: Start – Action 1 – Merge 1 – Decision 1 – Fork 1 – Merge 2 – Decision 2 – Action 2 – Merge 3 – Decision 3 – Merge 1 – Decision 1 – Fork 1 – Merge 2 – Decision 2 – Action 2 – Merge 3 – Decision 3 – Join 1 – FlowFinal.

Path 2: Start – Action 1 – Merge 1 – Decision 1 – Fork 1 – Decision 4 – Join 1 – FlowFinal.

The following two paths instead show the parallel execution of node "Action 2":

Path 1: Start – Action 1 – Merge 1 – Decision 1 – Fork 1 – Merge 2 – Decision 2 – Action 2 – Merge 3 – Decision 3 – Join 1 – FlowFinal.

Path 2: Start – Action 1 – Merge 1 – Decision 1 – Fork 1 – Decision 4 – Merge 1 – Decision 1 – Fork 1 – Merge 2 – Decision 2 – Action 2 – Merge 3 – Decision 3 – Join 1 – FlowFinal.

In this example, the edge (Decision 2, Merge 3) has not been traversed. It is not possible to traverse the edge (Start, Action 1) or the edge (Action 1, Merge 1) more than once.

It has been shown that the challenges of detecting nodes running in parallel to themselves and of cycle traversals cannot be solved in isolation. Only considering nodes that may run in parallel to themselves may lead to missing exit criteria for cycles, and to infinite *DAGs*. Only considering cycle traversals may lead to missing such nodes running in parallel to themselves. Thus, the following two challenges needs to be considered in combination:

1) all combinations of two edge traversals are needed to capture the parallel execution of an instance of a node with another instance of the same node (if possible).

2) how the challenge of cycles can be solved with a dynamic stopping criterion resulting from analyzing the elements within a specific cycle and considering the semantics of those elements, leading to the possibility to exit cycles (when possible) while preventing infinite cycle traversals.

Only elements that may have an outdegree or indegree > 1 may influence these criteria, as all other elements can just be traversed following the only outgoing or incoming edge. These are *ForkNodes*, *JoinNodes*, *DecisionNodes*, and *MergeNodes*.

MergeNodes are necessary for cycles to be possible. Such nodes have an indegree > 1. Assuming explicit modeling (which is guaranteed by the transformation step), at least one of the incoming edges can be traced back to the same *MergeNode*, if a cycle exists. The other node type possibly having an indegree > 1 is the *JoinNode*. However, a *JoinNode* cannot be used to initiate or close a cycle, as the *JoinNode* requires tokens offered on all edges to be traversed. This assumes the standard *ValueSpecification* being used. As the *ValueSpecification* cannot be considered in the analysis, this assumption holds. A *ValueSpecification* could be formulated, that uses an "OR" semantics, leading to only one of the incoming edges offering a token would be sufficient to traverse this node, basically simulating a *MergeNode*. However, it can be argued that this violates the purpose of a *JoinNode* to synchronize incoming flows [OM17]. Thus, such a construction can be considered as being *invalid* as per the definition provided in chapter 4.6.1. The analysis only considers valid *UML Activities*, and thus such a construction can be excluded without loss of generality.

ForkNodes start parallel flows. As such, these nodes are required for a node being able to be executed in parallel to itself. All outgoing edges of a *ForkNode* are traversed in parallel, and in the creation of *DAGs*, a *ForkNode* leads to initiating new *walks* within a *DAG*.

DecisionNodes offer alternative flows. As explained, such node types affect the creation of new *DAGs* and the number of *DAGs* created in the analysis of a *UML Activity*. These nodes also affect the traversals of edges, as only one of the outgoing edges is following per instance of such a node within a *DAG*. Thus, *DecisionNodes* affect challenge 1 mentioned above.

JoinNodes affect challenge 2 mentioned above, as explained. The indegree of *JoinNodes* within cycles, if existing, affects the ability to progress past these *JoinNodes*, and thus, to reach edges following such *JoinNodes* and eventually to exit, i.e., progress past such cycles. Since, as per challenge 1, each edge needs to be traversed at least twice, the relevant number for a *JoinNode* $\boldsymbol{u}$ in a UML Activity $\boldsymbol{D}$ is $2 * d_D^-(u)$. The criterion is then created by summarizing the numbers for all *JoinNodes* in a *UML Activity*. This criterion is a stopping criterion and is called *EmergencyExit*. If there is no *JoinNode*, then according to challenge 1, the limit is $\mathbf{2}$.

Both criteria are counted for each *DAG* and each edge in a *DAG* individually and only affect the current *DAG*. As for the criterion related to challenge 1, some edges are ignored for this limit, as explained above.

Still, *UML Activities* can be created, for which this criterion cannot hold alone, as there may also be edges after a *MergeNode*, which cannot be traversed twice or more often. Thus, the second criterion is required to be processed in combination.

As shown, no hard-coded, i.e., fixed, limit for the number of cycle traversals can be used. In this regard, the "each edge twice" criterion does not induce a fixed limit on the number of cycle traversals, as the number of needed cycle traversals for traversing each edge twice depends not only on the structure of the *UML Activity*, but also varies from *DAG* to *DAG* (as at each *DecisionNode*, for each outgoing edge, a new *DAG* is created).

As for the *EmergencyExit*, if at least one edge is traversed as often in a DAG, as the *EmergencyExit* is set to, then this *EmergencyExit* is reached and any further attempt to traverse the respective edge is stopped. Other parts of the *DAG*, i.e., other edge traversals continue.

The creation of a *DAG* does not fully stop when the *EmergencyExit* is reached, but just the traversal of the respective edge. This means that other parts of a *DAG* might continue to be created. Because of this, there might be several edges in one *DAG*, for which the *EmergencyExit* is reached.

When an *EmergencyExit* is reached, the respective *DAG* or walk in this *DAG* is called *incomplete*. For avoiding misinterpretations of incomplete walks, an "artificial *EmergencyExit* node" is inserted at the end of such walks, i.e., instead of the node that would follow the edge just traversed. This artificial node is identified during the analysis of walks and nodes contained in these walks. These artificial nodes are interpreted as *FlowFinalNodes*.

Even if it is possible to traverse an edge of the *UML Activity* twice or more in a single DAG, it is unclear when that edge will appear in a *DAG* more than once if it will at all. Because of this, the *EmergencyExit* has to be reached during the analysis to be stopped. Even if an edge is not traversed twice, and thus, the "two edge traversal" criterion is not reached, any traversal path through a *UML Activity* will either eventually lead to a *FinalNode* or end up in a cycle. In this cycle, some edge is continued to be traversed, and at some point in time, the *EmergencyExit* is triggering. Then, the further creation of walks will be stopped, and the problem of infinite *DAGs* is avoided.

It is not guaranteed that with the above criteria, a *DAG** is always created. In some cases, this is also not possible. Creating a *DAG** is on the one hand not enough, and on the other hand, not necessary to detect all nodes that may be executed in parallel to themselves

As shown above, it is not enough to obtain just any DAG* to find all pairs of nodes possibly running in parallel, because in some cases, it is not enough to traverse each edge twice, but a certain combination of edge traversals is necessary to obtain complete results.

It is, however, not necessary to traverse each edge twice and to analyze all possible combinations of edge traversals to find a certain pair of nodes. It is sufficient to traverse *some* edges twice and to fulfill *some* combination of edge traversals to find a certain pair of nodes (as shown above). Due to the systematic exploration of edges and traversals, these different combinations of traversals are captured in different *DAGs*. These *DAGs* are systematically constructed while following edges and counting traversals of single edges.

Considering the algorithm to create *DAGs* in chapter 4.6.1, step 2.(b) includes the creation of new *DAGs* when a *DecisionNode* is analyzed and the instruction to follow each of the outgoing edges in a separate *DAG*. This allows to explore a *UML Activity* in the breadth and then to extend the *DAGs* in the depth. Because of this, it is not necessary to wait for a *DAG** to be created (if it will be created), where possible. Instead, since *DAGs* with different combinations of traversals and different edge traversals are created, those edges traversed at least twice mentioned above as *some* and the combinations of traversals mentioned above as *some* are included in a *DAG* before a *DAG** would be created, if possible. Because of this, there is no need to construct a *DAG** to find all pairs of nodes possibly running in parallel. Instead, all edge traversals and all combinations of edge traversals required are reached with using the *EmergencyExit*.

After *DAGs* and walks have been created, tokens and sequences in the tokens are added to each walk for each *DAG*. This process of attaching tokens and sequences to walks follows the procedure defined in chapter 4.6.4.

With these tokens and sequences, walks, which may run in parallel, are calculated by comparing the sequences in the token for each walk with each other walk. This comparison follows the rules defined and proven to be correct in chapter 4.6.4. The calculation of tokens and sequences is conducted in between the post-analysis (as explained next).

Based on walks, which may run in parallel, it is trivial to extract nodes contained in these walks. These nodes are then collected, and cleaned from duplicates.

4.7 Post-Analysis of UML Elements

As mentioned, for some elements of *UML Activities*, it is not sufficient or not possible to handle the specific semantics during the transformation. Instead, a post-analysis handling is necessary. This post-analysis handling takes place after *DAGs* and walks have been created.

Some parts of the handling have to be conducted before calculating tokens and sequences, and some parts have to be executed while identifying nodes possibly running in parallel.

The following elements require a post-analysis handling:

- *CallBehaviorAction*
- *AcceptEventAction*
- *SendSignalAction*
- *StructuredActivityNode*
- *CrossingEdge*
- *ExpansionRegion*
- *InterruptibleActivityRegion*
- *InterruptingEdge*

A *CallBehaviorAction* is used to invoke other behaviors, in the context of this thesis other *UML Activities*. After creating *DAGs* and walks, each walk is analyzed for such *CallBehaviorActions*.

If the call is synchronous, all the elements of the called activity, except for *InitialNodes* and *FinalNodes*, are inserted into the current walk, replacing the *CallBehaviorAction*. Afterwards, this walk is again analyzed for *CallBehaviorActions*, as the called activity may contain other *CallBehaviorActions*. As per the assumption that all elements contained in a *UML Activity* can be executed, all elements of the called activity can be included. The order, in which the elements of the called activity are executed, is not important in this context, because this order is analyzed when analyzing this called *UML Activity* itself. If a *CallBehaviorAction* can be executed in parallel to another node, all elements of the called *UML Activity* may run in parallel to this node. After this handling, all *CallBehaviorActions* are eliminated.

It is possible that *CallBehaviorActions* in *UML Activities* form a cycle, i.e., a node in activity $\boldsymbol{C}$ calls an activity $\boldsymbol{D}$, in which activity $\boldsymbol{C}$ is called. In such a case, the insertion of nodes ends after inserting all the elements of activity $\boldsymbol{C}$ in the currently analyzed walk of activity $\boldsymbol{C}$. In case of such a cycle, all elements of activity $\boldsymbol{C}$ may run in parallel to any node possibly running in parallel to the *CallBehaviorAction* of activity $\boldsymbol{C}$ initiating this cycle. As all necessary information is already obtained and duplicated entries cannot provide additional information, it is not necessary to insert further elements, and further *CallBehaviorActions*, continuing this cycle, are just deleted in this *walk* and the cycle is resolved.

In case the call is asynchronous, the flow continues with the outgoing edge of the *CallBehaviorAction*, while the called *UML Activity* is executed in parallel. A new walk is created, which runs in parallel to the further elements of the walk containing this *CallBehaviorAction* instance. This is achieved by replacing the *CallBehaviorAction* with a *ForkNode*. The walk is then split at this *ForkNode*, and the elements of the called activity are placed in a new walk, which runs in parallel to the walk containing all the elements of the original walk, which follows the *CallBehaviorAction*. The newly formed walks follow the rules for walks stated in chapter 4.6.1. The reflections on cycles using *CallBehaviorActions* also hold for asynchronous calls.

AcceptEventActions and *SendSignalActions* are checked per *DAG*. As each signal sent by a *SendSignalAction* can only be received by one *AcceptEventAction*, it is necessary to track which signal sent by a *SendSignalAction* has already been received and used. For each *AcceptEventAction* to continue executing, a corresponding signal has to be received. Because of this, for each *AcceptEventAction* it is checked if there is a corresponding *SendSignalAction*, whose signal has not been consumed yet, and which has been executed before, or which is executing in parallel.

The corresponding *SendSignalAction* can be contained in the same walk or in another walk as the *AcceptEventAction* waiting for the signal. In the latter case, this walk has to be executed either before or in parallel to the walk containing the *AcceptEventAction*. If both are contained in the same walk, then the *SendSignalAction* has to be executed before the *AcceptEventAction* is executed.

If these conditions are not met, then an *AcceptEventAction* waits for a signal, which will not be sent, and the node cannot be executed. In such cases, the *DAG* and the walk containing such an AcceptEventAction are shortened. The walk containing this node is shortened at the position of this node. All walks, which are successors of this walk, are deleted, as they cannot execute. This shortened *DAG* can then be analyzed using tokens as described.

The post-analysis handling of *InterruptibleActivityRegions* and *InterruptingEdges* consists of using additional information obtained during the transformation. For each node contained in such an *InterruptibleActivitiyRegion*, the ID of this region is added to the node and the node is marked as "interruptible".

Afterwards, the *InterruptingEdges* are checked. For the node connected by the *InterruptingEdge* as head, and all nodes following this node in a *DAG*, the ID of the source region is added, and the flag "interruptingFollower" is set.

Afterwards, for each node it is checked, if it is interruptible. If yes, then it is checked if there are other nodes marked as "interruptingFollower". If this is the case, and the IDs of the regions are the same, then these nodes cannot run in parallel. Such an *InterruptibleActivityRegion* can be interrupted at different stages of execution, i.e., different nodes may have been executed when the execution of the nodes inside a region is interrupted, and this needs to be considered during the handling of such constructs.

There is a special semantics of *AcceptEventActions* in such regions (chapter 4.5.8). Also, *CrossingEdges* currently executing when the interrupt is triggered continue to execute.

StructuredActivityNodes, along with *CrossingEdges*, are again handled post-analysis using additional information. For each edge, the information whether this edge is a *CrossingEdge* is contained in the XML file. For the creation of the *DAGs*, these edges have been ignored.

For each *CrossingEdge*, all walks in each *DAG* are examined. If any of the nodes in a walk is the tail of this *CrossingEdge*, then all nodes before this node and the node itself in this walk are marked as "incoming" and the ID of the *CrossingEdge* is added. All nodes in predecessor walks of the current walk are also marked as "incoming" and the ID is added to each of these nodes.

Then, the node that is the head of this *CrossingEdge* is searched for. This node and all succeeding nodes in the walk containing this node, and all nodes in all successor walks are marked as "outgoing" and the ID of the *CrossingEdge* is added. Any node may be contained several times in a *DAG*, and it is crucial to not only identify the correct node, but also the correct instance of a node.

CrossingEdges define a sequential ordering. Thus, when a pair of nodes that may run in parallel is found, the IDs and "incoming" and "outgoing" marks are examined. If one of these nodes is "incoming" for an ID, for which the other node is "outgoing", they cannot run in parallel.

As mentioned in chapter 4.5.2, *ExpansionRegions* in parallel or streaming mode are transformed to *StructuredActivityNodes*, and the mode can be attached to this *StructuredActivityNode* for further analysis, if modelers additionally use assertions regarding duplicate object references in a collection. Analyzing such assertions is not implemented, as there is no standardized method for these assertions.

4.8 Discussion

As described in chapter 3.3, several approaches exist to analyze *UML Activities*. In contrast to those approaches, the solution provided in this thesis includes a formal and theoretic basis for analyzing *UML Activities*.

In the following, an assessment of the fulfillment of the requirements stated in chapter 3.3.12 is presented.

Requirement 1: only one of the approaches discussed considers *CallBehaviorActions* in the analysis [LTN19].The analysis is limited to synchronous execution of the called activity and does not consider cyclic calls of activities. In this regard, the analysis may not terminate in case of cycles. The solution advances the state of the art in that *CallbehaviorActions* with synchronous and asynchronous calls are considered, and in that the solution considers cyclic calls of activities, as shown in chapter 4.7. This ensures termination of the analysis without missing information with regard to the purpose of the analysis.

Requirement 2: two of the approaches discussed are not limited to a single *InitialNode* in the analysis [CLL07], [LTN19]. The solution presented in this thesis also considers multiple *InitialNodes*.

Requirement 3: as most of the approaches discussed, the solution presented in this thesis considers multiple *FlowFinalNodes* and *ActivityFinalNodes.*

Requirement 4: the challenge of cycle traversals is one of the key challenges to a complete analysis of *UML Activities* (see chapter 3.6). For the approaches discussed in chapter 3.3, there is either no information on handling cycles ([LWL08]), or a fixed limit for the traversals is set, e.g., a limit of one cycle execution ([Su15]), at most two cycle traversals ([SM08]), or traversing each edge once during a loop and twice overall ([CLL07]). The most flexible approach discussed uses a hard limit, which can be set by users ([Xu08]). Such limits are not sufficient and for each hard limit, UML Activities can be created, for which such algorithms produce incomplete results (see chapter 4.6.5). To solve this challenge, a dynamic limit based on the structure of the *UML Activity* has been introduced and discussed with regard to completeness of the results (chapter 4.6.5).

Requirement 5: most of the approaches discussed rely on pairs of *ForkNodes* and *JoinNodes* with regard to the analysis of *UML Activities* [Su08], [SZL09], [Su15], [SM08], [XLL05], [Xu08], [CLL07], [B011b], [KS09], [Ki07a], [VA14]. As such relations of *ForkNodes* and *JoinNodes* are not defined in the UML [OM17], this is a limitation. The solution presented in this thesis considers *ForkNodes* and *JoinNodes* independent of each other. Two of the approaches discussed also set no limitations on *ForkNodes* and *JoinNodes [LWL08], [LTN19]*.

Requirement 6: as shown in chapter 4.3, the UML offers a variety of elements for modelers. An approach for the analysis of UML Activities should therefore not be limited to a small subset of the elements, but consider most, if not all of the possibilities provided. Only two of the approaches discussed are able to consider more than the *basic elements* (see chapter 4.4.1). Xu et al. consider *ExpansionRegions* to a limited extent, *ExceptionHandlers* and *InterruptibleActivityRegions* [Xu08]. Lima et al. consider *CallBehaviorActions* to a limited extent, *SendSignalActions*, and *AcceptEventActions* [LTN19]. However, all of the other elements discussed in chapter 4.4.2 are not supported. The solution presented in this thesis instead advances the state of the art in that all elements of *UML Activities*, for which both semantics and a definition of its representation are given, are considered (see chapters 4.5, 4.6, and 4.7). Elements, which are not fully specified and standardized in the UML, cannot be considered due to a missing basis for identification and recognition. Examples of such elements are assertions for duplicate data references regarding *ExpansionRegions* in parallel and in streaming mode. Note that this may not lead to false positives, as the results of the analysis serve as an input for dynamic data race detection, and not to directly identifying data races. As the elements of *ExpansionRegions* are considered nevertheless, this may also not lead to false negatives in the sense of not identifying classes, which should be instrumented in dynamic data race detection.

The solution presented in this thesis tackles the challenges described in chapter 3.6 and closes the gaps left by the approaches discussed in chapter 3.3 and summarized in Table 1. The assessment shown in this table is extended with the solution presented in this thesis, resulting in Table 4.

	Require-ment 1	Require-ment 2	Require-ment 3	Require-ment 4	Require-ment 5	Require-ment 6
Approach	Multiple UML Activities	Multiple Initial Nodes	Multiple Final Nodes	Cycle Traversals	Forks and Joins	UML Elements
Sun	-	-	-	-	-	-
Sapna	-	-	-	-	-	-
Xu	-	-	+	0	-	0
Chandler	-	+	+	-	-	-
Lei	-	-	+	???	+	-
Boghdady	-	-	-	-	-	-
Kundu	-	-	+	-	-	-
Kim	-	-	-	-	-	-
Verma	-	-	+	-	-	-
Lima	0	+	+	-	+	0
Klaus	+	+	+	+	+	+

Table 4: Assessment of Existing Approaches and the Solution Presented in this Thesis

4.9 Summary

In this chapter, the analysis of *UML Activities* has been presented. As the analysis is based on a formal foundation using graph theory, the most important graph theoretic concepts have been presented.

The UML offers a variety of elements with regard to *UML Activities*. To analyze *UML Activities*, a three-step approach was introduced. First, the *UML Activities* are transformed, and the set of elements provided by the UML is reduced: some elements are preserved and other elements are transformed. The analysis is then conducted on the adapted *UML Activities*. However, not all elements of the UML can be transformed with preserving their complete semantics. Thus, they need to be considered in the post-analysis step.

This analysis is conducted by creating *DAGs* out of UML Activities. Tokens are then attached to the walks contained in each *DAG*. By analyzing these tokens, walks, which may be executed in parallel to other walks, are identified. Walks always represent a sequential execution of the *DANs* contained in these walks. Thus, by identifying such walks, nodes, which may be executed in parallel to other nodes, are identified, and the goal of the analysis is fulfilled.

The analysis of existing approaches in the analysis of UML Activities revealed three main gaps: analyzing sets of connected *UML Activities* (Requirement 1), handling *Cycle Traversals* within a *UML Activity* (Requirement 4), and support for a complete set of *UML Activity Elements* (Requirement 6). It has been shown how these gaps are closed by the solution presented in this thesis.

Handling cycle traversals within a *UML Activity* required introducing two criteria for traversals. On the one hand, a node may run in parallel to itself, which introduces one criterion regarding edge traversals. On the other hand, it is required that incoming edges of *JoinNodes* are considered, to enable the outgoing edge to be traversed. Both criteria are considered for edges in each *DAG*. *UML Activities* are explored in the breadth, and the *DAGs* are then extended in the depth. This ensures (1) that the necessary combinations of edge traversals to detect all nodes, which may be executed in parallel to themselves, are covered in a *DAG*, (2) that cycles can be left, when possible, and (3) that traversals are aborted when a certain amount of traversals is reached, thus preventing infinite *DAGs* and infinitely many *DAGs*. This amount is specific to each *UML Activity* and is calculated automatically during the analysis.

Finally, this chapter contained formal proofs with regard to the creation of *DAGs* and the analysis.

It has been proven that every execution instance of a *UML Activity* results in a unique *DAG.* Two parallel threads cannot be contained in one path in a *DAG*. A path in a *DAG* is always related to walks. If two walks w_1 and w_2 are executed in parallel then so are all pairs of instances of *DANs* from the *UML Activity* D (u_1, u_2) with $u_1 \in w_1$ and $u_1 \in w_2$.

The number of *DAGs* created with regard to *DecisionNodes* and cycles contained in a *UML Activity* has been defined and proven to be correct (without considering criteria for limiting edge traversals).

Finally, it has been proven how the tokens and the sequences in the tokens can be used to correctly and completely identify walks, which may be executed in parallel.

5 Analysis of Data Race Reports

In this chapter, the solution for the analysis of data race reports is presented. This solution is targeted at the second practical problem, as defined in chapter 1.2.

The solution reads in data race reports, and preprocesses the contents of those data race reports. The results are then visualized, enabling users to profit from the preprocessing and possible effects of this preprocessing on the efficiency of the analysis of data race reports.

5.1 Research Approach

The research approach consisted of several steps. As a first step, the characteristics of data race reports and the data races contained were analyzed.

Based on the insights gained, the next step was to examine how to utilize those characteristics with the goal to improve the efficiency in the analysis of data race reports. This utilization resulted in the preprocessing of data race reports.

Finally, a concept for the visualization of the preprocessed data race reports was developed and applied.

Following the research approach described above, the characteristics of data race reports are presented next (chapter 5.2).

The preprocessing of those data race reports is then introduced in chapter 5.3, followed by the visualization of the preprocessed data race reports (chapter 5.4).

This chapter closes with a summary and a comparison of the solution presented in this thesis with the approaches discussed in chapter 3.5, including an assessment with regard to the requirements for a solution defined in chapter 3.5.

5.2 Characteristics of Data Race Reports

Reconsidering the definition of a data race from chapter 2.2, originating from Netzer and Miller, a data race between two events $\boldsymbol{a}$ and $\boldsymbol{b}$ over a set of program executions $\boldsymbol{F}^6$ exists, if

1) A data conflict exists in a program execution P between a and b, and

2) There exists a program execution $P' \in F$, containing events a' and b', such that $a' \nrightarrow b'$ and $b' \nrightarrow a'$ [NM92].

Based on this definition, the following observations can be made with regard to characteristics of data races:

a) A data race is related to one shared memory location (due to the data conflict).

b) A data race is related to two accesses. Due to the ordering relations defined above, those accesses are concurrent.

c) An access can either be a read or a write access. In the following this is called *access type*. At least one access is of the access type write (due to the data conflict).

d) Since those accesses are found in a program execution, resulting from an execution of source code, the accesses are located in this source code. There is no restriction on where this access happens.

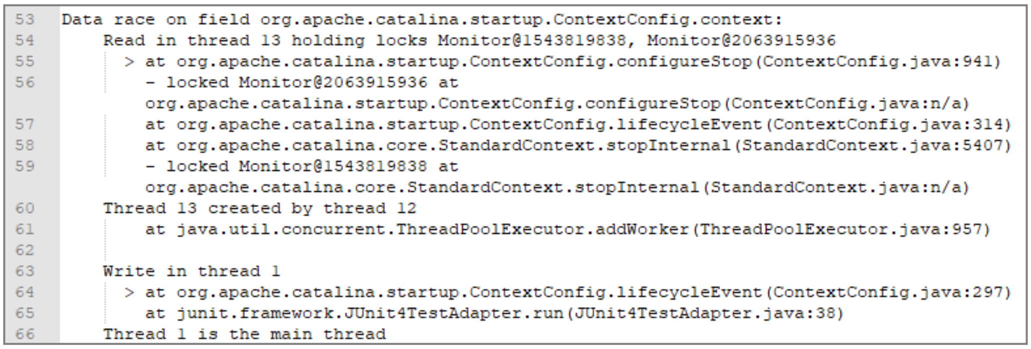
e) Furthermore, there is no restriction on the number of data races related to a shared memory location, or on the number of times an access may be related to a data race.

Analyzing data race reports, more information on characteristics can be identified. Most of those characteristics are general in the sense that many different data race reports stemming from different dynamic data race detection tools may allow to extract this information. However, the following insights are based on the data race reports produced by the publicly available dynamic data race detection tool `RV-Predict`, which has been used in the course of this thesis, and to which is referred in chapter 1.3, in the running example.

Any data race report may contain multiple entries, i.e., many different data races. The reasons for this are technical and not related to the characteristics

[6] Consider the relation of $\boldsymbol{F}$ and $\boldsymbol{P}$ defined in chapter 2.2: $\boldsymbol{F}$ contains the same events as a prefix of $\boldsymbol{P}$.

of data races. The following Figure 15 shows an exemplary entry for a detected data race.



```
53  Data race on field org.apache.catalina.startup.ContextConfig.context:
54      Read in thread 13 holding locks Monitor@1543819838, Monitor@2063915936
55        > at org.apache.catalina.startup.ContextConfig.configureStop(ContextConfig.java:941)
56          - locked Monitor@2063915936 at
            org.apache.catalina.startup.ContextConfig.configureStop(ContextConfig.java:n/a)
57          at org.apache.catalina.startup.ContextConfig.lifecycleEvent(ContextConfig.java:314)
58          at org.apache.catalina.core.StandardContext.stopInternal(StandardContext.java:5407)
59          - locked Monitor@1543819838 at
            org.apache.catalina.core.StandardContext.stopInternal(StandardContext.java:n/a)
60      Thread 13 created by thread 12
61          at java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:957)
62
63      Write in thread 1
64        > at org.apache.catalina.startup.ContextConfig.lifecycleEvent(ContextConfig.java:297)
65          at junit.framework.JUnit4TestAdapter.run(JUnit4TestAdapter.java:38)
66      Thread 1 is the main thread
```

Figure 15: Data Race Entry in a Data Race Report

The entry shows one data race (line 53) to a shared memory location `org.apache.catalina.startup.ContextConfig.context`. Two accesses can be seen, hierarchically subordinated below this data race. One access is listed in lines 54–61, and the second one in lines 63–66. Each of the accesses has an access type (lines 54 and 63). The source code location for each access is listed one line below (lines 55 and 64). The second listed access, e.g., is of type write, and located in the source code class `ContextConfig.java`, in source code line 297.

In addition to characteristics a) to e) described above, this entry allows to identify additional characteristics, related to those data race reports.

f) An access can be guarded by one or more locks (see chapter 2.3). This is shown in line 54.

g) In the data race report, an access has trace information, i.e., not only the source code location of the access itself is listed, but also source code locations related to the flow of execution (lines 55–61). In the following, this information is referred to as *call history*.

h) As can be seen in lines 55–61, the locks related to the access are acquired during the execution of the program, and at different locations than the access itself (lines 56 and 59).

i) Consistent to characteristic b), each access is executed by a different thread. The threads of the accesses are listed in lines 54 and 63.

j) Each thread has an origin. Such origins can be seen in lines 60 and 66. This can be either a source code location or the current thread is the main thread, i.e., it is created at the start of the system under test.

Summarized, the characteristics of data races contained in data race reports are shown in the following Figure 16.

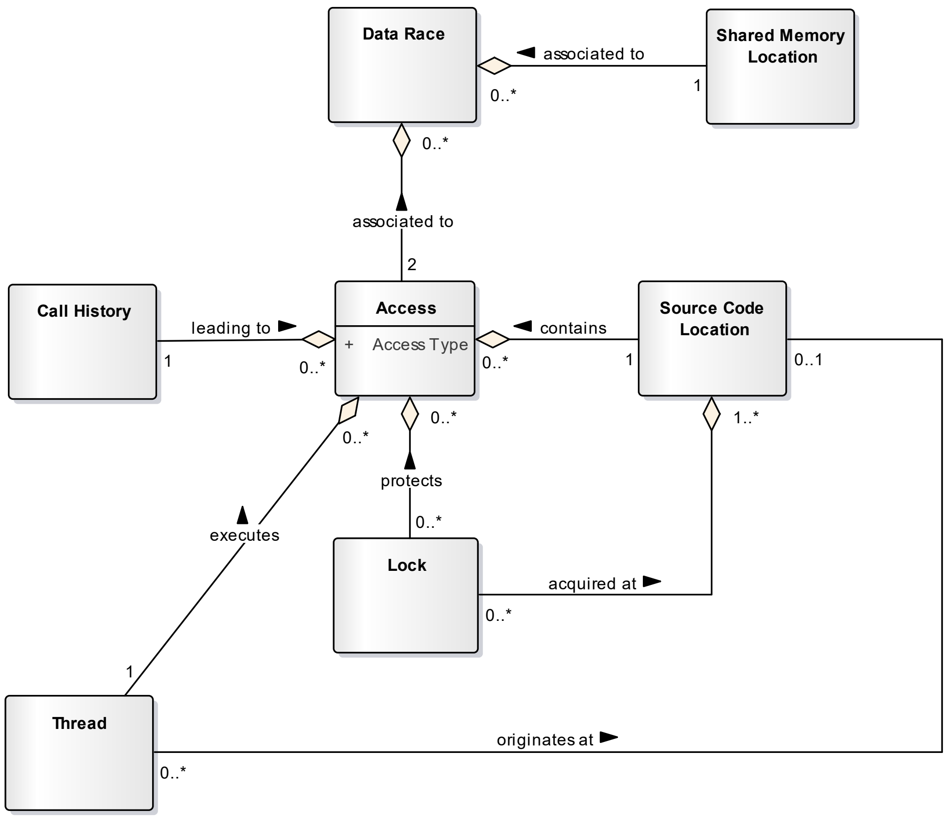


Figure 16: Characteristics of Data Races in Data Race Reports

The constraint that at least one of the accesses of a data race is a write access is not shown in this figure. The reason is that if this constraint is not fulfilled, then there is no data race, and thus the access is not contained in a data race report (assuming that no pairs of read-only accesses are contained in that data race report, as such pairs cannot form a data race).

5.3 Preprocessing Data Race Reports

Based on the relations defined in Figure 16, the central element chosen for the preprocessing was not a data race, but an access, as each element can be associated to a specific access. This is in line with the idea presented in [Ko15].

Furthermore, as there can be multiple data races with regard to one variable, it is not enough to focus the analysis of data race reports on a specific data race.

Instead, all accesses to that variable need to be considered for eliminating issues in the source code. The following Figure 17 shows an example, illustrated with results of the dynamic data race detection using `RV-Predict`.

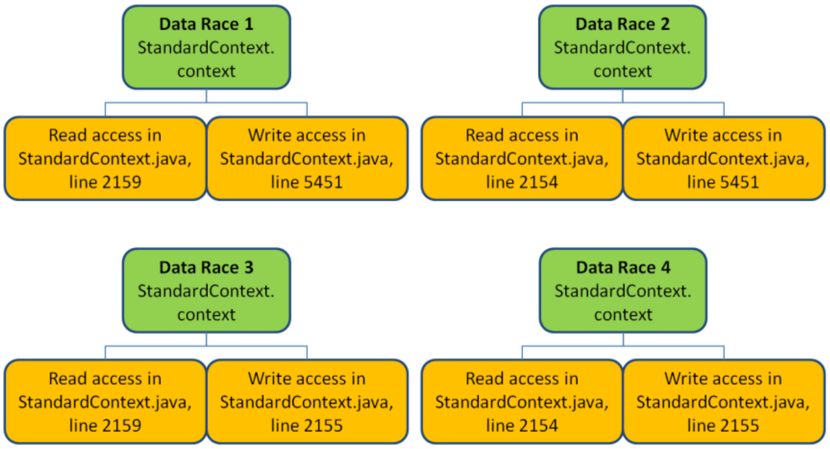


Figure 17: Exemplary Contents of a Data Race Report (Pictorial Representation)

This figure shows four data races, which are related to a total of four source code locations, all in source code class `StandardContext.java`. All these data races are related to the same variable `StandardContext.context`. Eliminating only one of those data races does not necessarily reduce the amount of source code locations related to a data race.

Assuming that Data Race 1 is eliminated by inserting locks protecting the accesses in line `2159` and in line `5451`, then those source code locations might still be related to data races (this depends on the exact mechanism used for eliminating the data race). The access in line `2159` is part of Data Race 3, and the access in line `5451` is part of Data Race 2. Further assuming that afterwards, Data Race 2 is eliminated by inserting locks then the access in line `2154` might still be related to a data race, as it is part of Data Race 4. Finally, if afterwards Data Race 3 is eliminated by inserting locks protecting the accesses in line `2159` and in line `2155`, Data Race 4 might still be present, although all the accesses contained in this data race already have been treated in the context of other data races. This holds, if the locks inserted for eliminating one data race are not consistent with the locks inserted for eliminating other data races.

Besides the need to analyze all data races with such an approach, the extensive usage of locks may lead to introducing deadlocks to the system.

When instead focusing on the distinct accesses related to a data race, four different source code locations can be identified, as shown in Figure 18. Due to this, the effort necessary may be reduced, since less source code locations are to be examined, as two of the source code locations were examined twice in the former example. In addition, the attention may be raised to take care of these accesses in a consistent way.

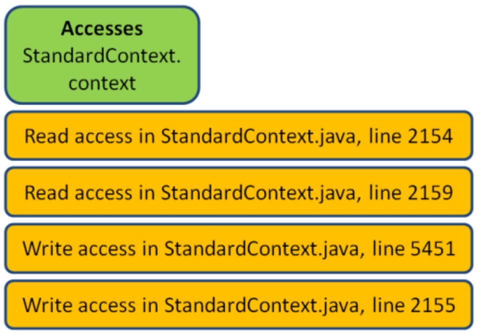

Accesses StandardContext. context
Read access in StandardContext.java, line 2154
Read access in StandardContext.java, line 2159
Write access in StandardContext.java, line 5451
Write access in StandardContext.java, line 2155

Figure 18: Exemplary Contents of a Data Race Report, Focused on Distinct Accesses (Pictorial Representation)

Thus, when focusing on accesses instead of data races as a whole, a view on issues in the source code may be induced. In addition, the example above has shown that such data race reports may contain multiple entries of the same access. Whenever there are at least two accesses related to the same variable, with the same access type, at the same location (source code class and line), these are called *duplicates*.

Any data race report may also contain duplicate data races, which can be broken down into sets of duplicate accesses. In addition, as a specific source code location may be executed by multiple threads, a single data race may also contain the same access twice.

Note that the definition of duplicates is only related to the access type, the variable, and the location. As such, these duplicate accesses may still differ in terms of the call history, and thus in terms of the locks acquired up to the specific access. As such, duplicates are identified during the preprocessing, but may not be deleted.

This information is used for preprocessing data race reports. The tool for preprocessing and visualizing data is written in C#. Users can import sets of data race reports in textual form, i.e., a root folder is selected, and all files and subfolders are automatically imported.

During the import, all the entries of the data race reports are read in and stored in a data structure. The import process relies on the syntax of data race reports as shown in Figure 15. The data structure is created based on the analysis of the characteristics of data races and data race reports, and follows Figure 16.

By analyzing all accesses, duplicate entries are identified and a flag is set. In addition, statistical information is processed.

5.4 Visualizing Preprocessed Data Reports

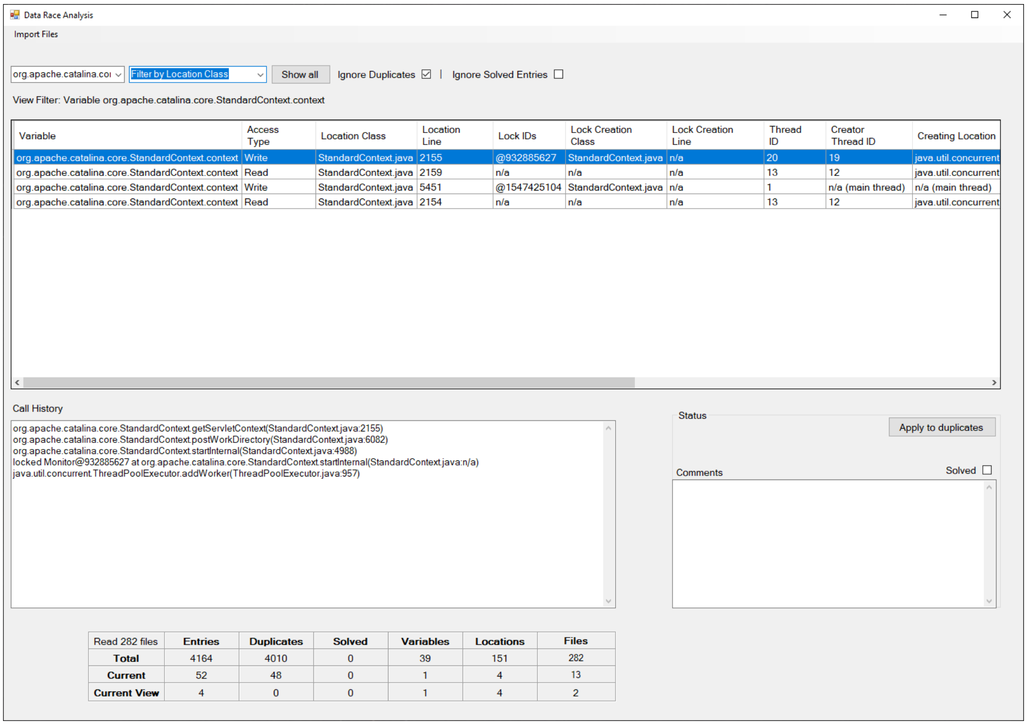
The information is then processed and provided to users in a tabular form. Each line in the table shows one of the accesses found. The columns show, among others, the variable, the source code class and line, the access type, the IDs of the locks held, the thread ID, and the origin of the thread. Below this table, the trace information of an access is provided in a separate field. Each line additionally shows if an access is contained multiple times, i.e., duplicates are marked in a separate column. The entries may be sorted alphabetically or by numbers in each column.

Users have the option to focus on one of the variables prone to data races, to focus on one the affected source code classes, or see all information. In addition, duplicates may be hidden. The tool also provides an option to add a status in free text to an access or to an access and all duplicates at once, and to mark an access or an access and all duplicates as solved. Such solved entries can also be hidden. However, the tool is still a prototype and there is currently no option to save and reload these results.

Finally, statistical information shows the number of files read, the number of accesses, and the number of different variables and source code locations affected, differentiated as overall numbers, numbers according to a filtered view, and according to a view with hidden entries. Figure 19 shows a screenshot of the tool.

The statistical information reveals that in summary 4,164 entries have been read in by the tool. As each data race is related to exactly two accesses, 2,082 data races have been processed. The current table contents, defined by setting the focus to all, a specific variable, or a specific source code class, in this example set to the variable `StandardContext.context`, encompass 52 entries (shown in the statistical information as "Current"). The current view, defined by selecting whether or not to ignore duplicate entries or solved entries, is set to ignoring duplicate entries, and contains four entries.

The entries shown in the screenshot are the same as presented in Figure 18. The current table contents relate to all data race reports with regard to the variable `StandardContext.context`, showing that the example shown in Figure 17 only represents a part of all data races with regard to that variable.



View Filter: Variable org.apache.catalina.core.StandardContext.context

Variable	Access Type	Location Class	Location Line	Lock IDs	Lock Creation Class	Lock Creation Line	Thread ID	Creator Thread ID	Creating Location
org.apache.catalina.core.StandardContext.context	Write	StandardContext.java	2155	@932885627	StandardContext.java	n/a	20	19	java.util.concurrent
org.apache.catalina.core.StandardContext.context	Read	StandardContext.java	2159	n/a	n/a	n/a	13	12	java.util.concurrent
org.apache.catalina.core.StandardContext.context	Write	StandardContext.java	5451	@1547425104	StandardContext.java	n/a	1	n/a (main thread)	n/a (main thread)
org.apache.catalina.core.StandardContext.context	Read	StandardContext.java	2154	n/a	n/a	n/a	13	12	java.util.concurrent

Call History

```
org.apache.catalina.core.StandardContext.getServletContext(StandardContext.java:2155)
org.apache.catalina.core.StandardContext.postWorkDirectory(StandardContext.java:6082)
org.apache.catalina.core.StandardContext.startInternal(StandardContext.java:4988)
locked Monitor@932885627 at org.apache.catalina.core.StandardContext.startInternal(StandardContext.java:n/a)
java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:957)
```

Read 282 files	Entries	Duplicates	Solved	Variables	Locations	Files
Total	4164	4010	0	39	151	282
Current	52	48	0	1	4	13
Current View	4	0	0	1	4	2

Figure 19: Screenshot of the Tool for the Analysis of Reported Data Races

5.5 Summary

The solution for the analysis of data race reports differs from all the approaches discussed in chapter 3.5 in that it is the only solution focusing on a simplified presentation of accesses in data race reports without the intent to visualize information. The approaches presented by Pande [Pa19b], Trümper [Tr14] and Koutsopoulos et al. [Ko15] focus on visualization aspects, and only one of those approaches [Ko15] is targeted at data races. Walker et al. also considered a need for a succinct presentation of the results, but focus on static analysis, and not on concurrency [Wa20].

Thus, the analysis and the comparison of the solution presented in this thesis to approaches presented in the literature had to be reduced to a comparison between two approaches – the one presented in this thesis and the approach presented by Koutsopoulos et al. [Ko15].

Requirement 1 - Duplicates: both approaches consider duplicate entries. However, while in [Ko15], the information is lost, the solution presented in this thesis provides a choice on whether duplicates are shown or just ignored, i.e., hidden. In addition, the solution presented in this thesis includes information on the total number of entries, the number of duplicate entries and the number of distinct entries.

Requirement 2 – Focus on shared variables: both approaches enable to focus on accesses related to shared variables, but differ in the presentation of that information. The approach of Koutsopoulos et al. presents all shared variables in one view, and focuses on the files, within which accesses to those shared variables happen. The authors admit the complexity of such a view [Ko15]. The solution presented in this thesis instead shows all the shared variables in a simple list. When such a shared variable is selected, the focus is on accesses, not on files. This emphasizes the focus of the solution presented in this thesis, to provide guidance on the elimination of data races. In addition, the knowledge about how many accesses exist with regard to such a shared variable, is directly visible, while in the approach presented in [Ko15], all the file listings need to be inspected for obtaining this information.

Requirement 3 – Focus on source code classes: the solution presented in this thesis is the only approach providing a focus on source code classes, again providing directly accessible information on problematic accesses and on statistical information.

Requirement 4 – Status tracking: no information could be found with regard to the ability of any of the approaches discussed in chapter 3.5 with regard to the tracking of the status of assessing the entries. The solution presented in this thesis allows setting the status of an access to "solved" and hiding such solved entries. This status can automatically be transferred to all duplicate entries. However, at the time of writing this thesis, the solution was still a prototype, and the ability to save a set of entries together with the status for later continuation of the work was not implemented.

Requirement 5 – Information preservation: the solution presented in this thesis does not delete any part of the information processed. All processed information is preserved and can be seen by users. The approach of Koutsopoulos et al. instead deletes duplicates [Ko15], and thus does not preserve all available information.

In addition to the fulfillment of the requirements introduced in chapter 3.5, the solution presented in this thesis is unique in that it provides knowledge in the form of statistical information. Whether such knowledge is useful, and if the fulfillment of those requirements is in fact leading to a higher efficiency in the analysis of data race reports is not guaranteed and has to be assessed. A controlled experiment assessing such potential efficiency improvements is presented in chapter 6.5.

6 Examinations of Efficacy

In this chapter, different aspects of the solution presented in this thesis are discussed and analyzed. The purpose of this chapter is to examine the analysis of *UML Activities*, to assess the completeness of this solution in practice, and to study possible effects of the solution on the efficiency.

Thus, this chapter contains a description of the research approach used for the examination of efficacy (chapter 6.1), followed by an exemplary application of the solution to demonstrate its practical usage (chapter 6.2). Aligned on the steps of the overall process, depicted in Figure 4, different important aspects are then examined. These examinations relate to the completeness of the analysis of *UML Activities* (chapter 6.3), possible efficiency improvements in the dynamic data race detection (chapter 6.4), and possible efficiency improvements in the analysis of data race reports (chapter 6.5).

6.1 Research Approach

The basis for the research conducted and presented in this thesis was a thorough understanding of the current state of the practice and the existing practical problems, as stated in chapter 1.2. A study of the state of the art revealed gaps, which have led to defining scientific problems and related goals. For these goals, research objectives had been defined, and hypotheses have been stated, defining benefits that were intended to be achieved and defining how the fulfillment of these goals shall be assessed. Summarized, the practical and scientific problems, and the goals were defined in chapters 1.2 and 1.3 as shown in the following Figure 20.

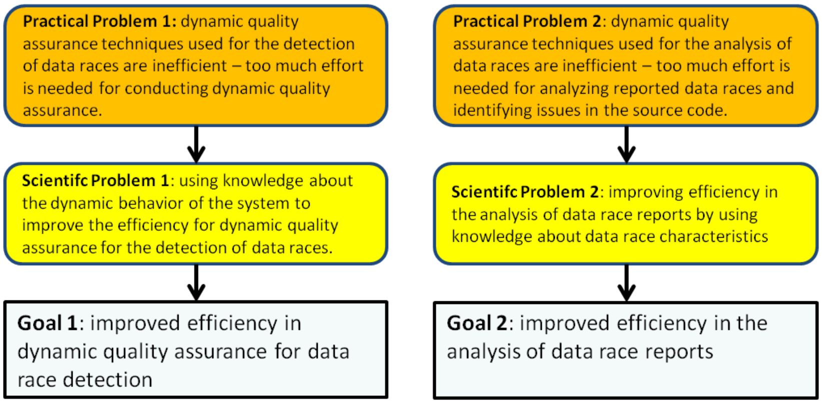

Practical Problem 1: dynamic quality assurance techniques used for the detection of data races are inefficient – too much effort is needed for conducting dynamic quality assurance.
Practical Problem 2: dynamic quality assurance techniques used for the analysis of data races are inefficient – too much effort is needed for analyzing reported data races and identifying issues in the source code.
Scientifc Problem 1: using knowledge about the dynamic behavior of the system to improve the efficiency for dynamic quality assurance for the detection of data races.
Scientifc Problem 2: improving efficiency in the analysis of data race reports by using knowledge about data race characteristics
Goal 1: improved efficiency in dynamic quality assurance for data race detection
Goal 2: improved efficiency in the analysis of data race reports

Figure 20: Problems and Goals Related to this Thesis

The overall goal of this thesis was the efficiency improvement in the quality assurance process for data races. Analogous to the quality assurance process being split up into finding data races and eliminating data races, two practical problems have been defined, both related to the efficiency in the respective field. The underlying scientific problems dealt with using knowledge about the systems under test and the detected data races to enable efficiency improvements.

Scientific problem 1 was covered by analyzing *UML Activities*, representing the dynamic behavior of a system, to gain the knowledge required to enable the required efficiency improvements in the dynamic data race detection. Scientific problem 2 was covered by analyzing data races and the reported data races, to provide guidance and knowledge about these detected data races, to enable efficiency improvements.

The goals were closely aligned with the problems defined, and were thus targeted at the intended efficiency improvements, both in the detection of data races and in the analysis of detected data races.

Based on these goals, hypotheses have been defined. As already stated in chapter 1.3, those hypotheses were:

Hypothesis 1 – Complete and Correct Analysis (Goal 1)

H_1. The analysis of the representation of the dynamic behavior of a system for parallel elements is complete and correct, .i.e., there are no false positives and no false negatives.

Hypothesis 2 – Efficiency Improvement in the Dynamic Detection of Data Races (Runtime Overhead) (Goal 1)

H_2. Using the focused approach for the instrumentation, the runtime overhead of test execution of a system under test is reduced compared to the runtime overhead using the same test execution technique on the same system under test without the focused approach. The reduction in the runtime overhead is at least inversely proportional to the amount of parallelism in the representation of that system under test.

Hypothesis 3 – Efficiency Improvement in Analyzing Data Races (Goal 2)

H_3. Using the log preprocessing, the effort for results analysis is at least 40% less with at least the same effectiveness compared to using the unprocessed log files.

Hypothesis 3 has been split up into:

Hypothesis 3.1 – Efficiency Improvement in Analyzing Data Race Reports for Data Races (Goal 2)

$H_{3.1}$. Using the log preprocessing, the number of analyzed reported data races in a given amount of time is at least 40% higher compared to using the unprocessed log files.

Hypothesis 3.2 – Efficiency Improvement in Analyzing Data Race Reports for Data Races affecting a Variable in the Source Code (Goal 2)

$H_{3.2}$. Using the log preprocessing, the number of analyzed reported data races related to specific variables in a given amount of time is at least 40% higher compared to using the unprocessed log files.

Hypothesis 3.3 – Efficiency Improvement in Analyzing Data Race Reports for Data Races affecting a Source Code Class (Goal 2)

$H_{3.3}$. Using the log preprocessing, the number of analyzed reported data races related to specific source code locations in a given amount of time is at least 40% higher compared to using the unprocessed log files.

A crucial step in the overall research was to decide, how to evaluate these hypotheses.

The first hypothesis was evaluated using a mixture of methods. Since this hypothesis targets an analysis, which is based on formal methods, the algorithms were formally proven to be correct and complete, as shown in chapter 4.6. However, this encompasses only the theoretic part. From a practical point of view, the challenge of limiting possibly infinite cycle traversals while retaining complete results is critical to the completeness of the results. Thus, it was decided to assess the completeness of the results using exemplary *UML Activities* of varying complexity (see chapter 6.3). It was not possible to conduct an assessment of *UML Activities* to an extent that would provide generalizable results, because the possibilities to combine elements of *UML Activities* are too high to provide a set of *UML Activities* adequately representing all possible combinations of elements and thus all possible *UML Activities*. Still, the solution to the analysis has been shown to be complete in theory, and regarding the practical application, the challenge can be reduced to the question whether the chosen stopping criteria allow obtaining complete results, or whether they need to be adjusted.

The second hypothesis was evaluated using an exemplary examination (see chapter 6.4). Such an exemplary examination does not allow drawing generalizable conclusions. Since applications can have different levels of parallelism, a representative experimental evaluation would need to include these different levels of parallelism. However, even when assuming that a representative set of applications could be composed, both test cases and models of the dynamic behavior would be required. In addition, single source code classes can be of varying complexity and thus may induce a varying overhead in the runtime when instrumenting such a class and analyzing the data collected for possible data races, and this needs to be considered when composing such a representative set of applications. As such, composing such a set of applications and conducting the necessary analyses requires too much time to be included in this thesis. This is therefore future work.

Hypothesis 3 targets the efficiency in the analysis of reported data races. Although theoretical discussions on the solution provided in this thesis indicate a possible improvement in the efficiency, this indication does not allow drawing any conclusions. Thus, it was decided to evaluate these hypotheses using a controlled experiment (see chapter 6.5).

6.2 Exemplary Application

This chapter demonstrates an exemplary application of the solution, highlighting how the different aspects of the solution support steps in the overall process of dynamic data race detection for data races.

Preconditions for the exemplary application of the solution presented in this thesis are the availability of the source code to test for data races, and of *UML Activities* representing the dynamic behavior of the system under test. As the solution presented in this thesis is not related to the dynamic data race detection itself, the availability of a tool for such dynamic data race detection together with test cases is a further requirement for the overall process.

The solution requires the input to be in a defined format. Although such a format may be changed, the current versions of the tool for the transformation of models requires the *UML Activities* to be modeled using `Enterprise Architect`, as mentioned in chapter 4. The tool for the analysis and processing of data race reports requires the reports in a format as created by `RV-Predict`, as mentioned in chapters 1.3 and 5.2.

The overall process is depicted in the following Figure 21. The *UML Activities* have been modeled using `Enterprise Architect`. These models are read in and analyzed using the algorithms discussed in chapter 4. The results are used for focusing the instrumentation. After the execution of the test cases, the log files are read in, and processed and presented to users as discussed in chapter 5. The preprocessed data race information can then be used for debugging the system under test.

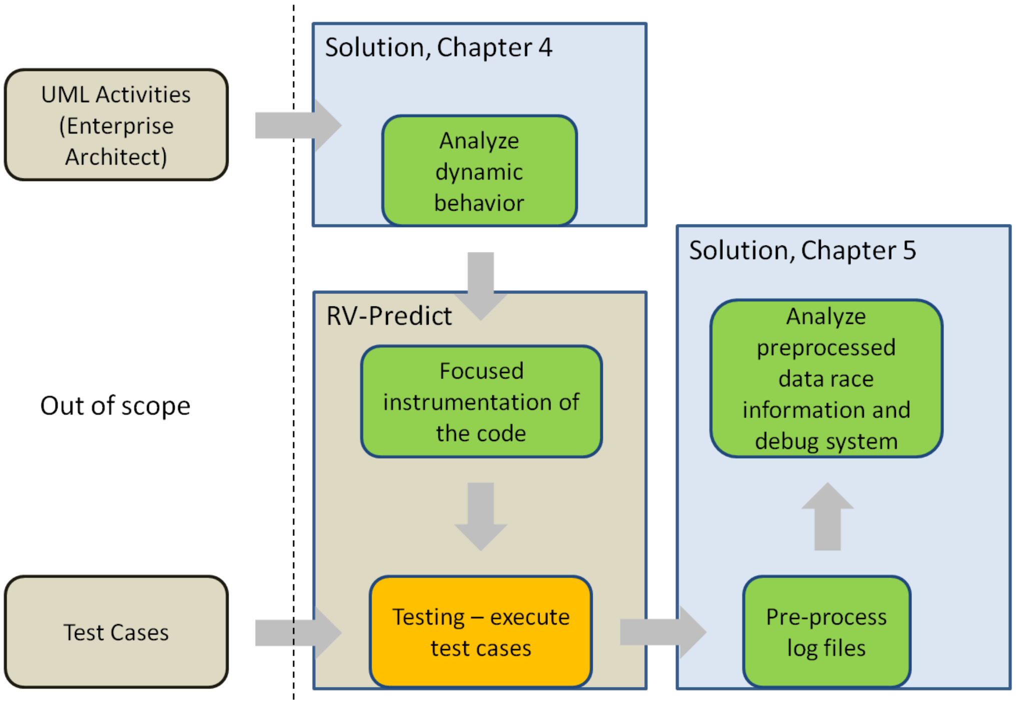

UML Activities (Enterprise Architect)
Out of scope
Test Cases
Solution, Chapter 4
Analyze dynamic behavior
RV-Predict
Focused instrumentation of the code
Testing – execute test cases
Solution, Chapter 5
Analyze preprocessed data race information and debug system
Pre-process log files

Figure 21: Simplified Process of Dynamic Quality Assurance for Data Races with Efficiency Improvements

The system under test used throughout this thesis, including this exemplary application, is `Apache Tomcat` in Version 8.0.26. This application is open source, and is shipped together with test cases. Those test cases are used for the dynamic data race detection. However, there were only rudimentary models of the dynamic behavior available. Thus, they had to be created manually.

This has been done by executing the shipped test cases in an instrumented test run. The logs, containing trace information, have been analyzed, and the call history has been analyzed in detail by inspecting the source code, to analyze how the control flow is exactly defined. As examples, it may not be clear from the log files, if a method call from one source code class to a different source code class is dependent on a decision (and a *DecisionNode* has to be set), or where such alternative flows are merged (and a *MergeNode* has to be set).

As the same behavior may be modeled in different ways, using different amounts of *UML Activities*, and different combinations of nodes, there is not a single solution. The completeness of the model can, however, be checked by comparing the results of the dynamic data race detection using complete instrumentation and using the focused instrumentation. As the tool used for the dynamic data race detection used predictive data race analysis, those results may, e.g., differ, when the focused instrumentation would not include a source code class containing an access related to a data race, or when the model would not correctly represent classes creating new threads, synchronization points, or decisions leading to alternative control flows. The model created contains 282 nodes and 314 edges. An excerpt of this model, meant for illustration purposes, and not for readability, can be seen in the following Figure 22.

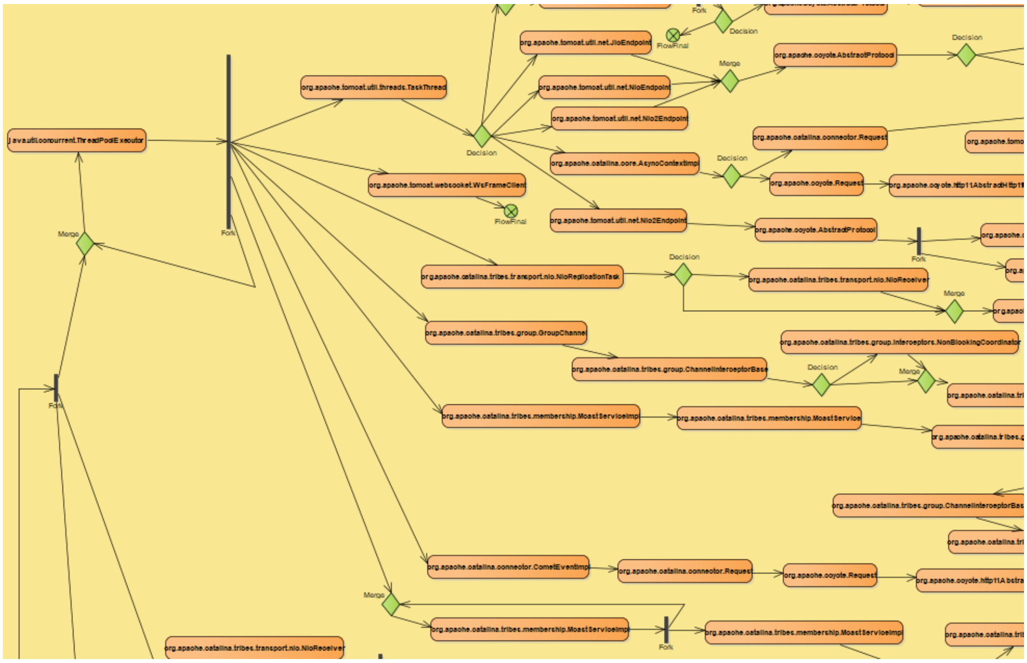
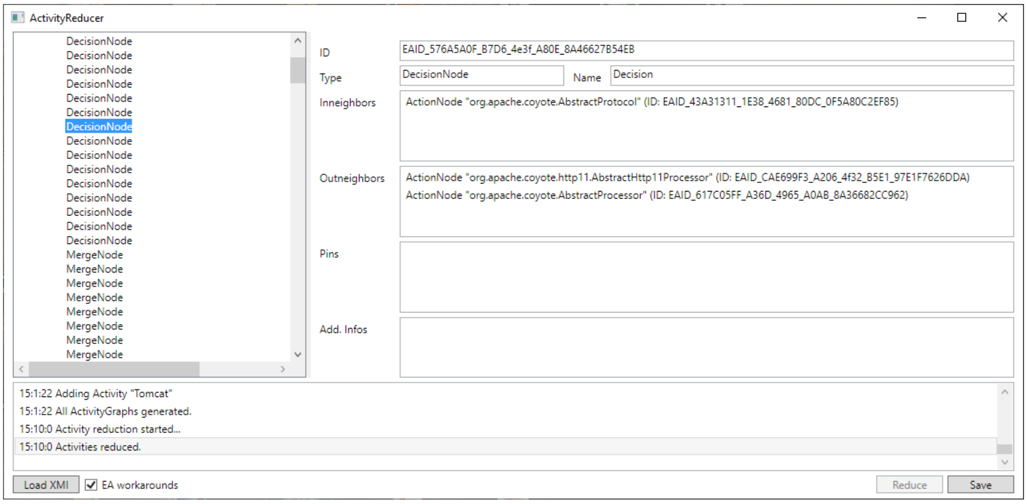

java.util.concurrent.ThreadPoolExecutor
Fork
Merge
Decision
FlowFinal
org.apache.tomcat.util.threads.TaskThread
org.apache.tomcat.util.net.JIoEndpoint
org.apache.tomcat.util.net.NioEndpoint
org.apache.tomcat.util.net.Nio2Endpoint
org.apache.catalina.core.AsyncContextImpl
org.apache.tomcat.websocket.WsFrameClient
org.apache.coyote.AbstractProtocol
org.apache.catalina.connector.Request
org.apache.coyote.Request
org.apache.catalina.tribes.transport.nio.NioReplicationTask
org.apache.catalina.tribes.transport.nio.NioReceiver
org.apache.catalina.tribes.group.GroupChannel
org.apache.catalina.tribes.group.ChannelInterceptorBase
org.apache.catalina.tribes.group.interceptors.NonBlockingCoordinator
org.apache.catalina.tribes.membership.McastServiceImpl
org.apache.catalina.tribes.membership.McastService
org.apache.catalina.connector.CometEventImpl

Figure 22: UML Activity Representing Apache Tomcat (Excerpt)

The model is exported as XMI file using standard functionality of `Enterprise Architect`. This file is read in by the tool `ActivityReducer`. This tool has been created as part of a master thesis, and is extensively described in [Zi16]. This tool conducts the transformation as defined in chapter 4.5 and saves the results as an XML file. A screenshot of this tool is shown in the following Figure 23.

This XML file then serves as input to the tool `ADDAG`. This tool creates DAGs in a first step and then analyzes those DAGs as defined in chapters 4.6 and 4.7. The emergency exit, defined in chapter 4.6.5, has been set to 4. The following Figure 24 shows a screenshot of the tool. The result of the analysis is a list of nodes. Those nodes represent source code classes, as defined in chapter 1.4, and can thus be directly used to focus the instrumentation of the dynamic data race detection.



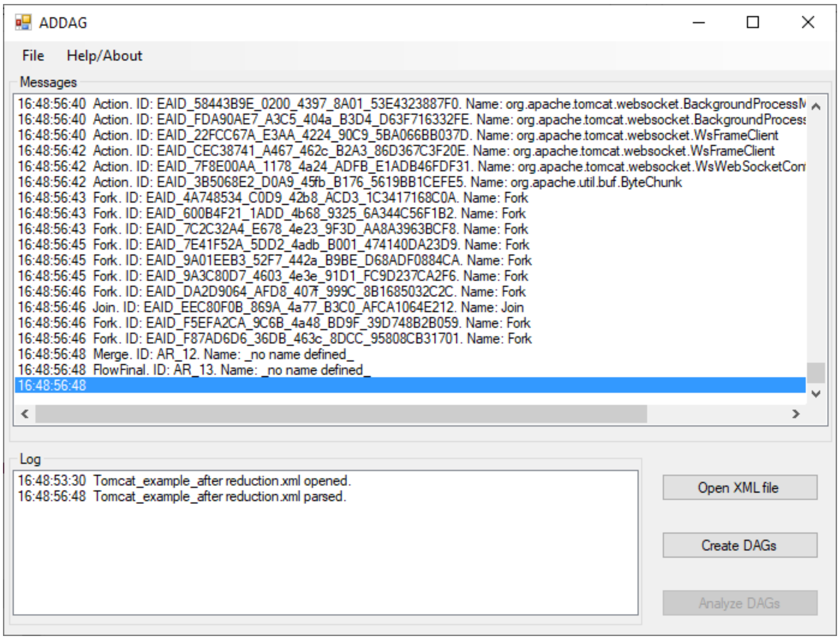
```
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
DecisionNode
MergeNode
MergeNode
MergeNode
MergeNode
MergeNode
MergeNode
MergeNode
MergeNode
```

ID: EAID_576A5A0F_B7D6_4e3f_A80E_8A46627B54EB
Type: DecisionNode Name: Decision
Inneighbors: ActionNode "org.apache.coyote.AbstractProtocol" (ID: EAID_43A31311_1E38_4681_80DC_0F5A80C2EF85)
Outneighbors: ActionNode "org.apache.coyote.http11.AbstractHttp11Processor" (ID: EAID_CAE699F3_A206_4f32_B5E1_97E1F7626DDA)
ActionNode "org.apache.coyote.AbstractProcessor" (ID: EAID_617C05FF_A36D_4965_A0AB_8A36682CC962)
Pins
Add. Infos

```
15:1:22 Adding Activity "Tomcat"
15:1:22 All ActivityGraphs generated.
15:10:0 Activity reduction started...
15:10:0 Activities reduced.
```

EA workarounds [✓]

Figure 23: Screenshot of ActivityReducer



Messages

```
16:48:56:40  Action. ID: EAID_58443B9E_0200_4397_8A01_53E4323887F0. Name: org.apache.tomcat.websocket.BackgroundProcessM
16:48:56:40  Action. ID: EAID_FDA90AE7_A3C5_404a_B3D4_D63F716332FE. Name: org.apache.tomcat.websocket.BackgroundProcess
16:48:56:40  Action. ID: EAID_22FCC67A_E3AA_4224_90C9_5BA066BB037D. Name: org.apache.tomcat.websocket.WsFrameClient
16:48:56:42  Action. ID: EAID_CEC38741_A467_462c_B2A3_86D367C3F20E. Name: org.apache.tomcat.websocket.WsFrameClient
16:48:56:42  Action. ID: EAID_7F8E00AA_1178_4a24_ADFB_E1ADB46FDF31. Name: org.apache.tomcat.websocket.WsWebSocketCon
16:48:56:42  Action. ID: EAID_3B5068E2_D0A9_45fb_B176_5619BB1CEFE5. Name: org.apache.util.buf.ByteChunk
16:48:56:43  Fork. ID: EAID_4A748534_C0D9_42b8_ACD3_1C3417168C0A. Name: Fork
16:48:56:43  Fork. ID: EAID_600B4F21_1ADD_4b68_9325_6A344C56F1B2. Name: Fork
16:48:56:43  Fork. ID: EAID_7C2C32A4_E678_4e23_9F3D_AA8A3963BCF8. Name: Fork
16:48:56:45  Fork. ID: EAID_7E41F52A_5DD2_4adb_B001_474140DA23D9. Name: Fork
16:48:56:45  Fork. ID: EAID_9A01EEB3_52F7_442a_B9BE_D68ADF0884CA. Name: Fork
16:48:56:45  Fork. ID: EAID_9A3C80D7_4603_4e3e_91D1_FC9D237CA2F6. Name: Fork
16:48:56:46  Fork. ID: EAID_DA2D9064_AFD8_407f_999C_8B1685032C2C. Name: Fork
16:48:56:46  Join. ID: EAID_EEC80F0B_869A_4a77_B3C0_AFCA1064E212. Name: Join
16:48:56:46  Fork. ID: EAID_F5EFA2CA_9C6B_4a48_BD9F_39D748B2B059. Name: Fork
16:48:56:46  Fork. ID: EAID_F87AD6D6_36DB_463c_8DCC_95808CB31701. Name: Fork
16:48:56:48  Merge. ID: AR_12. Name: _no name defined_
16:48:56:48  FlowFinal. ID: AR_13. Name: _no name defined_
16:48:56:48
```

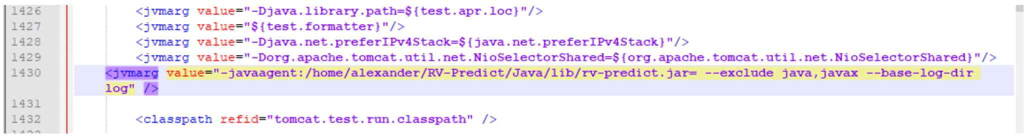
Log

```
16:48:53:30  Tomcat_example_after reduction.xml opened.
16:48:56:48  Tomcat_example_after reduction.xml parsed.
```

Figure 24: Screenshot of ADDAG

As mentioned, `Apache Tomcat` is shipped with test cases. These test cases are executed with `JUnit`. The source code can be used to build the executable file using `Apache Ant`. To instead execute the test cases, the command `ant test` is used. The test cases to execute, along with the tool to execute, are configured in the file `build.xml`, which is shipped together with the source code, and which is used to define different properties during the build process, which can be set without changing the source code.

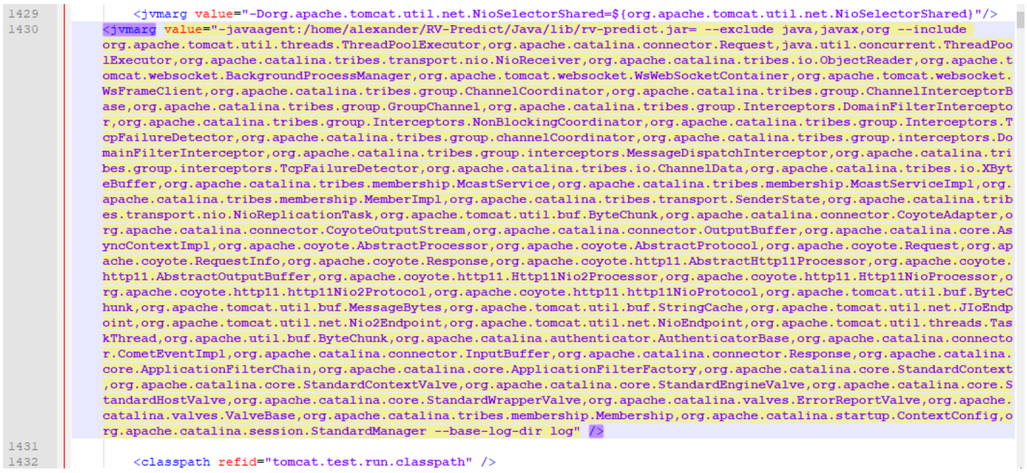
In this file, the properties for `JUnit` are set, and the dynamic data race detector can be inserted, as shown in Figure 25, line **1430**. A `java agent` is a jar file using the instrumentation API provided by the `Java Virtual Machine`. The dynamic data race detector `RV-Predict` is provided as such an agent. This allows executing the test cases for the dynamic data race detection.



```
1426        <jvmarg value="-Djava.library.path=${test.apr.loc}"/>
1427        <jvmarg value="${test.formatter}"/>
1428        <jvmarg value="-Djava.net.preferIPv4Stack=${java.net.preferIPv4Stack}"/>
1429        <jvmarg value="-Dorg.apache.tomcat.util.net.NioSelectorShared=${org.apache.tomcat.util.net.NioSelectorShared}"/>
1430    <jvmarg value="-javaagent:/home/alexander/RV-Predict/Java/lib/rv-predict.jar= --exclude java,javax --base-log-dir
        log" />
1431
1432        <classpath refid="tomcat.test.run.classpath" />
```

Figure 25: Screenshot of build.xml (Full Instrumentation)

As the dynamic data race detector can be configured using parameters in this file, the list of source code classes is added as parameter. Those parameters allow including and excluding source code classes. All source code classes obtained from the analysis of the behavioral model are included, and all others are excluded, as shown in Figure 26, line **1430**.

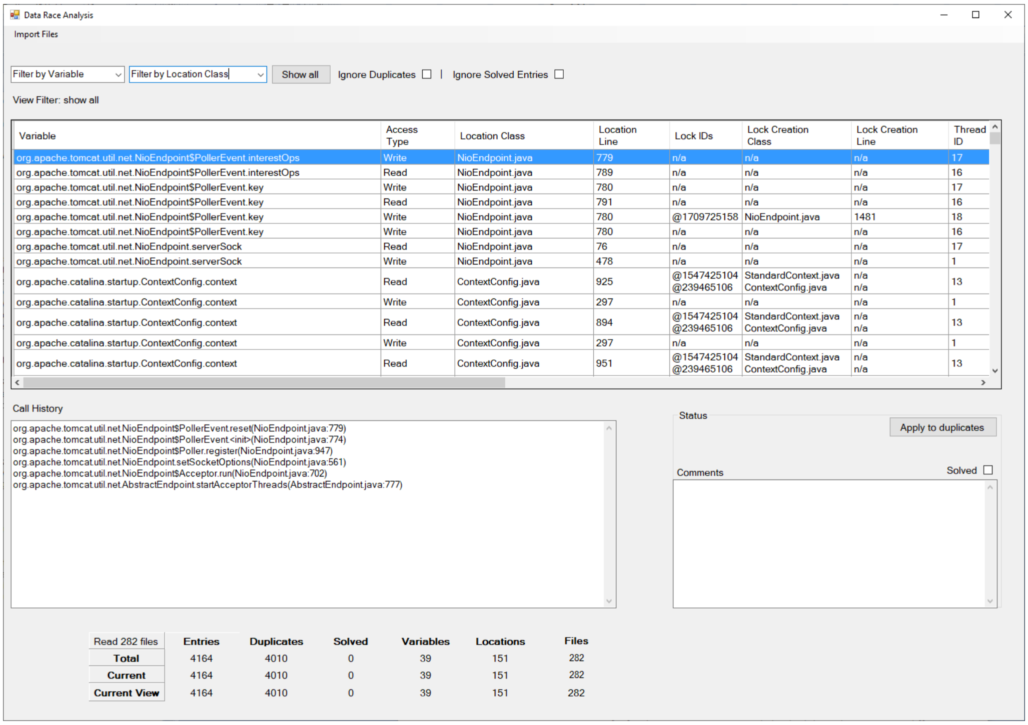


```
1429        <jvmarg value="-Dorg.apache.tomcat.util.net.NioSelectorShared=${org.apache.tomcat.util.net.NioSelectorShared}"/>
1430    <jvmarg value="-javaagent:/home/alexander/RV-Predict/Java/lib/rv-predict.jar= --exclude java,javax,org --include
        org.apache.tomcat.util.threads.ThreadPoolExecutor,org.apache.catalina.connector.Request,java.util.concurrent.ThreadPoo
        lExecutor,org.apache.catalina.tribes.transport.nio.NioReceiver,org.apache.catalina.tribes.io.ObjectReader,org.apache.t
        omcat.websocket.BackgroundProcessManager,org.apache.tomcat.websocket.WsWebSocketContainer,org.apache.tomcat.websocket.
        WsFrameClient,org.apache.catalina.tribes.group.ChannelCoordinator,org.apache.catalina.tribes.group.ChannelInterceptorB
        ase,org.apache.catalina.tribes.group.GroupChannel,org.apache.catalina.tribes.group.Interceptors.DomainFilterIntercepto
        r,org.apache.catalina.tribes.group.Interceptors.NonBlockingCoordinator,org.apache.catalina.tribes.group.Interceptors.T
        cpFailureDetector,org.apache.catalina.tribes.group.channelCoordinator,org.apache.catalina.tribes.group.interceptors.Do
        mainFilterInterceptor,org.apache.catalina.tribes.group.interceptors.MessageDispatchInterceptor,org.apache.catalina.tri
        bes.group.interceptors.TcpFailureDetector,org.apache.catalina.tribes.io.ChannelData,org.apache.catalina.tribes.io.XByt
        eBuffer,org.apache.catalina.tribes.membership.McastService,org.apache.catalina.tribes.membership.McastServiceImpl,org.
        apache.catalina.tribes.membership.MemberImpl,org.apache.catalina.tribes.transport.SenderState,org.apache.catalina.trib
        es.transport.nio.NioReplicationTask,org.apache.tomcat.util.buf.ByteChunk,org.apache.catalina.connector.CoyoteAdapter,o
        rg.apache.catalina.connector.CoyoteOutputStream,org.apache.catalina.connector.OutputBuffer,org.apache.catalina.core.As
        yncContextImpl,org.apache.coyote.AbstractProcessor,org.apache.coyote.AbstractProtocol,org.apache.coyote.Request,org.ap
        ache.coyote.RequestInfo,org.apache.coyote.Response,org.apache.coyote.http11.AbstractHttp11Processor,org.apache.coyote.
        http11.AbstractOutputBuffer,org.apache.coyote.http11.Http11Nio2Processor,org.apache.coyote.http11.Http11NioProcessor,o
        rg.apache.coyote.http11.http11Nio2Protocol,org.apache.coyote.http11.http11NioProtocol,org.apache.tomcat.util.buf.ByteC
        hunk,org.apache.tomcat.util.buf.MessageBytes,org.apache.tomcat.util.buf.StringCache,org.apache.tomcat.util.net.JIoEndp
        oint,org.apache.tomcat.util.net.Nio2Endpoint,org.apache.tomcat.util.net.NioEndpoint,org.apache.tomcat.util.threads.Tas
        kThread,org.apache.util.buf.ByteChunk,org.apache.catalina.authenticator.AuthenticatorBase,org.apache.catalina.connecto
        r.CometEventImpl,org.apache.catalina.connector.InputBuffer,org.apache.catalina.connector.Response,org.apache.catalina.
        core.ApplicationFilterChain,org.apache.catalina.core.ApplicationFilterFactory,org.apache.catalina.core.StandardContext
        ,org.apache.catalina.core.StandardContextValve,org.apache.catalina.core.StandardEngineValve,org.apache.catalina.core.S
        tandardHostValve,org.apache.catalina.core.StandardWrapperValve,org.apache.catalina.valves.ErrorReportValve,org.apache.
        catalina.valves.ValveBase,org.apache.catalina.tribes.membership.Membership,org.apache.catalina.startup.ContextConfig,o
        rg.apache.catalina.session.StandardManager --base-log-dir log" />
1431
1432        <classpath refid="tomcat.test.run.classpath" />
```

Figure 26: Screenshot of build.xml (Focused Instrumentation)

The test cases and the dynamic data race detection with focused instrumentation are then conducted using the command as shown above. This test case execution resulted in a total of 771 folders created by the dynamic data race detector, with each folder containing a `results.txt` file and a `debug.log` file.

These files can then be read in by the tool `DataRaceAnalyzer`. The data races contained in those report files are processed and presented to users as defined in chapter 5 and shown in Figure 27.



View Filter: show all

Variable	Access Type	Location Class	Location Line	Lock IDs	Lock Creation Class	Lock Creation Line	Thread ID
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.interestOps	Write	NioEndpoint.java	779	n/a	n/a	n/a	17
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.interestOps	Read	NioEndpoint.java	789	n/a	n/a	n/a	16
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.key	Write	NioEndpoint.java	780	n/a	n/a	n/a	17
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.key	Read	NioEndpoint.java	791	n/a	n/a	n/a	16
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.key	Write	NioEndpoint.java	780	@1709725158	NioEndpoint.java	1481	18
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.key	Write	NioEndpoint.java	780	n/a	n/a	n/a	16
org.apache.tomcat.util.net.NioEndpoint.serverSock	Read	NioEndpoint.java	76	n/a	n/a	n/a	17
org.apache.tomcat.util.net.NioEndpoint.serverSock	Write	NioEndpoint.java	478	n/a	n/a	n/a	1
org.apache.catalina.startup.ContextConfig.context	Read	ContextConfig.java	925	@1547425104 @239465106	StandardContext.java ContextConfig.java	n/a n/a	13
org.apache.catalina.startup.ContextConfig.context	Write	ContextConfig.java	297	n/a	n/a	n/a	1
org.apache.catalina.startup.ContextConfig.context	Read	ContextConfig.java	894	@1547425104 @239465106	StandardContext.java ContextConfig.java	n/a n/a	13
org.apache.catalina.startup.ContextConfig.context	Write	ContextConfig.java	297	n/a	n/a	n/a	1
org.apache.catalina.startup.ContextConfig.context	Read	ContextConfig.java	951	@1547425104 @239465106	StandardContext.java ContextConfig.java	n/a n/a	13

Call History

```
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.reset(NioEndpoint.java:779)
org.apache.tomcat.util.net.NioEndpoint$PollerEvent.<init>(NioEndpoint.java:774)
org.apache.tomcat.util.net.NioEndpoint$Poller.register(NioEndpoint.java:947)
org.apache.tomcat.util.net.NioEndpoint.setSocketOptions(NioEndpoint.java:561)
org.apache.tomcat.util.net.NioEndpoint$Acceptor.run(NioEndpoint.java:702)
org.apache.tomcat.util.net.AbstractEndpoint.startAcceptorThreads(AbstractEndpoint.java:777)
```

Status

Comments

Read 282 files	Entries	Duplicates	Solved	Variables	Locations	Files
Total	4164	4010	0	39	151	282
Current	4164	4010	0	39	151	282
Current View	4164	4010	0	39	151	282

Figure 27: Screenshot of DataRaceAnalyzer

6.3 DAGs as Result of Analyzing UML Activities

In this chapter, the numbers of *DAGs* created during the analysis of *UML Activities* and the results of the analysis itself are assessed using ten exemplary activities, representing different classes of complexity. Additionally, the completeness of the results with regard to the *EmergencyExit* and a possible *DAG** are discussed. This chapter thus targets hypothesis 1, as explained in chapter 6.1.

For the examinations, *UML Activities* have been created and manually analyzed. Afterwards, these activities have been processed with the tools for the transformation and analysis of *UML Activities*. This approach can only show fulfillment for a small sample of *UML Activities*. The external validity is thus limited. Additionally, manual analysis of *UML Activities* always includes the possibility of missing node instances, which may run in parallel to other node instances. However, for none of the activities analyzed, ADDAG could find additional results not found in manual analysis.

Each *UML Activity* was analyzed using different settings for the *EmergencyExit*. The *EmergencyExit* was set as defined in chapter 4.6.5, and was manually set for analyzing the effects on the analysis. Each *DAG* created for a *UML Activity* is identified by a number. For each analyzed activity, ADDAG was used to calculate:

- the number of *DAGs* created,
- the EmergencyExit,
- the number of *DAGs* necessary to obtain all findings,
- the first *DAG*, which is a *DAG**, and
- the necessary EmergencyExit number to obtain a *DAG**.

Note that the number of *DAGs* necessary to find a certain pair of nodes, and also to create a *DAG*, which is a *DAG**, differs depending on how the edges are traversed during the analysis. When encountering a *DecisionNode*, then for each outgoing edge, a separate *DAG* is used, as explained in chapter 4.6. During the analysis of *UML Activities* using the tool, the first edge, according to the order in the XML file, is always used for the existing *DAG*, and for other edges, new *DAGs* are created and used. If this order would be changed, the analysis would result in the same *DAGs*, but in a different order. The number of DAGs necessary to find a certain pair of nodes may then differ (as their order has been changed). This behavior could be observed during the experiments.

The *UML Activity* used to demonstrate the behavior concerning the number of *DAGs* in relation to the order of edges in the XML file is shown in Figure 14. For this analysis, the order of both outgoing edges of node "Decision 2" has been changed in the XML file.

Table 5 shows the results of the experiments. In each line of the table, the results for one *UML Activity* are shown. Each activity is identified by a number in the first column. In the following two columns, the results in terms of DAGs created for an *EmergencyExit* set to "2 * number of incoming join node edges", and the value for the maximum number of edge traversals until the *EmergencyExit* is reached for that specific *UML Activity* are shown. Since an experimentation regarding manually set limits for edge traversals has been conducted, exact numbers for the *EmergencyExit* value necessary to create a *DAG*, which is a *DAG**, can be provided. Finally, it is shown how many *DAGs* have been necessary to find all nodes, which may run in parallel to other nodes or to themselves using an *EmergencyExit* factor of 2, and how many *DAGs* were necessary until a *DAG* has been created, which is a *DAG**.

UML Activity	DAGs created	EmergencyExit	Necessary Emergency Exit to find a DAG*	DAGs to find all parallel nodes	First to be a DAG*
1	1	2	1	1	1
2	1	4	1	1	1
3	1	16	1	1	1
4	5	4	3	2	3
5	17	8	n/a	5	n/a
6	35	4	6	21	118
7	1,273	6	8	1,271	14,916
8a	8,460	4	6	1	153,859
8b	8,460	4	6	137	153,859
9	9,385	8	10	3,082	53,764

Table 5: Results of Experiments on DAGs and DAG*

Activities 8a and 8b represent the same *UML Activity*, but with a changed order of outgoing edges, as mentioned above. For both activities, the same numbers of *DAGs* are created. For activity 8a, all relevant nodes are found with the first *DAG*, while for finding the same nodes in 8b, 137 *DAGs* were necessary. For both activities, the first *DAG** was found in *DAG* 153,859. Although the order, in which edges are traversed differs for these two *UML Activities*, the amount of traversals of each edge, which are necessary for obtaining a *DAG**, is reached with the same *DAG*.

Activity 5 represents a not yet discussed situation. In this activity, a *DecisionNode* is contained, which can only be traversed once per *DAG*. This *UML Activity* additionally contains a cycle, and another *DecisionNode*, which can be traversed multiple times. Due to one *DecisionNode* only being traversable once per *DAG*, there is no *DAG**, as in each *DAG*, one edge is not traversed. Still, all nodes running in parallel to other nodes or to themselves could be identified. This shows that it cannot be a criterion to obtain a *DAG** to find all nodes, which may run in parallel to other nodes or to themselves. If it would, this *UML Activity* could not have been analyzed completely.

The results show that it is not necessary to create *DAGs* until a *DAG** is reached to find all nodes possibly running in parallel to other nodes or to themselves. With the *EmergencyExit*, all nodes, which may run in parallel to other nodes or to themselves, are obtained. This is especially demonstrated with *UML Activity* 8a, for which all results were already obtained with *DAG* 1. All 153,858 further *DAGs* until a *DAG** is created did not reveal any additional nodes possibly running in parallel to other nodes or to themselves.

The reasons for these results are as follows. For being a *DAG**, each edge has to be traversed twice in a *DAG*, where possible. As shown in chapter 4.6.5, it is not enough to obtain just any *DAG** to find all nodes possibly running in parallel to other nodes or to themselves, because in some cases, it is not enough to traverse each edge twice, but a certain combination of edge traversals is necessary to obtain complete results, and in other cases, it is not even possible to obtain a *DAG**.

It is not necessary to traverse each edge twice and to analyze all possible combinations of edge traversals to find a certain pair of nodes. It is sufficient to traverse some edges twice and to fulfill some combination of edge traversals to find certain nodes running in parallel to other nodes or to themselves. Due to the systematic exploration of edges and traversals in `ADDAG`, these different combinations of traversals are captured in different *DAGs*. These *DAGs* are systematically constructed while following edges and counting traversals of single edges. *UML Activities* are explored in the breadth and then the *DAGs* are extended in the depth. Because of this, there is no need to wait for a *DAG** to be created (if it will be created), where possible. Instead, since *DAGs* are constructed with different combinations of traversals and different edge traversals, those edges traversed at least twice mentioned above as "some" and the combinations of traversals mentioned above as "some" are traversed, before a *DAG**, if possible, is obtained. Because of this, there is no need to construct a *DAG** to find all nodes running in parallel to other nodes or to themselves. All edge traversals and all combinations needed are obtained with using the *EmergencyExit*, and before a *DAG** is created (if it will be created).

6.4 On the Efficiency Improvement of Focused Data Race Detection

To assess the solution with regard to hypothesis 2 (see chapter 6.1), an exemplary examination using `Apache Tomcat 8.0.26` was performed. There are two reasons for using this system in this specific version. There exists an extensive bug database for `Apache Tomcat` [No21b]. In addition, as an examination using `RV-Predict` has shown, version `8.0.26` is known to contain a number of data races [Da16]. Those data races have been fixed after they have been reported. The same tool is used for the examination presented in this thesis. This tool is configured for focusing on code parts identified using the approach to analyze *UML Activities*, as demonstrated in chapter 6.2. This allows to not only compare the runtime overhead (no instrumentation versus full instrumentation versus focused instrumentation), but also to check completeness of the findings, i.e., the data races identified.

Since *UML Activities* for `Apache Tomcat` were not available, they had to be manually created, as explained in chapter 6.2. Such an approach can only be used to analyze code parts actually executed when running the test cases. However, since the same test cases were used to perform the data race detection, possible incompleteness of the traces does not affect the results of the examination: completeness is given with regard to those code parts examined using the test cases.

Afterwards, the resulting *UML Activities* were analyzed using the approach presented in chapter 4. All steps of this examination were performed on a notebook with a Core i5-6200U processor and 8 GB RAM. During the fully automated analysis, 382 *DAGs* were created. The creation required 11.02 seconds. Analyzing those *DAGs* took additional 14.89 seconds.

A crucial step of the approach is the mapping of nodes in the *UML Activities* to source code parts. In the exemplary examination, the mapping has been created on basis of the traces used for creating the *UML Activities*. Because of this, the source code parts to focus the data race analysis on were directly identified by identifying nodes in the *UML Activities*. The existence of such a direct connection has been stated as one of the assumptions in chapter 1.4. Out of the 97 different source code classes referenced in the model, 65 different source code classes have been identified as possibly being involved in data races. The analysis thus resulted in a reduction of 33% of source code classes in need to be instrumented, as 67% of all source code classes contained in the model have been selected for focusing instrumentation on.

For the exemplary examination, three test runs have been performed. The runtime has been measured automatically within the context of the test execution. These test runs together with their runtime were:

1) Execution of the test cases without any instrumentation: 51 minutes.

2) Execution of the test cases with focused instrumentation (i.e., instrumentation limited to the code parts identified): 211 minutes and 52 seconds.

3) Execution of the test cases with full instrumentation (i.e., no focus for the instrumentation is set): 339 minutes and one second.

This means the runtime for the focused instrumentation was reduced by 127 minutes and nine seconds (i.e., 37.4%) compared to the fully instrumented test execution. Based on the execution of test cases without any instrumentation, the runtime for the focused instrumentation was 415% of the runtime without instrumentation. This means the runtime overhead for the focused instrumentation is 315%. The runtime for the full instrumentation was 664% of the runtime without instrumentation. The runtime overhead for the full instrumentation is 564%.

Comparing those numbers, it can be concluded that the focused instrumentation reduces the runtime by 37.4% and the runtime overhead by

44.1% (related to the runtime overhead of the fully instrumented execution of the test cases). Hypothesis 2 stated that the reduction in the runtime overhead is at least inversely proportional to the amount of parallelism in the representation of that system under test. As stated above, 67% of all different source code classes represented in the model are included in a parallel computation. As such, the inverse proportion is 33%. As the achieved reduction in the runtime overhead using focused instrumentation compared to full instrumentation is higher than that, the hypothesis can be confirmed in the specific case examined.

Note that the authors of [Da16] mention a runtime of roughly 260 minutes for the fully instrumented test run, and of 50 minutes for the test run without instrumentation. It is unclear why these differences appear. Still, using these numbers for a fully instrumented test execution, a reduction of the runtime of 48 minutes (i.e., 18.46%) and of the runtime overhead of 22.9% can be calculated. However, since the fully instrumented test execution was reported in [Da16] to be much faster than the fully instrumented test execution reported in this thesis, chances are that the test execution with focused instrumented also would be much faster. This would again result in higher reductions of the runtime and the runtime overhead.

Since runtime reduction can simply be achieved by ignoring code parts, and shall not result in false negatives (i.e., data races found with full instrumentation, but not with focused instrumentation), the data races found during both data race detection runs had to be compared. This comparison has shown that all data races found using full instrumentation are also found during focused instrumentation.

As this examination only covers one application, a generalization of the results is not possible. Since a runtime reduction can only be achieved by sorting out code parts irrelevant for data race detection, the runtime reduction achievable is closely related to the type of application in terms of the amount of parallelism inside an application. It can be expected that the runtime reduction for a highly parallel application is less than that for an application with only few code parts possibly running in parallel.

Finally, manually creating the *UML Activities* induces a possible bias, as those *UML Activities* were created with the intent to demonstrate the achievable reduction of the runtime overhead. This bias has been tackled by relating to the traces created during test case execution. In addition, comparing the number of data races reported when executing the data race detection with full instrumentation to the number of data races reported when using focused instrumentation gives insight into the completeness of the *UML Activities* in relation to the code parts executed.

6.5 Controlled Experiment on Analyzing Data Race Reports

To measure possible efficiency improvements in the identification of data races, more precise in the identification of problematic source code locations related to data races, a controlled experiment has been conducted. As understanding and fixing problems in the source code can be very time consuming, this important part in the overall process could not be included in the experimental evaluation. This chapter is thus focused on evaluating hypothesis 3 and its subordinated hypotheses 3.1, 3.2, and 3.3, as explained in chapter 6.1.

6.5.1 Improvement Possibilities

When thinking about improvement possibilities, the following cases to consider can be identified, with two variants for each:

1. A single data race
    a. Not a specific data race: one data race, regardless of the variable or the location (i.e., the variable and the location are not defined upfront).
    b. A specific data race: it is known that a data race exists for a certain pair of accesses or locations (either the locations or the variables are defined upfront), but more information is required (i.e., the reported data race). A pair of accesses with a data race is by definition always related to the same variable. Thus, this case can also be specified as: a single specific pair of accesses to the same variable.
2. A single pair of accesses in a location in the code with reported data races
    a. Not a specific location (the location is not defined): since by definition the accesses reported in a pair of accesses with a data race have to take place in some location in the code, this case can also be specified as: a single data race (not a specific data race)
    b. A specific location (the location is defined)
3. A single pair of accesses to a variable
    a. Not a specific variable (the variable is not defined)
    b. A specific variable (the variable is defined)
4. All accesses in a specific (defined) location in the code with reported data races

    a. Affecting the same variable (one of possibly several variables)

    b. Affecting different variables (in case of access to several variables in the same line of code or same operation). This means: all accesses to a specific location independent of the variable affected.

5. All accesses to the same variable (in different or the same location) with reported data races

    a. Not a specific variable (the variable is not defined upfront)

    b. A specific variable (the variable is defined upfront)

Case 1 number of represents a typical task during analyzing and debugging results of data race detection. Cases 2 and 3 both represent specific cases, which may be of interest in certain situations. This may seem rather unlikely, as this would in most of the cases represent an incomplete phase of analyzing and debugging data race reports. Nevertheless, these cases are included to obtain a more complete overview on improvement possibilities. Finally, cases 4 and 5 represent relevant tasks for the practice, as they are targeted at all reported data races with regard to a single variable or to a single location. These cases in sum also subsume the case "all reported data races", as data races always are related to a variable and to locations in the code: analyzing all accesses to all variables or in all locations results in having analyzed all reported data races.

Based on an analysis of those cases, three tasks to consider for the controlled experiment can be identified. The tasks to consider are:

1. Identifying distinct accesses related to a variable

2. Identifying distinct accesses within a source code class

3. Identifying distinct accesses independent of a specific variable or source code class

For the experimental evaluation, there were no other tools to compare the solution presented in this thesis with, as explained in chapter 3.5. Due to this, a basic approach, i.e., using traditional office tools was selected. Thus, the experimental evaluation can only assess the assumption, that tool support focused on specific tasks can improve the efficiency in conducting those tasks in comparison to general-purpose tools. Still, such an assumption, as obvious as it may be, cannot be relied on and needs to be scientifically assessed.

For the tasks identified, it was theoretically analyzed, how many and which steps are needed to fulfill a task. Those steps were then transformed into a hypothetical required effort in terms of clicks. This analysis did only include the process steps of identifying problematic accesses, not the steps of analyzing the source code for reasons for issues and ways to fix possible problems. Since the information provided is the same in both approaches to compare, and the time needed for analyzing and fixing problems can be very high, too high for inclusion in the experimental evaluation, and since this time is strongly dependent on a specific situation, those steps were not included.

6.5.2 Goals, Questions and Metrics

For our experimental evaluation, the GQM approach was used. The main hypothesis is, as shown in chapters 1.3 and 6.1:

H3: using the log preprocessing, the effort for results analysis is at least 40% less with the same effectiveness compared to using the unprocessed log files.

This main hypothesis was then split up for the identified tasks, and measurement goals were formulated.

H3.1: using the log preprocessing, the number of analyzed data races in a given amount of time is at least 40% higher compared to using the unprocessed log files.

H3.2: using the log preprocessing, the number of analyzed reported data races related to specific variables in a given amount of time is at least 40% higher compared to using the unprocessed log files.

H3.3: using the log preprocessing, the number of analyzed reported data races related to specific source code locations in a given amount of time is at least 40% higher compared to using the unprocessed log files.

MG3.1: Analyze the analysis of reported data races using the approach for preprocessing of log files and the analysis of the same reported data races without any preprocessing for the purpose of comparison with regard to the number of analyzed accesses in a given amount of time from the viewpoint of software developers in the context of a controlled experiment.

MG3.2: Analyze the analysis of reported data races using the approach for preprocessing of log files and the analysis of the same reported data races without any preprocessing for the purpose of comparison with regard to the number of analyzed accesses related to specific variables in a given amount of time from the viewpoint of software developers in the context of a controlled experiment.

MG3.3: Analyze the analysis of reported data races using the approach for preprocessing of log files and the analysis of the same reported data races without any preprocessing for the purpose of comparison with regard to the number of analyzed accesses related to a specific source code location in a given amount of time from the viewpoint of software developers in the context of a controlled experiment.

These measurement goals lead to the following research questions:

RQ3.1: does the proposed preprocessing lead to efficiency improvement with the same effectiveness for analysis of reported data races compared to analyzing the same reported data races without preprocessing?

RQ3.2: does the proposed preprocessing lead to efficiency improvement with the same effectiveness for analysis of reported data races related to specific variables compared to analyzing reported data races related to the same variables without preprocessing?

RQ3.3: does the proposed preprocessing lead to efficiency improvement with the same effectiveness for analysis of reported data races related to specific source code locations compared to analyzing reported data races related to the same source code locations without preprocessing?

Finally, the null hypotheses and alternative hypotheses can be formulated as follows, considering the relation of data races and accesses:

H0,3.1: the proposed preprocessing does not lead to an improvement of the number of analyzed reported accesses of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

H1,3.1: the proposed preprocessing leads to an improvement of the number of analyzed reported accesses of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

H0,3.2: the proposed preprocessing does not lead to an improvement of the number of analyzed accesses directly affecting a specific variable of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

H1,3.2: the proposed preprocessing leads to an improvement of the number of analyzed accesses directly affecting a specific variable of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

H0,3.3: the proposed preprocessing does not lead to an improvement of the number of analyzed accesses related to a specific location in the source code of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

H1,3.3: the proposed preprocessing leads to an improvement of the number of analyzed accesses related to a specific location in the source code of at least 40% with at least the same effectiveness in a given amount of time compared to analyzing the same reported accesses without preprocessing.

To be able to answer the research questions and to evaluate the hypotheses, several measures need to be taken during the controlled experiment:

- M1: Number of analyzed reported accesses
- M2: Number of analyzed reported accesses related to specific variables
- M3: Number of analyzed reported accesses related to specific source code locations
- M4: Time needed for analysis of the reported accesses

Derived from these measures:

- M5: Average time needed for analysis of a reported access
- M6: Average time needed for analysis of a reported access related to a specific variable
- M7: Average time needed for analysis of a reported access related to a specific source code location

Analysis in this context means identifying the location of a reported access. Specific in this context means that in the controlled experiment, a variable or a source code location (source code class) is predefined and only these predefined items are of interest. The term effectiveness refers to an amount (of analyzed accesses or data races), while the term efficiency refers to an amount in relation to the time needed.

6.5.3 Experimental Design

In the controlled experiment, one factor was examined: the technique executed for analyzing reported data races. There were two factor alternatives, namely (1) analyzing the unprocessed reported data races, i.e., the log files, using office tools, and (2) analyzing the preprocessed reported data races using the newly created tool.

The experimental unit is the analysis phase in the software engineering process, after execution of test cases has taken place. Experimental subjects are the individuals applying the techniques, i.e., personnel in software development.

Similar to the measures defined above, the response variables are defined:

- Effort for analysis of reported accesses: the time needed for analysis of the reported accesses is measured. This measure is observed as a number (ratio scale).
- Number of analyzed reported accesses (ratio scale)
- Number of analyzed reported accesses related to a specific variable (ratio scale)
- Number of analyzed reported accesses related to a specific source code location (ratio scale)

Derived from those measures are:

- Average time needed to analyze one access: this measure is calculated from the above measures and is calculated three times (for the number of analyzed reported accesses and the time needed, for the number of analyzed reported accesses related to a specific variable and the time needed, and for the number of analyzed reported accesses to a specific source code location and the time needed).
- Average number of analyzed accesses per time unit, i.e., minutes: this measure is also calculated from the above measures and is again calculated three times (for the number of analyzed reported accesses and the time needed, for the number of analyzed reported accesses related to a specific variable and the time needed, and for the number of analyzed reported accesses to a specific source code location and the time needed).

Both of these two derived measures are related to the same data and these derived measures can replace each other. As an example, six analyzed accesses within three minutes would mean an average time of 30 seconds per finding, and an analysis rate of two findings per minute. Therefore, both derived measures have been calculated, but only the analysis rate has been used for the statistical analysis.

There are several parameters to consider, and not all of them are under control.

- Complexity of the software under analysis: under control, as the same software is used for both alternatives.
- Programming language used: under control, as the same software, and thus the same programming language, is used for both alternatives.
- Process for the analysis: under control, as for both factors, the tasks to do are defined, and explained to the experimental subjects.
- Number of reported data races: under control, as the reported data races, i.e., the input for the controlled experiment, are chosen according to several characteristics, as explained below.
- Difficulty of the analysis of reported data races selected by experimental subjects: not under control, and eliminated by using a substitute for this task (main characteristics of an access are noted down instead of analyzing a reported access in detail)
- Familiarity and experience of subjects with debugging, code reading, reading bug reports: not under control, and considered as blocking variable. Hence, a block design (within-subject design) with a random assignment of factor alternatives and experimental subjects is used.

The controlled experiment is set up as a block design, with two groups and two rounds, as depicted in Table 6. In each round, three tasks are to be conducted, in accordance with the tasks identified. The tasks are:

1. Find all distinct accesses related to the variable X
2. Find all distinct accesses related to the source code class Y
3. Find as many distinct accesses as possible

Note that "X" and "Y" in the text above are placeholders. The tasks for both groups contain one variable and one source code class each, with those entries being different for each group and each round. The term distinct refers to the characteristics of data races in terms of the variable affected, the access type, the source code class, and the source code line. If for two accesses at least one of these characteristics is different, then those accesses are considered as being distinct.

	Group 1	Group 2
Round 1	Factor alternative 1 Data race set 1	Factor alternative 2 Data race set 1
Round 2	Factor alternative 2 Data race set 2	Factor alternative 1 Data race set 2

Table 6: Groups and Rounds

Due to the controlled experiment consisting of two rounds, a possible learning effect of the experimental subjects needed to be considered as thread to validity. If experimental subjects analyze the same data races or accesses twice, a learning effect may happen. This learning effect may substantially affect the results in terms of effectiveness and efficiency. To avoid this thread to validity while not injecting another thread to validity, the data races or accesses need to be different for the two experimental runs. Therefore, two different sets of reported data races were prepared. Each set is used in one of the two rounds, and for each set, one variable and one source code class is selected for usage in the tasks, as described above.

Both sets used are based on the same set of reported data races, which were gained by conducting dynamic data race detection on an application using test cases shipped with the source code of that application. To retain comparability when using two different sets of data races, those sets do not differ in terms of the following characteristics:

- Number of reported data races
- Number of duplicates
- Number of files with reported data races

Although the characteristics for the data races related to the source code class and to the variable were the same in each of the two sets, it was not possible to obtain sets with completely identical characteristics. Set 1 consisted of 36 files, which contained 838 accesses. Of those, 807 were duplicates. The data races contained were spread upon 12 different variables and 30 different locations (source code class and line). Set 2 consisted of 36 files, which contained 838 accesses, with 807 being duplicates. The data races were related to eleven different variables and 29 different locations. The relation of data races to duplicate data races for the two sub sets is the same as in the complete set.

For task 1 in set 1, three distinct accesses out of 86 accesses had to be identified, spread over 29 files (two files considering only distinct accesses).

For task 2 in set 1, five distinct accesses out of 202 accesses had to be identified, spread over 29 files (two files considering only distinct accesses). For task 3 in set 1, 31 distinct accesses out of all 838 accesses had to be identified, spread over 36 files (five considering only distinct accesses).

For task 1 in set 2, three distinct accesses out of 128 accesses had to be identified, spread over all 36 files (two files considering only distinct accesses). For task 2 in set 2, five distinct accesses out of 212 accesses had to be identified, spread over 29 files (two files considering only distinct accesses). For task 3 in set 2, 31 distinct accesses out of all 838 accesses had to be identified, spread over 36 files (five considering only distinct accesses).

The characteristics for both sets are similar in many cases, but not completely the same. Although, based on the design of the controlled experiment, no negative effects were expected on the results, it was planned at this stage to not only compare the two factor alternatives, but also to examine and compare the results for the two sets with regard to possible differences in the outcome.

In addition to the task lists (see Appendix A and B), a questionnaire (see Appendix C) has been created to obtain an insight into the perception of the participants concerning the tasks fulfilled. This questionnaire asked for the professional experience in the area of software engineering, and contained four statements for each round:

1. When thinking about the analysis of data race reports with the unprocessed log files...

   - ... I feel confidence in the completeness of the results
   - ... I feel confidence in the correctness of the results
   - ... the analysis was difficult for me
   - ... the analysis was exhausting for me

2. When thinking about the analysis of data race reports with the preprocessed data race results...

   - ... I feel confidence in the completeness of the results
   - ... I feel confidence in the correctness of the results
   - ... the analysis was difficult for me
   - ... the analysis was exhausting for me

These statements could be rated using a five point Likert scale, with the values "totally disagree" (1), "rather disagree" (2), "neither agree not disagree" (3), "rather agree" (4), and "totally agree" (5).

6.5.4 Conduct of the Controlled Experiment

The controlled experiment was conducted with professionals in the software engineering domain. As their availability was limited, the time was restricted to 90 minutes. This has been split up into an introduction into the topic (30 minutes), and two rounds of 30 minutes each. In both rounds, three tasks had to be conducted, which were limited to ten minutes each.

As the controlled experiment was conducted during the pandemic, it was not possible to have in-person meetings with the participants. Instead, individual video calls were used. Since there were no other possibilities to observe the participants, it was required that the call was not only a voice call, but always accompanied with live video.

Each participant received the two sets of data races, the tool to use, one of two task lists (one for each group), and the questionnaire. Each task list contained a short description of the scenario (i.e., the participant is part of software engineering, test case execution has taken place, and now, the findings shall be analyzed), and the tasks themselves together with instructions how to document their findings. The time needed for each task was measured, and the execution had to be stopped for each task after ten minutes. The time needed was then noted for each task and each round. After both rounds, the participants were asked to fill out the questionnaire. The filled-out task lists and questionnaires had to be sent back directly within the video conference.

6.5.5 Assessment of the Results of the Controlled Experiment

Due to the necessity of video calls, the controlled experiment was conducted over several weeks, with eight participants. As a first step of the assessment, all entries were manually checked for correctness. When, during software development, a wrong location would be identified as being part of a data race, the following review of the source code would reveal this issue, as the location would not match the characteristics of the reported accesses. This would result in additional work, i.e., the reported access would have to be examined again. Thus, it was decided to sort out wrong entries in the task lists. This had to be done for two participants, with two entries each. This situation occurred one time when using factor alternative 1, and one time when using factor alternative 2. The raw data of the controlled experiment, with the corrected number of entries, can be found in Table 7. For all the statistical calculations, an α of 0.05 was used.

	Participant ID	1	2	3	4	5	6	7	8
Unprocessed (Factor alternative 1)	Set	1	1	1	1	2	2	2	2
	Task 1 – Corrected #	3	3	2	2	2	3	3	2
	Task 1 – Time (minutes and seconds)	09:00	05:00	10:00	10:00	09:30	05:40	08:05	10:00
	Task 2 – Corrected #	5	5	5	4	5	5	5	4
	Task 2 – Time (minutes and seconds)	09:00	06:00	10:00	10:00	10:00	08:40	08:45	10:00
	Task 3 – Corrected #	17	20	14	12	11	11	25	9
	Task 3 – Time (minutes and seconds)	10:00	10:00	10:00	10:00	10:00	10:00	10:00	10:00
Processed (Factor alternative 2)	Set	2	2	2	2	1	1	1	1
	Task 1 – Corrected #	3	3	3	3	3	3	3	3
	Task 1 – Time (minutes and seconds)	03:00	01:05	00:45	01:40	01:30	00:50	02:30	03:25
	Task 2 – Corrected #	5	5	5	5	5	5	5	5
	Task 2 – Time (minutes and seconds)	02:00	00:50	01:00	01:45	01:30	01:10	01:32	03:30
	Task 3 – Corrected #	29	31	31	21	21	31	30	19
	Task 3 – Time (minutes and seconds)	10:00	06:30	07:55	10:00	10:00	07:30	10:00	10:00

Table 7: Raw Data of the Controlled Experiment

Based on those data, the time per finding and the findings per minute for each task and each participant were calculated. These numbers are shown in Table 8. Although these numbers are displayed with three decimal digits, for all statistical calculations except the statistical power, ten decimal digits had been used. For the statistical power, eight decimal digits have been considered.

	Participant ID	1	2	3	4	5	6	7	8
Unprocessed (Factor alternative 1)	Set	1	1	1	1	2	2	2	2
	Task 1 – Time per finding	03:00	01:40	05:00	05:00	04:45	01:53	02:41	05:00
	Task 1 – Findings per minute	0.333	0.600	0.200	0.200	0.211	0.529	0.371	0.200
	Task 2 – Time per finding	01:48	01:12	02:00	02:30	02:00	01:44	01:45	02:30
	Task 2 – Findings per minute	0.556	0.833	0.500	0.400	0.500	0.577	0.571	0.400
	Task 3 – Time per finding	00:35	00:30	00:42	00:50	00:54	00:54	00:24	01:06
	Task 3 – Findings per minute	1.700	2.000	1.400	1.200	1.100	1.100	2.500	0.900
Processed (Factor alternative 2)	Set	2	2	2	2	1	1	1	1
	Task 1 – Time per finding	01:00	00:21	00:15	00:33	00:30	00:16	00:50	01:08
	Task 1 – Findings per minute	1.000	2.769	4.000	1.800	2.000	3.600	1.200	0.878
	Task 2 – Time per finding	00:24	00:10	00:12	00:21	00:18	00:14	00:18	00:42
	Task 2 – Findings per minute	2.500	6.000	5.000	2.857	3.333	4.286	3.261	1.429
	Task 3 – Time per finding	00:20	00:12	00:15	00:28	00:28	00:14	00:20	00:31
	Task 3 – Findings per minute	2.900	4.769	3.916	2.100	2.100	3.100	3.000	1.900

Table 8: Rates Calculated from Raw Data

Then, the means, the variances and the standard deviations for each task and each factor alternative were calculated. For each task, a box plot, a p-p plot and a histogram were created to visually check the data and the distribution of values. The box plots are shown in Figure 28.

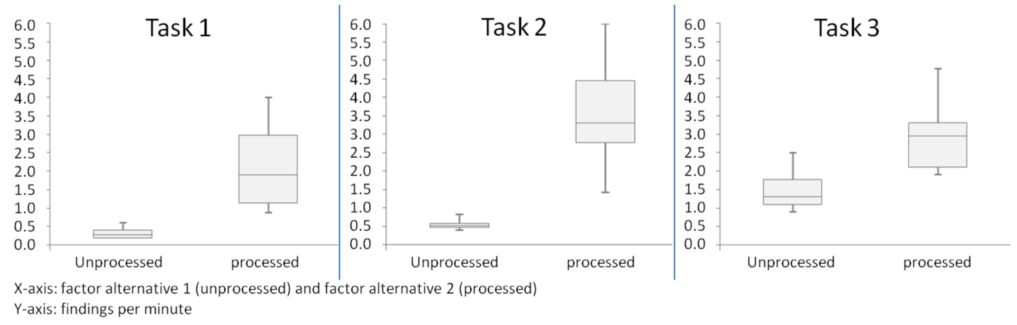


Figure 28: Box Plots for the Tasks 1, 2, and 3

The visual inspection indicated that there is a difference in the means, and that the distribution is skewed for all tasks. Calculated values for the skewness for task 1 are 0.887 for factor alternative 1, and 0.562 for factor alternative 2. For task 2, the skewness is 1.403 for factor alternative 1 and 0.345 for factor alternative 2. For task 3, the skewness is 0.994 for factor alternative 1 and 0.807 for factor alternative 2.

Focusing on the tails of the distributions, the kurtosis was calculated. For task 1, the kurtosis is -0.765 for factor alternative 1 and -1.236 for factor alternative 2. For task 2, the kurtosis is 2.945 for factor alternative 1 and -0.207 for factor alternative 2. For task 3, the kurtosis is 0.176 for factor alternative 1 and 0.053 for factor alternative 2.

As this led to doubts regarding the assumption of a normal distribution, and as the sample size is small, a Shapiro-Wilk test was conducted to see if the sample data originate from a normally distributed population. Those tests revealed an ambivalence, as for all data for tasks with factor alternative 1 the null hypothesis of normal distribution was rejected (Task 1: $W = 0.136$, Task 2: $W = 0.113$; Task 3: $W = 0.491$; critical $W = 0.818$). For all data for tasks with factor alternative 2 instead, the null hypothesis was accepted (Task 1: $W = 1.087$, Task 2: $W = 1.387$; Task 3: $W = 0.884$; critical $W = 0.818$). It is unclear, why there is such a clear separation in the data. It has been concluded that for further calculations, non-parametric tests have to be used.

Due to the paired design, a two-tailed Wilcoxon signed-rank test was used to check, whether there are differences in the efficiency of the task execution for each task when using factor alternative 1 compared to using factor alternative

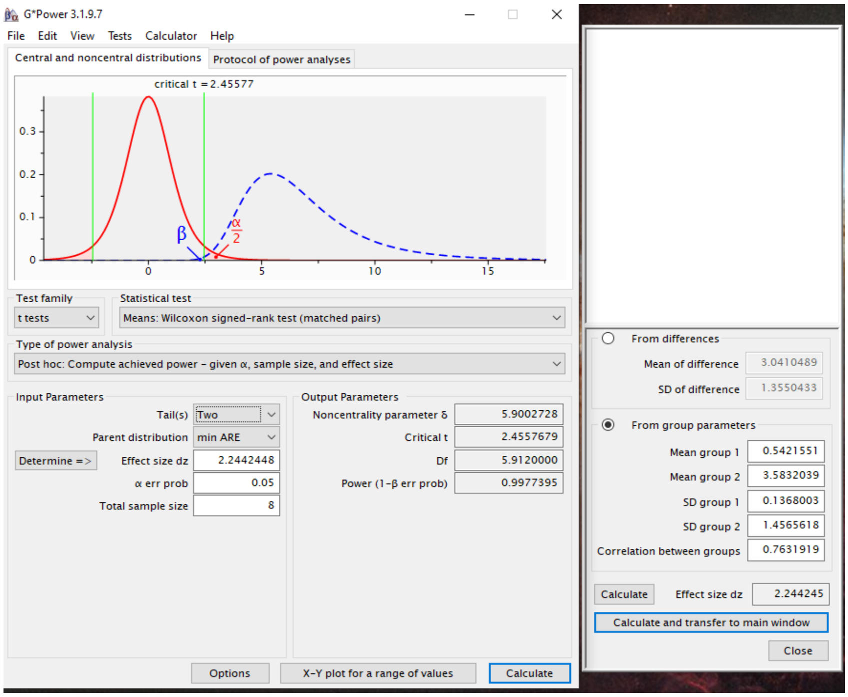
2, assuming that there are no significant differences (null hypothesis). Again, the derived number of findings per minute was used for the calculations.

For all three tasks, the calculated ranks were $\mathbf{36}$ and $\mathbf{0}$. As for a two-tailed test with a sample size of 8, the critical value for the rank is $\mathbf{3}$ using an $\boldsymbol{\alpha} = \mathbf{0.05}$, the null hypothesis is rejected for tasks 1, 2 and 3.

Finally, the tool `G*Power` [Fa09] was used for a post-hoc analysis of the statistical power achieved and the effect size for each task. For the calculation of the statistical power, a parent distribution had to be selected. However, it can only be stated that there is no normal distribution. Out of the options "Normal, Laplace, Logistic, and min ARE", the option "min ARE" has been selected, resulting in a "theoretical minimum of the power" [No21a]. This results in the actual statistical power being at least as high as the calculated value.

The results of these computations were surprising, as for all three tasks, a high statistical power was calculated although the sample size was low. The reason for this was always the calculated effect size, which turned out to be extremely high. Note that the effect size numbers presented below are still within the sizes reported by Sawilowsky [Sa09].

For task 1, the calculated effect size was $\mathbf{1.586}$, resulting in a statistical power of the analysis of $\mathbf{0.930}$. For task 2, the calculated effect size was even higher, with $\mathbf{2.244}$. This resulted in a statistical power of the analysis of $\mathbf{0.9978}$. A screenshot of the calculation with `G*Power` can be found below in Figure 29. Finally, for task 3, the calculated effect size was $\mathbf{1.786}$, and the statistical power was calculated with $\mathbf{0.971}$.


G*Power 3.1.9.7
File Edit View Tests Calculator Help
Central and noncentral distributions
Protocol of power analyses
critical t = 2.45577
β
α/2
Determine =>
Calculate
Calculate and transfer to main window
Close
Options
X-Y plot for a range of values
Calculate

Test family: t tests
Statistical test: Means: Wilcoxon signed-rank test (matched pairs)
Type of power analysis: Post hoc: Compute achieved power – given α, sample size, and effect size

Input Parameters
Tail(s): Two
Parent distribution: min ARE
Effect size dz: 2.2442448
α err prob: 0.05
Total sample size: 8

Output Parameters
Noncentrality parameter δ: 5.9002728
Critical t: 2.4557679
Df: 5.9120000
Power (1−β err prob): 0.9977395

From differences
Mean of difference: 3.0410489
SD of difference: 1.3550433

From group parameters
Mean group 1: 0.5421551
Mean group 2: 3.5832039
SD group 1: 0.1368003
SD group 2: 1.4565618
Correlation between groups: 0.7631919
Effect size dz: 2.244245

Figure 29: Screenshot of G*Power for Task 2

As mentioned above, there are slight differences in the sets of data races for the two groups. Thus, possible differences in the efficiency using set 1 or set 2 for each task were calculated, for both factor alternatives (i.e., six examinations).

Again, descriptive statistics in the form of calculating the means, the variances, and the standard deviations was used as a first step. The box plots were almost the same for the tasks 1 and 2 with factor alternative 1 and indicated a slight difference for task 3 with factor alternative 1, with set 2 having lower values. For factor alternative 2, in contrast, the values for set 2 were slightly higher for all tasks.

Once again, supported by the p-p plot, the histogram, and calculations of skewness and kurtosis, a normal distribution could not be confirmed, and thus, non-parametric tests were used for the further calculations. Note that in these calculations, the sample sizes were always 4, as the data had to be split up with regard to the set used.

To check whether there are differences in the efficiency of conducting the tasks with using set 1 or set 2, a Mann-Whitney-U test was used.

For factor alternative 1 and all three tasks, the test revealed no statistically relevant difference, and the null hypotheses had to be accepted with $U = 17$ for task 1, $U = 18$ for task 2, and $U = 14$ for task 3, all using a two-tailed test with $\alpha = 0.05$. The same holds for factor alternative 2, with $U = 16$ for task 1, $U = 16$ for task 2, and $U = 14.5$ for task 3.

Because of these measurements, it can be concluded that the sets had no statistically significant influence on the efficiency of conducting the tasks. Due to the within-subject design, the experience of experimental subjects is also excluded as influencing factor. Thus, the differences in the efficiency of conducting the three tasks can be attributed to the factor alternatives.

Finally, the efficiency improvements observed in the controlled experiment were calculated by comparing the mean values for the derived measure of findings per minute. For task 1, an efficiency improvement of 6.522 could be observed, i.e., the efficiency improved from 100% for factor alternative 1 to 652% for factor alternative 2. For task 2, a similar value was observed, with an efficiency improvement of 6.609 when using factor alternative 2. Finally, for task 3, an efficiency improvement of 1.999 when using factor alternative 2 could be observed.

As these values are much higher than the hypothesized efficiency improvements of at least 0.40, all three null hypotheses stated in chapter 6.5.2 are rejected, and the alternative hypotheses are accepted.

Besides analyzing the data of the analysis of data race reports, the questionnaire was analyzed with regard to the ratings of the participants concerning the statements mentioned in chapter 6.5.3. The raw data of these ratings can be seen in the following Table 9:

	Participant ID	1	2	3	4	5	6	7	8
	Experience (in years)	5	10	6	2	8	13	12	17
Unprocessed (Factor alternative 1)	Confidence in completeness	2	2	1	1	2	3	2	2
	Confidence in correctness	4	2	2	2	3	4	4	2
	Difficult	1	2	5	3	4	3	2	4
	Exhausting	1	4	5	4	5	4	5	5
Processed (Factor alternative 2)	Confidence in completeness	5	5	5	5	3	5	5	5
	Confidence in correctness	5	5	5	4	3	5	5	5
	Difficult	1	1	1	1	1	1	1	2
	Exhausting	1	1	1	1	1	2	2	2

Table 9: Raw Data of the Questionnaire

The experience was not further analyzed statistically. This data was instead used to examine whether there are any dependencies between the experience and the efficiency in the analysis of data race reports. Such a dependency could not be found.

As the data received from using the Likert scale is ordinal data, the possibilities to analyze the data are more limited than when using a ratio scale. A Mann-Whitney-U test has been used to analyzing the data and comparing the ratings for the factor alternatives 1 and 2. It was hypothesized that there is no difference between the ratings for both factor alternatives (zero hypothesis). The statistical analyses have been conducted with an $\alpha = 0.05$, and due to both n_1 and n_2 being 8, $U_{crit} = 13$ for the Mann-Whitney U Test.

All hypotheses had to be rejected. The hypothesis regarding the confidence in the completeness of the results had to be rejected with $U_{min} = 0.5, z = -3.308,$ and $p = 0.001$. The analysis regarding the confidence in the correctness of the results revealed values of $U_{min} = 5, z = -2.866,$ and $p = 0.005$. Computing the rank sums for the ratings regarding the perception of the tasks being difficult and being exhausting resulted in the same rank sums, and the same values of $U_{min} = 5.5, z = -2.783,$ and $p = 0.005$. The calculated effect sizes were 0.827 for the confidence in the completeness of the results, 0.709 for the confidence in the correctness of the results, and 0.696 both for the tasks being difficult and being exhausting.

Summarized, it can be concluded that the participants had more confidence both in the completeness and in the correctness of the results of the analysis when using the preprocessed data race reports compared to when using the unprocessed log files. In addition, the participants perceived these tasks less difficult and less exhausting when using the preprocessed data race reports.

6.5.6 Discussion of the Results

Although the sample size for the controlled experiment was low, the immense effect sizes allowed drawing conclusions with a high statistical power. Still, the small sample size is a threat to the validity. Due to the randomization of experimental subjects and group assignment, the risk can be assessed to be acceptable. An influence by learning effects due to the experimental subjects getting used to analyze data races and accesses has been mitigated by using a within-subject design.

Another possible threat to validity might be related to the sets of data races selected for the controlled experiment. Although the sets were selected out of a real-world set of data races, the characteristics of the selected sets of data races match the source set of reported data races, and those characteristics at least in terms of duplicate data races match what is reported for data races in general in literature, it cannot be excluded that with other sets of reported data races different values would have been observed.

Thus, the controlled experiment should be replicated with an increased sample size, and with a different set of reported data races.

It is interesting to note that the efficiency improvements for tasks 1 and 2 are much higher than for task 3, and that the efficiency improvement for task 2 is higher than for task 1. A possible interpretation is as follows.

For tasks 1 and 2, specific affected variables and source code locations had to be searched for. While the tool presented in this article allows to just filtering the data for such criteria, it is a time consuming task to read through all the reported data races. For each entry in the result files, it has to be checked whether the affected variables or source code locations are those searched for. In addition, there is a high number of duplicates, which need to be read and compared to what has already been found, which again is time consuming. The tool instead allows hiding those duplicate entries. These factors contribute to the high efficiency improvement.

The difference in the efficiency improvements in tasks 1 and task 2 might stem from the fact, that for task 1, a certain variable was specified, and for task 2, a source code class was specified. As a variable is connected to a data race, and a source code class to each of the two accesses building a data race, the effort for examining specific source code classes is higher than for examining specific variables.

When conducting task 3, those specific variables and source code classes are not of interest, and thus do not influence the efficiency. However, the duplicates still are present for factor alternative 1 and can be hidden in factor alternative 2. This contributes to the difference in the efficiency of conducting task 3 with factor alternative 1 or 2.

Finally, the tool presents all results in a tabular format, providing a view on all the accesses at once (although scrolling may be needed), while for factor alternative 1, the accesses were spread across many different files. In addition, another important influence on the efficiency stems from the knowledge provided by factor alternative 2 on how many distinct accesses exist, with regard to a specific variable or source code location, and in general. Without such a knowledge, all reported data races had to be examined to gain confidence on how many distinct data races or accesses exist, and that all existing distinct data races or accesses have been found. It can be assumed that this strongly influences the efficiency in data race analysis.

The results from analyzing the questionnaires support these conclusions. The participants profited from the knowledge gained by using the tool in that they felt more confident in their results regarding completeness and correctness. When using the tool, the participants were able to compare their results with the entries the tool displayed. Without having such an overview, there is no basis for that judgment except for an overview on how many folders were analyzed and how many folders were remaining. However, the number of files or folders does not allow a reliable assessment, as there may be different numbers of reported data races within each file. The only concerns that were raised during the video calls were that the participants had no possibility to review the source code of the tool for the preprocessing itself and thus had to trust on its reliability.

Furthermore, having an organized overview on all the results at one place, and not having to browse through several folders and reading files supported the participants in their task in that the task was perceived as being less difficult and less exhausting. A reason may be that with the presentation in the tool, the participants could focus on reading the details of the accesses and data races and there was no necessity to concentrate on such surrounding tasks.

The higher confidence in the completeness of the results when using the preprocessing could be confirmed in that the participants could in summary indeed detect more accesses or data races in the same amount of time. However, the higher confidence in the correctness of the results could not be confirmed when analyzing the results of the data race report analysis. For both factor alternatives each, one participant had two wrong entries, as stated in chapter 6.5.5. For factor alternative 2 (preprocessed), one participant wrote down a wrong source code line in two cases. For factor alternative 1 (unprocessed), another participant inserted two wrong entries in the task list. These entries could not be attributed to a specific source of error, e.g., writing down a wrong number for the source code line, or reading a wrong variable.

It has to be noted that the comparison of the efficiency could not be done between two tools specialized exactly on the purpose of analyzing data races. As such, the controlled experiment can only show efficiency improvements when using such a specialized tool compared to not using such a specialized tool.

6.6 Summary

As the solution presented in this thesis was targeted at efficiency improvements in the quality assurance process for data races, the effects of the solution have been studied in terms of achievable efficiency improvements. Based on the problems and goals, hypotheses have been formulated. Some properties of the solution could be formally proven. Other properties of the solution have been studied using exemplary evaluations and a controlled experiment.

As for hypothesis 1, the complete and correct analysis has been formally proven in chapter 4.6 in theory. As explained, in the practical application, cycle traversals need to be limited to prevent infinite DAGs and infinitely sized DAGs. This limitation has been implemented as shown in chapter 4.6.5. An additional assessment of the completeness in the practical application of the analysis has thus been conducted. Although this assessment confirmed the hypothesis, as shown in chapter 6.3, it is not possible to generalize those results, as explained in chapter 6.1.

The efficiency improvement in the dynamic detection of data races, as assumed in hypothesis 2, has been assessed using an exemplary examination on one system. Although this system is a real-world system, and not created for the specific purpose of this thesis, it cannot be seen as representative for all applications that exist. As such, hypothesis 2 could be confirmed in that case, as explained in chapter 6.4. These results are again not generalizable.

A controlled experiment has been conducted with regard to hypothesis 3 and the underlying hypotheses 3.1, 3.2, and 3.3, as presented in chapter 6.5. This controlled experiment revealed a very high effect size, and due to this, the results have a high statistical power despite the rather low sample size. With achieved efficiency improvements of 652% (relating to hypothesis 3.1), 661% (relating to hypothesis 3.2), and 199% (relating to hypothesis 3.3), the achieved efficiency improvement in analyzing data race reports is much higher than what was expected.

Table 10 summarizes the results. As explained, experimental evaluations could not be conducted for all hypotheses defined in this thesis. To further raise the knowledge regarding the achieved effects of the solution presented in this thesis, additional experimental evaluations can be set up especially regarding hypotheses 1 and 2. This should be addressed in future work.

Hypotheses	Confirmation achieved
Hypothesis 1: Complete and correct analysis	(partially) Formally proven (theory) and confirmed exemplarily (practice) No experimental evaluation
Hypothesis 2: Efficiency improvement in the dynamic data race detection of data races (runtime overhead)	(specific case) Confirmed exemplarily No experimental evaluation
Hypothesis 3:Efficiency improvement in analyzing data races	Indirectly by evaluating subordinated hypotheses
Hypothesis 3.1: Efficiency improvement in analyzing data race reports for data races	Controlled experiment
Hypothesis 3.2: Efficiency improvement in analyzing data race reports for data races affecting a source code class	Controlled experiment
Hypothesis 3.3: Efficiency improvement in analyzing data race reports for data races affecting a variable in the source code.	Controlled experiment

Table 10: Summary of Achieved Confirmation of Hypotheses

7 Summary and Future Work

In this thesis, a solution for efficiency improvements in the quality assurance process for data races was presented. The purpose of this chapter is to explain the contributions made, related to the research objectives stated in chapter 1.3, and how the state of the art has been extended. Furthermore, directions for future work are given.

7.1 Contributions

The solution presented in this thesis consists of two main parts, surrounding the test execution in the quality assurance process for data races:

1. Analyzing the dynamic behavior of a system under test, with the goal to identify nodes, representing source code classes that may be executed in parallel to other nodes or to themselves. This information can then be used to focus instrumentation for the dynamic data race detection, enabling a reduction in the runtime overhead without affecting its effectiveness, and thus an efficiency improvement in the dynamic data race detection.

2. Analyzing characteristics of data races in general and of data race reports, with the goal to process these data race reports, and to provide the results of the preprocessing to users. This preprocessing and the presentation of the results enable a more efficient analysis of detected data races.

Several contributions have been made while conducting the research in the context of this thesis, aligned with the objectives of the research (see chapter 1.3), as shown in the following.

1) Assess the representations of dynamic behavior of a system with the goal to select one representation for the analysis.

   Several different representations of the dynamic behavior of a system exist. In this thesis, the most prominent types of representations, according to the literature, have been examined. As a result, *UML Activities* have been chosen as the representation of the dynamic behavior of a system used in the context of this thesis.

2) Analyze the characteristics of the chosen representation of the dynamic behavior with the goal to understand the specifics and how these specifics influence the interpretation of this representation.

The UML specification has been examined in detail and all elements, for which a definition exists, have been analyzed with the goal to understand the influence of these elements on the *UML Activities*. This not only includes nodes and edges, but also regions, pins, parameters, combinations of elements, and combinations of *UML Activities*. As a result, all elements, for which a definition exists, could be considered in the analysis.

3) Create a technique to analyze the chosen representation of the dynamic behavior of a system for parallel elements, with the goal to obtain an algorithmic approach for the analysis.

   Based on a systematic literature review, the current state of the art in the analysis has been identified. The analysis of the state of the art revealed several research gaps, which could be closed by the solution presented in this thesis. As such, the state of the art could be advanced in this thesis. A core contribution to advancing the state of art is the possibility to obtain complete analysis results while limiting possibly infinite cycle traversals.

   Based on the knowledge gained by analyzing the UML superstructure, the elements could be categorized into basic and additional elements. While basic elements could be directly considered in the analysis of *UML Activities*, additional elements required a specific handling, and those elements either had to be transferred to other elements, considered after the analysis, or both.

   The solution for the analysis therefore consists of three steps: transformation, analysis, and post-analysis. The algorithmic approaches forming the core of the analysis and the post-analysis steps are formal approaches based on graph theoretic concepts, and thus, the algorithms for the analysis could be clearly defined.

4) Show the completeness and correctness of the obtained algorithmic approach, with the goal to establish confidence in this approach.

   As the analysis of *UML Activities* presented in this thesis is a formal approach, formal proofs could be used to show certain characteristics of the solution, including correctness of the results. The completeness of the approach has been proven from a theoretic viewpoint. The implementation of this approach has been examined using several *UML Activities* representing different levels of complexity, and completeness and correctness of the results could be confirmed for these examples.

5) Evaluate the effects on efficiency of the test execution of the algorithmic approach compared to not using the algorithmic approach to focus instrumentation.

As an evaluation of the effects on the efficiency of the test execution with the necessary external validity to draw generalizable results would require an immense effort, this could not be included in this thesis. However, an exemplary examination has been conducted, and for this specific case, a reduction of the runtime overhead when using the focused instrumentation compared to using full instrumentation could be shown.

6) Enable the practical application of the algorithmic approach, with the goal to obtain a fully automated tool implementing this algorithmic approach.

   The algorithmic approach for the analysis of *UML Activities* has been implemented prototypically in `ADDAG`. The usage of `ADDAG` has been shown with an exemplary application, and the approach itself has been used to demonstrate the completeness of the results of the analysis and to demonstrate possible improvements in the runtime overhead in dynamic data race detection.

7) Analyze characteristics of data race reports, with the goal to obtain knowledge about how to preprocess data races reports so that the analysis is more efficient.

   Based on a formal definition of data races from literature, the characteristics of data races have been derived. These characteristics have been complemented by the results of the analysis of data race reports, provided by the dynamic data race detector used in this thesis. The knowledge gained has been transferred to a UML model. In addition, possible duplicates in data race reports have been considered in the analysis.

8) Enable the practical application of the knowledge obtained on characteristics of data race reports, with the goal to obtain tool support.

   Based on the knowledge gained concerning data races and data race reports, the application `DataRaceAnalyzer` has been developed, which preprocesses data race reports, and presents the results of this preprocessing to users. Since no other applications exist that are targeted at guiding the analysis of data race reports and the elimination of such defects despite a need for such tools, the state of the art could be advanced.

9) Empirically evaluate the effects of the preprocessing of the data race reports and of the tool support on the efficiency on the analysis of data race reports compared to analyzing unprocessed data race reports.

   The effects of the preprocessing of data race reports on the efficiency of the analysis of data race reports have been empirically evaluated in a controlled experiment with eight participants. Despite the low sample size, the effect size was very high, and thus, a high statistical power could be achieved. The results of this controlled experiment have shown that immense improvements in the efficiency of the analysis of data race reports could be enabled when using the preprocessing compared to analyzing unprocessed data race reports.

7.2 Open Questions and Future Work

The open questions and directions for future work can be categorized into methodologies, tool support, and empirical evaluation.

In terms of methodologies, multiple opportunities for future work are given. In this thesis, it was assumed, that behavioral representations of the system under test exist. As this is not always given, an open question is how to automatically derive such models, if they have not been created during development, on the basis of the source code of an application. In the same regard, the completeness of the behavioral representation was assumed. Based on such a methodology to derive such models, an approach to check the completeness of these behavioral descriptions would be valuable.

The algorithm and the solution for analyzing behavioral descriptions are focused on *UML Activities*. Since other representations exist, algorithms to analyze such other types of representations could enable an extension of the solution, providing more flexibility in terms of the prerequisites. In the same manner, and interesting direction is to include other models of the UML itself in the analysis, such as sequence diagrams or class diagrams.

Once methodologies to derive behavioral representations in the form of *UML Activities* have been created, the next logical step is to implement these methodologies to create tool support, and in the best case a fully-automated approach. The solution presented in this thesis can be extended in this regard, to provide better support for applications under test by lowering the prerequisites for usage.

The same holds with regard to tool support for deriving other types of behavioral representations of a system. In that case, the solution for the analysis of *UML Activities* could be extended to broaden its scope. Since a three step approach is used, and the first step is a transformation, the tool is already prepared in that the analysis relies on general node types for the analysis. Thus, another transformation would be required. If a mapping from the elements of a representation of the dynamic behavior to the elements already used in the analysis can be created, this analysis could be reused. However, the post-analysis phase might require to be extended, depending on the elements of that newly supported type of representation.

With regard to the analysis of reported data races, the tool presented in this thesis can be further improved for continuous usage. Although a status can be assigned to each entry in the results list, it is not possible to export or import those lists together with the status information. Such a feature allows working on the results across several sessions. Another opportunity for future work is to prepare the tool for multi user operation. By splitting up responsibilities for data races related to certain variables to several users, it would be possible to work on the same list in parallel, thus saving time in case multiple users need to analyze and eliminate data races for the same application.

Open questions remain with regard to the empirical evaluations. Although the correctness and completeness of the analysis of *UML Activities* has been formally proven, more assessments in terms of the practical application of the implementation are desirable. In the course of this thesis, such an assessment has been made using ten activities. However, the possibilities to combine UML elements are vast, and a more intensive assessment allows a better understanding of the completeness of the analysis with regard to the implementation, and would thus provide more confidence in the usage of the tool.

Furthermore, the efficiency improvement of dynamic data race detection using a focused instrumentation could only be assessed exemplarily with one application. Thus, the external validity of this assessment is low. Additional assessments using different applications of various sizes can provide a better insight into the benefits of using focused instrumentation, and thus strengthen the knowledge regarding its effects.

Finally, the controlled experiment with regard to the assessment of efficiency improvements in the analysis of data race reports could only be done by comparing `DataRaceAnalyzer` to office tools. At the time of conducting this controlled experiment, there was no other tool available, which focused on the analysis of data race reports in terms of enabling efficiency improvements. In the future, other tools supporting the analysis of data race reports may be released. It would then be very interesting to compare those tools to `DataRaceAnalyzer`, to obtain insight into different approaches and their effects. As `DataRaceAnalyzer` was the first tool with such a focus, the insights gained may then allow additional improvements, with the ultimate goal to further improve the efficiency of data race analysis and eventually the whole quality assurance process for data races.

References

[AB10] Adve, S. V.; Boehm, H.-J.: Memory Models: A Case For Rethinking Parallel Languages and Hardware. Communications of the ACM 8/53, pp. 90–101, 2010.

[Ab17] Abbaspour Asadollah, S.; Sundmark, D.; Eldh, S.; Hansson, H.: Concurrency bugs in open source software: a case study. Journal of Internet Services and Applications 1/8, 2017.

[ABF04] Arisholm, E.; Briand, L. C.; Foyen, A.: Dynamic coupling measurement for object-oriented software. IEEE Transactions on Software Engineering 8/30, pp. 491–506, 2004.

[ABF08] Almeida, P. S.; Baquero, C.; Fonte, V.: Interval Tree Clocks: A Logical Clock for Dynamic Systems. In (Baker, T. P.; Bui, A.; Tixeuil, S. Eds.): Principles of Distributed Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 259–274, 2008.

[Ad10] Adve, S.: Data races are evil with no exceptions. Communications of the ACM 11/53, p. 84, 2010.

[An94] Andersen, L. O.: Program Analysis and Specialization for the C Programming Language. Dissertation, Copenhagen, 1994.

[As15] Asadollah, S. A.; Hansson, H.; Sundmark, D.; Eldh, S.: Towards Classification of Concurrency Bugs Based on Observable Properties. In: Proceedings of the First International Workshop on Complex faUlts and Failures in LargE Software Systems (COUFLESS '15). IEEE Press, pp. 41–47, 2015.

[AS15] Atkey, R.; Sannella, D.: ThreadSafe: Static Analysis for Java Concurrency, 2015.

[Au97] Audenaert, K.: Clock trees: logical clocks for programs with nested parallelism. IEEE Transactions on Software Engineering 10/23, pp. 646–658, 1997.

[Ba06a] Banerjee, U.; Bliss, B.; Ma, Z.; Petersen, P.: A theory of data race detection. July 17 - 20, 2006, Portland, Maine, USA. ACM Press, New York, NY, 2006.

[Ba06b] Banerjee, U.; Bliss, B.; Ma, Z.; Petersen, P.: Unraveling Data Race Detection in the Intel® Thread Checker, 2006.

[BA08] Boehm, H.-J.; Adve, S. V.: Foundations of the C++ concurrency memory model. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). Association for Computing Machinery, New York, NY, USA ACM, New York, NY, 2008.

[BCM10] Bond, M. D.; Coons, K. E.; McKinley, K. S.: PACER: Proportional Detection of Data Races. ACM SIGPLAN Notices 6/45, pp. 255–268, 2010.

[BH08] Bodden, E.; Havelund, K.: Racer: Effective Race Detection Using AspectJ. Association for Computing Machinery, New York N.Y., 2008.

[BHO20] Blondin, M.; Haase, C.; Offtermatt, P.: Directed Reachability for Infinite-State Systems, Springer International Publishing, 2020.

[Bi17] Biswas, S.; Cao, M.; Zhang, M.; Bond, M. D.; Wood, B. P.: Lightweight data race detection for production runs. In: Proceedings of the 26th International Conference on Compiler Construction (CC 2017). Association for Computing Machinery, New York, NY, USA, pp. 11–21, 2017.

[BK20] Burzynski, P.; Karagiannis, D.: bee-up – A teaching tool for fundamental conceptual modelling. Joint Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers, pp. 217–221, 2020.

[Bl18] Blackshear, S.; Gorogiannis, N.; O'Hearn, P. W.; Sergey, I.: RacerD: compositional static race detection. In: Proceedings of the ACM on Programming Languages OOPSLA/2, pp. 1–28, 2018.

[Bo11a] Boehm, H.-J.: How to miscompile programs with "benign" data races. HotPar'11: Proceedings of the 3rd USENIX conference on Hot topic in parallelism, 2011.

[Bo11b] Boghdady, P.N. et al. Eds.: An enhanced test case generation technique based on activity diagrams. Cairo, Egypt, 29 November - 1 December 2011. IEEE, Piscataway, NJ, 2011.

[Bo12a] Boehm, H.-J.: Position paper: Nondeterminism is unavoidable, but data races are pure evil. In (Black, A. P. et al. Eds.): Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability - RACES '12. ACM Press, New York, New York, USA, p. 9, 2012.

[Bo12b] Boehm, H.-J.: Position Paper: Nondeterminism is unavoidable, but data races are pure evil. RACES '12: Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability, p. 9, 2012.

[Bo19] Bo, L.; Jiang, S.; Qian, J.; Wang, R.; Yao, Y.: Performance Evaluation of Data Race Detection Based on Thread Sharing Analysis With Different Granularities: An Empirical Study. IEEE Access 7, pp. 73819–73829, 2019.

[Br73] Brinch Hansen, P.: Operating System Principles. Prentice-Hall, Inc., New Jersey, 1973.

[BRJ96] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language for Object-Oriented Development. Unix Review 13/14, 1996.

[Ch09] Chen, Q.; Wang, L.; Yang, Z.; Stoller, S. D.: HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In (Chechik, M.; Wirsing, M. Eds.): Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 425–439, 2009.

[Ch76] Chen, P. P.-S.: The entity-relationship model—toward a unified view of data. ACM Transactions on Database Systems 1/1, pp. 9–36, 1976.

[CL10] Chew, L.; Lie, D.: Kivati: Fast Detection and Prevention of Atomicity Violations. Proceedings of the EuroSys 2010 Conference, Paris, France, April 13-16, 2010. Association for Computing Machinery, New York, 2010.

[CLL07] Chandler, R.; Li, H.; Lam, C. P.: Generating Usage Scenarios Automatically from UML Activity Diagrams. Technical Report: TR-SERG-06-01, Mount Lawley, 2007.

[Da16] Daian, P.; Guth, D.; Hathhorn, C.; Li, Y.; Pek, E.; Saxena, M.; Şerbănuţă, T. F.; Roşu, G.: Runtime Verification at Work: A Tutorial. In (Falcone, Y.; Sánchez, C. Eds.): Runtime Verification. Springer International Publishing, Cham, pp. 46-67, 2016.

[DB03] Drummond, L. M.; Barbosa, V. C.: On reducing the complexity of matrix clocks. Parallel Computing 7/29, pp. 895–905, 2003.

[Di65a] Dijkstra, E. W.: Cooperating sequential processes. Technical Report EWD-123, 1965.

[Di65b] Dijkstra, E. W.: Solution of a problem in concurrent programming control. Communications of the ACM 9/8, p. 569, 1965.

[DMM98] Diwan, A.; McKinley, K. S.; Moss, J. E. B.: Type-based alias analysis. ACM SIGPLAN Notices 5/33, pp. 106–117, 1998.

[Ef12] Effinger-Dean, L.; Lucia, B.; Ceze, L.; Grossman, D.; Boehm, H.-J.: IFRit: interference-free regions for dynamic data-race detection. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12). Association for Computing Machinery, New York, NY, USA, 2012.

[EM72] Eisenberg, M. A.; McGuire, M. R.: Further comments on Dijkstra's concurrent programming control problem. Communications of the ACM 11/15, p. 999, 1972.

[Er10] Erickson, J.; Musuvathi, M.; Burckhardt, S.; Olynyk, K.: Effective Data-Race Detection for the Kernel, 2010.

[Fa09] Faul, F.; Erdfelder, E.; Buchner, A.; Lang, A.-G.: Statistical power analyses using G*Power 3.1: tests for correlation and regression analyses. Behavior research methods 4/41, pp. 1149–1160, 2009.

[FF09] Flanagan, C.; Freund, S. N.: FastTrack: Efficient and Precise Dynamic Race Detection. ACM SIGPLAN Notices 6/44, pp. 121–133, 2009.

[FF20] Flanagan, C.; Freund, S. N.: The anchor verifier for blocking and non-blocking concurrent software. Proceedings of the ACM on Programming Languages OOPSLA/4, pp. 1–29, 2020.

[Fi88] Fidge, C. J.: Timestamps in Message-Passing Systems That Preserve the Partial Ordering. Australian Computer Science Communications No. 1/Vol. 10, pp. 56–66, 1988.

[Fi91] Fidge, C.: Logical time in distributed computing systems. Computer 8/24, pp. 28–33, 1991.

[FM82] Fischer, M. J.; Michael, A.: Sacrificing serializability to attain high availability of data in an unreliable network. In: Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems (PODS '82). Association for Computing Machinery, New York, NY, USA, 1982.

[Fo10] Fowler, M.: UML distilled. A brief guide to the standard object modeling language. Addison-Wesley, Boston, MA, 2010.

[FQ03] Flanagan, C.; Qadeer, S.: A type and effect system for atomicity. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03). Association for Computing Machinery, New York, NY, USA, 2003.

[Ge19] Genç, K.; Roemer, J.; Xu, Y.; Bond, M. D.: Dependence-aware, unbounded sound predictive race detection. Proceedings of the ACM on Programming Languages OOPSLA/3, pp. 1–30, 2019.

[GG21] Guizani, K.; Ghannouchi, S. A.: An approach for selecting a business process modeling language that best meets the requirements of a modeler. Procedia Computer Science 181, pp. 843–851, 2021.

[GKM20] Gharat, P. M.; Khedker, U. P.; Mycroft, A.: Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis. ACM Transactions on Programming Languages and Systems 2/42, pp. 1–78, 2020.

[GN08] Godefroid, P.; Nagappan, N.: Concurrency at Microsoft – An Exploratory Survey. Microsoft Research Technical Report MSR-TR-2008-75, 2008.

[HMR14] Huang, J.; Meredith, P. O.; Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In (O'Boyle, M.; Pingali, K. Eds.): Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 337–348, 2014.

[Ho72] Holt, R. C.: Some Deadlock Properties of Computer Systems. ACM Computing Surveys 3/4, pp. 179–196, 1972.

[Ho74] Hoare, C. A. R.: Monitors. Communications of the ACM 10/17, pp. 549–557, 1974.

[HS09] Hammer, C.; Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security 6/8, pp. 399–422, 2009.

[HZZ13] Huang, J.; Zhou, J.; Zhang, C.: Scaling predictive analysis of concurrent programs by removing trace redundancy. ACM Transactions on Software Engineering and Methodology 1/22, pp. 1–21, 2013.

[Je82] Jensen, K.: High-Level Petri Nets. Applications and Theory of Petri Nets. Informatik-Fachberichte 66, pp. 166–180, 1982.

[JT14] Jannesari, A.; Tichy, W. F.: Library-Independent Data Race Detection. IEEE Transactions on Parallel and Distributed Systems 10/25, pp. 2606–2616, 2014.

[Ka17] Kasikci, B.; Cui, W.; Ge, X.; Niu, B.: Lazy Diagnosis of In-Production Concurrency Bugs: Proceedings of the 26th Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 582–598, 2017.

[KE14] Klaus, A.; Elberzhager, F.: Retrieving the state of the art and of the practice in QA for data inconsistencies. In (Büren, G. et al. Eds.): MetriKon 2014 - Praxis der Software-Messung. Tagungsband des DASMA Software Metrik Kongresses ; MetriKon 2014, 06.-07. November 2014, Stuttgart. Shaker, Aachen, pp. 127–136, 2014.

[Ki07a] Kim, H.; Kang, S.; Baik, J.; Ko, I.: Test Cases Generation from UML Activity Diagrams: Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007). IEEE, pp. 556–561, 2007 - 2007.

[Ki07b] Kitchenham, B.; Charters, S.; Budgen, D.; Brereton, P.; Turner, M.; Linkman, S.; Jorgensen, M.; Mendes, E.; Visaggio, G.: Guidelines for performing Systematic Literature Reviews in Software Engineering. Version 2.3. EBSE Technial Report. EBSE-2007-01, 2007.

[Kl12] Klaus, A.: Stakeholder-orientierter Software Test für Geschäftsanwendungen. Softwaretechnik-Trends 1/32, pp. 8–9, 2012.

[Kl13] Klaus, A.: Analyse und Test konkurrierender Zugriffe auf Daten bei Geschäftsanwendungen - Konzept zur Evaluierung. In (Büren, G. et al. Eds.): MetriKon 2013 - Praxis der Software-Messung. Tagungsband des DASMA Software Metrik Kongresses ; MetriKon 2013, 14.-15. November 2013, Kaiserslautern. Shaker, Aachen, pp. 313–318, 2013.

[KMV17] Kini, D.; Mathur, U.; Viswanathan, M.: Dynamic race prediction in linear time. In (Cohen, A.; Vechev, M. Eds.): Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 157–170, 2017.

[KNS92] Keller, G.; Nüttgens, M.; Scheer, A.-W.: Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)". Scheer, A.-W. (Hrsg.): Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes 89, 1992.

[Ko15] Koutsopoulos, N.; Northover, M.; Felden, T.; Wittiger, M.: Advancing data race investigation and classification through visualization. Bremen, Germany, 27-28 September 2015. IEEE, Piscataway, NJ, 2015.

[KO20] Kharitonov, D. I.; Odyakova, D. S.: Modelling race conditions in multithreading programs in terms of Petri nets. IOP Conference Series: Materials Science and Engineering 734, p. 12030, 2020.

[KS09] Kundu, D.; Samanta, D.: A Novel Approach to Generate Test Cases from UML Activity Diagrams. The Journal of Object Technology 3/8, p. 65, 2009.

[KZC12] Kasikci, B.; Zamfir, C.; Candea, G.: Data Races vs. Data Race Bugs: Telling the Difference with Portend. Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 3-7, 2012, London, England, UK. ACM Press, New York N.Y., 2012.

[KZC13] Kasikci, B.; Zamfir, C.; Candea, G.: RaceMob: Crowdsourced Data Race Detection. In (Kaminsky, M.; Dahlin, M. Eds.): Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 406–422, 2013.

[KZC15] Kasikci, B.; Zamfir, C.; Candea, G.: Automated Classification of Data Races Under Both Strong and Weak Memory Models. ACM Transactions on Programming Languages and Systems 3/37, pp. 1–44, 2015.

[La07] Landes, T.: Tree clocks: an efficient and entirely dynamic logical time system: Proceedings of the 25th IASTED International Multi-Conference: parallel and distributed computing and networks, pp. 375–380, 2007.

[La10] Ladani, A. J.: Dynamic Race Detection in Parallel Programs. Dissertation, Karlsruhe, 2010.

[La74] Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 8/17, pp. 453–455, 1974.

[La78] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 7/21, pp. 558–565, 1978.

[La79] Lamport, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers 9/C-28, pp. 690–691, 1979.

[LD19] Lidbury, C.; Donaldson, A. F.: Sparse record and replay with controlled scheduling. In (McKinley, K. S.; Fisher, K. Eds.): Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 576–593, 2019.

[Li19] Li, G.; Lu, S.; Musuvathi, M.; Nath, S.; Padhye, R.: Efficient scalable thread-safety-violation detection. In (Brecht, T.; Williamson, C. Eds.): Proceedings of the 27th ACM Symposium on Operating Systems Principles. ACM, New York, NY, USA, pp. 162–180, 2019.

[Lo17] Lopez, C. T.; Marr, S.; Mössenböck, H.; Boix, E. G.: A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs. In: Ricci A.; Haller P. (eds) Programming with Actors. Lecture Notes in Computer Science, vol 10789. Springer, Cham. 2017.

[LTN19] Lima, L.; Tavares, A.; Nogueira, S. C.: A framework for verifying deadlock and nondeterminism in UML activity diagrams based on CSP, 2019.

[Lu08] Lu, S.; Park, S.; Seo, E.; Zhou, Y.: Learning from Mistakes —A Comprehensive Study on Real World Concurrency Bug Characteristics. Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 1-5, 2008, Seattle, Washington, USA. Association for Computing Machinery, New York N.Y., 2008.

[LWL08] Lei, B.; Wang, L.; Li, X.: UML Activity Diagram Based Testing of Java Concurrent Programs for Data Race and Inconsistency: 2008 International Conference on Software Testing, Verification, and Validation. IEEE, pp. 200–209, 2008 - 2008.

[Ma04] Manson, J.: The Java Memory Model. Dissertation, Maryland, 2004.

[Ma88] Mattern, F.: Virtual Time and Global States of Distributed Systems. In (Corsnard M. et al. Ed.): Proceedings of the International Workshop on Parallel and Distributed Algorithms. Elsevier Science Publishers B. V., pp. 120–134, 1988.

[Me15] Melo, S. M.; Souza, S. R. S.; Silva, R. A.; Souza, P. S. L.: Concurrent software testing in practice: a catalog of tools. In (Vos, T.; Eldh, S.; Prasetya, W. Eds.): Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation. ACM, New York, NY, USA, pp. 31–40, 2015.

[Mi08] Minkel, J. R.: The 2003 Northeast Blackout--Five Years Later. Scientific American, 2008.

[MKV18] Mathur, U.; Kini, D.; Viswanathan, M.: What Happens - After the First Race? Enhancing the Predictive Power of Happens - Before Based Dynamic Race Detection. In: Proceedings of the ACM on Programming Languages, Volume 2, Issue OOPSLA. ACM, New York, NY, USA, 2018.

[MMN09] Marino, D.; Musuvathi, M.; Narayanasamy, S.: LiteRace: Effective Sampling for Lightweight Data-Race Detection. ACM, New York, NY, 2009.

[MPV20] Mathur, U.; Pavlogiannis, A.; Viswanathan, M.: The Complexity of Dynamic Data Race Prediction. In (Hermanns, H. et al. Eds.): Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science. ACM, New York, NY, USA, pp. 713–727, 2020.

[Na07] Narayanasamy, S.; Wang, Z.; Tigani, J.; Edwards, A.; Calder, B.: Automatically classifying benign and harmful data races using replay analysis. ACM, New York, NY, 2007.

[NA07] Naik, M.; Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07). Association for Computing Machinery, New York, NY, USA, 2007.

[NM92] Netzer, R. H. B.; Miller, B. P.: What are race conditions? ACM Letters on Programming Languages and Systems 1/1, pp. 74–88, 1992.

[No21a] No author mentioned: G * Power 3.1 manual. https://www.psychologie.hhu.de/fileadmin/redaktion/Fakultaeten/Mathematisch-Naturwissenschaftliche_Fakultaet/Psychologie/AAP/gpower/GPowerManual.pdf, accessed 14 Nov 2021.

[No21b] No author mentioned: Apache Bug Database. https://bz.apache.org/bugzilla/query.cgi, accessed 14 Nov 2021.

[NZ98] Nüttgens, M.; Zimmermann, V.: Geschäftsprozeßmodellierung mit der objektorientierten Ereignisgesteuerten Prozeßkette (oEPK). In (Maicher, M.;

Scheruhn, H.-J. Eds.): Informationsmodellierung. Deutscher Universitätsverlag, Wiesbaden, pp. 23–35, 1998.

[O'05] O'Callahan, R.; Jones, C.; Froyd, N.; Huey, K.; Noll, A.; Partush, N.: Engineering Record And Replay For Deployability. USENIX Association, Berkeley, Calif., 2005.

[OA07] OASIS WS-BPEL Technical Committee: Web Services Business Process Execution Language, 2007.

[OGH20] Oortwijn, W.; Gurov, D.; Huisman, M.: An Abstraction Technique for Verifying Shared-Memory Concurrency. Applied Sciences 11/10, p. 3928, 2020.

[OM08] Otto, F.; Moschny, T.: Finding synchronization defects in java programs: Proceedings of the International Conference on Software Engineering & co-located workshops Leipzig, Germany, May 10 - 18, 2008. ACM, New York, NY, p. 41, 2008.

[OM13] OMG: Business Process Model and Notation (BPMN), Version 2.0.2, 2013.

[OM17] OMG: Unified Modeling Language, v2.5.1, 2017.

[Pa19a] Pavlogiannis, A.: Fast, Sound and Effectively Complete Dynamic Race Prediction. In: Proceedings of the ACM on Programming Languages, Volume 4, Issue POPL. Association for Computing Machinery, New York, NY, USA, 2019.

[Pa19b] Pande, M.: Visual Analytics Tool for Java_Virtual Machine Execution Traces. Master Thesis, Stockholm, 2019.

[Pa83] Parker, D. S.; Popek, G. J.; Rudisin, G.; Stoughton, A.; Walker, B. J.; Walton, E.; Chow, J. M.; Edwards, D.; Kiser, S.; Kline, C.: Detection of Mutual Inconsistency in Distributed Systems. IEEE Transactions on Software Engineering 3/SE-9, pp. 240–247, 1983.

[Pe62] Petri, C. A.: Kommunikation mit Automaten. Dissertation, Darmstadt, 1962.

[Pe81] Peterson, G. L.: Myths about the mutual exclusion problem. Information Processing Letters 3/12, pp. 115–116, 1981.

[PG01] Praun, C. von; Gross, T. R.: Object race detection. ACM SIGPLAN Notices 11/36, pp. 70–82, 2001.

[PG08] Patil, R. V.; George, B.: Tools And Techniques to Identify Concurrency Issues. MSDN Magazine 2008, 2008.

[PLZ09] Park, S.; Lu, S.; Zhou, Y.: CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In: Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS XIV). Association for Computing Machinery, New York, NY, USA, 2009.

[PMS20] Pereira, J. C.; Machado, N.; Sousa Pinto, J.: Testing for Race Conditions in Distributed Systems via SMT Solving. In (Ahrendt, W.; Wehrheim, H. Eds.): Tests and Proofs. Springer International Publishing, Cham, pp. 122–140, 2020.

[PS08] Park, C.-S.; Sen, K.: Randomized active atomicity violation detection in concurrent programs. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering (SIGSOFT '08/FSE-16). Association for Computing Machinery, New York, NY, USA, 2008.

[Re12] Reuters: Spike in deaths blamed on 2003 New York blackout. Reuters, 2012.

[RGB18] Roemer, J.; Genç, K.; Bond, M. D.: High-coverage, unbounded sound predictive race detection. In (Foster, J. S.; Grossman, D. Eds.): Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 374–389, 2018.

[RGB20] Roemer, J.; Genç, K.; Bond, M. D.: SmartTrack: efficient predictive race detection. In (Donaldson, A. F.; Torlak, E. Eds.): Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, New York, NY, USA, pp. 747–762, 2020.

[RH15] Rajagopalan, A. K.; Huang, J.: RDIT: race detection from incomplete traces. In (Di Nitto, E.; Harman, M.; Heymans, P. Eds.): Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, New York, NY, USA, pp. 914–917, 2015.

[Ro19] Roemer, J.: Practical High-Coverage Sound Predictive Race Detection. Dissertation, Ohio, 2019.

[RVS13] Raychev, V.; Vechev, M.; Sridharan, M.: Effective race detection for event-driven programs. In (Hosking, A.; Eugster, P.; Lopes, C. V. Eds.): Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications. ACM, New York, NY, USA, pp. 151–166, 2013.

[Sa09] Sawilowsky, S. S.: New Effect Size Rules of Thumb. Journal of Modern Applied Statistical Methods 2/8, pp. 597–599, 2009.

[Sa11] Said, M.; Wang, C.; Yang, Z.; Sakallah, K.: Generating Data Race Witnesses by an SMT-Based Analysis: NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings, pp. 313–327, 2011.

[Sa97] Savage, S.; Burrows, M.; Nelson, G.; Sobalvarro, P.; Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Transactions on Computer Systems 4/15, pp. 391–411, 1997.

[SAB19] Späth, J.; Ali, K.; Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized Pushdown systems. Proceedings of the ACM on Programming Languages POPL/3, pp. 1–29, 2019.

[Sc02] Scheer, A.-W.: ARIS - vom Geschäftsprozess zum Anwendungssystem. Springer, Berlin, 2002.

[Sc21] Schäffer, E.; Stiehl, V.; Schwab, P. K.; Mayr, A.; Lierhammer, J.; Franke, J.: Process-Driven Approach within the Engineering Domain by Combining Business Process Model and Notation (BPMN) with Process Engines. Procedia CIRP 96, pp. 207–212, 2021.

[SCR08] Serbănută, T. F.; Chen, F.; Rosu, G.: Maximal Causal Models for Multithreaded Systems. Technical Report UIUCDCS-R-2008-3017, 2008.

[Se08] Sen, K.: Race directed random testing of concurrent programs. ACM, New York, NY, 2008.

[SF07] Schattkowsky, T.; Förster, A.: On the Pitfalls of UML 2 Activity Modeling. International Workshop on Modeling in Software Engineering (MISE'07: ICSE Workshop 2007), p. 8, 2007.

[SH20] Schnoor, H.; Hasselbring, W.: Comparing Static and Dynamic Weighted Software Coupling Metrics. Computers 2/9, p. 24, 2020.

[SI09] Serebryany, K.; Iskhodzhanov, T.: ThreadSanitizer – data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09). Association for Computing Machinery, New York, NY, USA, 2009.

[SK18] Smaragdakis, Y.; Kastrinis, G. Eds.: Defensive Points-To Analysis: Effective Soundness via Laziness. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2018.

[SK20] Sophocleous, R.; Kapitsaki, G. M.: Examining the Current State of System Testing Methodologies in Quality Assurance. In (Stray, V. et al. Eds.): Agile Processes in Software Engineering and Extreme Programming. Springer International Publishing, Cham, pp. 240–249, 2020.

[SM08] Sapna, P. G.; Mohanty, H.: Automated Scenario Generation Based on UML Activity Diagrams: 2008 International Conference on Information Technology. IEEE, pp. 209–214, 2008 - 2008.

[Sm12] Smaragdakis, Y.; Evans, J.; Sadowski, C.; Yi, J.; Flanagan, C.: Sound predictive race detection in polynomial time. POPL'12 ; January 25-27, 2012, Philadelphia, PA, USA. ACM, New York, NY, 2012.

[Sp16] Späth, J. et al. Eds.: Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2016.

[Sp19] Späth, J.: Synchronized Pushdown Systems for Pointer and Data-Flow Analysis. Dissertation, Paderborn, 2019.

[St18] Stallings, W.: Operating systems. Internals and design principles. Pearson, Harlow, Essex, 2018.

[St20] Striewe, M.; Houy, C.; Rehse, J.-R.; Ullrich, M.; Fettke, P.; Schaper, N.; Oberweis, A.: Towards an Automated Assessment of Graphical (Business Process) Modelling Competences: A Research Agenda. Lecture Notes in Informatics (LNI), pp. 665–670, 2020.

[St96] Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages

(POPL '96). Association for Computing Machinery, New York, NY, USA, 32–41. pp. 32–41, 1996.

[Su08] Sun, C.: A Transformation-Based Approach to Generating Scenario-Oriented Test Cases from UML Activity Diagrams for Concurrent Applications: 2008 32nd Annual IEEE International Computer Software and Applications Conference. IEEE, pp. 160–167, 2008 - 2008.

[Su15] Sun, C.; Zhao, Y.; Pan, L.; He, X.; Towey, D.: A transformation-based approach to testing concurrent programs using UML activity diagrams. Software: Practice and Experience 4/46, pp. 551–576, 2015.

[Sz88] Szymanski, B. K.: A simple solution to Lamport's concurrent programming problem with linear wait. In (Lenfant, J. Ed.): Proceedings of the 2nd international conference on Supercomputing - ICS '88. ACM Press, New York, New York, USA, pp. 621–626, 1988.

[SZL09] Sun, C.; Zhang, B.; Li, J.: TSGen: A UML Activity Diagram-Based Test Scenario Generation Tool: 2009 International Conference on Computational Science and Engineering. IEEE, pp. 853–858, 2009 - 2009.

[Tr14] Trümper, J.: Visualization techniques for the analysis of software behavior and related structures. Dissertation, Potsdam, 2014.

[VA14] Verma, V.; Arora, V.: A novel approach for automatic test sequence generation for java fork/join from activity diagram. IEEE, Piscataway, NJ, 2014.

[Va20] Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Gall, H. C.; Zaidman, A.: How developers engage with static analysis tools in different contexts. Empirical Software Engineering 2/25, pp. 1419–1457, 2020.

[va98] van der AALST, W. M. P.: THE APPLICATION OF PETRI NETS TO WORKFLOW MANAGEMENT. Journal of Circuits, Systems and Computers 01/08, pp. 21–66, 1998.

[vH05] van der Aalst, W.; Hofstede, A. ter: YAWL: yet another workflow language. Information Systems 4/30, pp. 245–275, 2005.

[Wa20] Walker, A.; Coffey, M.; Tisnovsky, P.; Cerny, T.: On Limitations of Modern Static Analysis Tools. In (Kim, K. J.; Kim, H.-Y. Eds.): Information Science and Applications. Springer Singapore, Singapore, pp. 577–586, 2020.

[Wi08] Wimmel, H.: Entscheidbarkeit bei Petri Netzen. Überblick und Kompendium. Springer, Berlin, Heidelberg, 2008.

[WLW17] Wu, Z.; Lu, K.; Wang, X.: Surveying concurrency bug detectors based on types of detected bugs. Science China Information Sciences 3/60, 2017.

[WS06] Wang, L.; Stoller, S. D.: Runtime analysis of atomicity for multithreaded programs. IEEE Transactions on Software Engineering 2/32, pp. 93–110, 2006.

[XLL05] Xu, D.; Li, H.; Lam, C. P.: Using adaptive agents to automatically generate test scenarios from the UML activity diagrams. Proceedings 15-17 December 2005, Taipei, Taiwan. IEEE Computer Society, Los Alamitos Calif., 2005.

[Xu08] Xu, D.; Liu, W.; Liu, Z.; Philbert, N.: Tool Support to Deriving Test Scenarios from UML Activity Diagrams: 2008 International Symposium on Information Science and Engineering. IEEE, pp. 73–76, 2008 - 2008.

[Xu20] Xu, M.; Kashyap, S.; Zhao, H.; Kim, T.: Krace: Data Race Fuzzing for Kernel File Systems: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1643–1660, 2020.

[XZL21] Xiang, D.; Zhao, F.; Liu, Y.: DICER 2.0: A New Model Checker for Data-Flow Errors of Concurrent Software Systems. Mathematics 9/9, p. 966, 2021.

[YRC05] Yu, Y.; Rodeheffer, T.; Chen, W.: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. ACM, New York, NY, 2005.

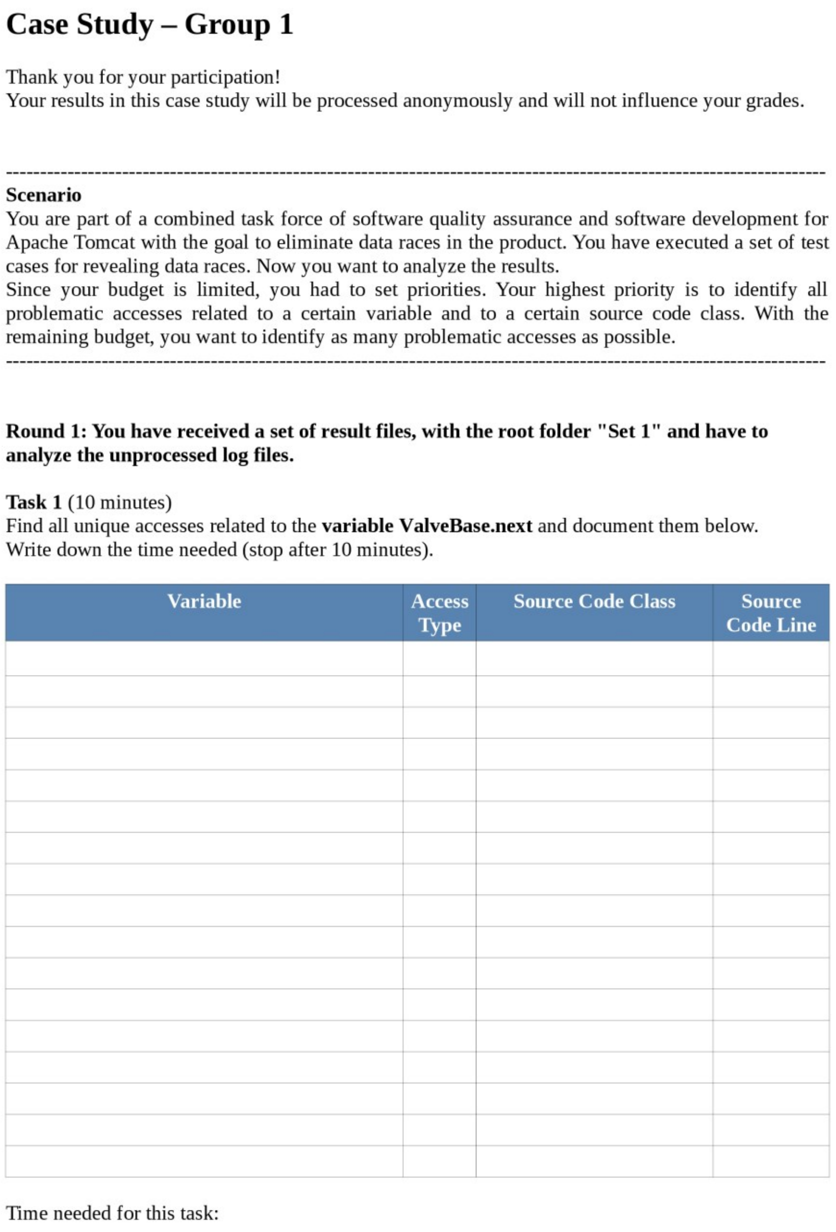
[Zh11] Zhang, J.; Xiong, W.; Liu, Y.; Park, S.; Zhou, Y.; Ma, Z.: ATDetector: improving the accuracy of a commercial data race detector by identifying address transfer. ACM, New York, NY, 2011.

[Zi16] Zimmer, G.: Reduction of UML 2 Activity Diagrams to a Corresponding Representation using a Limited Subset of Elements for Concurrency Analysis. Master Thesis, Kaiserslautern, 2016.

[ZSL10] Zhang, W.; Sun, C.; Lu, S.: ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems, March 13-17, 2010, Pittsburgh, PA, USA. ACM Press, New York N.Y., 2010.

Appendix

Appendix A: Task List for the Controlled Experiment – Group 1



Case Study – Group 1

Thank you for your participation!
Your results in this case study will be processed anonymously and will not influence your grades.

Scenario
You are part of a combined task force of software quality assurance and software development for Apache Tomcat with the goal to eliminate data races in the product. You have executed a set of test cases for revealing data races. Now you want to analyze the results.
Since your budget is limited, you had to set priorities. Your highest priority is to identify all problematic accesses related to a certain variable and to a certain source code class. With the remaining budget, you want to identify as many problematic accesses as possible.

Round 1: You have received a set of result files, with the root folder "Set 1" and have to analyze the unprocessed log files.

Task 1 (10 minutes)
Find all unique accesses related to the **variable ValveBase.next** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 2 (10 minutes)
Find all unique accesses related to the **class ValveBase.java** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 3 (10 minutes)
Find **as many unique accesses as possible** (including results from previous tasks).
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Round 2: You have received a set of result files, with the root folder "Set 2" and have to analyze the preprocessed results using the Data Race Analysis Tool.

Task 1 (10 minutes)
Find all unique accesses related to the **variable StandardContext.cookieProcessor** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 2 (10 minutes)
Find all unique accesses related to the **class WsFrameClient.java** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 3 (10 minutes)
Find **as many unique accesses as possible** (including results from previous tasks).
Write down the time needed (stop after 10 minutes).

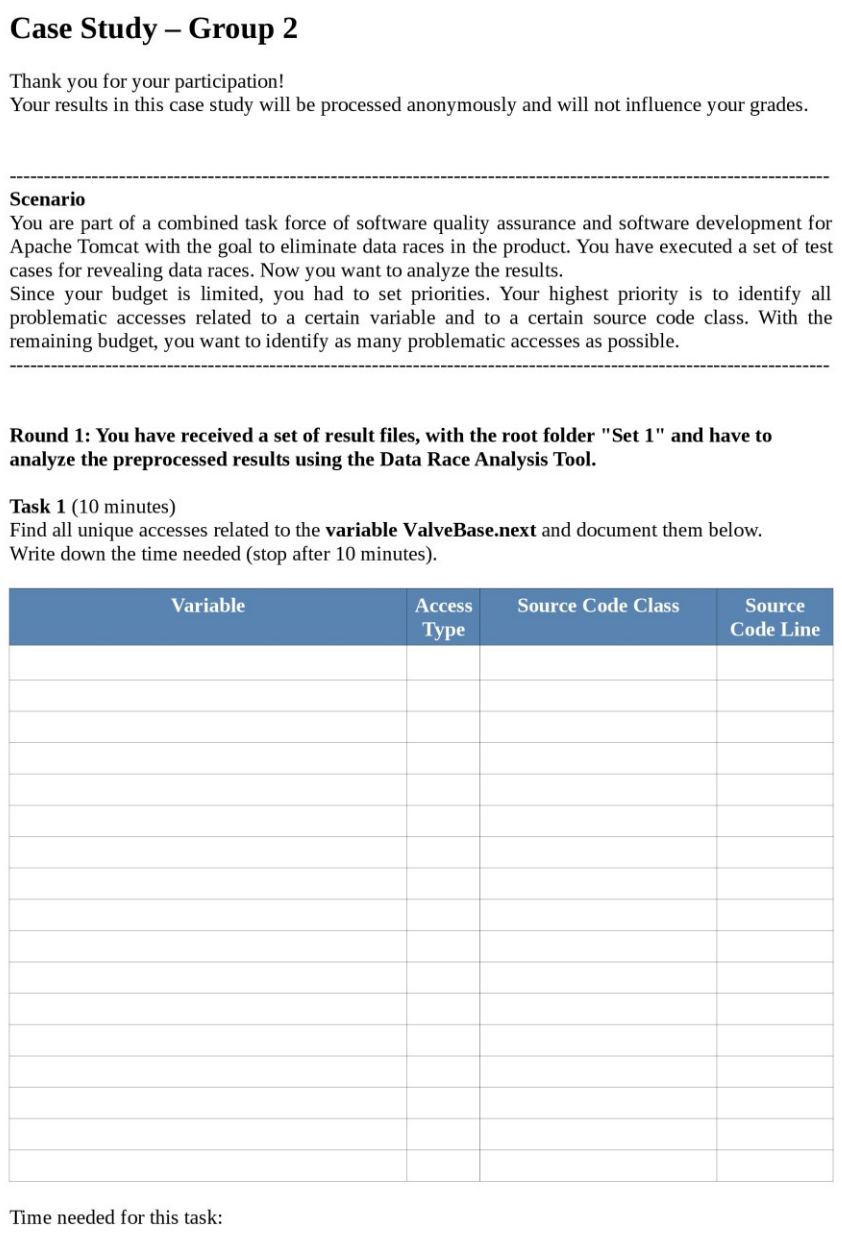
Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Thank you very much for participating in this case study!

Appendix B: Task List for the Controlled Experiment – Group 2



Case Study – Group 2

Thank you for your participation!
Your results in this case study will be processed anonymously and will not influence your grades.

Scenario
You are part of a combined task force of software quality assurance and software development for Apache Tomcat with the goal to eliminate data races in the product. You have executed a set of test cases for revealing data races. Now you want to analyze the results.
Since your budget is limited, you had to set priorities. Your highest priority is to identify all problematic accesses related to a certain variable and to a certain source code class. With the remaining budget, you want to identify as many problematic accesses as possible.

Round 1: You have received a set of result files, with the root folder "Set 1" and have to analyze the preprocessed results using the Data Race Analysis Tool.

Task 1 (10 minutes)
Find all unique accesses related to the **variable ValveBase.next** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 2 (10 minutes)
Find all unique accesses related to the **class ValveBase.java** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 3 (10 minutes)
Find **as many unique accesses as possible** (including results from previous tasks).
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Round 2: You have received a set of result files, with the root folder "Set 2" and have to analyze the unprocessed log files.

Task 1 (10 minutes)
Find all unique accesses related to the **variable StandardContext.cookieProcessor** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 2 (10 minutes)
Find all unique accesses related to the **class WsFrameClient.java** and document them below.
Write down the time needed (stop after 10 minutes).

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Task 3 (10 minutes)
Find **as many unique accesses as possible** (including results from previous tasks).
Write down the time needed (stop after 10 minutes).

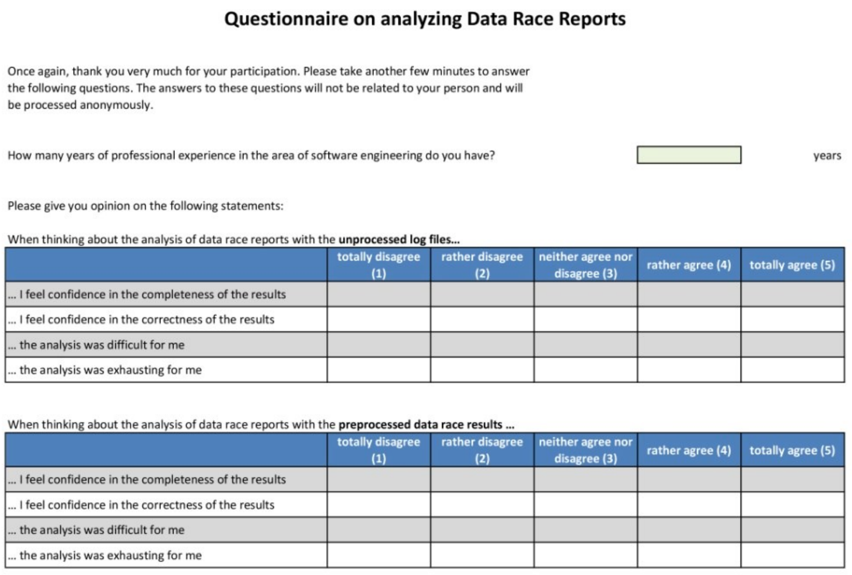
Variable	Access Type	Source Code Class	Source Code Line

Variable	Access Type	Source Code Class	Source Code Line

Time needed for this task:

Thank you very much for participating in this case study!

Appendix C: Questionnaire for the Controlled Experiment



Questionnaire on analyzing Data Race Reports

Once again, thank you very much for your participation. Please take another few minutes to answer the following questions. The answers to these questions will not be related to your person and will be processed anonymously.

How many years of professional experience in the area of software engineering do you have? ______ years

Please give you opinion on the following statements:

When thinking about the analysis of data race reports with the **unprocessed log files...**

	totally disagree (1)	rather disagree (2)	neither agree nor disagree (3)	rather agree (4)	totally agree (5)
... I feel confidence in the completeness of the results					
... I feel confidence in the correctness of the results					
... the analysis was difficult for me					
... the analysis was exhausting for me					

When thinking about the analysis of data race reports with the **preprocessed data race results ...**

	totally disagree (1)	rather disagree (2)	neither agree nor disagree (3)	rather agree (4)	totally agree (5)
... I feel confidence in the completeness of the results					
... I feel confidence in the correctness of the results					
... the analysis was difficult for me					
... the analysis was exhausting for me					

Lebenslauf

Name	Alexander Klaus	
Schulbildung	1986-1990	Carl-Bosch-Grundschule, Frankenthal (Pfalz)
	1990-1999	Albert-Einstein-Gymnasium, Frankenthal (Pfalz) Abschluss: Abitur
Studium	2000-2007	Studium der Wirtschaftsinformatik Universität Mannheim Abschluss: Diplom
Berufstätigkeit	2007-2016	Wissenschaftlicher Mitarbeiter Fraunhofer Institut für Experimentelles Software Engineering, Kaiserslautern
	2016-heute	Angestellter EXCO GmbH, Frankenthal (Pfalz)

Maxdorf, den 08. April 2022