HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

Inter-Procedural Static C++ Concurrency Checker

Bachelor Thesis

Department of Computer Science
University of Applied Sciences Rapperswil

Spring Semester 2017

Students:	Frank Jordi, Hansruedi Patzen, Fabian Schläpfer
Supervisor:	Prof. Peter Sommerlad
Technical Advisor:	Mario Meili
Expert:	Martin Botzler, Siemens AG
Proofreader:	Prof. Oliver Augenstein
Time period:	20.02.2017 - 16.06.2017

Abstract

The rise of concurrent and parallel programming has lead to an increased demand for native programming language support. C++ added a wide variety of these constructs to the standard library starting with C++11. When using concurrency one needs to be careful not to introduce data races. Data races are dangerous because they are undefined behavior and, due to their non deterministic nature, hard to find and solve. There are different approaches to detect them. One approach is to do a dynamic analysis which comes with a run-time overhead. Another approach is to do a static analysis by doing a control flow analysis using the raw source code. This bachelor thesis takes the ConditionR tool, a static data race analysis prototype developed in a master thesis, and develops it further to make it usable with real world projects. An open source project with at least one known data race was chosen to evaluate the capabilities at the beginning and the end of the thesis. To do so the three main parts of the prototype need to be developed further. The groundwork is done in a Clang Static Analysis checker which outputs the control flow graph needed for the data race detection. The analysis is written in Scala based on an algorithm patented by Prof. Dr. Luc Bläser. Finally the detected data races are reported to the Cevelop IDE plug-in. The plug-in visualizes the results, giving developers an overview of all found races and their relation with each other.

Management Summary

Introduction

The rise of multi-core systems has lead programming paradigms to shift towards concurrent and parallel programming. Most programming languages support multi-threaded programming by providing concurrent program constructs. C++ introduced native concurrency support with the C++11 standard. Prior to that the POSIX threads C library was typically used for this kind of application. One pitfall of parallel programming is that it opens up programs to new error types like data races or deadlocks. Data races are especially dangerous because they result in undefined behavior. They are also non-deterministic in nature and can manifest themselves in different places at different times making them hard to find and solve.

Approach / Technologies

There are multiple approaches to find data races. One way is to do a run-time analysis whereby memory access is checked while the program is running. In addition to the need for specific compilers supporting this feature this creates an additional overhead that can lead to missing a data race due to timing differences with the original code. Another approach is to use static analysis to check the program on a source code basis. This creates no additional run-time overhead but the implementation can lead to many false positives or worse false negatives missing the detection. The ConditionR, which started as a master thesis with additional features done in a semester thesis, is a project which uses static analysis to check C++ code for data races. It does so using the Clang static analysis framework to create a control flow graph which is analyzed using a patented algorithm by Prof. Dr. Luc Bläser implemented in Scala. The results are then visualized inside the Cevelop IDE.

Results

The ConditionR tool handles a wide variety of C++ specific constructs. The original version, being a basic prototype, had many shortcomings which are now mostly eliminated. This includes the support for multiple translation units and function calls across translation units. It now also supports class member functions and variables and lambdas. The data race visualization inside the Cevelop IDE has been completely revamped and improved to allow for an easy overview of the given problems found inside a project. Testing and continuous integration has also been updated and fully automated. The original goal of getting the project into a market ready state could not be completely reached due to an overestimation of the already implemented capabilities at the beginning of the thesis.

Outlook

With the current state of the ConditionR project one can analyze simple C++ projects with some restriction. The current approach with using a Clang static analyzer checker should probably be reconsidered. Simply because it creates a strong dependency on a specific compiler and version. Also Clang has problems when dealing with loops which could be handled more easily when dealing with the abstract syntax tree directly. The checker is also missing support for some high-level C++ constructs like `std::async` or `std::future`. Lambdas are only fully supported when defined directly in the `std::thread` constructor, otherwise its captures are not recognized. The current version has some difficulty dealing with large control flow graphs that cause high memory usage making it unusable with bigger projects. Finally, there is always the option to implement deadlock detection as described in the patented algorithm.

Contents

1 Introduction

1.1 Motivation

The rise of multi-core systems has lead programming paradigms to shift towards concurrent and parallel programming. Most programming languages support multi-threaded programming by natively providing concurrent programming constructs such as threads. Concurrent software may suffer from unknown data races, which lead to dangerous undefined behavior and data corruption.

1.2 Context

A prototype of the ConditionR tool was created in 2015 in a masters thesis by Silvano Brugnoni [Bru16]. It is a software to detect data races in C++ projects. The implementation of the algorithm for the data race detection is based on a patented algorithm by Prof. Dr. Luc Bläser. The plug-in could initially only handle basic concurrency concepts like `std::thread` and `std::mutex`. Multiple translation units (TUs), lambdas and function calls with arguments were not supported. Further, the visualization in Eclipse was not very sophisticated and was not usable on Linux.

In 2016, during a student research thesis by Fabian Schläpfer and Samuel Jost [SJ16], more concepts like lambdas were investigated and implemented. Additionally, new visualization concepts were developed and many dependencies like Clang, LLVM or Eclipse were upgraded.

1.3 Goals

The goal of this bachelor thesis is to develop the current ConditionR prototype further towards a market-ready product. To achieve this, the visualization has to be reworked and improved. To allow the usage in a real world environment, the support of multiple TUs is necessary and the algorithm must be extended accordingly. It should further be possible to detect POSIX read/write locks and C++11 lambdas. To ensure the real world applicability of the plug-in, it is evaluated with open-source projects at the beginning and the end of the thesis.

The thesis goals are divided into minimal and extended goals. The minimal goals should be reached during the bachelor thesis for it to be considered a success. The extended goals may be addressed in case there is enough time available after completing the minimal goals.

- Minimal goals:
  - Usable and appealing visualization.
  - Cross-Translation-Unit: Allow the analysis of a project containing multiple source files.
  - Suport for POSIX read/write locks.
  - Lambdas as threads (capture lists).
  - Selection of suitable open-source projects to evaluate the ConditionR in a real world environment.
- Extended goals:
  - Support for C++14/17 read/write locks.
  - Lambdas as functions.
  - Detection according the lockable requirements (see C++ standard).
  - Evaluation of the ConditionR tool using the selected projects.

1.4 Background

This section gives an overview of topics relevant to static analysis and data race detection and introduces some terminology used in the later chapters.

1.4.1 Flow Graphs

A *flow graph* is a graph that represents all possible execution paths through a program.

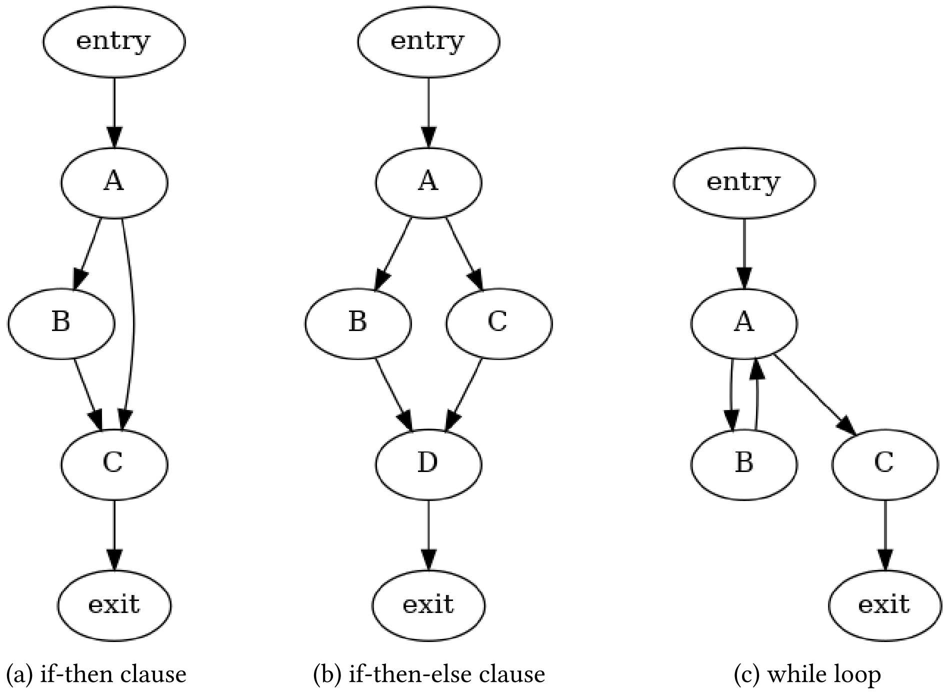


Figure 1.1: Theoretical flow graph examples. (a) depicts an if-then clause, where one of the execution paths includes B and one does not. (b) shows an if-then-else clause, where one path contains B and the other contains C. (c) corresponds to a while loop, whose statements are evaluated at least zero, up to an infinite number of times.

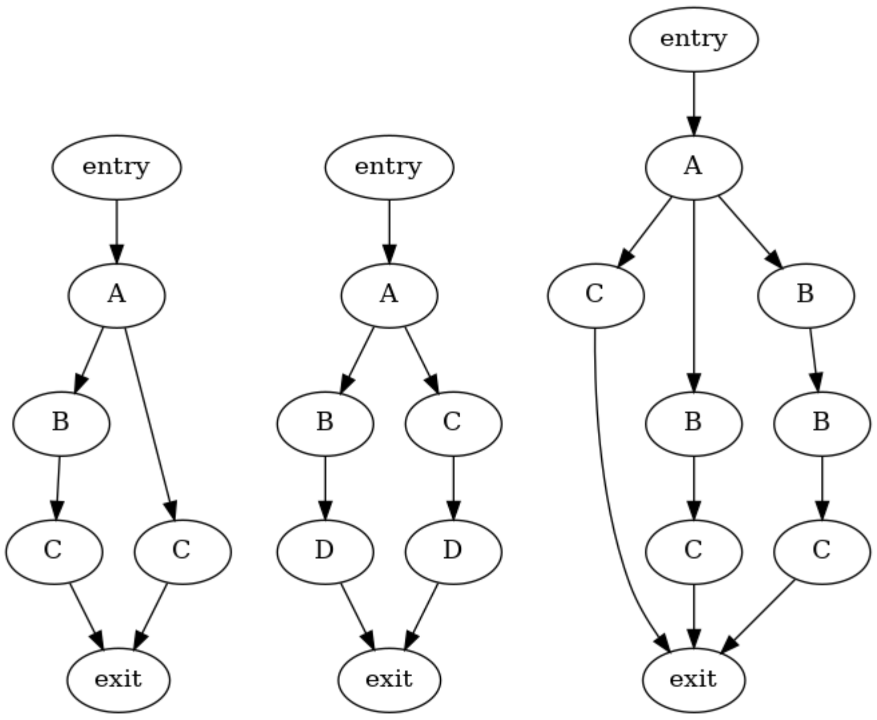
The nodes of the flow graph represent the *basic blocks* of the program. A basic block is defined as a sequence of statements that contains no jump or jump target except at the entry or exit of the block.

Each flow graph begins with a special *entry* node that itself has no predecessors, and has the first actual block of the graph as its successor. Similarly, a special *exit* node completes each flow graph, having as its predecessors all final blocks of the execution paths.

Figure 1.1 visualizes three examples of simple fragments of code and their representation as a flow graph. It must be noted, however, that these graphs are a theoretical representation. In practice, conditional blocks and potential loop iterations are unrolled into explicit separate paths of the following basic blocks. Figure 1.2 shows the actual flow graphs constructed by the extraction component for the same graphs discussed above.

1.4.2 Data-flow Analysis

The data race detection algorithm employs a technique called *data-flow analysis* to find data races. This technique regards the execution of a program as



(a) if-then clause (b) if-then-else clause (c) while loop

Figure 1.2: Actual flow graph examples. (a) depicts an if-then clause, where one of the execution paths includes B and one does not. (b) shows an if-then-else clause, where one path contains B and the branch successors and the other contains C and the branch successors. (c) corresponds to a while loop, whose statements B are evaluated an unpredictable number of times. In practice, branches are created for a fixed number of potential iterations.

a series of state transformations applied to an initial empty state. Each statement transforms the program state. Every program state is represented as a *program point* that contains additional information such as the threads active in this state. A sequence of program points corresponds to an *execution path* through a program.

Flow-Sensitivity

An analysis is considered *flow-sensitive* if the order of instructions of the program under analysis is relevant. Data-flow analysis is flow-sensitive as the series of state transformations have a well-defined order that is taken into account during the analysis.

Path-Sensitivity

An analysis is further considered *path-sensitive* if all possible execution paths are taken into account. This improves the accuracy of an analysis as more of the complexity of a program is captured. A disadvantage is that the number of execution paths grows exponentially with the number of branches in the program.

2 Architecture

This chapter provides an overview of the architecture of the Software. It describes shortly the three components of the ConditionR tool, the extraction part, the analysis part and the Eclipse CDT visualization plug-in. For a detailed description of the single parts please refer to their chapters.

2.1 Overview

The ConditionR tool comprises three components to extract the flow graph from the C++ code, find data races in there and display them for a user.

2.1.1 Communication

The communication between the components is described in the following list. Figure 2.1 visualizes the described communication.

1. A user starts the data race analysis in the Eclipse CDT plug-in with a command.
2. The Eclipse plug-in locates all relevant source files and sends them together with a Clang configuration to the algorithm module.
3. The extractor is passed the source files together with the Clang configuration.
4. The extractor determines the required includes for Clang and calls it with the Command Line Interface (CLI) flags.
5. The extraction checker passes all control flow graphs to the extractor.
6. The extractor post processes the control flow graphs into a valid JSON and delivers them to the data race detection algorithm.
7. The detected data race source locations and call stack are returned to the visualization plug-in.

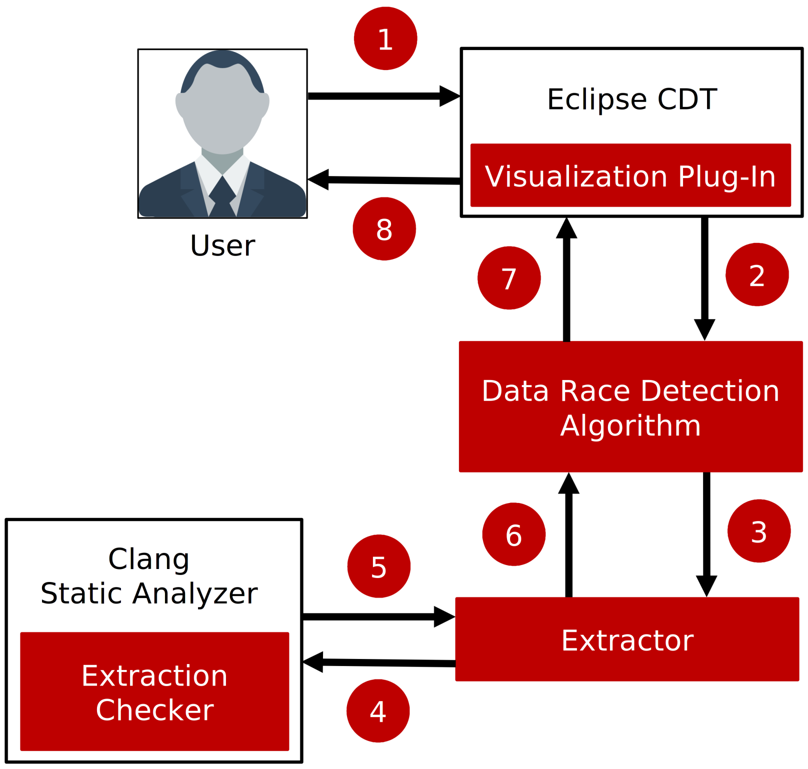


Figure 2.1: Communication and interaction between the components of the ConditionR software. This figure was adapted from the student research thesis.

8. The potential data races are displayed to the user in a data race view and as code markers in the editor.

2.1.2 Automated Integration Testing

To ensure that the latest versions of all components work together properly, an integration test was put in place on the build server. The integration test verifies that the communication between all components, as described in section 2.1.1, works as expected.

2.2 Extraction

The extraction component is responsible for extracting the control flow graph from the received source code files. It receives the source code files to be extracted and a Clang configuration from the algorithm and sends back the extracted control flow graph in the JSON format. The extractor is written in C++.

2.3 Algorithm

The algorithm receives the control flow graph as JSON from the extraction component. It performs the data race analysis based on the control flow graph. After the analysis, the detected potential data races are sent to the visualization plug-in. The algorithm is written in Scala and is an implementation of the patented algorithm by Prof. Dr. Luc Bläser.

2.4 Visualization Eclipse Plug-In

The visualization plug-in is the point of interaction between the user and the analysis tool. It starts the data race analysis and displays the results. The plug-in is written in Java and extends the Eclipse CDT user interface with a data race view, some commands and a preference page.

3 Initial Evaluation

This chapter describes the evaluation of the ConditionR tool using real-world open source software projects.

At the outset of this thesis, an initial evaluation was conducted. This was done to determine the initial capabilities of the ConditionR tool. At the end of this thesis, a final evaluation was carried out. Its purpose was to capture the work achieved in this thesis and also to uncover any remaining shortcomings that are yet to be resolved in the future.

3.1 Motivation

The motivation behind this initial evaluation is multifold.

First of all, the initial capabilities of the tool were somewhat unclear. The masters thesis hints at many open issues and deficits but does not provide detailed information as to what features exactly are missing and what the behavior of the tool is when an unsupported concurrency construct or language feature is encountered.

Furthermore, ConditionR has never been used with real world source code, but only with a relatively small number of hand-crafted test cases. In the past it has become clear that simply changing minor details in the source code of a test case will lead to a crash of the checker or algorithm. Using the tool with actual real-world code will hopefully uncover many more such issues, which can then be used to construct a list of deficiencies in the initial implementation. From that, a plan of action can then be derived.

Lastly, the visualization has never been tested using more than a handful of data races. The conservative nature of the algorithm implies that a potentially large number of false positives are reported. This will allow the visualization to be judged on the basis of how well it scales with the number of data races.

3.2 Project Selection

Before conducting an evaluation, a list of suitable candidate open source projects needed to be determined. From these candidates, one project was then selected to be the subject of the initial evaluation. Using that project, the evaluation was then conducted and its results were captured.

3.2.1 Methodology

An initial list of candidate projects was obtained using GitHub. GitHub was searched for issues containing the keywords "*data race*". The associated projects were then assessed on the basis on a number of criteria explained in detail in section 3.2.2.

The reason behind focusing on open source projects with at least one known data race is that this allows the evaluation to verify that ConditionR is indeed able to find that specific data race. Otherwise, a data race would first have to be found manually, which could prove to be quite time consuming.

3.2.2 Criteria

The following criteria were used to assess the suitability of projects:

Complexity For this initial evaluation, the selected project should be fairly complex but not too complex. This is, on the one hand, due to the limited capabilities of the initial version of ConditionR. On the other hand, the time spent for this evaluation should be minimized such that there is enough time to implement new features and to actually achieve an improvement over the initial version. Selecting a project that is not too complex minimizes the time needed to understand the code and judge whether a reported data race is correct or a false positive.

Checker support The extraction checker in its initial version only supported a subset of C++11 language features and concurrency constructs. Analyzing code that makes use of elements not supported by the checker is pointless as this will make it impossible to extract any useful information from that code at all.

Popularity It is desirable that the selected project be somewhat popular. This is viewed as an indirect measure of the code being somewhat actively maintained and representative of real world source code used in production systems.

3.2.3 Candidates

The following candidates were determined using the methodology described in section 3.2.1.

OpenCV *A popular computer vision library.*
Uses boost::barrier and a custom mutex and lock implementation.
Known race(s):
`https://github.com/opencv/opencv/issues/8149`
`https://github.com/opencv/opencv/issues/5175`

CppMicroServices *An OSGi-like C++ dynamic module system and service registry.*
Uses custom wrapper classes for std::mutex and other C++11 concurrency primitives.
Known race(s):
`https://github.com/CppMicroServices/CppMicroServices/issues/173`

HPX *A C++ concurrency and parallelism library for high-performance computing applications.*
Known race(s):
`https://github.com/STEllAR-GROUP/hpx/issues/2517`

Chess *An AI-driven chess TCP-client.*
Uses std::future, std::async and C++11 lambdas as threads.
Based on `https://github.com/siggame/chess`.
Known race(s):
`https://github.com/BryceMehring/Chess/issues/11`

Etherum *An Etherum client written in C++.*
Uses `std::thread` and `pthread_mutex_lock()`.
Uses a custom wrapper around `std::lock_guard`.

Known race(s):
`https://github.com/ethereum/cpp-ethereum/issues/2287`

OpenSSL *A TLS/SSL cryptography library.*
Uses a custom wrapper around POSIX threads functions.
Also uses POSIX readers/writer locks. It might be possible to disable these by means of `PTHREAD_RWLOCK_INITIALIZER` and `USE_RWLOCK`.
Known race(s):
`https://github.com/openssl/openssl/issues/2457`

highfidelity *A client/server software for creating shared VR environments.*
Known race(s):
`https://github.com/highfidelity/hifi/pull/8859`

cf *A C++ library providing composable futures.*
Known race(s):
`https://github.com/rpz80/cf/issues/2`

3.2.4 Decision

The main criteria used to select a subject for the initial evaluation were those of complexity and checker support.

OpenCV uses various concurrency constructs from *boost*, for example `boost::barrier`. *OpenSSL* uses POSIX readers-writer locks. As these constructs were not initially supported by the checker, these two candidates could not easily be analyzed without spending additional effort to either rewrite the code or implement the missing features in the checker.

All other projects except for the *Chess* project generally were too complex and the already known races were relatively. While evaluating ConditionR with more complex race conditions *is* one of the goals of this thesis, that was scheduled for the final evaluation. This initial evaluation serves the purpose of establishing a simple baseline.

In the end, the *Chess* project emerged as having the right complexity for the initial evaluation. The known data race in that project is a straightforward and classic example of a data race and therefore a good benchmark for an early version of ConditionR. The *Chess* project and the data race in question are explained in detail in section 3.3.1.

3.3 Evaluation

3.3.1 The *Chess* Project

The *Chess* project selected as the subject of this evaluation has a fairly straightforward structure.

The context is a game of chess played over the network. The server is provided by a different software package and the *Chess* project is simply a chess client talking to the server.

The main logic of the chess client is as follows: First, a connection to the server is established and the game is set up. Then, for each turn, an `AI` object is asked to pick the next move given the current board and the game history. The `AI` class does this using a custom parallel implementation of the *Minimax* algorithm.

3.3.2 Preprocessing

Since the initial version of the algorithm only supported a single TU, all TUs found in the project needed to be merged into a single one. This was done by simply concatenating all `.cpp` source files into one resulting source file, that can then be passed to ConditionR for analysis, along with all header files.

Furthermore, the chess project uses `std::async` and `std::futures`. However, the checker in its initial version does not support these constructs yet. Hence these were replaced with `std::threads` and a `std::condition_variable`. It should be noted, however, that this transformation is only applicable due to an implementation detail of the checker. In its initial version, `std::condition_variables` are not officially supported.

3.3.3 Problems

Identical Memory Location Names for Distinct Memory Locations

It quickly became clear that one major shortcoming of the initial version of ConditionR is that the `memoryLocations` reported by the checker are essentially just variable names. This means that two variables in completely different contexts with the same name are regarded as being the same variable and thus memory location.

Distinct Memory Location Names for Identical Memory Locations

On the other hand, accessing the *same* memory location using *different* identifiers is not recognized as referring to the same memory location. Listing 3.1 shows a read access and a concurrent write access in the source code. Listing 3.2 is the output of the checker for these two statements. The checker is unable to deduce that the target of the read and write access is identical because the identifiers used are different. Consequently, the algorithm is unable to detect the known data race in the chess project.

```
// ...
bool AI::MiniMax(int playerID, bool bCutDepth,
   BoardMove& moveOut)
{
  unsigned int depthLimit = (bCutDepth ? 3 :
     m_depth); // <== Read access here.
    // ...
}

// ...

std::uint64_t AI::GetTimePerMove()
{
// ...
    m_depth = 4; // <== Write access here.
// ...
```

Listing 3.1: Concurrent read and write accesses to the same location in memory.

```
}, {
  "type": "Write",
  "memoryLocation": "&ai->m_depth",
    ...
}, {

...

```

```
}, {
  "type": "Read",
  "memoryLocation": "&SymRegion{reg_$20<ai>}->
     m_depth",
    ...
}, {
```

Listing 3.2: The output of the original checker for listing 3.1.

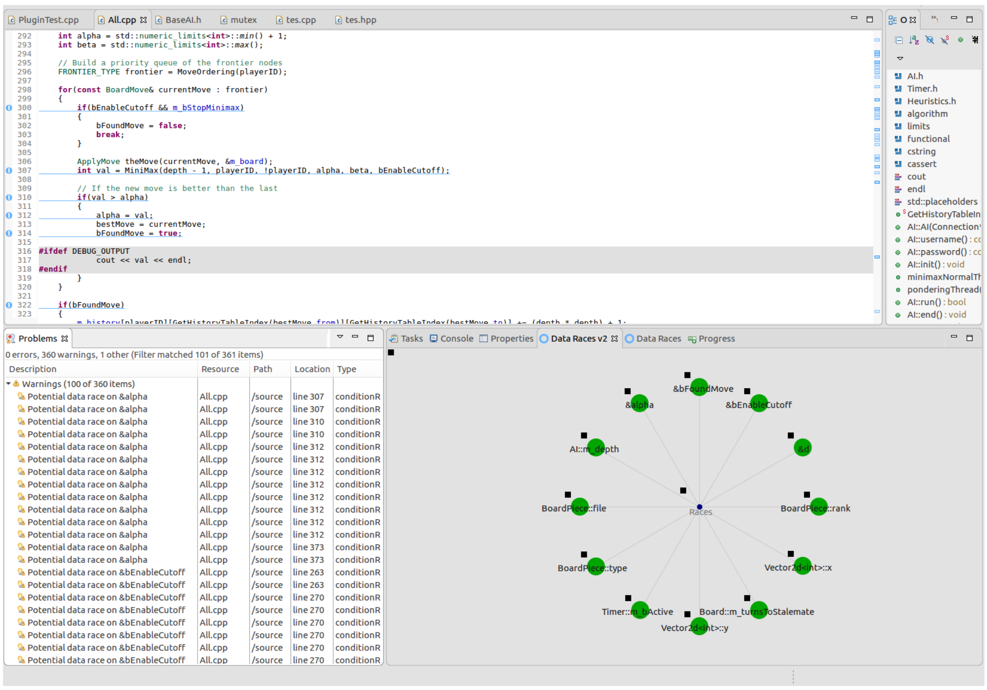
This particular limitation was initially worked around by modifying the checker to output the memory locations of member variables in the format `Class::member`. This allowed the algorithm to recognize that the two concurrent statements access the same memory location and correctly report the data race. The resulting output is shown in listing 3.3.

```
}, {
  "type": "Write",
  "memoryLocation": "AI::m_depth",
    ...
}, {

...


}, {
  "type": "Read",
  "memoryLocation": "AI::m_depth",
    ...
}, {
```

Listing 3.3: The output of the modified checker for listing 3.1.

Program Points Computation

When running the ConditionR tool on the preprocessed chess AI source code for the first time, a stack overflow exception occurred in the algorithm Scala code. A quick investigation revealed that the computation of the program points was implemented in a recursive fashion. However, the function definition was not tail-recursive, which prevented tail call optimization from being applied. As a quick workaround, the amount of memory allocated for the stack was increased from 64MB to 512MB, which allowed the analysis for this specific source code to run without error. Later on, this problem was properly solved by rewriting the function in a tail-recursive way.



```
    int alpha = std::numeric_limits<int>::min() + 1;
    int beta = std::numeric_limits<int>::max();

    // Build a priority queue of the frontier nodes
    FRONTIER_TYPE frontier = MoveOrdering(playerID);

    for(const BoardMove& currentMove : frontier)
    {
        if(bEnableCutoff && m_bStopMinimax)
        {
            bFoundMove = false;
            break;
        }

        ApplyMove theMove(currentMove, &m_board);
        int val = MiniMax(depth - 1, playerID, !playerID, alpha, beta, bEnableCutoff);

        // If the new move is better than the last
        if(val > alpha)
        {
            alpha = val;
            bestMove = currentMove;
            bFoundMove = true;

#ifdef DEBUG_OUTPUT
            cout << val << endl;
#endif
        }
    }

    if(bFoundMove)
    {
        m_history[playerID][GetHistoryTableIndex(bestMove.from)][GetHistoryTableIndex(bestMove.to)] += (depth * depth) + 1;
```

Problems

0 errors, 360 warnings, 1 other (Filter matched 101 of 361 items)

Description	Resource	Path	Location	Type
Warnings (100 of 360 items)				
Potential data race on &alpha	All.cpp	/source	line 307	conditionR
Potential data race on &alpha	All.cpp	/source	line 307	conditionR
Potential data race on &alpha	All.cpp	/source	line 310	conditionR
Potential data race on &alpha	All.cpp	/source	line 310	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 312	conditionR
Potential data race on &alpha	All.cpp	/source	line 373	conditionR
Potential data race on &alpha	All.cpp	/source	line 373	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 263	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 263	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR
Potential data race on &bEnableCutoff	All.cpp	/source	line 270	conditionR

Figure 3.1: Data race visualization of the initial evaluation. Default view displayed when the analysis has finished.

3.4 Results

This section presents the results of the initial evaluation.

3.4.1 Visualization

The visualization of the data races found during the analysis is displayed in figure 3.1. The black squares in the top left corner of every node are an artifact of a temporary workaround to get the visualization to work under Linux and can be ignored. Clicking on on the node representing the memory location `Vector2d<int>::x` on the bottom right results in the view shown in figure 3.2. Expanding on the rightmost access node displays the expanded view shown in figure 3.3.

3.4.2 Data races

The analysis reported 360 data races on 12 unique memory locations. Of those data races, only one is a true data race, namely the race on `AI::m_depth`. The rest are false positives.

On the other hand, a data race on `Connection::playerID` found by clang thread sanitizer was not reported at all.

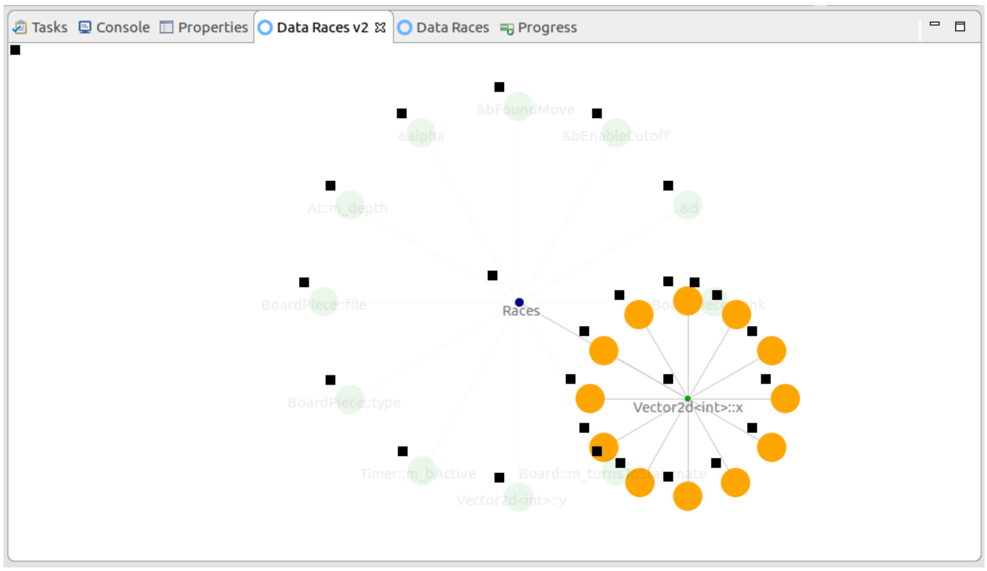


Figure 3.2: Data race visualization of the initial evaluation. View expanded on one data race.

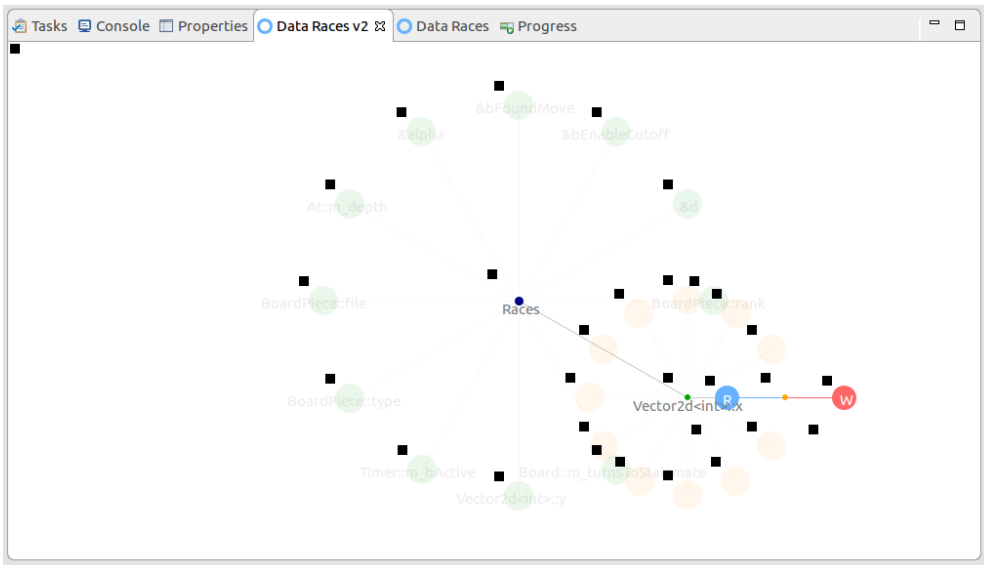


Figure 3.3: Data race visualization of the initial evaluation. View expanded on one memory access of one data race.

3.4.3 Extractor

During the evaluation, a number of bugs in the custom clang extraction checker were found and fixed in order to analyze the entire source code without crashing.

3.5 Conclusion

Clearly, there is a lot of room for improvements in all components.

The visualization currently offers little value above what a trivial list would provide.

The extraction checker uses a very crude memory location naming concept which does not allow the algorithm to properly correlate read and write accesses needed to detect a data race.

The algorithm can only work with a single TU. In the evaluation setup this was not a huge problem as concatenating all source files into one was enough to overcome this limitation in the current version. For real world applications though, multiple TUs with explicit function calls are required.

The final evaluation will show what improvements this thesis was able to deliver.

4 Control Flow Graph Extractor

The extractor does the groundwork for the ConditionR. Its task is to create a control flow graph from C++ source code. This is achieved using the Clang Static Code Analyzer infrastructure. The extractor is written in C++.

4.1 Starting Position

This section deals with the state of the Clang extraction checker condition when this bachelor thesis started. It gives an overview of the capabilities and about the known missing features.

4.1.1 Capabilities

The ConditionR tool the end of the master thesis supported only a small number of C and C++ constructs. The student research thesis added support for POSIX thread start, join and lock statements. The following list gives an overview of all supported constructs:

- Support for races on global variables.
- Basic support for `std::thread` and POSIX threads.
- Basic support for `std::mutex` and POSIX lock and unlock.
- Support for locking mechanisms like `std::lock_guard` due to the function call inlining of Clang.

4.1.2 Missing features

To be able to analyze real world code, there are still quite a few features missing. The following list shows the known shortcomings including the ones found during the thesis:

- No support for C++ variables and function overloading.

- No support for C++ constructors, destructors and operators.
- Incorrect output if a function is found within two different TUs.
- Lambdas can not be analyzed because they are not named.
- When multiple locks are locked, the named mutex will always be named the same. This is because the name is determined based on the variable inside the locks implementation
- Arguments and parameters of a function cannot be mapped due to the Clang memory location naming.
- No data race detection when objects are used.
- Only very basic multi TU support, basically just global variables.

4.2 Requirements

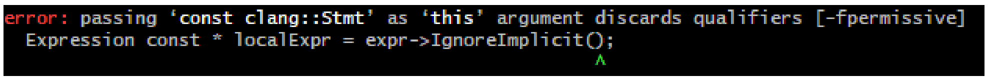
This section covers the requirements to build and run the extractor and checker in its latest form.

4.2.1 Linux

There are many different Linux distributions available and not every one of them provides a built package containing the latest versions of Clang and LLVM. To be able to run all tests successfully and use the extraction checker, Arch Linux or one of its derivative like Manjaro are preferred. These come with the latest versions per default and allow for an easy setup. For distributions like Ubuntu one needs to install the experimental packages to get the correct versions. If no packages are available at all, there is still the possibility to build and install these two tools directly from the source code. If one has enough space and does not want to install Clang and LLVM directly, a docker container could be used as well. (See listing 4.1 for a simple `Dockerfile` example.)

```
FROM base/archlinux
RUN pacman -Syyuu --noconfirm
RUN pacman -S clang llvm cmake make vim --noconfirm
RUN pacman -Sc --noconfirm
```

Listing 4.1: Minimal Dockerfile to create a working clang and llvm environment needed to build the extraction checker.



```
error: passing 'const clang::Stmt' as 'this' argument discards qualifiers [-fpermissive]
  Expression const * localExpr = expr->IgnoreImplicit();
                                                      ^
```

Figure 4.1: Error during the extraction checker build process on Windows using Cygwin and Clang 3.9.1.

4.2.2 MacOS

Running and compiling the checker on MacOS requires that Clang and LLVM with version 4.0.0 is installed on the system. This can be done through either homebrew or using the sources directly. After this, *cmake* will handle the rest except that one needs to define `LLVM_PATH` when running it for the first time.

4.2.3 Windows

There is no official installation binary for Clang and LLVM on Windows. One approach to get it working is to use Cygwin. This method worked up until the memory location name started to be determined by walking through the abstract syntax tree (AST) of the given expression. The error during the compilation is shown in figure 4.1. Because of the different versions of Clang and LLVM this is most likely solved after updating from version 3.9.1 to 4.0.0. Like with MacOS, additional libraries need to be linked during compilation and `LLVM_PATH` must be defined for *cmake* to run. Another approach would be to build it directly from source using Visual Studio. [Cla]

4.3 Architecture

The extractor is built as a Clang Static Code Analyzer checker which processes specific events to create control flow graph JSON output, which can then be passed to the algorithm for further analysis.

4.3.1 Basics

When building a checker, one should follow the guidelines given by the Clang static analyzer. [Che]

- Let the checker inherit from the Clang `Checker<...>` class.
- Specify the events one is interested in, by passing them as a template parameter.

- Register the checker using the `registerChecker<...>()` method and passing it as the only template parameter.

The `registerChecker<...>()` must be called inside the given `clang_registerCheckers(...)` method which has external C linkage. One should further define the clang version the library is compiled against using the `char const clang_analyzerAPIVersionString[]`.

4.3.2 Implementation

Herein follow the specification and implementation details of the extractor code checker. A big picture overview of the whole implementation can be seen in figure 4.2. The checker can be divided in three major parts. The extraction checker itself, the graph classes containing all the building blocks needed in the control flow graph and at separate printing class which buffers and then prints all graphs after each TU.

Extraction

The extraction part contains the actual checker implementation. Figure 4.3 shows which hooks from the `clang::ento::check` namespace the `ExtractionChecker` implements. Initially the checker code was contained inside a single C++ source file. During this thesis the structure has been broken up and the checker hook methods contain little logic themselves. Code belonging to the graph part has been put into the `TreeGraph` class and its dependencies. To print the graph to `std::cout` the `JSONPrinter` class is used. Finally to create the correct statements from a given expression the `PreCallHandler` and `LocationHandler` classes are called.

Graph

The graph part provides the data structure which will later be serialized. The `TreeGraph` holds a list of all `BasicBlocks` and also the special entry and exit block. The list containing the blocks is only used during the initial extraction because the checker only saves the list index of the last `BasicBlock` in its state. When using a deserialized graph, the entry block must be used directly. If a list containing all blocks is needed the `getBlocks()` function can be used. This function flattens the graph data structure starting with the entry block in order. Since each graph represents a single function it further contains a list of `Parameter` objects and a `SourceLocation`. The latter allows

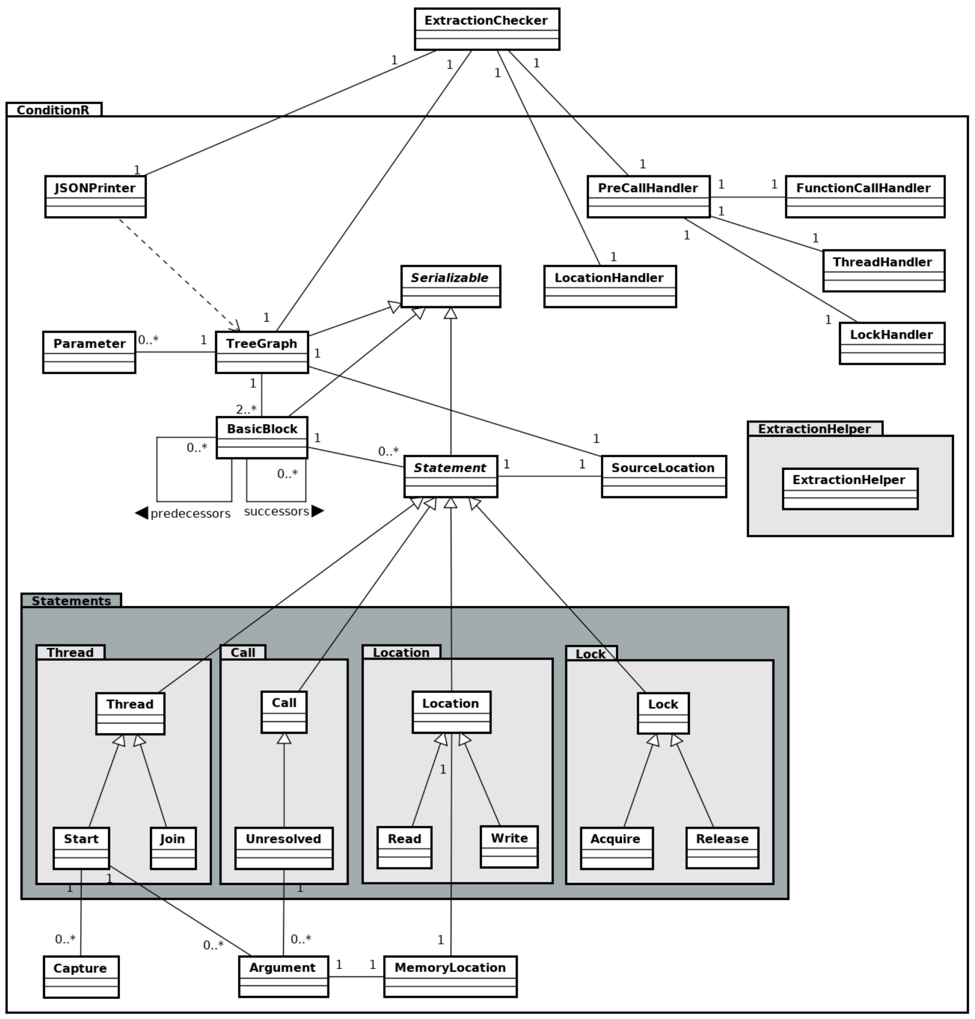


Figure 4.2: Extraction checker global overview.

the algorithm to determine the entry point the user has selected inside the Cevelop IDE. Figure 4.4 gives a basic overview of the dependencies to other classes.

Printer

The `JSONPrinter` buffers the graph of each function after it has been analyzed. When the TU is fully processed, the graphs will then be printed to `std::cout`. figure 4.5 shows that the current implementation depends directly on `rapidjson` [Rap] as a back end for serialization. This strong dependency to a third party library can be weakened if an adapter is used. This allows for an easy replacement if another JSON library is needed.

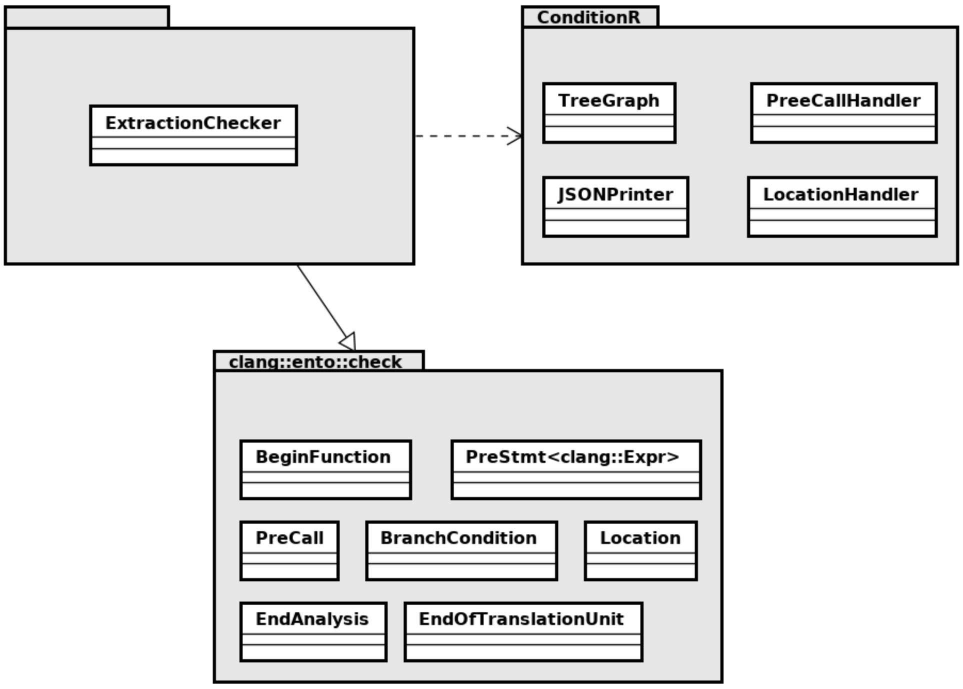


Figure 4.3: Extraction plug-in extraction overview showing the direct dependencies from the Clang `Checker` class and also the classes used to implement most of the logic.

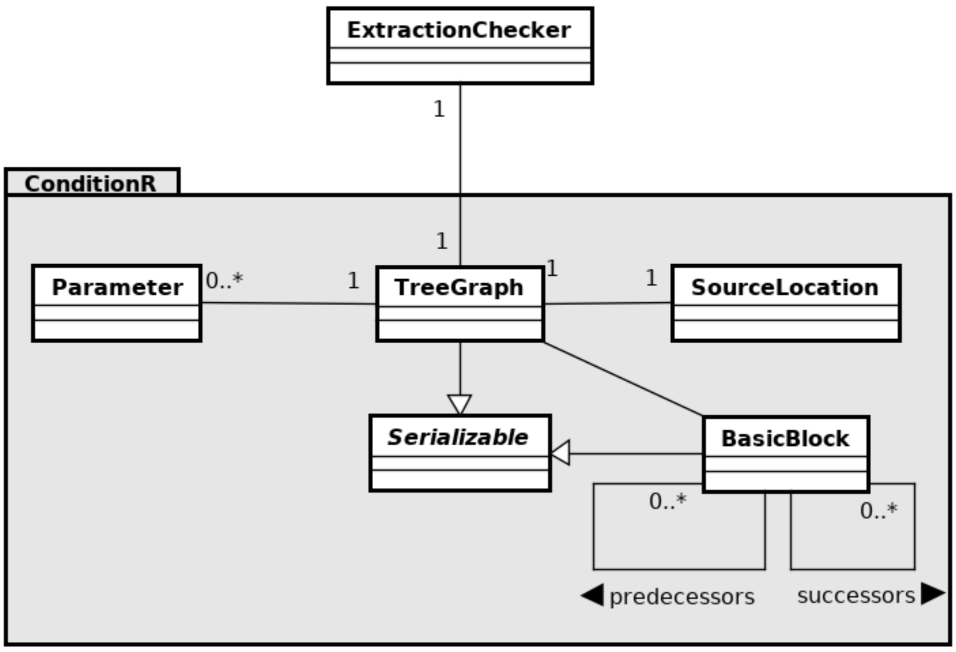


Figure 4.4: Extraction graph part overview with its dependencies.

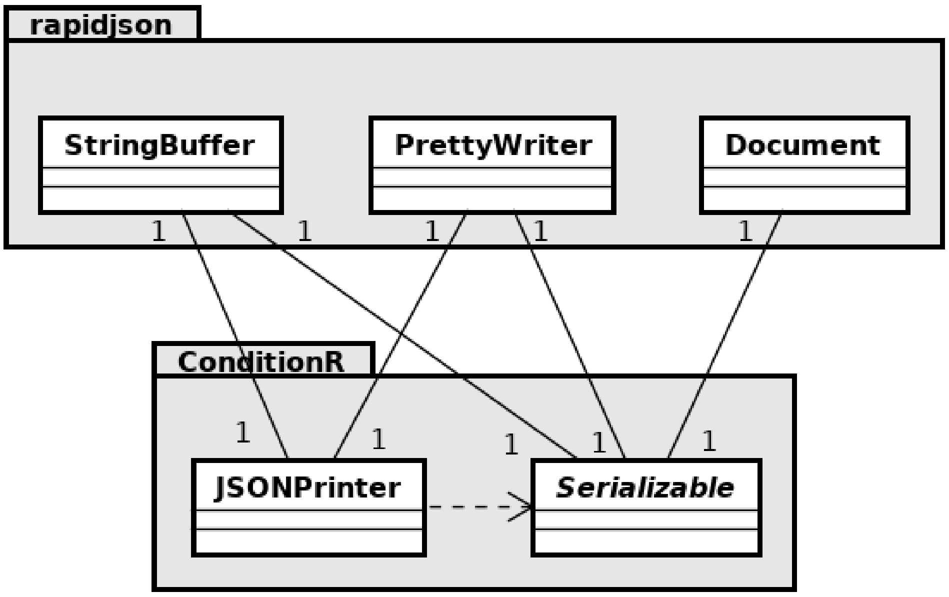


Figure 4.5: Extraction checker printer part overview, showing the dependency on the rapidjson library.

4.4 Extraction

After analyzing the architecture of the checker in section 4.3, this section will give an overview of how the user or the algorithm can interact with the extractor.

4.4.1 Extractor

The extractor is a small CLI tool created to allow for an easier interaction with the actual Clang checker. Normally the plug-in must be called using a lot of arguments. Some of those, like the system include paths, must be manually looked up on each system. Additionally some post processing of the checker output, like fixing the invalid JSON output is done. If desired, all `UnresolvedCall` nodes, which point to graphs not contained inside the checker output, can be removed. For additional space optimization JSON output can be compressed by removing all empty basic blocks. Refer to table 4.1 for a full list of supported CLI arguments.

As a word of warning, one must be careful not to pass malicious content as the Clang binary or checker library argument because it will be invoked directly inside a `std::system` call.

Figure 4.6 gives an overview on how the extractor is used within the ConditionR tool. It can also be used directly from the CLI to get the JSON data, if the required arguments are provided.

`Usage:`	
`-c <clang binary>`	`[Required] Clang binary which is used to extract the CFG.`
`--clang`	`See -c.`
`-l <extraction library>`	`[Required] Extraction checker library which needs to be loaded.`
`--checker-lib`	`See -l.`
`-f <list of files>`	`[Required] Files which need to be analyzed.`
`--files`	`See -f.`
`-o "<more flags>"`	`Further optional flags. (Example -stdlib=libc++). Must be enclosed in "".`
`--other-flags`	`See -o.`
`--raw-output`	`Get the raw checker plug-in output (JSON format will still be fixed).`
`--fno-ucall-remove`	`Do not remove calls for which there are no graphs found."`
`--fno-emptyblock-remove`	`Do not remove blocks containing no statements."`
`-h`	`Show this usage message.`
`--help`	`See -h.`

Table 4.1: Usage of the `cr-extractor` tool.

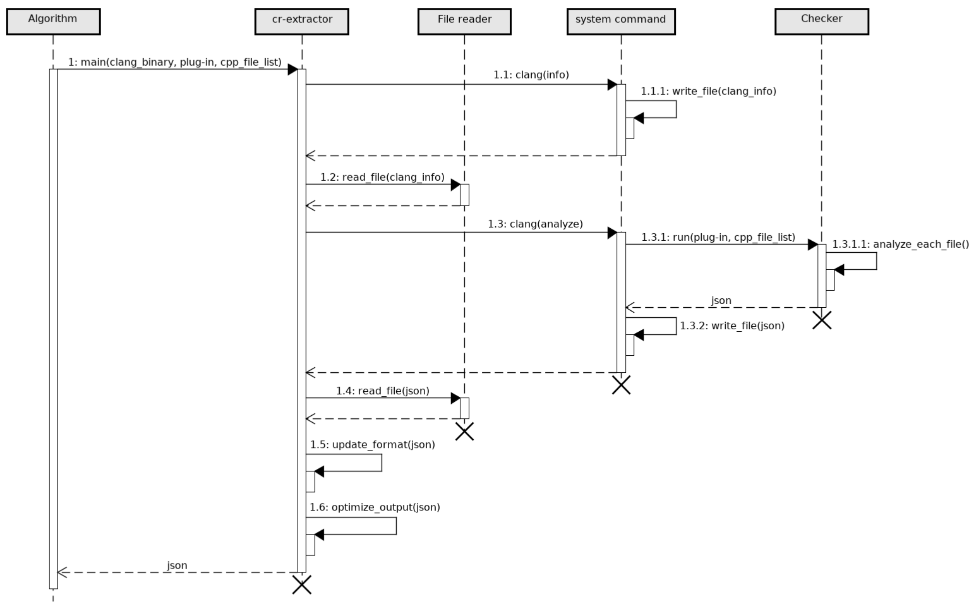

Algorithm
cr-extractor
File reader
system command
Checker
1: main(clang_binary, plug-in, cpp_file_list)
1.1: clang(info)
1.1.1: write_file(clang_info)
1.2: read_file(clang_info)
1.3: clang(analyze)
1.3.1: run(plug-in, cpp_file_list)
1.3.1.1: analyze_each_file()
json
1.3.2: write_file(json)
1.4: read_file(json)
1.5: update_format(json)
1.6: optimize_output(json)
json

Figure 4.6: Sequence diagram showing the interaction between the algorithm, extractor and the checker

4.4.2 Checker

The Clang Static Analyzer Checker library has been completely refactored. When this thesis started, everything was contained inside a single C++ file, making it difficult to get a good overview of the current code base. This file was also included inside the test suite but never actually used for any tests. Which meant that every required Clang and LLVM library had to be linked with it as well.

During the thesis the code has been separated and put into different classes inside the ConditionR namespace. For general methods, used in different parts of the extraction, the `ExtractionHelper` namespace was created. The existing functionality has been moved into the `PreCallHandler` and `LocationHandler` classes to keep files at a manageable size.

Major Changes

The biggest change was the removal of Clangs function inlining capabilities. This was done to allow the output to be consistent when analyzing multiple translation units. Normally Clang would follow a call if it was in the same translation unit and generate `UnresolvedCall` nodes otherwise. This created problems with memory locations since they had different names and no correlation could be found anymore, which meant no data race could be detected even though the same memory location had been accessed.

The JSON output changed significantly as well. New nodes had to be added to each graph for example a block for the current function location and parameters. The `UnresolvedCall` and `ThreadStart` statements now also contain a list of argument objects and a `MemoryLocation` if it's a named variable.

4.5 Output

After the analysis of each function is done, the checker serializes the current `TreeGraph` and buffers its output in a `rapidjson::StringBuffer` object inside the `JSONPrinter` class. The buffer will then be printed to `std::cout` after every TU with an added comma. This allows the extractor to simply read the output, remove the last comma and add '[' before and ']' after the string to get a valid JSON. The definition of this output can be found in appendix A

`Usage:`	
`-c <clang binary>`	`[Required] Clang binary which is used to extract the CFG.`
`--clang`	`See -c.`
`-l <extraction library>`	`[Required] Extraction checker library which needs to be loaded.`
`--checker-lib`	`See -l.`
`-o "<other flags>"`	`Further optional flags. (Example -stdlib=libc++). Must be enclosed in "".`
`--other-flags`	`See -o.`
`--update-test-data`	`Update the JSON test data for all tests.`
`-h`	`Show this usage message.`
`--help`	`See -h.`

Table 4.2: Usage of the `cfg-extraction-test`.

4.6 Testing

To test that the extraction checker actually works, the `cfg-extraction-test` project has been updated and populated with more test cases. The old CLI has also been updated. Refer to table 4.2 to get an overview of the available CLI arguments. Since *cfg-extraction-test* actually calls the *cr-extractor* the same precautions concerning the clang binary and the extraction checker library must be taken. The `--update-test-data` argument can be used to update all the current test data. This can further be used when the output format or the output in general has changed. Of course one needs to validate that the new data is actually correct, otherwise the whole point of these tests would be useless. Updating the test data manually is labor intensive due to the memory locations using file byte offsets which are tedious to determine. The JSON output will get quite big when branching or loops occur. Every test calls Clang with the relevant flags using the header files of the extractor and passing it the relevant arguments. All tests use the raw and not the post-processed output. In a future update the tests could be updated to allow using the graphs directly and check their correctness without relying on simple string comparisons.

4.7 Future Development

This section describes future development of the extraction checker and the extractor. The current version is still not fully finished and maybe some concepts need to be rethought to move it into a market ready state.

4.7.1 Loop handling

Handling loops is a known issue with the clang project, that is not fully resolved [Loo]. For example listing 4.2 will not generate the correct statements, meaning that the `z = 200` write statement will never be recorded since Clang determines that the loop will run at maximum 20 times but the extractor limits it to two iterations.

```
int main() {
  int z { };
  for (int i { 0 }; i < 20; ++i) {
    if (i > 10) {
      z = 200;
    }
  }
}
```

Listing 4.2: Code where the current solution to the loop issues results in a wrong graph.

A possible solution to most of the problems concerning Clang could be to work directly with the AST instead of relying on Clang to do the control flow analysis. To get this working the extraction checker could likely be replaced entirely.

4.7.2 Deployment

Deployment is tricky as described in section 4.2. The hard dependency on specific Clang versions with their constant API changes makes stable development difficult. One possible solution could be to bundle Clang with the checker but this requires further research.

5 Data Race Detection Algorithm

This chapter describes the algorithm component of the ConditionR tool.

First, an overview of the components history and scope is given. Then, the phases of the data race detection algorithm are described. Finally, the work done in this thesis and potential future work is summarized.

5.1 Overview

Once the extraction component has extracted the control flow graph from the source code of the project under analysis, the graph must be further analyzed in order to detect potential data races. These data races are then reported to the visualization component, which in turn displays the analysis results to the user.

The algorithm was invented and patented by Dr. Luc Bläser. [Blä15] The implementation used in this project is written in Scala and was created in the masters thesis.

This chapter gives a high level overview of the algorithm and a detailed description of the aspects modified or added in this thesis. For a detailed overview of the inner workings of the algorithm left untouched in this thesis, please refer to [Bru16].

5.2 Goals

The initial version of the algorithm had several limitations that prevented it from being applied to real world C++ projects. One of the goals of this thesis was to reduce these limitations.

First, the ability to analyze projects that consist of multiple TUs is vitally needed. This requires the algorithm to be able to resolve function calls, not just within a TU but also across TUs.

Furthermore, the algorithm should track accesses to memory locations even if they are not globally defined but rather accessed using a pointer passed as an argument to a function, as this is common practice in C++ code.

Finally, for each data race the algorithm should provide the sequence of function calls that lead to the data race. This information can then be used by the visualization to guide the user through the source code.

The following sections detail the functionality of the algorithm and the implementation of the improvements described above.

5.3 Phases

The algorithm consists of the following phases:

1. The function calls in the control flow graph obtained from the extractor need to be *resolved*.
2. A *thread-graph* containing thread starts and joins is constructed from the resolved control flow graph.
3. The locks held during memory accesses are recorded in a *Lockset*.
4. A flow-sensitive analysis determines potential data races between temporally related threads using the constructed thread-graph and lockset.
5. A fully-concurrent analysis determines potential data races between threads without a direct temporal relationship.

These phases are described in more detail in the following sections.

5.3.1 Function Call Resolution

The first step is to resolve function calls present in the initial control flow graph.

Interaction with the extractor

In order to support the new functionality, the interface between the extraction component and the algorithm was revised.

Previously, the algorithm would instruct the extractor to analyze the analysis entry point, starting with the `main` function, and in return receive the control flow graph for all statements in this TU reachable from this function. Function calls to targets within the same TUs were directly inlined by the Clang static analyzer and not explicitly recorded. Calls to functions not defined in the same TUs are recorded in the control flow graph. The algorithm would then check for any unresolved function calls and iteratively call the extractor to analyze the called function in the context of its TU. This process was repeated until all unresolved function calls were resolved.

The advantage of this method is that most of the heavy lifting is outsourced to the extractor, or more precisely, the Clang static analyzer. This greatly simplifies the algorithm implementation.

The downside to this method is that function calls within a TU are automatically inlined by the extractor and are thus not visible to the algorithm. This meant that when a data race was reported to the user, no further details about the data race were known. Specifically, the sequence of function calls that lead to the data race would be very useful to the user to track down the cause of the data race. Another disadvantage of this method is related to the capabilities of the Clang static analyzer. Attempting to resolve an unresolved function call to a constructor, destructor, operator or lambda was previously impossible. This is because these special functions are not yet supported as analysis target. As a consequence, detecting data races that include a function call of this kind was previously not possible.

In the new version of the algorithm developed in this thesis, the exchange of data between the algorithm and the extractor has been modified to allow for more fine grained control. The extractor no longer resolves function calls directly but rather inserts explicit function call statements. This allows the algorithm to track function calls and resolve them during analysis. It also enables the algorithm to detect data races on memory shared by passing pointers as arguments to functions in entirely different TUs, which previously also was not possible. Finally, even calls to constructors, destructors, operators and lambdas in different TUs are now resolved in the algorithm. This is because when started without an explicit analysis target, the extractor simply analyzes every function in the given TU. This provides the algorithm with a more detailed control flow graph that allows the data race detection to be improved.

Translation Units

The new interface to the extractor described in the previous section enables multiple TUs to be supported with relatively little effort required by the algorithm. Unlike the previous data exchange format, new format described in detail in the extractors chapter takes into account the fact that functions reside in a TU and provides access to all of them. When parsing the control flow graph, all the graphs are extracted and organized in a single data structure. This allows the algorithm to operate independently of TUs and reduces the problem to function call resolution.

Inlining

Since the extractor no longer immediately inlines intra-TU function calls, the algorithm now has to do all of the inlining, whereas before only inter-TU calls needed to be inlined outside of the extractor in the algorithm.

In the proof-of-concept implementation of the previous version, each function was traversed recursively and independently of all other functions in an arbitrary order. Each unresolved function call is substituted with the statements of the called function. This method is not suited to the new approach where there are a much larger number of functions and calls since resolving everything independently would result in a lot of work being done multiple times without any benefit.

Rather than simply resolving everything without any particular strategy, the new implementation starts the process at functions that do not contain any further function call and simply marks them as resolved. Next, functions that only contain calls to these resolved functions are resolved. This process is repeated until the analysis entry point has been resolved.

A fundamental drawback of this approach, however, is that recursive functions do not fit into this model without any further effort, as they can not be inlined. A control flow graph containing recursive function calls can be interpreted as a directed graph. In order for inlining to be possible in all cases, the graph would have to be acyclic. This however is not the case for source code containing recursive procedures. A different approach would be to forgo inlining the call graph before starting the flow analysis altogether. This would require a complete redesign of the program points computation in the lockset construction, which is outside the scope of this thesis. For the time being, this issue was addressed by employing a recursion limit for the function call resolution.

Multiple Identical Function Calls

Another issue that revealed itself in more complex test cases is resolving multiple calls to the same target within the same function. In the original version, the blocks of the target function were inserted without modification into the source function. The predecessor of the first inserted block is set to the caller block, which contains the call currently being resolved, and the successor of the caller block is set to the first inserted block. The same principle is applied to the last block of the target function and the successor of the caller block. This concept is analogous to insertion for doubly linked lists.

The problem with this approach is that when multiple calls to the same function exist in the source function, the blocks inserted while processing the second call will be identical to the ones inserted during the first call. This lead to infinite loops and subsequently a stack overflow error in the old version.

The revised version creates a separate copy of the target function blocks for each call and tags these blocks with the caller block. This makes them distinguishable during traversal later.

Function Parameters and Arguments

Accesses to memory allocated outside the current function via pointers or references passed in as arguments should be properly correlated with concurrent accesses to the same memory locations using a different name outside the function context. This is solved by looking up the name of the function parameter at the same position as the argument passed in to the function when it is called. Accesses to the parameter inside the function are then correlated with accesses to the location passed to the function under the name of the argument. This allows the algorithm to detect data races even when function calls and no global variables are involved, which previously was the only case in which data races could be detected.

5.3.2 Thread Graph Construction

After resolving function calls in the control flow graph received from the extractor, a thread graph is constructed. This additionally requires an entry point for the analysis. The analysis then assumes that there is one main thread running when the given entry point is entered. Subsequent thread starts add a thread to the thread graph.

This aspect of the algorithm remains mostly unchanged from the previous version.

5.3.3 Program Points

Section 1.4.2 introduced the notion of *program points* as a sequence of program state transformations. These program points are calculated based on the resolved control flow graph and the thread graph. The resulting program points represent the possible execution paths through the program. They are used in the lockset analysis to construct a lockset. This process is described in the corresponding section below.

Deeply Nested Function Calls

It was observed that when analyzing projects containing a data race only reachable via deeply nested function calls, the data race would often be missed. An investigation into the issue showed that the way the program points calculation in the lockset construction kept track of the active threads was flawed. An empty basic block containing no statements would cause the calculation to forget what threads are currently active. This issue has been fixed in the current version of the algorithm.

5.3.4 Lockset Analysis

Following the thread graph construction, a so called *lockset* is constructed. This process is traditionally called *lockset analysis*. The lockset contains all memory accesses along with the thread entries that performed them and the locks held by the thread when accessing the memory. This lockset can then be used in combination with the thread graph to detect data races between concurrent threads. The lockset analysis is flow- and path-sensitive, that is, it traverses every execution path. To construct the lockset, the program points computed in the previous phase are used.

5.3.5 Flow-Sensitive Data Race Analysis

Given the extensive preparation work done in the previous phases, the actual data race analysis is relatively straightforward. As already discussed, every program point describes a possible state of the program and includes a memory access, the locks held and the active threads. This is all the information needed to finally detect the data races.

The flow-sensitive data race analysis aims to find data races between conditionally concurrent threads. Two threads are considered conditionally concurrent if one thread may or may not start another thread. Since the algorithm can not determine whether or not the second thread is actually started, it will always assume that it is in fact started to ensure not missing any potential data races.

The analysis traverses every program point and reports a data race whenever two or more threads concurrently access the same memory location without sharing a common lock, and at least one access is a write access.

This first part of the data race analysis also detects unconditionally concurrent threads and stores them for the second part of the data race analysis. Two threads are regarded as unconditionally concurrent if there is no apparent relationship between the two threads, i.e., there is no start edge from one thread to the other in the thread graph. For these threads, the analysis will assume that both are always running concurrently.

5.3.6 Fully Concurrent Flow-Sensitive Data Race Analysis

Using the unconditionally concurrent threads determined in the previous analysis, this final step will employ a final data race detection for these threads.

For each pair of unconditionally concurrent threads, a data race is reported for a shared memory access if the conditions described in the previous section are met.

5.4 Architecture

This section gives an overview of the high-level architecture of the algorithm and shows the details of the graph package which was the main subject of extension during this thesis.

5.4.1 Overview

The high-level architecture of the analysis component remains largely the same as in the version provided in the masters thesis. Figure 5.1 provides a white-box view of the analysis component.

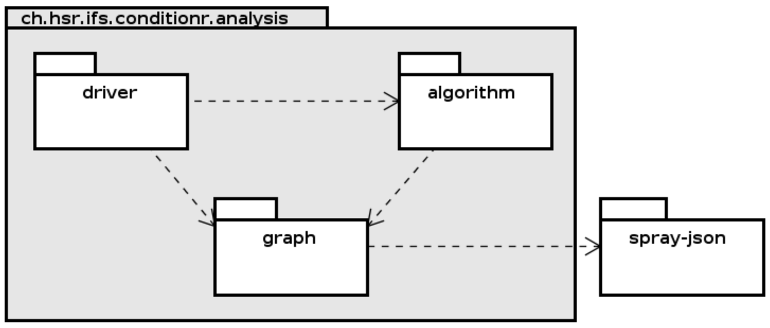


Figure 5.1: White-box view of the analysis component.

5.4.2 Graph

The graph package outlined in figure 5.2 was extended for function parameters and arguments. Much of the work done on extending the algorithm functionality was independent of architectural changes.

5.5 Summary

The algorithm implementation was extended to employ proper cross-translation unit function call inlining. Pointers to memory locations passed as arguments to other functions are tracked and considered during the data race detection. Nested function calls of arbitrary depth are correctly followed. Multiple calls to the same function from the source function are not an issue for the algorithm anymore. Various bugs have been fixed to improve the stability of the implementation.

As the number of execution paths during a path-sensitive analysis grows exponentially with the number of conditionals, analyzing larger projects using the current approach comes with a considerable performance penalty. The current approach should be reconsidered to address these issues in order to be viable for large-scale projects.

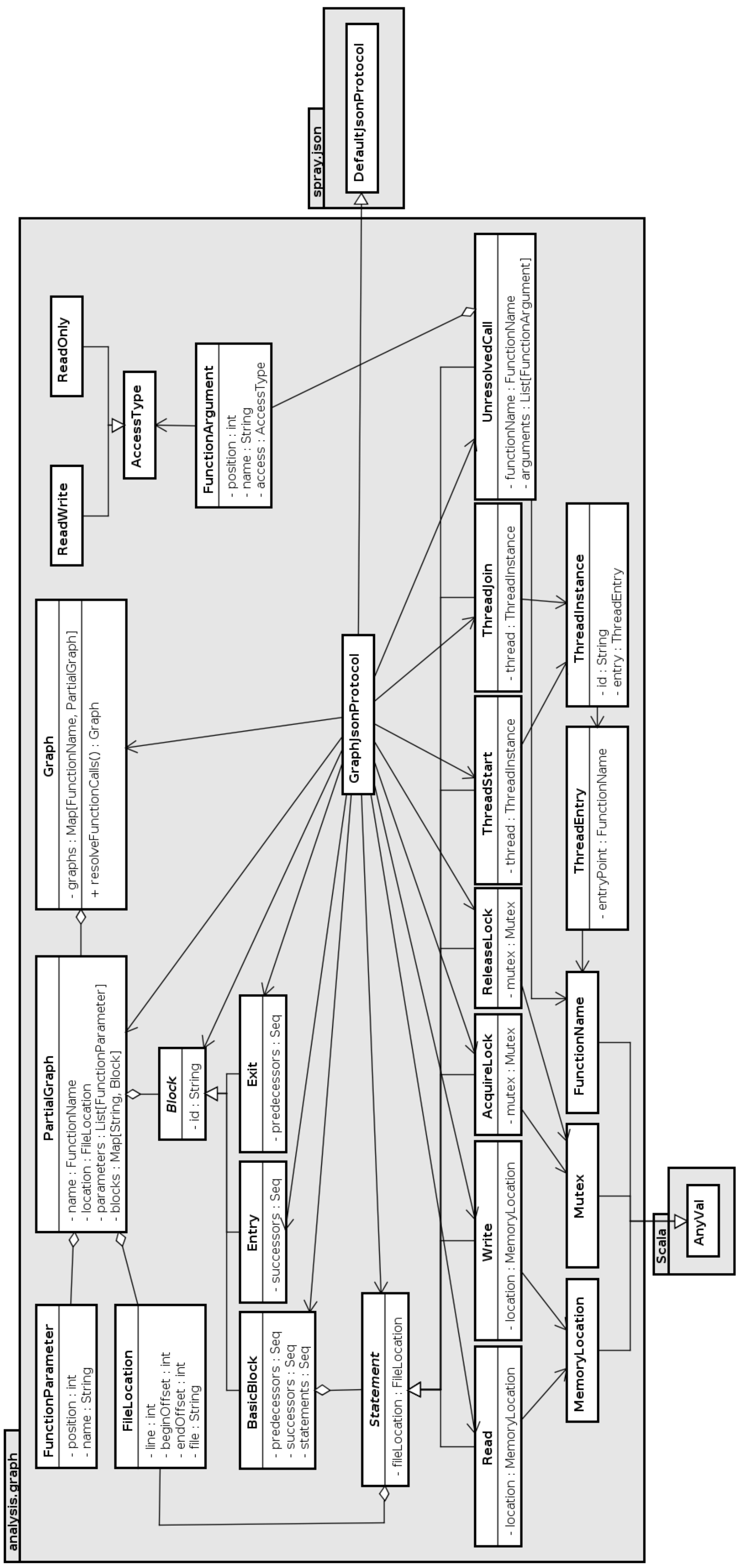


Figure 5.2: Class diagram of the graph package.

6 Visualization

This chapter documents the visualization plug-in of the ConditionR tool. First, the initial UI of the masters thesis as well as the developed concepts from the student research thesis are analyzed. Then, new concepts from this bachelor thesis are proposed and analyzed. After that, the decision of which design was chosen for the implementation is documented. In the final part of this chapter, the implementation is described in detail.

6.1 Overview

After the user started the data race analysis for a given project, the analysis returns a list of potential data races. These results are then visualized in the IDE to assist the user with his goal of developing correct software. This closes the loop of the user experience of the ConditionR tool.

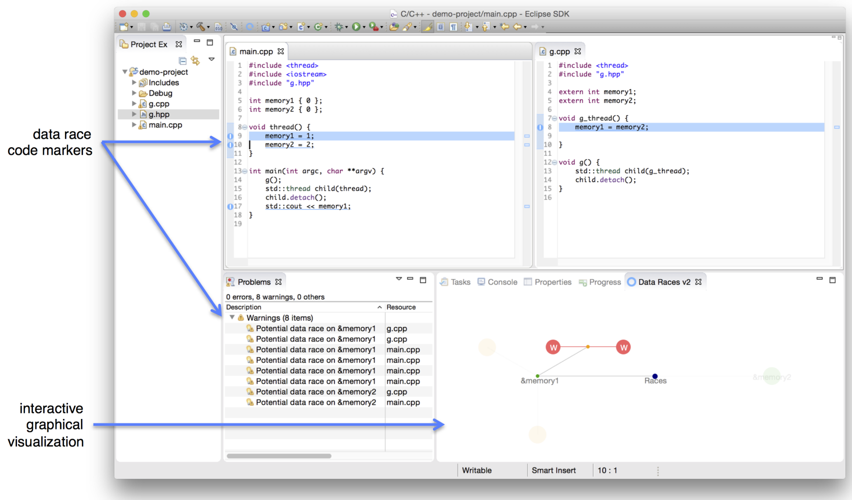
In order to deliver maximum value to the user, it is vital that the analysis results are communicated to the user as effectively as possible.

6.2 Master Thesis UI Version

The version of the visualization presented in the masters thesis has at its core a dedicated view within the Eclipse view. The results of the data race analysis are shown as a graph directly in that view. In addition to this view, warning messages and editor code markers concerning the detected data races are added. The analysis is started using a button added to the main toolbar by the plug-in.

6.2.1 Analysis

Looking at a very simple example as shown in figure 6.1, the visualization seems fairly straightforward. Once the plug-in is applied to a more complex



```
#include <thread>
#include <iostream>
#include "g.hpp"

int memory1 { 0 };
int memory2 { 0 };

void thread() {
    memory1 = 1;
    memory2 = 2;
}

int main(int argc, char **argv) {
    g();
    std::thread child(thread);
    child.detach();
    std::cout << memory1;
}
```

```
#include <thread>
#include "g.hpp"

extern int memory1;
extern int memory2;

void g_thread() {
    memory1 = memory2;

}

void g() {
    std::thread child(g_thread);
    child.detach();
}
```

0 errors, 8 warnings, 0 others

Description	Resource
Warnings (8 items)	
Potential data race on &memory1	g.cpp
Potential data race on &memory1	g.cpp
Potential data race on &memory1	main.cpp
Potential data race on &memory1	main.cpp
Potential data race on &memory1	main.cpp
Potential data race on &memory1	main.cpp
Potential data race on &memory2	g.cpp
Potential data race on &memory2	main.cpp

Figure 6.1: Screenshot of the visualization component from the master thesis documentation

example such sa the *Chess* project described in section 3.3.1, the visualization of the results appears fairly confusing and overwhelming.

Graph View

For each data race, the main view only shows the locations of the two memory accesses that lead to the data race. It does not give any further details on why there is a data race and how it occurred. Furthermore, only one data race connection for a given access is shown at a time. If an access belongs to a data race with several other locations, these are not visible.

Another issue with this view is apparent in cases where a memory location is involved in multiple data races. An example of this is shown in figure 6.2. The data races for a memory location are laid out in a circle around that location. Each race is represented as a solid, indistinguishable circle. Therefore it is impossible for the user to tell which node corresponds to which race, which makes the process of investigating a given data race very difficult. The same problem presents itself if there are many memory locations that belong to data races. Once more than eight memory locations are involved in (potentially distinct) data races, it is possible for them to overlap visually. This is due to the way the graph is laid out and makes it very hard for the user to understand what is going on.

The next point of criticism is the inconsistency of the navigation inside the view. The user has to click in the graph on each node to show more information. To show the location of a data race access in the editor, hovering over it using a mouse is required.

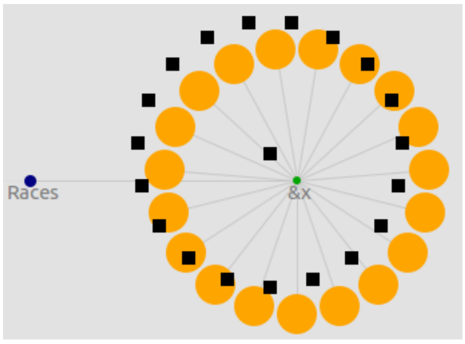
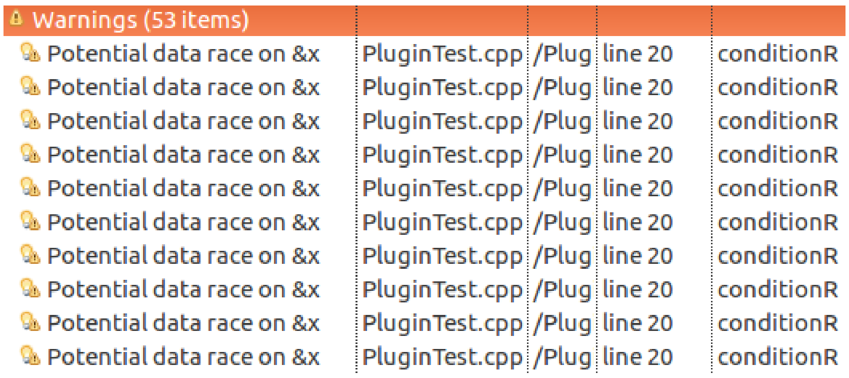


Figure 6.2: View example of multiple data races on memory location x



Warnings (53 items)				
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR
Potential data race on &x	PluginTest.cpp	/Plug	line 20	conditionR

Figure 6.3: Eclipse problems view with duplicated warning messages

A further issue with operating the visualization is that the graph view has to be manually opened *before* the data race analysis is completed. If the view has not been opened manually or has been opened after the analysis has already completed, no results will be displayed in the view.

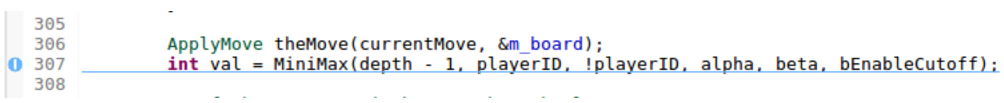
Warning Messages

The warning messages for the data races could also be improved. For each access location of a data race, one warning message is generated. If a location has multiple data races, the exact same warning message for the data races is generated multiple times. An example of this issue is shown in figure 6.3.

Editor Markers

Eclipse allows markers to be set on arbitrary lines of code in the editor. A marker consists of an icon on the left hand side of the editor, and an emphasis on a fragment of code on that line.

For each data race reported by the analysis, a marker is created. This marker covers the entire line of code, and not just the fragment involved in the data race. While this makes it easier to spot the line of code involved in



```
    ApplyMove theMove(currentMove, &m_board);
    int val = MiniMax(depth - 1, playerID, !playerID, alpha, beta, bEnableCutoff);
```

Figure 6.4: Unclear marking of a data race

the data race, it does not allow the user to easily determine where exactly on that line the data race occurs. Additionally, if there are multiple locations involved in data races on the same line, this can not be recognized visually.

The same problem exists for example if there is a method call with multiple parameters which will be copied. An example of this is shown in figure 6.4.

6.2.2 UI Bug

The initial visualization plug-in, while suffering from many drawbacks, works as intended on MacOS systems.

In the current version of Ubuntu (16.04) and Eclipse (4.6.x), the visualization of the analysis results does not work properly. The entire data race graph was drawn on a single `Composite` object. Each node of the graph, which was clickable, was covered by an "invisible" rectangle to make the node clickable. But this rectangle turned out not to be invisible on Linux. The reason for that was a bug in the Standard Widget Toolkit (SWT) library with newer GIMP-Toolkit (GTK) versions. The clickable rectangles used the `NO_BACKGROUND` flag which should made the background invisible and reveal the actual node below. Inside the SWT package, the Composite class of SWT deactivates some style-flags, as shown in listing 6.1.

```
static int checkStyle (int style) {
  if (OS.INIT_CAIRO) {
    style &= ~SWT.NO_BACKGROUND;
  }
  style &= ~SWT.TRANSPARENT;
  return style;
}
```

Listing 6.1: snippet from the SWT library

The `NO_BACKGROUND` style is disabled if Cairo or a GTK version newer than 2.17.0 is used. On recent Linux systems, the style-flag `NO_BACKGROUND` was ignored because of the usage of a newer GTK version. Even without the use of cairo and an older GTK version it did not work properly. Instead of being invisible, the rectangle now was gray. So the attempt to force SWT to set the

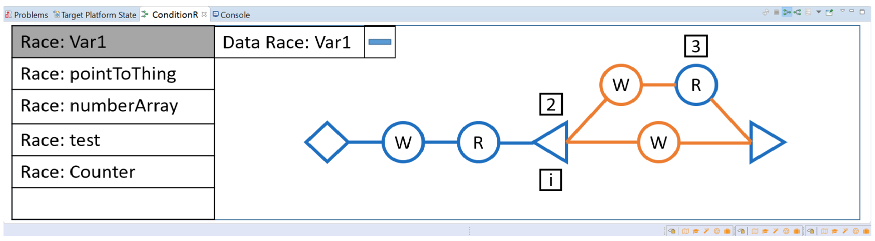


Figure 6.5: UI concept of the thread view with two concurring threads

NO_BACKGROUND style did not work. Using the TRANSPARENT flag instead does not work either, because this will be ignored anyway.

Possible Fixes

An easy hotfix for this view would be to, instead of making the whole nodes clickable, only add a small clickable square at the left top of the node. For the initial evaluation (see section 3.3.1), this hotfix was implemented in order to get at least some usable results in the view. This is also the reason why in every screenshot of the old view, apparently meaningless black squares are visible.

If the graph view should work and look identical to the master thesis UI, the view has to be rewritten. The nodes should be drawn on the clickable rectangle instead of the graph view composite.

6.3 Student Research Thesis Proposals

In the student research thesis, two new visualization concepts were developed. First, a thread view which, for each memory location with data races, shows a thread graph. Secondly, a spread view that mainly focuses on the variables and how they are mapped.

6.3.1 Thread View Concept

The concept of the thread view contains the following components as figure 6.5 shows:

- A list of racy variables where the user can select a variable to be shown in the graph-part of the view
- A graph view where the selected variable is drawn with all involved read and write accesses, thread starts and joins.

The benefits of this concepts are that it shows in which threads the racy variable is used and presents a complete overview for a given racy variable. Additionally, it also reveals whether a thread joins correctly. Another good idea is that all racy variables are listed separately. Only information about one racy variable at a time is shown in the graph view. This makes it much more clear and useful.

The concept also took into account complex cases like large number of threads or too many uncritical read accesses. With buttons over some nodes, the graph could be extended or collapsed to show only the critical sections. But even with the support of such complex cases, the view could still be too confusing and potentially unclear if there are too much data races for a variable.

6.3.2 Spread View Concept

Unlike the thread view, the spread view focuses on the variables and how they are mapped. The idea of this concept is to give the user more information about how a race occurred. Figure 6.6 shows such a view. The spread graph shows the involved methods `main();` and `method1(var test);` and their variables. The variable `a` is passed to a new method, which uses it as `test`. The underlying memory location is identical.

The following two components are used in a spread view:

- A list of racy variables where the user can select a variable to be shown in the graph-part of the view
- A spread graph which shows the involved methods with their variables in a UML style.

The problem of this view is that it is only usable in easy examples with few methods involved. If a data race involves a large number methods, it would be too cluttered to display the whole data race.

Also, this concept would not work for static or global variables, because there is no mapping to display.

6.4 UI Improvement

This section documents the improvement ideas and also their implementation for the visualization which came up during this bachelor thesis.

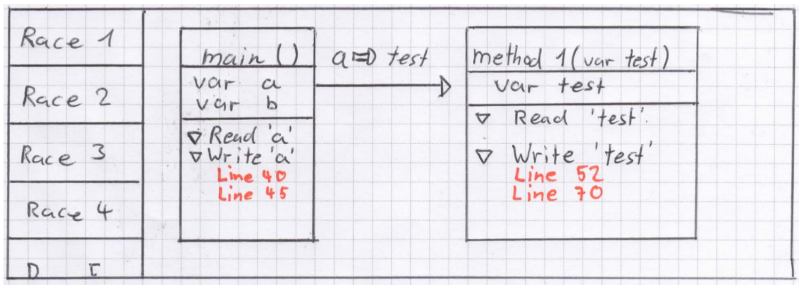
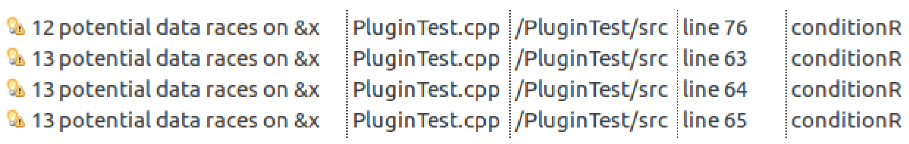


Figure 6.6: UI concept of the spread view with two methods and a passed variable



12 potential data races on &x	PluginTest.cpp	/PluginTest/src	line 76	conditionR
13 potential data races on &x	PluginTest.cpp	/PluginTest/src	line 63	conditionR
13 potential data races on &x	PluginTest.cpp	/PluginTest/src	line 64	conditionR
13 potential data races on &x	PluginTest.cpp	/PluginTest/src	line 65	conditionR

Figure 6.7: Improved result of the warning messages in the problem view

6.4.1 Warning Messages

As discussed during the analysis of the UI presented in the masters thesis in section 6.2.1, a warning message was generated for all locations of a data race. If a location was involved in several data races, several warnings with the same message were generated.

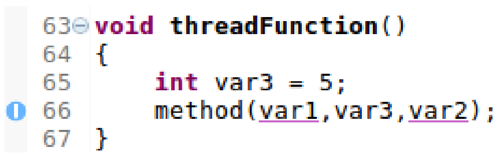
It was not a bad idea to have the same error message occur multiple times but for the user this was not very helpful as screen real estate is limited. To improve the user experience, identical warning messages should be merged into one and the number of occurrences should be displayed along with the message. Figure 6.7 shows this improvement. The warning for line 76 is shown only once instead of twelve times.

6.4.2 Editor Markers

The code markings from the master thesis had a lot of potential for improvements. The biggest flaw was that for a data race found in the code, the whole line was marked. There was no information about where on the line the data race was. Also if two or more data races were on the same line, there was no way of telling.

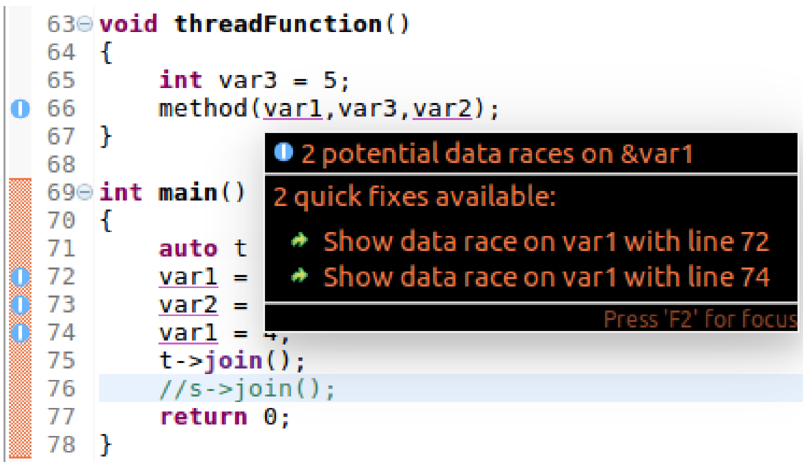
The best improvement to remove this flaw was to underline the racy variable only. Figure 6.8 shows the solution of the code markers revision. This involved some code changes in the extractor and algorithm, because more detailed information about the location was needed.

Next, the tooltip information while hovering over the code markers had to be revisited. The UI version of the master thesis only showed a single mes-



```
63 void threadFunction()
64 {
65     int var3 = 5;
66     method(var1,var3,var2);
67 }
```

Figure 6.8: Improved result of the code marking in the editor



```
63 void threadFunction()
64 {
65     int var3 = 5;
66     method(var1,var3,var2);
67 }
68
69 int main()
70 {
71     auto t
72     var1 =
73     var2 =
74     var1 = 4;
75     t->join();
76     //s->join();
77     return 0;
78 }
```

2 potential data races on &var1

2 quick fixes available:

Show data race on var1 with line 72

Show data race on var1 with line 74

Press 'F2' for focus

Figure 6.9: Hovering over var1 on line 66 shows with which other lines a conflict exist

sage with an internal id of the last data race of this line which was of no use to the user. The reworking idea was to show a list of messages with information about every other location this variable access has a data race with. Figure 6.9 shows the result of this. Clicking on a list entry will cause the data race view to move focus to the selected data race to get more information.

6.4.3 View

The view is the core of the ConditionR Eclipse plug-in visualization. It shows more information about how a specific data race in the code occurs.

Because the implementation of the master thesis was on one hand not usable on Unix systems and on the other hand not terribly useful for the understanding of the data races, the view had to be totally reworked. Since the proposals of the student research thesis had their flaws, new proposals were developed based on their ideas.

One Graph for each Memory Location

This proposal shows all data races for a given memory location in a single graph. The basic idea was to, instead of drawing only separate data race pairs like in the master thesis, sum up all the pairs in a connected graph.

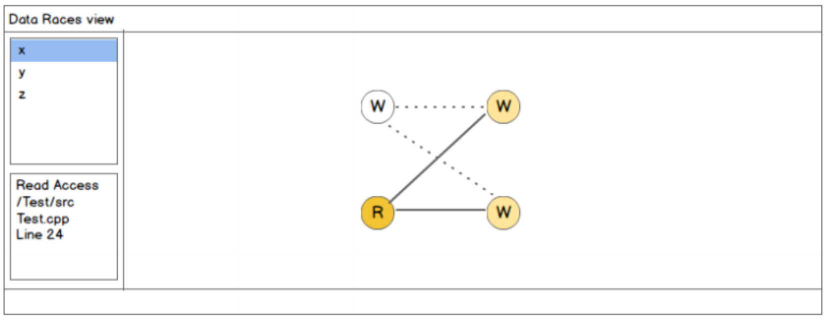


Figure 6.10: Mockup where a read access location of x is selected

This reduces duplicated race locations and gives a useful overview of all conflicting race locations for a location.

Figure 6.10 shows an example of this view. It contains three parts. First, a list on the left side where all memory locations with data races are listed, based on the idea of the thread view concept variable list described in section 6.3.1. Secondly, an information box on the left bottom, where more information about a node could be displayed. And finally, a graph view where the data race graph for the selected memory location will be displayed.

If a memory location has been clicked in the list, its graph will be drawn. The graph contains nodes and edges. A node stands for a race location of the memory location in the code and shows also whether it is a read (R) or write (W) access. An edge stands for a potential data race between two race locations. The nodes are clickable. A click shows more information about the location in the information box, highlighting the node and its connections and jumps in the code editor to the location.

Like the other proposals, this view also has flaws. One disadvantage is that it is only visible which positions in the code are involved in a data race. Showing additional information, like a stack trace, would be difficult in the view. Another problem is the scaling. If there are many data races for a memory location, the graph would be to big. This would be difficult to draw in a small view without overlapping nodes and edges. This is the reason why this view was not further developed and implemented.

One Graph for each Data Race

The biggest problem was the scaling of the view when there are a large number of data races and displaying helpful additional information about a data race. So the basic idea to fix the scaling problem was to show only one data race in a separate section of the view, because the amount of data races for a memory location can increase but a single data race can not grow.

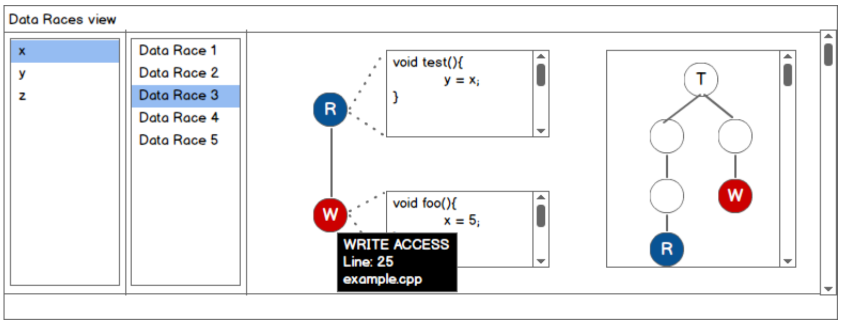


Figure 6.11: Mock-up where a read access location of x is selected

Figure 6.11 shows the mock-up for this view. On the left side it contains two lists. A list with memory locations and a list with all data races for the chosen memory location. On the right side the data race information sub-view is displayed. This sub-view is split into three parts. The left part shows the two accesses for a race as a blue R for Read access or a red W for write access. The middle part shows for both accesses their code snippet. In the right part, a call stack is displayed to give more information about how the data race arises. More information about the functionality of this view is described in section 6.5.

The visualization of the call stack evolved during development. In the mock-up it is displayed as a graph. In the final implementation it has been changed into a tree list because it shows more information on the first look.

6.5 Functionality

This chapter documents the functionality of the visualization plug-in from the users perspective. It describes how to configure the plug-in, how to start a data race analysis and how to interpret the results.

6.5.1 Running Analysis

An analysis can be started in two different ways. The standard way to run the analysis is to click on the button in the toolbar (see figure 6.12). This starts the analysis with the main method of the project as entry point for the algorithm. If no main method is found, the analysis will be canceled and an error window will be displayed to inform the user.

For project with no existing main method, like libraries, or if a user only want to run an analysis only for part of a project, there is another solution.

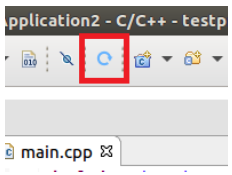
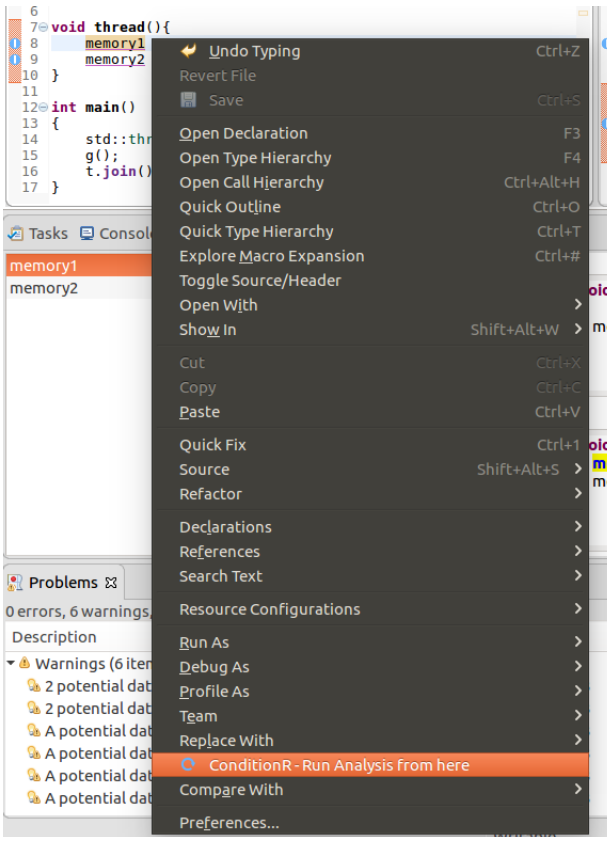


Figure 6.12: Added button from the plug-in to start a data race analysis with the main method as entry point for the algorithm.



```
void thread(){
    memory1
    memory2
}

int main()
{
    std::thr
    g();
    t.join()
}
```

Figure 6.13: Added drop-down menu entry to start a data race analysis from the actual cursor position in the editor.

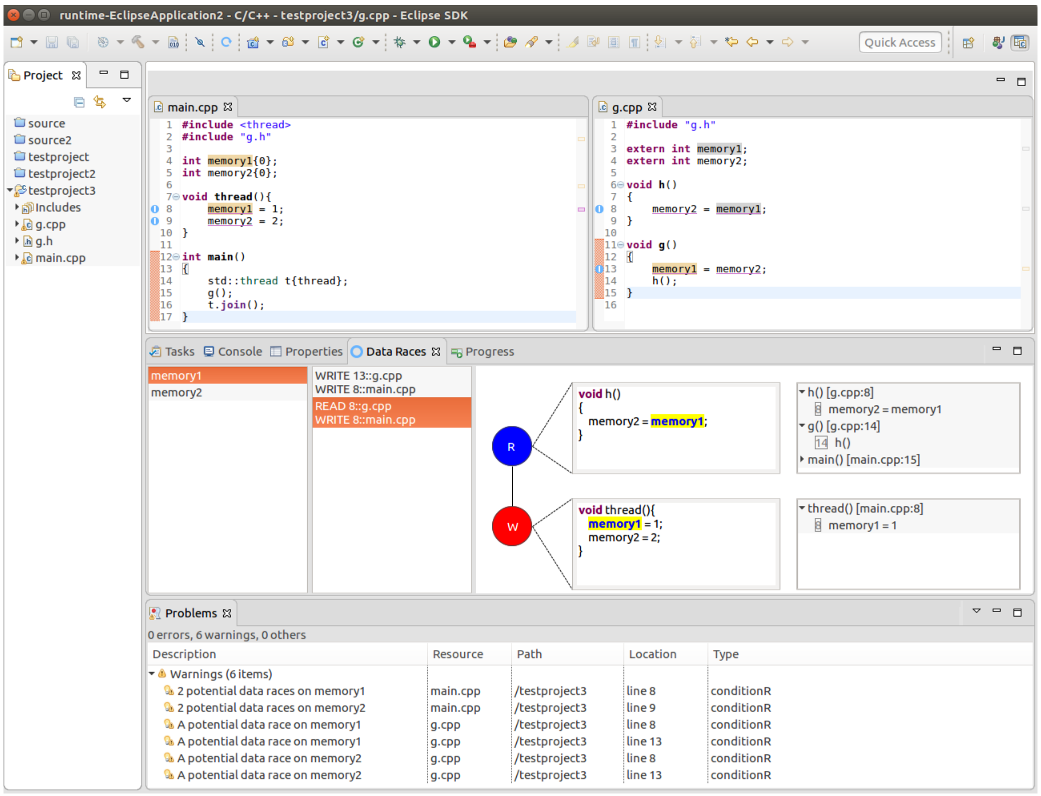
The user can select a position inside the editor in a method body and right click to start the analysis from this method. This is shown in figure 6.13.

6.5.2 Data Race Visualization

In this section, the usage of the data race visualization is described. This description is based on an example project which is illustrated in figure 6.14.

Data Race View

After a successful data race analysis, the visualization automatically opens the data race view for the user. The view contains two lists. One list for



main.cpp

```
#include <thread>
#include "g.h"

int memory1{0};
int memory2{0};

void thread(){
    memory1 = 1;
    memory2 = 2;
}

int main()
{
    std::thread t{thread};
    g();
    t.join();
}
```

g.cpp

```
#include "g.h"

extern int memory1;
extern int memory2;

void h()
{
    memory2 = memory1;
}

void g()
{
    memory1 = memory2;
    h();
}

```

Data Races

```
memory1
memory2
```

```
WRITE 13::g.cpp
WRITE 8::main.cpp
READ 8::g.cpp
WRITE 8::main.cpp
```

```
void h()
{
  memory2 = memory1;
}
```

```
h() [g.cpp:8]
  8  memory2 = memory1
g() [g.cpp:14]
  14  h()
main() [main.cpp:15]
```

```
void thread(){
  memory1 = 1;
  memory2 = 2;
}
```

```
thread() [main.cpp:8]
  8  memory1 = 1
```

0 errors, 6 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (6 items)				
2 potential data races on memory1	main.cpp	/testproject3	line 8	conditionR
2 potential data races on memory2	main.cpp	/testproject3	line 9	conditionR
A potential data race on memory1	g.cpp	/testproject3	line 8	conditionR
A potential data race on memory1	g.cpp	/testproject3	line 13	conditionR
A potential data race on memory2	g.cpp	/testproject3	line 8	conditionR
A potential data race on memory2	g.cpp	/testproject3	line 13	conditionR

Figure 6.14: Screenshot of an analyzed C++ project with data races.

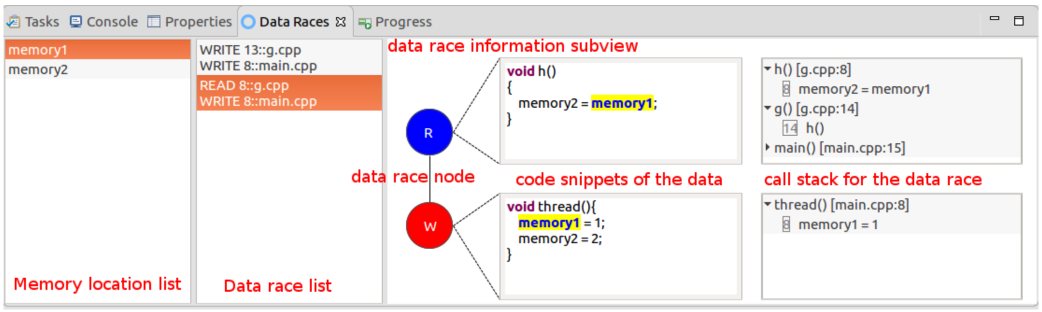
all memory location that contain data races and another for all data race pairs of the selected memory location. Additionally, a data race information subview is opened (see figure 6.15). If a memory location is selected in the first view, the data race list will be updated with all races for that memory location. If an data race pair is selected in the data race list, the data race will be displayed inside the data race information subview.

The data race subview is composed of three different parts.

In the left part, the two data race locations are displayed as circles. Blue for read access and red for a write access of the memory location. Hovering over a circle will show a tool-tip view with information about the file name, the line number inside the file of the location and the access type. Clicking on a circle will jump to the location in the editor and highlight the data race.

In the middle part, the whole function where the data race occurs is displayed for both location. This simplifies the comparison of the two location, because a user does not have to jump to the data race in the editor.

In the right part, the call stack for the data race is listed. Here, the user can see in which function the data race or the function of the data race is called. This helps the user to better understand the data race.



```
memory1
memory2
```

```
WRITE 13::g.cpp
WRITE 8::main.cpp
READ 8::g.cpp
WRITE 8::main.cpp
```

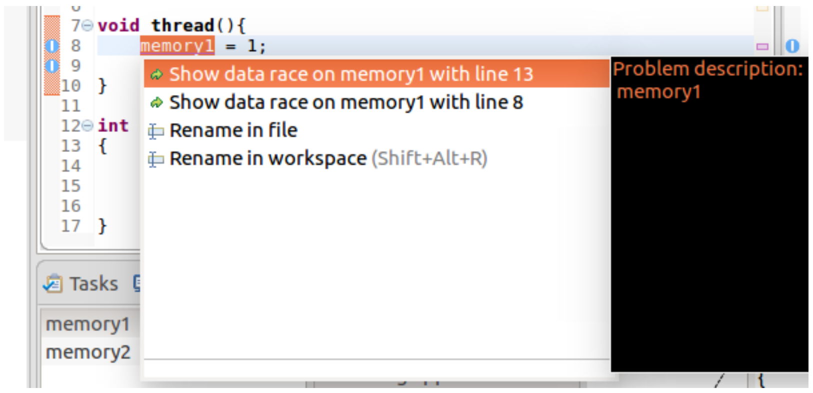
```
void h()
{
  memory2 = memory1;
}
```

```
h() [g.cpp:8]
  memory2 = memory1
g() [g.cpp:14]
  h()
main() [main.cpp:15]
```

```
void thread(){
  memory1 = 1;
  memory2 = 2;
}
```

```
thread() [main.cpp:8]
  memory1 = 1
```

Figure 6.15: The data race view with analyzed input of a project.


Show data race on memory1 with line 13
Show data race on memory1 with line 8
Rename in file
Rename in workspace (Shift+Alt+R)
Tasks
memory1
memory2

```
void thread(){
  memory1 = 1;

}

int
{



}
```

Problem description: memory1

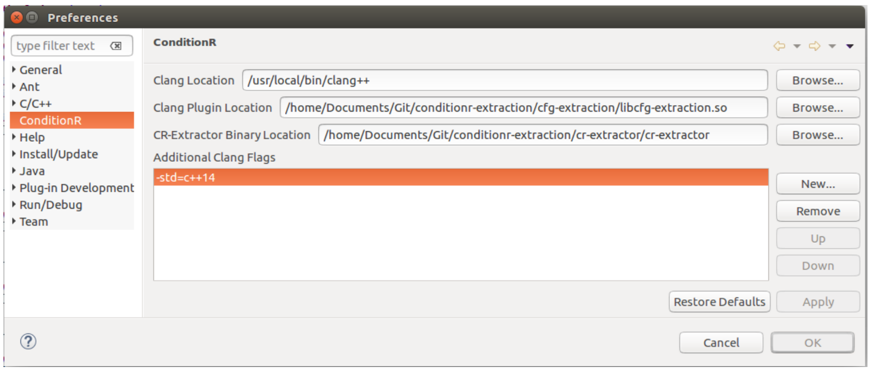
Figure 6.16: Displayed editor marking of a location with is in conflict with two other lines.

Editor Markings

The editor markings in the editor are created or updated after a successful data race analysis. The markings consist of a purple underlining of the data race locations and an icon on the left side of the editor on each line where an data race was found. Hovering over a underlined location in the editor will show an tooltip which lists all other locations where this line has a conflict with. Clicking on a list element will show the data race view with the selected race in the information subview. Another way to show the data race is clicking on the icon on the left side of the editor. This shows also an list of all conflicting other locations. From here the user can jump into the data race view (see figure 6.16).

6.5.3 Configuration

To use the *ConditionR* plug-in, paths to clang, the checker library and the cr-extractor binary must be provided. This can be done via the *preferences* window in Eclipse. A user can open the window with `Window > Preferences > ConditionR` and add the files. Figure 6.17 shows this window. Additionally, a user has the possibility to add some optional flags for clang here, for



Clang Location /usr/local/bin/clang++
Clang Plugin Location /home/Documents/Git/conditionr-extraction/cfg-extraction/libcfg-extraction.so
CR-Extractor Binary Location /home/Documents/Git/conditionr-extraction/cr-extractor/cr-extractor
Additional Clang Flags
-std=c++14

Figure 6.17: Preference window for the configuration of the plug-in.

example `-std=c++14`.

6.6 Design and Architecture

This section documents the design and architecture of the visualization plug-in. The major part of the visualization architecture was taken from the master thesis [Bru16]. For components that have not been changed greatly, only a short summary is given. If more information is needed, please refer to the masters thesis.

6.6.1 Component Overview

The source code is organized in five packages. Four of them were taken from the master thesis with some changes. In the following list a short overview of the packages is given.

- `preferences`. This package contains the logic for the preference page where a user can modify settings of the plug-in.
- `common`. This package contains several classes that are used by more than one of the other packages. For example the `EventBus` for the communication between UI subcomponents.
- `commands`. This package contains the commands for the plug-in.
- `datarace`. This is the newly added package which contains the data race class for the view and the conversion for the data races from the algorithm.
- `views`. This package contains the classes that implement the in section 6.5 described data race view.

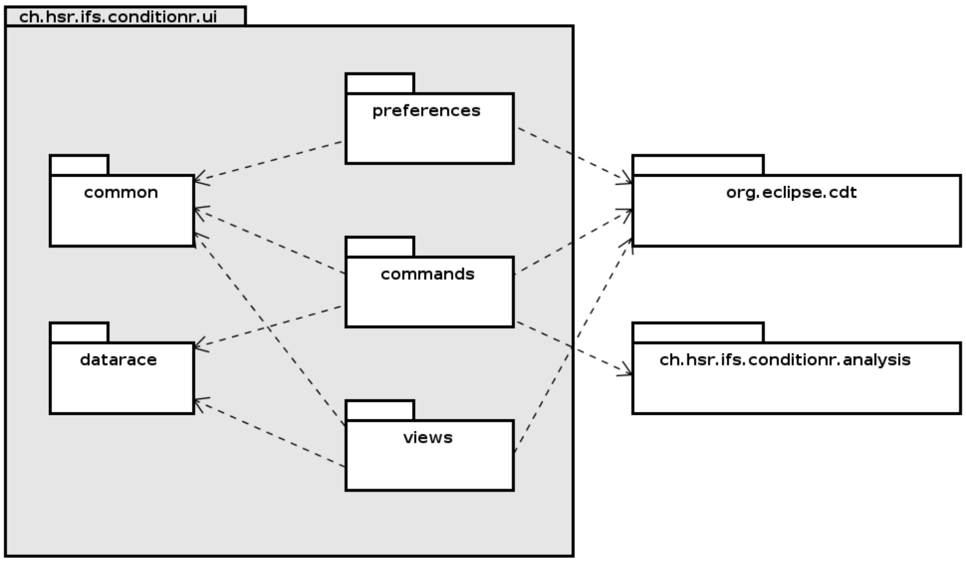


Figure 6.18: White-box view of the UI components.

The white-box view of the UI component is illustrated in figure 6.18. The following sections describe the contents of the commands, datarace and views packages. The other two packages have not been changed and more information about them is given in the master thesis.

6.6.2 commands

This package contains the two commands `RunAnalysisCommand` and `RunAnalysisFromEntryPoint` which uses the `org.eclipse.ui.commands` extension point. It also contains the point of interaction with the algorithm component (`ch.hsr.ifs.conditionr.analysis`). The content of the `commands` package, including important dependencies, is illustrated in figure 6.19. In the following list are the classes of this packages described.

- `RunAnalysisCommand`. This command is called if a user wants to run an analysis via the toolbar in the user interface. All this class does is to instantiate and schedule an Analysis Job with the main function as entry point for the algorithm.

- `RunAnalysisCommandFromEntryPoint`. This command is called if a user want to run an analysis via the selected location in the code editor. It extends the `RunAnalysisCommand` with the selected function as entry point.

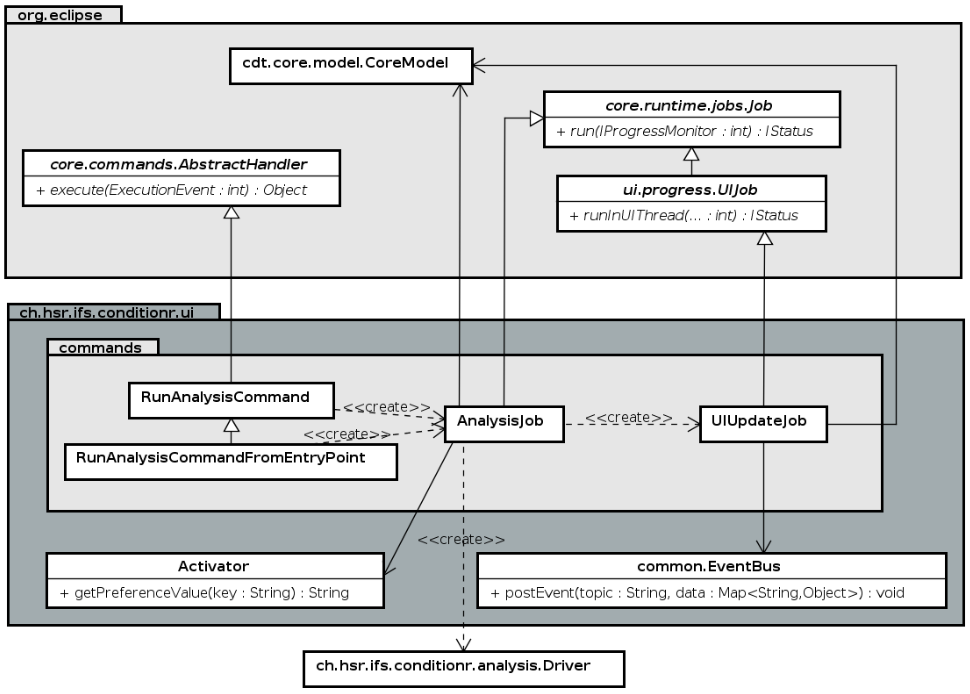

org.eclipse
cdt.core.model.CoreModel
core.runtime.jobs.Job
+ run(IProgressMonitor : int) : IStatus
core.commands.AbstractHandler
+ execute(ExecutionEvent : int) : Object
ui.progress.UIJob
+ runInUIThread(... : int) : IStatus
ch.hsr.ifs.conditionr.ui
commands
RunAnalysisCommand
RunAnalysisCommandFromEntryPoint
<<create>>
<<create>>
AnalysisJob
<<create>>
UIUpdateJob
<<create>>
Activator
+ getPreferenceValue(key : String) : String
common.EventBus
+ postEvent(topic : String, data : Map<String,Object>) : void
ch.hsr.ifs.conditionr.analysis.Driver

Figure 6.19: Package-Diagram for the classes of the commands package and their dependencies.

- `AnalysisJob`. This job is in charge of performing the data race analysis. If the job has been started, it goes through the following steps. First it gets all source files of the current active Eclipse project from the `CoreModel`. Then it checks the entry point. If the entry point is not valid, the job will be canceled. Afterwards it loads the tool configuration from the plug-in store via the `Activator`. Then it instantiates the `Driver` from the analysis component and runs the analysis. When the analysis is complete, the received result is converted into the data race model described in section 6.6.3 and schedules a `UIUpdateJob` with the converted result.

  The `AnalysisJob` runs asynchronously in a separate thread to prevent it from blocking the UI.

- `UIUpdateJob`. This job displays the result in Eclipse. It creates code markers for the editor, that underline the source locations of the potential data races and publishes the result on the `common.EventBus` which informs the view to display the data races.

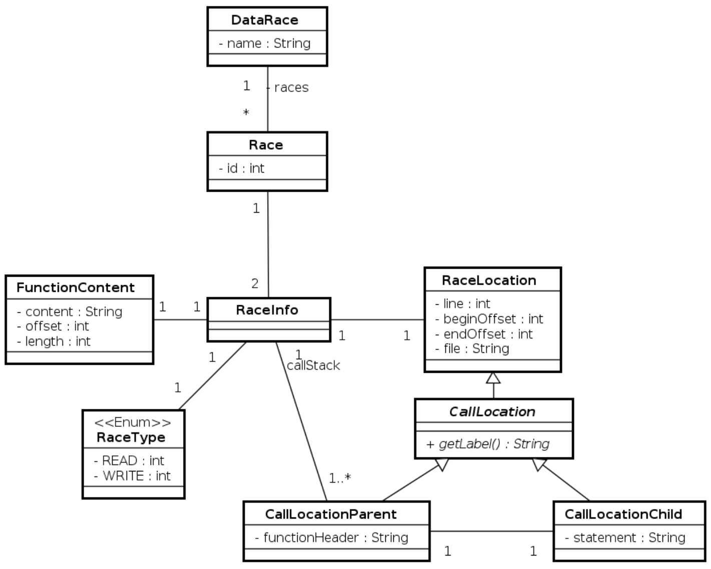


Figure 6.20: Domain model of the data race for the view.

6.6.3 datarace

This package contains the data race model for the view and a sub-package `datarace.util`, which converts the data races received from the algorithm component and provides additional information about a data race from the AST. Figure 6.20 shows the domain model of a data races for the view. The following list describes each class of the domain model.

- `DataRace`. This class contains all information about data races for one memory location. It has an attribute `name`, which contains the name of the location and a list of `races` with all associated data races.
- `Race`. This class contains the information about a data race. Each race has an internal `id` and two `RaceInfo`s, one for each location of the data race.
- `RaceInfo`. This class contains all information about a specific data race location. It has a `RaceType`, the `FunctionContent`, the `RaceLocation` and a `callStack`, which
- `RaceLocation`. This class contains the position of a data race in a file. The position is represented by the attributes `line` for the line number, a `beginOffset` and an `endOffset` for the marking of the location, and a `file` which contains the absolute path of the file.
- `FunctionContent`. This class contains information about the function where the data race occurs. The `content` attribute contains the whole

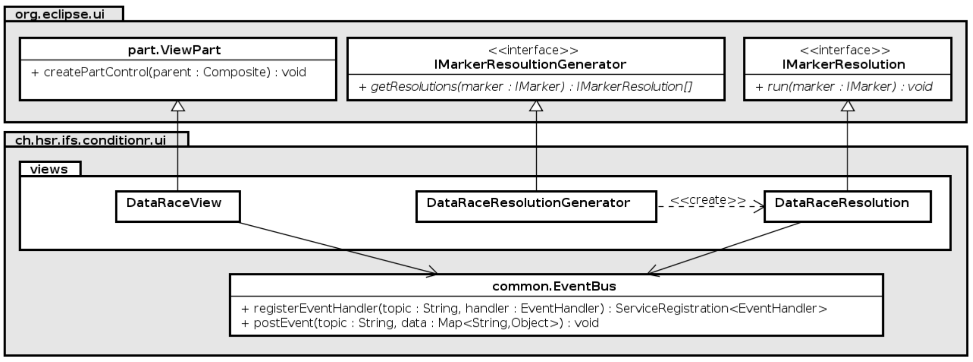

org.eclipse.ui
part.ViewPart
+ createPartControl(parent : Composite) : void
<<interface>>
IMarkerResoultionGenerator
+ getResolutions(marker : IMarker) : IMarkerResolution[]
<<interface>>
IMarkerResolution
+ run(marker : IMarker) : void
ch.hsr.ifs.conditionr.ui
views
DataRaceView
DataRaceResolutionGenerator
<<create>>
DataRaceResolution
common.EventBus
+ registerEventHandler(topic : String, handler : EventHandler) : ServiceRegistration<EventHandler>
+ postEvent(topic : String, data : Map<String,Object>) : void

Figure 6.21: Package-Diagram for the classes of the view package and their dependencies.

function code with declaration and body. The `offset` and `length` attributes describes the position of the data race inside the function. The `FunctionContent` class is used by the view to display a code snippet of the data race.

- `RaceType`. This is an enumeration type with the values `WRITE` for a write access, and `READ` for a read access of a memory location.
- `CallLocation`. This is an abstract class which extends the `RaceLocation` with an `getLabel()` method which is used for displaying the call stack as a `TreeView`.
- `CallLocationParent`. This class is used for the `TreeView` of a call stack as a parent component. It contains information about the function (`functionHeader`) and on which line the call inside the function is.
- `CallLocationChild`. This class is used for the `TreeView` of a call stack as a child component for a `CallLocationParent`. It contains information about the `statement` of a call.

6.6.4 view

This package contains the classes for the view and for the marker resolutions. At the end of this bachelor thesis it only contains a view for the data races. In the future, this package could be expand with new views like a view for deadlocks. Each view implements the `org.eclipse.ui.part.ViewPart`, which is the abstract base implementation of all workbench views. The content of the `views` package, including important dependencies, is illustrated in figure 6.21. In the following list the classes of this packages are described.

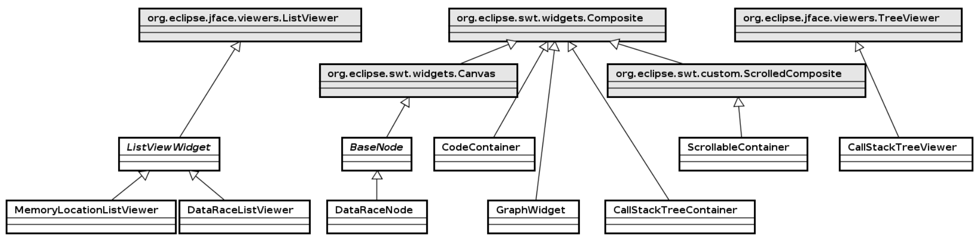

org.eclipse.jface.viewers.ListViewer
org.eclipse.swt.widgets.Composite
org.eclipse.jface.viewers.TreeViewer
org.eclipse.swt.widgets.Canvas
org.eclipse.swt.custom.ScrolledComposite
ListViewWidget
BaseNode
CodeContainer
ScrollableContainer
CallStackTreeViewer
MemoryLocationListViewer
DataRaceListViewer
DataRaceNode
GraphWidget
CallStackTreeContainer

Figure 6.22: Diagram of the `view.widgets` package with their class dependencies.

- `DataRaceView`. This class implements the view for the data races. It contains all widgets which are described in section 6.6.4. The class also subscribes to the `common.EventBus` to get informed which data races it has to display.

- `DataRaceResolutionGenerator`. Factory class for new `DataRace-Resolution` instances. This class implements the `IMarkerResolution-Generator` interface.

- `DataRaceResolution`. This class is the marker resolution for a data race. When the resolution is invoked by a user, it publishes the data race on the event bus to inform the view.

view.widgets

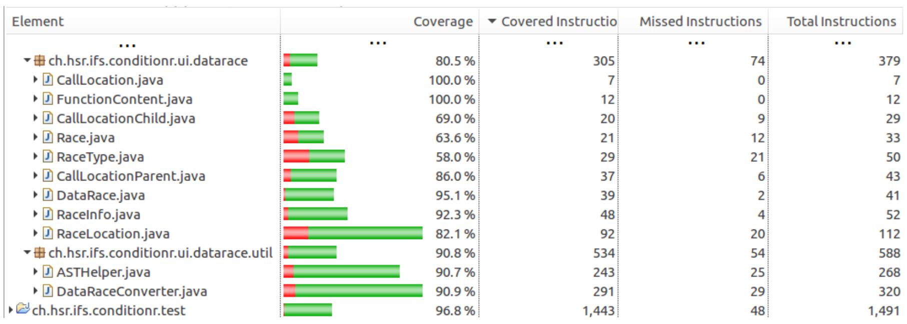
This package contains all custom widgets of the plug-in which are used to create the `DataRaceView`. Most of the widgets extend some SWT classes, but there are also some more complex widgets which uses the JFace library. Most code ideas for writing the widgets originate from the website `http://www.vogella.com/tutorials/eclipse.html` [Gmb]. Figure 6.22 shows a diagram with all created widgets and their dependencies from SWT and JFace.

In the following list the most important widgets are briefly described.

- `MemoryLocationListViewer`. This widget class extends the JFace `List-Viewer`. It lists, after a data race analysis, all memory locations, which contain data races.

- `DataRaceListViewer`. This widget class is like the `MemoryLocationList-Viewer`, but instead of the memory locations, it lists all data races for the chosen memory location.

- `DataRaceNode`. The `DataRaceNode` is a clickable widget element. It uses the `RaceInfo`, described in section 6.6.3, to display information about



Element	Coverage	Covered Instructio	Missed Instructions	Total Instructions
ch.hsr.ifs.conditionr.ui.datarace	80.5 %	305	74	379
CallLocation.java	100.0 %	7	0	7
FunctionContent.java	100.0 %	12	0	12
CallLocationChild.java	69.0 %	20	9	29
Race.java	63.6 %	21	12	33
RaceType.java	58.0 %	29	21	50
CallLocationParent.java	86.0 %	37	6	43
DataRace.java	95.1 %	39	2	41
RaceInfo.java	92.3 %	48	4	52
RaceLocation.java	82.1 %	92	20	112
ch.hsr.ifs.conditionr.ui.datarace.util	90.8 %	534	54	588
ASTHelper.java	90.7 %	243	25	268
DataRaceConverter.java	90.9 %	291	29	320
ch.hsr.ifs.conditionr.test	96.8 %	1,443	48	1,491

Figure 6.23: Test coverage, created with EclEmma, of the `ch.hsr.ifs.conditionr.ui.datarace` and `datarace.util` packages only of the project.

the race. If the race is a read access, it is a blue circle with an `R`, if the race is a write access, it is a read circle with an `W`.

- `CodeContainer`. This widgets displays the code snippet of a data race in an editor and highlights the location. The editor is an `org.eclipse.cdt.internal.ui.editor.CSourceViewer`;
- `GraphWidget`. This widget is the ground view of the right site of the data race view. It creates all `CodeContainers`, `DataRaceNodes` and `CallStackTreeViewers`, and updates them, if the view has been resized or an new data race has to be displayed.
- `CallStackTreeViewer`. This widget extends the JFace `TreeViewer`. It lists for a data race location their call stack. A parent of tree is a `CallLocationParent` (see section 6.6.3) and a child of a parent a `CallLocationChild`.

6.6.5 Testing

The visualization plug-in is tested using the JUnit framework. In the version provided by the masters thesis, nothing was tested. Most of the newly added logic, like the data race domain model or the conversion from the algorithm data race to the data race view model, is tested. The views package is not tested, because most of the code there is creation of widgets or positioning of them. The effort for writing view tests would have outweighed the benefits. Also excluded from testing are packages which only contain static constant strings or minimal logic like `preferences` or `common`. The `commands` package only has a few tests, because most of the methods there only use Eclipse infrastructure, for example to retrieve project files or start a job.

Figure 6.23 shows the test coverage of the `datarace` and datarace.util packages. Simple getters and setters are not tested. This is why the coverage of the `datarace` package is only around 80%.

7 Final Evaluation

This chapter describes the final evaluation of the improved ConditionR tool available at the end of this thesis.

First, the results of the initial evaluation are recapitulated. Then, the project used during the final evaluation is described. Finally, the results are presented.

7.1 Motivation

The goal of this second evaluation is to give a concise overview of what was achieved during this thesis to improve the user experience.

At the beginning of the thesis, an initial evaluation was conduced. It revealed many issues with the original implementation of all components. The results of that initial evaluation served as a basis on which to improve the ConditionR tool towards a state in which real-world project can be analyzed.

The following sections

7.2 Project Selection

Due to time constraints, a number of performance issues with the new implementation could not be fully addresses at the time of finishing this thesis. This makes it currently infeasible to apply the new version of the ConditionR tool to the Chess project evaluated in the initial evaluation. Nevertheless, many fundamental issues with the original implementation were resolved, which enables the tool to be applied to C++ source code much closer to what actual real-world projects typically look like.

The final evaluation is thus conducted using a C++ project developed to showcase the improvements of the new implementation. This code is slightly more complex than a typical test case as it contains multiple aspects, but much less complex than a real-world C++ project. However, it does embody many aspects that were previously not at all supported by the ConditionR tool but now are.

7.2.1 Example Project

This section presents the source code of the example project used in this evaluation.

Structure

The project consists of two TUs, one defined in the source file `main.cpp`, shown in listing 7.1, called *main*, and a second one in `foo.cpp` called *foo*, shown in listing 7.3. *Main* includes the header file `foo.h` to import the function `void foo(int*)` defined in *foo*.

The TU *main* contains a global variable of type `int` named `global` and the definition of the main function `int main(void)`. The main function first starts a thread defined using an anonymous lambda function. The thread increments the integer variable `global`. Incrementing a number entails first reading its current value, adding one to it, and finally writing the new value to the original memory location. After starting a thread, it calls the function `foo(int*)`, passing as an argument a pointer to the variable `global`.

The function `foo(int*)` defined in the TU *foo* dereferences the pointer received as an argument and reads the current value stored at that memory location. It then increments the value by one and writes the result to the same memory location.

Since the pointer passed to `foo` points to the memory location `global` written to in the thread started by `main`, a data race is to be expected. The algorithm correctly determines that this is indeed a data race.

C++ source code

```cpp
#include "foo.h"

#include <thread>

int global;

int main() {
  std::thread t { []() { global++; } };
  foo(&global);
  t.join();
}
```

Listing 7.1: The file `main.cpp` of the evaluation project. It contains a global variable of type int and the definition of the function `main`. The `main` function starts a thread defined using an anonymous lambda function. The thread increments the global variable. After starting the thread the function `foo` located in a different translation unit is called.

```cpp
#ifndef FOO_H
#define FOO_H

void foo(int*);

#endif
```

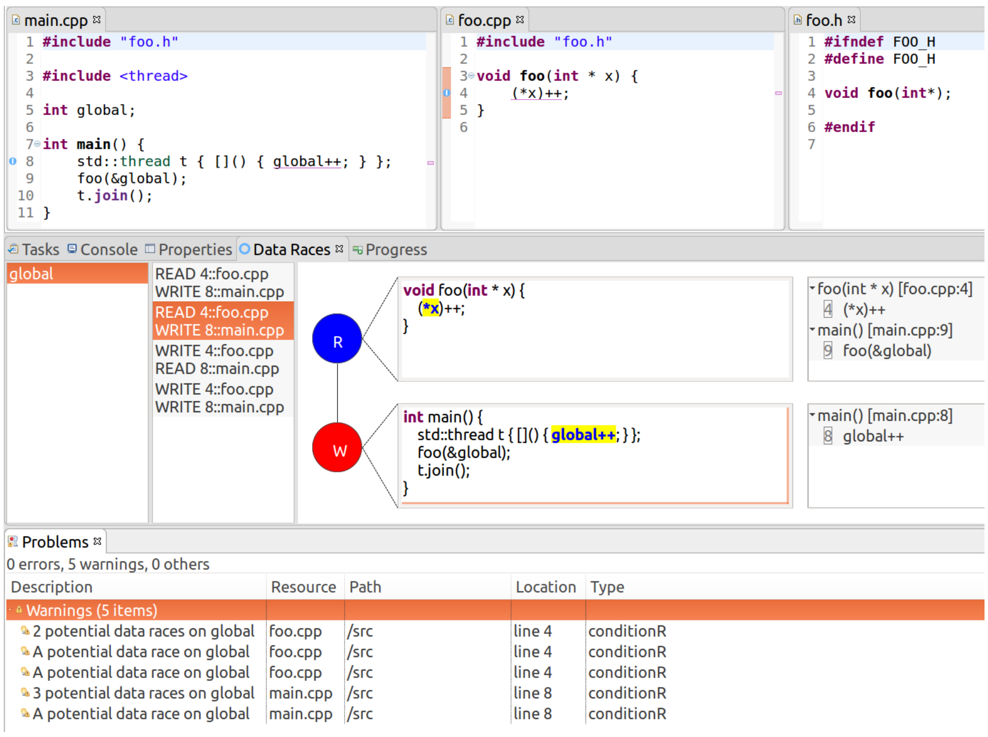
Listing 7.2: The file `foo.h` of the evaluation project. It declares the function `foo` and is included in both translation units `main` and `foo`.

```cpp
#include "foo.h"

void foo(int * x) {
  (*x)++;
}
```

Listing 7.3: The file `foo.cpp` of the evaluation project. It contains the definition of the function `foo` which takes a pointer parameter of type `int*` and increments the referenced integer.

7.3 Results

The initial implementation of the ConditionR tool lacked support for many basic C++ programming constructs as well as basic aspects such as function calls across TUs. For many of these aspects, if they were present in the program, the extraction step would crash and an analysis could not even


main.cpp

```
 1 #include "foo.h"
 2
 3 #include <thread>
 4
 5 int global;
 6
 7 int main() {
 8     std::thread t { []() { global++; } };
 9     foo(&global);
10     t.join();
11 }
```

foo.cpp

```
1 #include "foo.h"
2
3 void foo(int * x) {
4     (*x)++;
5 }
6
```

foo.h

```
1 #ifndef FOO_H
2 #define FOO_H
3
4 void foo(int*);
5
6 #endif
7
```

```
global    READ 4::foo.cpp
          WRITE 8::main.cpp
          READ 4::foo.cpp
          WRITE 8::main.cpp
          WRITE 4::foo.cpp
          READ 8::main.cpp
          WRITE 4::foo.cpp
          WRITE 8::main.cpp
```

```
void foo(int * x) {
    (*x)++;
}
```

```
int main() {
    std::thread t { []() { global++; } };
    foo(&global);
    t.join();
}
```

```
foo(int * x) [foo.cpp:4]
  4  (*x)++
main() [main.cpp:9]
  9  foo(&global)
```

```
main() [main.cpp:8]
  8  global++
```

Problems

0 errors, 5 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (5 items)				
2 potential data races on global	foo.cpp	/src	line 4	conditionR
A potential data race on global	foo.cpp	/src	line 4	conditionR
A potential data race on global	foo.cpp	/src	line 4	conditionR
3 potential data races on global	main.cpp	/src	line 8	conditionR
A potential data race on global	main.cpp	/src	line 8	conditionR

Figure 7.1: A screenshot showing the visualization of the example project used in the final evaluation. The data race on the memory location `global` is correctly detected and precisely marked in the source code.

be partially executed. In the version of ConditionR available now, a project like this can be fully analyzed and data races are correctly reported. The visualization resulting from the project outlined here is shown in figure 7.1.

8 Conclusion

This chapter reviews the project described in this document. It summarizes the results of this thesis and concludes with an outlook for future work on this project.

8.1 Result

During this bachelor thesis, the static analysis tool ConditionR has been improved and extended with the following results:

- Support for source code located in multiple TUs and function calls across TUs.
- Support for C++ member functions and variables was added.
- Support for lambdas created in the `std::thread` constructor.
- The data race visualization for the Eclipse plug-in has been completely revamped and improved to give a user an informative overview of all data races found inside a project.
- The data race analysis can be started using an arbitrary function as the entry point.
- For future development, continuous integration on a build server was set up.

The *productizing* goal of getting the project into a market ready state could not be completely reached due to an overestimation of the already implemented capabilities at the beginning of the thesis. Nevertheless, a significant step towards that goal was made.

8.2 Outlook

The analysis tool is not completely market ready yet. To allow the program to be used with any kind of real world code further improvements are needed. The following list gives an overview example for futures developments of this tool.

- Increasing the number of supported C++ constructs.
- Support for lambdas captures when used as a normal function.
- Fixing currently known bugs and improve performance.

The deployment in its current form only fully supports Linux with recent Clang versions. The current approach of using a Clang checker to generate the control flow graph should perhaps be reconsidered.

8.3 Acknowledgments

We want to thank our advisor Prof. Peter Sommerlad for his guidance and valuable feedback during this bachelor thesis.

We also want to thank Mario Meili for the helpful discussions during the project meetings and for giving us insightful feedback on the documentation.

Glossary

Clang C based language front-end supporting C, C++, Objective C/C++, OpenCL C, and others for the LLVM compiler. 19, 21, 25, 65

EclEmma A free Java code coverage tool for Eclipse. 58

Eclipse An integrated development environment (IDE) to write C++ and Java code. 6, 8, 39, 46, 51, 58, 64

Jenkins Jenkins is an open source automation server written in Java. 72

JFace JFace is a set of helpful complex widgets relying on SWT. 57, 58

JUnit A unit testing framework for the Java. 58

Scala Scala is a multi-paradigm programming language based on the JVM. 15, 30

Acronyms

AST abstract syntax tree. 21, 29, 55

CI Continuous Integration. 72

CLI Command Line Interface. 6, 25, 28

GTK GIMP-Toolkit. 42

JSON JavaScript Object Notation. 69, 71

SWT Standard Widget Toolkit. 42, 57

TU translation unit. 1, 2, 13, 18, 20, 22, 23, 27, 30, 32, 33, 61, 62, 64

Bibliography

[Blä15] Luc Bläser. *Swissreg - Eidg. Institut für Geistiges Eigentum.* May 4, 2015. URL: `https://www.swissreg.ch/srclient/faces/jsp/patent/sr300.jsp?language=de&section=pat&id=CH711035` (visited on 06/12/2017).

[Bru16] Silvano Brugnoni. "ConditionR - A Static Data Race Detection Tool for C++11". Masters thesis. University of Applied Sciences Rapperswil, May 19, 2016. eprint: `https://eprints.hsr.ch/id/eprint/507`.

[Che] *Checker Developer Manual.* URL: `http://clang-analyzer.llvm.org/checker_dev_manual.html` (visited on 02/27/2017).

[Cla] *Clang - Getting started.* URL: `http://clang.llvm.org/get_started.html` (visited on 02/27/2017).

[Gmb] Vogella GmbH. *Eclipse, RCP, Plugin and OSGi Development.* URL: `http://www.vogella.com/tutorials/eclipse.html` (visited on 03/02/2017).

[Loo] *Handling of loops in the Clang Static Analyzer.* URL: `http://clang-developers.42468.n3.nabble.com/Handling-of-loops-in-the-Clang-Static-Analyzer-td4055475.html` (visited on 04/30/2017).

[Rap] *RapidJSON.* URL: `http://rapidjson.org/` (visited on 02/27/2017).

[SJ16] Fabian Schläpfer and Samuel Jost. "Studienarbeit ConditionR". Student Research Project thesis. University of Applied Sciences Rapperswil, Dec. 23, 2016.

A JSON Format

listing A.1 shows the current JavaScript Object Notation (JSON) output format of the `cr-extractor`.

```
// Basic buildup
[
  [
    { <graph> }
  ]
]

// graph
{
  "graph" : "<function_name>",
  "fileLocation" : { <file_location> },
  "parameters" : [ { <parameter> } ],
  "nodes" : [ { <basic_block> } ]
}

// file_location
{
  "line" : 1..n,
  "beginFileOffset" : 1..n,
  "endFileOffset" : 1..n,
  "file": "<file_path>"
}

// basic_block
{
  "id" : "<block_id>",
  "predecessors" : [ "<block_id>" ],
  "successors" : [ "<block_id>" ],
  "statements" : [ { <statement> } ]
}

statement : <call> | <lock> | <memory_access> |
   <thread_start> | <thread_join>

```

```
// call
{
  "type" : "UnresolvedCall",
  "functionName" : "<function_name>",
  "captures" : [ { <capture> } ],
  "arguments" : [ { <argument> } ],
  "fileLocation" : { <file_location> }
}

// lock
{
  "type" : "<lock_type>",
  "mutex" : "<variable_name>",
  "fileLocation" : { <file_location> }
}

// memory_access
{
  "type" : "<access_type>",
  "memoryLocation" : { <memory_location> },
  "fileLocation" : { <file_location> }
}

// thread_start
{
  "type" : "ThreadStart",
  "threadId" : "<variable_name>",
  "entryPoint" : "<function_name>",
  "captures" : [ { <capture> } ],
  "arguments" : [ { <argument> } ],
  "fileLocation" : { <file_location> }
}

// thread_join
{
  "type" : "ThreadJoin",
  "threadId" : "<thread_id>",
  "entryPoint" : "<function_name>",
  "fileLocation" : { <file_location> }
}

// parameter
{
  "type" : "<parameter_type>",
  "position" : 0..n,
  "name" : "<parameter_name>"
}

```

```
// argument
{
  "type" : "<argument_type>",
  "position" : 0..n,
  "memoryLocation" : { <memory_location> }
}

// capture
{
  "type" : "<capture_type>",
  "isThisCapture" : true | false,
  "memoryLocation" : { <memory_location> }
}

// memory_location
{
  "name" : "<location_name>",
  "scope" : { <scope> }
}

// scope
{
  "type" : "<scope_type>",
  "parent": { <scope> }
}

scope_type : global | local | member
capture_type : byref | byval
argument_type : readonly | readwrite
parameter_type : parameter type including const
   etc.

access_type : Read | Write
lock_type : AcquireLock | ReleaseLock

location_name : variable name
parameter_name : name of the function parameter,
   can be empty

block_id : <function_name>/<id>
id : entry | 1..n | exit
function_name : mangled function name | main
```

Listing A.1: JSON format overview

B Continuous Integration

This part serves as an overview of things to keep in mind when dealing with Continuous Integration (CI) and the ConditionR tool. CI is done using Jenkins as the server platform. The different projects all have specific configurations needed to building and running the tests.

Full integration testing means that a call to the algorithm with some given files, Clang and the checker is simulated and the final result is checked. This requires that every part is successfully built and also that a compatible Clang binary is used.

```bash
#!/bin/bash

UPDATE_SITE="http://sinv-56089.edu.hsr.ch/jenkins/job/conditionr-dependencies-updatesite/ws/ch.hsr.ifs.conditionr.dependencies/target/repository/plugins/"
TARGET_FILE="../ch.hsr.ifs.conditionr.target/ch.hsr.ifs.conditionr.target.target"

PLUGIN_NAME="ch.hsr.ifs.conditionr-analysis"
SNAPSHOT_VERSION="[0-9]*[.][0-9]*[.][0-9]*[.][0-9]*"
PREFIX_VERSION="[0-9]*[.][0-9]*"
VERSION_PATTERN="${PREFIX_VERSION}[_]${SNAPSHOT_VERSION}"

CURRENT=`wget -O - ${UPDATE_SITE} \
        | grep -oP ${PLUGIN_NAME}_${VERSION_PATTERN} \
  | head -n 1 \
  | grep -oP ${VERSION_PATTERN}`

CURRENT_SNAPSHOT=`echo ${CURRENT} | grep -oP ${SNAPSHOT_VERSION}`
CURRENT_PREFIX=`echo ${CURRENT} | grep -oP ${PREFIX_VERSION} | head -n 1`

```

```
sed -i "s/<unit
    id=\"${PLUGIN_NAME}_${PREFIX_VERSION}\"
    version=\"${SNAPSHOT_VERSION}\"\/>/<unit
    id=\"${PLUGIN_NAME}_${CURRENT_PREFIX}\"
    version=\"${CURRENT_SNAPSHOT}\"\/>/g"
    ${TARGET_FILE}
```

Listing B.1: Shell script used to automatically update the Algorithm target before running the integration tests.