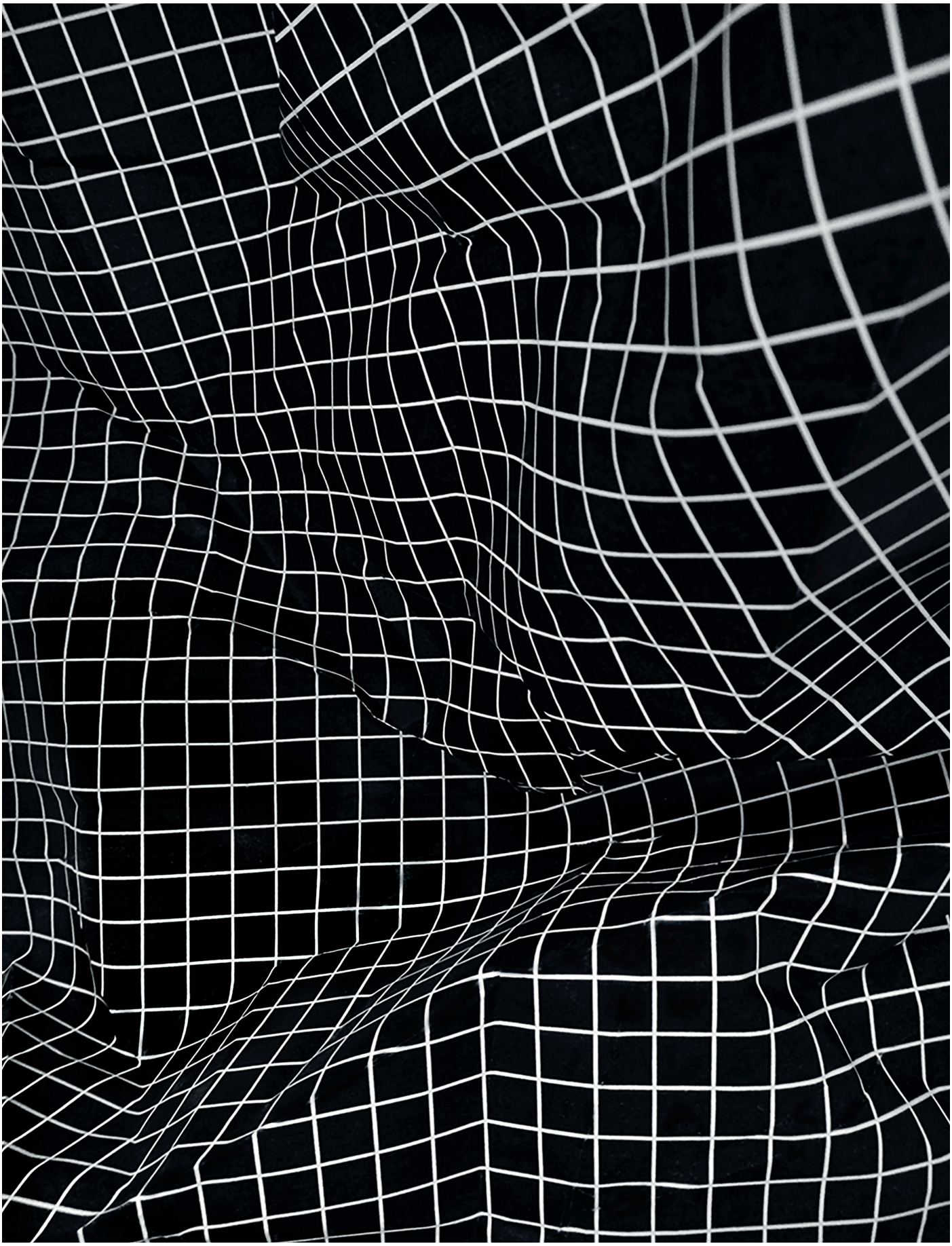
ConditionR

A Static Data Race Detection Tool for C++11

MASTER THESIS
University of Applied Sciences Rapperswil

Author: Silvano Brugnoni
Supervisor: Prof. Peter Sommerlad
Technical Advisor: Thomas Corbat



UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL

MASTER THESIS

ConditionR
A Static Data Race Detection Tool for C++11

Author:
Silvano Brugnoni

Supervisor:
Prof. Peter Sommerlad

Technical Advisor:
Thomas Corbat

August 31, 2015

Abstract

The widespread use and popularity of multiprocessors in today's computing world has incentivized many developers to start writing concurrent software, which in turn has lead many modern object-oriented languages to provide support for multi-threading. With the release C++11 standard, the C++ language has followed this trend as well by introducing a brand new threading library .

However, writing and debugging multi-threaded code is difficult. Simple errors in synchronization can produce timing-dependent data races that can take weeks or months to track down.

The goal of this project is to build a static analysis tool that provides automatic detection of data races in concurrent C++11 source code, in order to help developers write multi-threaded C++ code. The foundation for this tool is an algorithm that was originally developed by Prof. Dr. Luc Bläser at the HSR Institute for Software, and currently has a patent pending under the name *Efficient Static Thread-Start-Join-Sensitive Detection of Low-Level Data Races and Deadlocks* at the Swiss Federal Institute of Intellectual Property.

The resulting software tool - named *ConditionR* - is able to detect data races in programs that use the most common C++11 concurrency features, and provides rudimentary support for interprocedural analysis across translation units. The tool is integrated into Eclipse CDT. Because the interpretation of the algorithm results is not straightforward, the software tool features an interactive graphical visualization of the detected data races.

Management Summary

This thesis describes the design and development of a static analysis tool that provides automatic detection of data races in concurrent C++11 programs. The thesis was conducted at the Institute for Software at the University of Applied Sciences Rapperswil.

Motivation

The ever-growing popularity and availability of multiprocessors incentivized many developers to start writing concurrent software. Many modern object-oriented languages have reacted to this trend by providing either language-level or library-based support for multi-threading.

However, writing multi-threaded programs is error prone and difficult. It is easy to make mistakes in synchronization that produce data races, and yet these mistakes can be extremely hard to find during debugging. To help with this issue, Prof. Dr. Luc Bläser at the Institute of Software has developed an algorithm that is able to statically detect data races in program code. The purpose of this algorithm is to help programmers writing concurrent code by exposing data races, a type of error that is notoriously hard to find for humans. An implementation of this algorithm currently exists for C#.

With the C++11 standard, C++ has also followed the trend towards concurrency, by introducing a thread-aware memory model and a platform-independent threading library. However, the previously described difficulties for concurrent programming also apply to C++11. It would be great, if a tool with the power of Prof. Dr. Luc Bläser's algorithm existed to help developers write correct multi-threaded programs, and promote concurrent programming for C++.

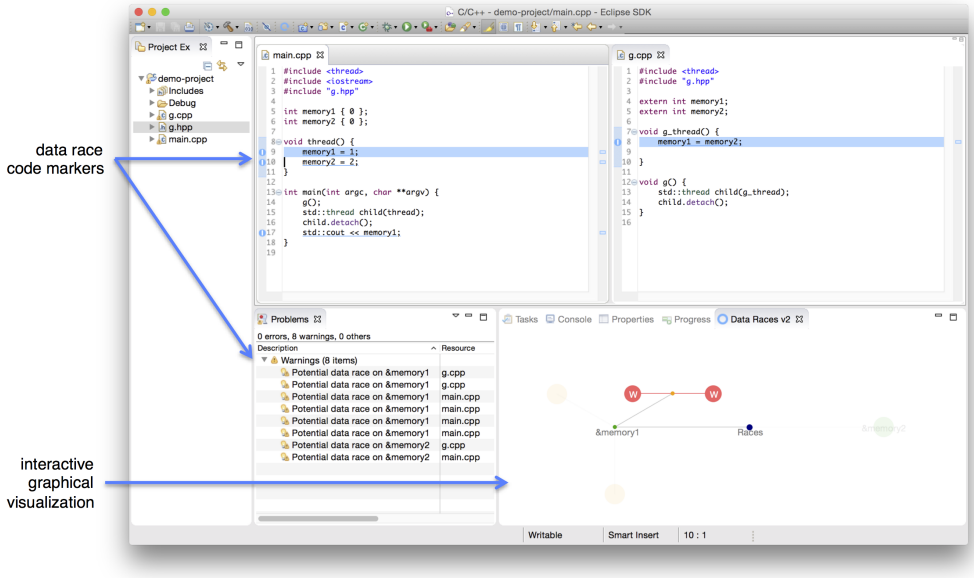
Goals

The main goal of this thesis is to build a tool that helps developers write correct multi-threaded programs by providing automatic detection of data races in C++ source code, based on the aforementioned algorithm. Given the complex nature of C++, combined with its still comparatively weak tooling and infrastructure support, the nature of this project is far from being a matter of simply re-typing the algorithm for C++. Many C++ specific challenges must be addressed, which are not present in most other languages. In fact, it is not required nor expected that the tool supports 100% of the C++ language features by the end of this project.

Results

At the end of the project, an analysis tool - named *ConditionR* - was delivered with the following features:

- The algorithm implementation is based on Clang and its static analysis engine. The tool is designed is such a way that Clang can be swapped out and replaced by a different backend, if needed.
- The user interface is implemented as an Eclipse CDT plug-in. As such, it employs common Eclipse UI elements, such as code markers and quick assists, making it easy to learn and use for developers familiar with Eclipse.
- The analysis supports programs that were built using the most common C++11 concurrency constructs `std::thread` and `std::mutex`.
- The tool supports cross-translation unit analysis for regular functions. This feature serves as a proof-of-concept to show that cross-TU analysis is possible.
- The analysis results are displayed using Eclipse code markers. Because the interpretation of the information contained in these markers is not straightforward, the tool also provides an interactive graphical visualization help developers understand and connect the analysis result data.



The developed analysis tool features an Eclipse-based user interface.

The analysis tool is currently not ready to be applied on real-world C++ programs, as it does not yet support all C++ concurrency constructs. However, it is a first step towards a world with better C++ tooling.

Declaration of Authorship

I, Silvano Brugnoni, declare that this thesis and the work presented in it is my own, original work. All the sources I consulted and cited are clearly attributed. I have acknowledged all main sources of help.

Location, Date: _____

Signature: _____

Contents

1. Introduction

The objective of this master thesis is to build a tool that helps developers write correct multi-threaded programs by providing automatic detection of data races in C++ source code. The foundation for this tool is an algorithm originally developed by Prof. Dr. Luc Bläser at the HSR Institute for Software. This algorithm currently has a patent pending under the name *Efficient Static Thread-Start-Join-Sensitive Detection of Low-Level Data Races and Deadlocks* at the Swiss Federal Institute of Intellectual Property.

This Chapter will first introduce the terminology and concepts pertaining to the theoretical aspects of this thesis. Afterwards, it will summarize related works and provide background information on the analysis techniques used in this thesis, as well as some competing approaches. The chapter will conclude with a motivation statement and a formal definition of the goals for this thesis.

It is assumed that the reader is familiar with the concepts of concurrency and multi-threaded programming with shared state. The introductory part is kept brief, in order to prevent too much digression and to keep the text concise and on topic. In many places, sources to papers and books will be provided that expand on the relevant subject matter in more detail.

1.1. Data Races

The widespread use and popularity of multiprocessors in today's computing world and their advantages has incentivized many developers to start writing concurrent software [PG01]. This has lead many modern object-oriented languages such as C++, C#, or Java to provide either language-level or library-based support for multi-threading.

However, this shift away from sequential programs towards concurrency is not without risks. It is widely accepted that writing correct multi-threaded programs using the *shared state concurrency model*, which most object-oriented languages do to express concurrency, is tremendously challenging [FF00][SBN+97]. In order to reason about these difficulties in a more generalized way and to show that they are inherent not only to a single language but an entire family thereof, they will be illustrated using the underlying programming model rather than a concrete language.

1.1.1. Shared State Concurrency Model

The shared state concurrency model consists of multiple *threads* that share a common memory space. A thread is simply a sequence of *instructions* that are executed in order. These instructions are operations that potentially access or mutate the shared memory space [RH04]. An example of a shared memory environment with multiple active threads is illustrated in Figure 1.1.

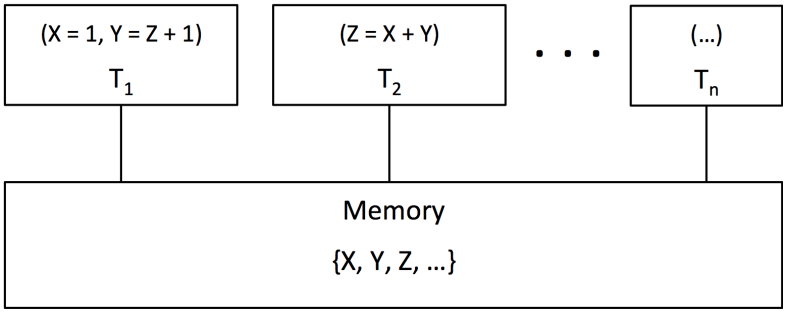


Figure 1.1.: The shared state concurrency model. The instruction sequences associated with Threads T_1 to T_n concurrently access and mutate cells in a shared memory space: T_1 contains two instructions. First, it stores the value 1 at the memory cell X. Second, it adds the value from cell Z and 1 and stores it at the memory cell Y. T_2 only contains one instruction: It adds the values from X and Y and stores them at the memory location Z. T_n contains an unknown sequence of instructions.

1.1.2. Concurrency and Parallelism

In the context of multi-threaded programming it is important to distinguish *concurrency* and *parallelism*. The two terms are strongly related, but not synonymous. Parallel program execution means truly simultaneous execution of independent threads, which is only possible on multiprocessors. Concurrent program execution describes a condition where multiple independent threads are active in overlapping time periods, but not necessarily executing in parallel. Concurrency is strictly speaking a more generalized form of parallelism; a parallel program is also concurrent but the opposite is not always true [AS83].

1.1.3. Instruction Interleaving and Non-Determinism

Concurrent programs with multiple active threads do not have a predetermined global instruction execution order. Instead, the program is executed using *instruction interleaving*. All concurrently active threads arbitrarily alternate in executing their instruction sequences, while their internal instruction order is preserved [KSY05]. This random interleaving of instructions from multiple threads results in a *non-deterministic* [EP88] global execution order. Van Roy [RH04] defines an execution order as non-deterministic "if there is an execution state in which there is a choice of what to do next, i.e, a choice of which thread to reduce [continue]". Therefore, the execution becomes non-deterministic as soon as more than one thread is concurrently active. Figure 1.2 shows an example of two concurrent threads doing instruction interleaving and some possible global execution orders, and Figure 1.3 illustrates the emergence of one possible global execution order in more detail.

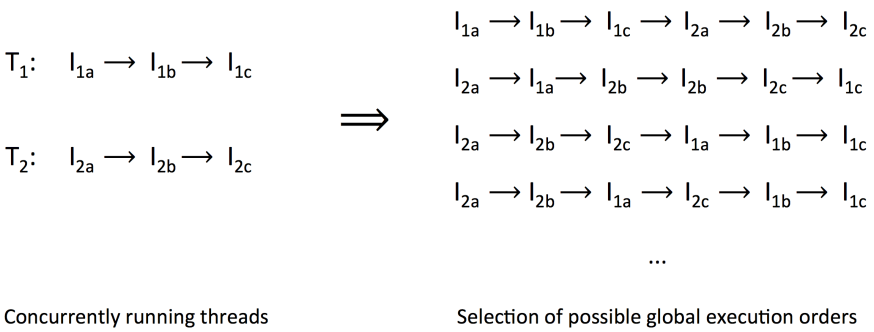


Figure 1.2.: Instruction Interleaving. The instruction sequences of the two threads T_1 and T_2 are executed in parallel. Due to random instruction interleaving, many different possible global execution orders can result from executing this program.

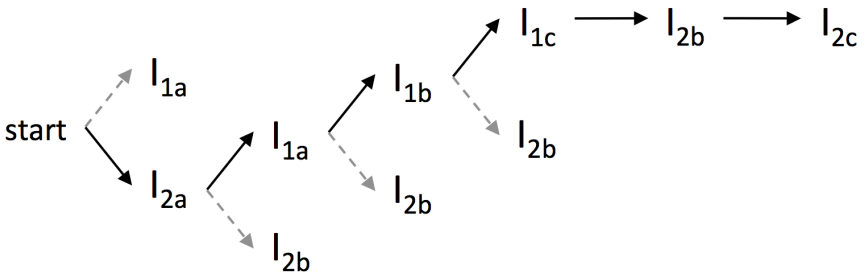


Figure 1.3.: Non-deterministic execution. The above illustration shows one possible global execution path that could result from executing the example in Figure 1.2. The dashed arrows represent alternative execution paths that were not taken

Sequential Consistency

Order-preserving instruction interleaving results in a *sequentially consistent* global execution [Lam79], given the executed program does not contain *data races* (explained later in Section 1.1.5). Sequential consistency has the advantage of being very intuitive to reason about. Unfortunately, it also inhibits many common compiler and hardware optimizations [AG96] because it severely restricts instruction reordering, and thus enforcing sequential consistency potentially has a strong impact on program performance [BA08]. This has lead many languages, including C++11, to provide the option to abandon sequential consistency in favor of more efficient program execution. However, giving up sequential consistency usually results in a less intuitive programming model.

The remainder of this chapter assumes sequentially consistent execution for the sake of simplicity. This fact is only relevant to a limited extent, because all of the properties shown still apply if the execution order is not sequentially consistent. Section 1.2.1 will elaborate more on the C++ memory model.

1.1.4. Thread Synchronization

The shared state concurrency model offers synchronization mechanisms to coordinate multi-threaded access to the shared memory. Using these mechanisms, program parts that must not be executed concurrently can be encapsulated into *critical sections*. A critical section is a sequence of instructions that is executed as if it was one *atomic* instruction. Atomic execution means that the resulting program state after critical section only depends on the program state at the start of the section and on the instructions inside the section [NM92].

[Ber66] states that atomic execution of a critical section is guaranteed if none of the shared variables that are read or modified inside the critical section are modified by any other concurrently executing thread.

Locks

The simplest way to implement a critical section is to protect it with a *lock*, also known as *mutex*. Locks enforce *mutually exclusive* access; they ensure that only one thread can enter the critical section at a time. Before entering a critical section guarded by a lock, the active thread needs to acquire the lock. If the lock is currently free, the thread is allowed enter to the critical section. If the lock is already taken by another thread, the active thread must wait at the entrance of the critical section until the lock is released by the thread currently holding it.

Monitors

Simple mutual exclusion may not be enough to express the desired synchronization logic. Van Roy [RH04] illustrates this using a bounded buffer: It is not sufficient to simply protect the buffer with a lock, because the access conditions are more complex. For instance, a thread that wants to insert an element into a full bounded buffer must wait until more space is available, even if no other threads are competing for the access at the moment. This cannot be expressed with simple locks.

Monitors extend the lock semantics with the ability to control the access to critical sections based on freely programmable conditions. The concept of monitors was first introduced in [Hoa74] and continues to be the standard way of coordinating threads in the shared state concurrency model [RH04].

1.1.5. Race Conditions

The inherently non-deterministic execution order of programs written using the shared state concurrency model is the cause if their natural susceptibility to a class of errors called *race conditions* [NM92]. A race condition exists anywhere at least two concurrently active threads access shared memory in a non-deterministic order.

Data Races

Race conditions that cause non-atomic execution of critical sections are commonly called *data races* [PG01]. They are almost always considered programming errors, and they have proven to be incredibly difficult to locate and fix [RB00][MC91][SBN+97]. In C++11, data races are considered to be *undefined behavior*.

General Races

[NM92] formally defines a different kind of race condition called a *general race*, where incorrect thread synchronization causes programs to exhibit non-deterministic behavior, even though no data races exist. For instance, a parallel for-loop that has data dependencies between each iteration may behave non-deterministically if these dependencies are not properly expressed.

In Listings 1.1 and 1.2, such a general race is exemplified using a partial sum function. Listing 1.1 shows a straight-forward sequential implementation. The function takes a sequence defined by two $InputIterators$ as input and writes the result to an $OutputIterator$. The iterative part of the computation is implemented using an std::for_each and a lambda function. Listing 1.2 shows a parallel implementation of the same function, where the std::for_each is swapped with a parallel_for (cf. Appendix B.1 for the implementation details) to parallelize the iteration steps. Mutually exclusive access to the shared variables from the concurrently running iteration steps is enforced by a std::mutex. This parallel implementation is free of data races. However, the synchronization does not respect the temporal data dependencies: The iteration steps are executed in an unpredictable order, and because each iteration step depends on the data provided of the previous one, the function output becomes unpredictable.

Figure 1.4 depicts a simplified view of the parallel computation in order to illustrate the general race more clearly. The instructions I_n get executed atomically. After I_1 the execution order becomes non-deterministic. A race condition is present because the execution order influences the result of the function. It is not a data race however, as there are no violations of critical sections. The thread synchronization statements in the Listing have been omitted.

Listing 1.1: A simple sequential implementation of a partial sum function in C++. The function computes the partial sum of an arbitrary sequence defined by the two InputIterators *begin* and *end*, and writes the results the OutputIterator *output*.

```
#include <iterator>
#include <algorithm>
using namespace std;

template<class InputIt, class OutputIt>
void sequential_partial_sum(InputIt begin, InputIt end, OutputIt output) {
  using value_type = typename iterator_traits<InputIt>::value_type;
  value_type sum{};
  for_each(begin, end, [&](value_type current) {
    sum = sum + current;
    *output++ = sum;
  });
}
```

Listing 1.2: Naive parallel implemenation of the partial sum function in Listing 1.1. Instead of std::for_each, this implementation uses a parallel_for (cf. Appendix B.1 for the implementation) in order to execute the iteration steps concurrently. The lines shaded in blue have been added to synchronize the access and mutation of the shared variables *sum* and *output*.

```
#include <iterator>
#include <thread>
#include <vector>
using namespace std;

template<class InputIt, class UnaryFunction>
UnaryFunction parallel_for(InputIt first, InputIt last, UnaryFunction f);

template<class InputIt, class OutputIt>
void concurrent_partial_sum(InputIt begin, InputIt end, OutputIt output) {
  using value_type = typename std::iterator_traits<InputIt>::value_type;
  value_type sum{};
  mutex lock{};
  parallel_for(begin, end, [&](value_type current) {
    lock_guard<mutex> guard {lock};
    sum = sum + current;
    *output++ = sum;
  });
}
```

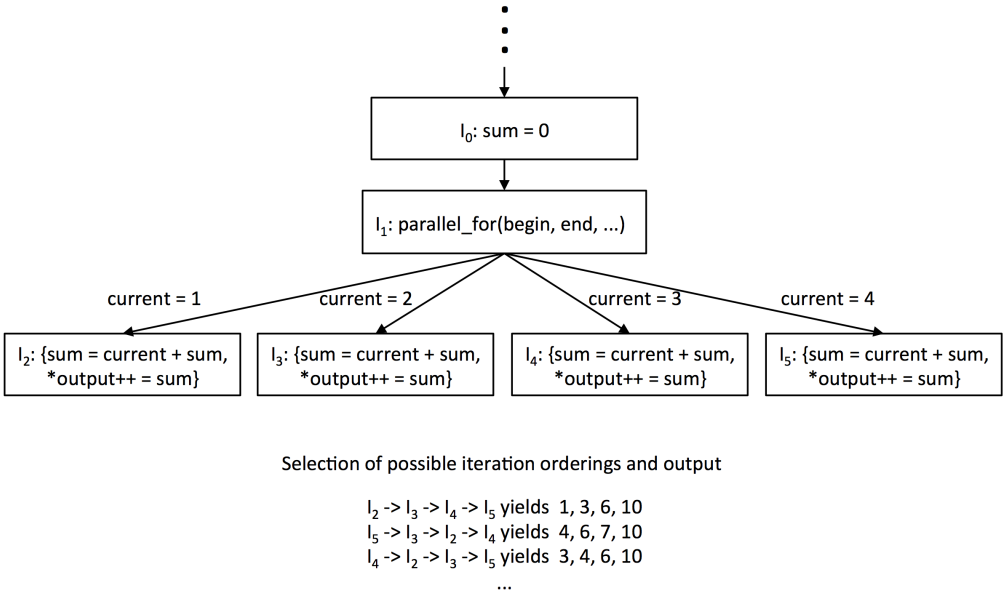


Figure 1.4.: Visualization of the general race exemplified in Listing 1.2. The instructions I_n get executed atomically, but after I_1, the execution order becomes non-deterministic. A race condition is present because the execution order influences the result of the function. The thread synchronization statements have been omitted.

1.2. C++11 Concurrency

Prior to C++11, the C++ language was defined as a single-threaded language without any reference to threads. Multi-threaded applications had to rely on the thread libraries provided by the operating systems, for instance the POSIX Thread API, also known as *pthread* [65013]. In accordance, the compilers were largely unaware of threads and generated code that was optimized for single-threaded execution. The lack of restrictions pertaining to thread communication allowed them to perform code transformations and almost unconstrained instruction reordering that did not preserve the meaning of multi-threaded programs [AG96][SS88]. In order to address this issue and support multi-threaded programming, the OS thread libraries provided synchronization operations such as mutexes and monitors themselves (cf. Section 1.1.4). These operations were implemented in a way that prevented them from being reordered with respect to regular memory accesses [BA08].

However, this informal specification of multi-threaded semantics was considered ambiguous, difficult to reason about, and lacking both in terms of portability and performance [Boe05]. In an effort to address these issues, the semantics for multi-threaded programs were formally specified and released as part of the C++11 language standard. The key results of this specification are a memory model for multi-threaded programs in combination with a platform-agnostic thread API.

1.2.1. Memory Model

There is a tension between the usability of sequential consistency, which has the downside of precluding many hardware and compiler optimizations, and the performance benefits of weaker semantics that are in turn more difficult to reason about. The C++ memory model addresses this issue by providing three distinct synchronization semantics:

- Traditional lock-based, sequentially consistent synchronization primitives are available for typical application code that does not use atomics.
- Sequentially consistent atomics are available for programs that need concurrent memory accesses but performance is not critical.
- Low-level atomics featuring weaker, but less expensive synchronization semantics are available for programs that require high-performance concurrency.

Accordingly, the memory model itself is not sequentially consistent by default. Instead, the total ordering of all memory actions with respect to each other is specified using *happens-before* relations. Happens-before is an accumulation of many other relationships between the memory actions, which identify thread synchronizations, visible side effects, and so on. These relationships are derived partly from the concurrency semantics specified by the programmer, and from the evaluation semantics. They generally constrain the order in which the actions can be executed. The rigor of these constraints determines the freedom the compiler is given with respect to instruction reordering when optimizing a program. Memory models based on this principle are called *weak* or *relaxed* [BOS+11]. Programs based on a relaxed memory model can be made sequentially consistent by constraining the relationships between the memory actions in such a way that only allows sequentially consistent global execution orders.

This thesis will focus on C++ programs that are written using either traditional lock-based synchronization primitives, or sequentially consistent atomics. It will refrain from supporting weaker consistencies for two reasons. The race detection algorithm currently does only support sequential consistency semantics, and programmers that rely on relaxed consistency models generally have to have a good idea of what they are doing anyway.

1.2.2. Threading Library and Concurrency Features

This section introduces the new classes for managing threads, protecting shared data, and synchronizing operations between threads, which have been added to the C++ Standard Library with the C++11 standard. Since these additions were quite extensive, only the ones directly related to the project will be covered. A very good resource for additional information regarding C++11 Concurrency is Anthony Williams' "C++ Concurrency in Action" [Wil12], which is also the major source for the content of this section.

Managing Threads

Most thread-management tasks are performed through the `std::thread` object associated with a particular thread. This includes launching, joining, and detaching threads to run in the background.

A new thread is started by constructing a `std::thread` object and specifying the task to run on that thread. In the simplest case, that task is a `void` function without parameters[1]:

```
void task();
std::thread T(task);
```

In addition to functions, the `std::thread` constructor accepts any other types that satisfy the `Callable` concept, such as functors, lambda expressions, call wrappers like `std::bind`, and so on.

Once a thread has been started, there are two choices. The program can either wait for the thread to complete, or leave it to run on its own in the background. Calling `join()` on an `std::thread` object causes the execution to block at this statement until the thread has finished:

```
#include <thread>
void task();

int main() {
  std::thread T(task);
  T.join() // execution waits here until T is completed.
}
```

If the function `detach()` is called instead of `join()`, the associated thread gets transformed into a background thread:

```
std::thread T(background_task);
T.detach();
assert(!T.joinable());
```

The control and ownership of a detached thread are moved to the C++ Runtime Library, and it becomes impossible to obtain a `std::thread` instance referencing the thread. Thus, threads cannot be joined once they have been detached.

Protecting Shared Data

In addition to threads, C++11 also introduced synchronization mechanisms to protect shared data from concurrent access. The most general data-protection mechanism in C++11 are mutexes, which have already been discussed in Section 1.1.4.

A mutex in C++ can be created by constructing an instance of `std::mutex`. A call to the member function `lock()` locks the mutex, and conversely, `unlock()` unlocks it. However, it is not recommended to call these member functions directly, because it may lead to programming errors. A common mistake is forgetting to call `unlock()` on every execution path, especially on those caused by exceptions.

The Standard Library provides a utility class `std::lock_guard`, which implements the RAII idiom[2] for mutexes. This class aims to simplify management of mutexes. It locks the supplied

[1] The `<thread>` header must be included so that the compiler can see the definition of the `std::thread` class.

[2] Resource Acquisition Is Initialization (RAII): According to this idiom, the holding of a resource should be tied to the object lifetime: resource allocation should be done during object creation by the constructor, and resource release during object destruction.

mutex during construction and releases it during destruction, ensuring that `unlock()` is never forgotten. The following code listing shows how a `std::mutex` and `std::lock_guard` can be used to protect a shared variable from concurrent access:

```
#include <thread>
std::mutex m;
int shared{};

void update_shared(int new_value) {
  std::lock_guard<std::mutex> guard(m); // mutex is locked
  shared = new_value;
} // lock is released when 'guard' goes out of scope
```

C++11 offers additional mechanisms that extend the basic `std::mutex` implementation with means to mitigate and prevent the risk of deadlocks, effectively deal with exceptions, and support for recursion. These extended mechanisms are not covered by the current implementation of the analysis algorithm, and their descriptions is omitted here.

Condition Variables

In addition to the simple protection of shared variables against concurrent accesses, C++11 also offers more advanced synchronization facilities, which allow to coordinate actions between threads.

The first of these synchronization facilities are *condition variables*. They are the C++ variant of monitors, which have discussed in Section 1.1.4. Condition variables are mutexes with an associated condition. If a thread encounters a condition variable, it must *wait* for the condition to be satisfied, before it can proceed processing. When a thread as determined that the condition is satisfied, it can *notify* one or more of the waiting threads and allow them to continue. Listing 1.3 shows an example of thread synchronization using a condition variable in the context of a bank account. Multiple threads add and subtract money to and from the account balance. A condition variable is used to prevent the balance from turning negative. When a thread wants to withdraw money, the condition checks beforehand if the balance is sufficient. If it's not, the thread must wait at the condition variable. When money is deposited, all waiting threads get notified, and proceed to check the condition again.

Note that this code example wraps the mutex in a `std::unique_lock` before it is passed to the conditional variable. `std::unique_lock` is simply a mutex wrapper that allows to temporarily transfer the mutex ownership to the condition variable.

Listing 1.3: Example of thread synchronization using a condition variable. Multiple threads add and subtract money to and from the account balance. A condition variable is used to prevent the balance from turning negative

```
#include <thread>
#include <condition_variable>
#include <iostream>

int balance{};
std::mutex m;
std::condition_variable cond;

void withdraw(int funds) {
  std::unique_lock<std::mutex> guard(m);
  cond.wait(guard, [&]{return balance >= funds;});
  balance -= funds;
  std::cout << "withdrew " << funds << std::endl;
  std::cout << "new balance: " << balance << std::endl;

}

void deposit(int funds) {
  std::unique_lock<std::mutex> guard(m);
  balance += funds;
  std::cout << "deposited " << funds << std::endl;
  std::cout << "new balance: " << balance << std::endl;
  cond.notify_all();
}

int main() {
  balance = 5;
  std::thread t1([]{withdraw(5);});
  std::thread t2([]{withdraw(1);});
  std::thread t3([]{deposit(2);});
  t1.join();
  t2.join();
  t3.join();
  std::cout << "-------------\n";
  std::cout << "final balance " << balance;
}
```

Futures

The second advanced synchronization mechanism is a concept called *futures*. A future is an object representing an event that has not yet occurred. Futures allow threads to wait for specific events, that occur as part of asynchronous computations. They also provide a means to return values from these operations.

Futures are implemented in the Standard Library using the class template `std::future<T>`, which represents a future of type `T`. The value of a future can be retrieved using the member function get(), which synchronizes the program by waiting for the result. Alternatively, futures can be periodically checked it to see if the associated value has become available using the member function `wait_for(duration)`.

`std::futures` are used as the return types for a series of abstractions for asynchronous computations:

- `std::async`. This is the most convenient and straight-forward abstraction to perform an asynchronous computations. `std::async` takes a function and executes it asynchronously, potentially in a separate thread. It returns a `std::future` that will eventually hold the result of the function call. `std::async` gives no guarantee that the supplied function will be run on a different thread.

  The Program in Listing 1.4 exemplifies the use of `std::async` and `std::future`: An asynchronous task is started, and its result is obtained using an `std::future`.

- `std::packaged_task`. This template class is a wrapper that ties a future to a `Callable` target, so that it can be invoked asynchronously. Compared to `std::async`, `std::packaged_task` provides more control over the execution of the asynchronous operation. It is left to the programmer to decide, how and when a `std::packaged_task` should be run.

  An example program, which illustrates how `std::packaged_task` can be used to run an asynchronous operation is illustrated in Listing 1.5. This program shows how an `std::packaged_task` is created, and subsequently executed in a concurrently running thread. After the operation is complete, the result retrieved via an `std::future`.

- `std::promise`. This class is essentially the counterpart to the `std::future`. It provides a means of setting a value, which can later be read using an associated `std::future`. Promise and future form a one-way communication channel. A value can be put on the channel by storing it in the promise, and retrieved by getting it from the future. This information exchange is synchronized, and can happen across threads.

  Listing 1.6 shows an example of a `std::promise`/`std::future` pair. First, a promise and its corresponding are created. Then, the future is handed of to a different thread. Finally, the promise is fulfilled by setting it to the value 42, which is read and printed from the other thread.

This concludes the digression into the C++ concurrency features. This section has introduced the C++ thread management facilities (`std::thread`) and the basic synchronization mechanisms (`std::mutex`). It has also peaked at the more advanced features the C++11 Standard Library has to offer.

Listing 1.4: Program showing the use of `std::async` and `std::future`: An asynchronous task is started, and its result is retrieved using an `std::future`.

```
#include <future> / <iostream> // condensed to preserve space

int compuation(); // definition must be visible

int main() {
  std::future<int> result = std::async(computation);
  std::cout << "compuation result: " << result.get();
}
```

Listing 1.5: Program showing the use of `std::packaged_task`: A `std::packaged_task` is created, and executed in a concurrently running thread. After the operation has completed, t result is obtained using an `std::future`

```
#include <future> / <thread> / <iostream>

int operation(); // definition must be visible

int main() {
  std::packaged_task<int> task(operation);
  std::future<int> task_result = task.get_future();
  std::thread task_thread(std::move(task));
  task_thread.join();
  std::cout << "task result: " << result.get();
}
```

Listing 1.6: Program showing the use of a `std::promise`/`std::future` pair: First, a promise and its corresponding are created. The future is then handed of to a different thread. Finally, the promise is fulfilled by setting it to the value 42, which is read and printed from the other thread.

```
#include <iostream> / <thread> / <future> / <functional>

void async_print(std::future<int>& future) {
  std::cout << "value: " << future.get() << '\n';
}

int main() {
  std::promise<int> promise;
  std::future<int> future = promise.get_future(); // create the future
  std::thread t(async_print, std::ref(future)); // put future into thread
  promise.set_value(42); // fulfill promise, future becomes ready
  t.join();
}
```

1.3. Related Work

It is very it difficult to observe data races, because they often violate data constraints without immediate visible damage. The effects of these violations tend to cause problems and errors much later in the program execution, which makes it hard to trace the observed errors back to their actual root cause.

Data races often occur only during very specific instruction orderings with a low probability of happening, which makes detection using traditional methods of automated regression testing very challenging. Automated tests tend follow more or less exactly the same execution sequence every time they are run. Because of this property, there is a high risk that data races will elude in-house testing and will only be discovered when the software is shipped to the end users [EA03].

In the past, a lot of work has been done to detect data races. The following sections introduce the spectrum of analysis methods that have been applied to this problem, and highlight their properties, advantages, and shortcomings.

1.3.1. Dynamic Analysis

Dynamic program analysis is an approach, where programs are analyze during their execution. The strength of dynamic analysis tools is that by operating at runtime they only visit feasible paths and have accurate views of the values of variables and aliasing relations. Because of this extensive information, dynamic program analyses rarely produce false positive error reports.

Unfortunately, dynamic analysis has a heavy computational price, making it time consuming to run test cases and impossible on programs that have strict timing requirements. In addition, dynamic program analyses can only find errors on executed paths. This means that in practice only a tiny fraction of all feasible paths can be explored and analyzed, which causes error reports from dynamic program analysis tools usually to be incomplete [EA03].

The best known dynamic tool is the Eraser data race detector [SBN+97]. Eraser dynamically tracks the set of locks held during program execution. This information is used to compute the intersection of all locks held when accessing shared state. All shared locations having an empty intersection are flagged as not being consistently protected.

1.3.2. Static Analysis

At the opposite end of the spectrum is static program analysis. While this technique operates with far less precise local information, it still features significant advantages, especially for large code bases. Unlike dynamic approaches, static analysis does not require executing code: it immediately finds errors, even in code paths that are difficult to reach during actual program execution. Because static analysis happens offline, it can be perform analyses that would be impractical, or too time consuming at runtime.

Because static program analyses operate offline, they almost certainly have less accurate information regarding the program state than dynamic analyses. Having to rely in incomplete information often forces static analyses to make assumptions and generalizations, which makes them prone reporting false positives.

The best known static tool is RacerX [EA03], a static tool that uses flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks. According to the authors, RacerX aggressively infers checking information such as which locks protect which operations, which code contexts are multi-threaded, and which shared accesses are dangerous. It tracks a set of code features which it uses to sort errors both from most to least severe.

Other Approaches

In between the spectrum of static and dynamic analyses, there exist a few other approaches that have been applied to data race detection.

Post-mortem techniques [HM94] analyze logging or trace data after the program under analysis has finished execution. This approach is similar in nature to dynamic analysis techniques. Despie the fact that post-mortem techniques can affect performance less than dynamic analyses, they suffer from the same limitations. They can only find errors along executed paths.

Another technique that has been used to find data races is model checking [CGP99], a formal verification technique that can be viewed as a more comprehensive form of dynamic testing. Based on a formal description of a system and a specification, model checking can verify whether the system model meets the given specification. Originally, This technique grew out of the need to explore the massive state spaces in hardware circuits to verify their correctness. Unfortunately, getting a large system into a model checker requires significant effort to both to specify the system and in scaling it down enough to execute in the model checked environment. Applying model checking to large and complex software systems is still infeasible today.

Notable analysis tools that use model checking to find data races in Java programs are the Java PathFinder [VHB+03] and Bandera [CDH+00].

1.4. Background Information

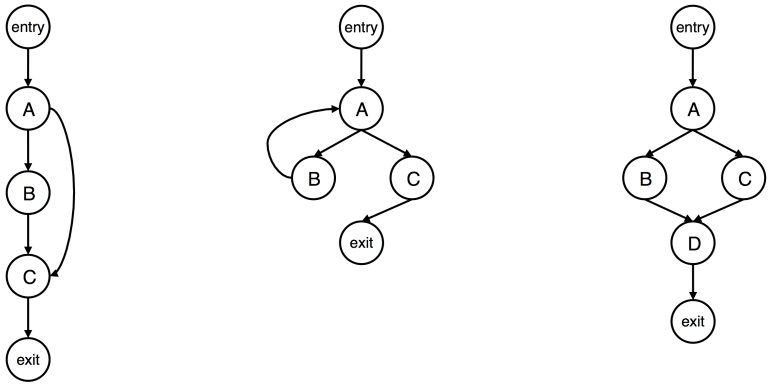
The objective of this section is to provide the reader with an overview of the techniques used in this thesis.

1.4.1. Flow Graphs

A *flow graph* is a representation of all possible paths though a program that the execution may follow. Flow graphs are essential to many compiler optimizations and static program analyses [ALSU06].

Basic Blocks

The nodes in a flow graph represent the *basic blocks* of the program. A basic block is a maximal sequence of consecutive program instructions. There must not be any jumps into or out of the middle of the block. The flow of control may only enter a block through its first instruction and leave after its last instruction.



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

Figure 1.5.: Flow Graph Examples. 1.5a shows an if-else clause: the execution of block B is not guaranteed. In 1.5b, blocks A and B form a while-loop, which will be traversed an indeterminate number of times, ranging from zero to infinity. 1.5c shows an if-then-else clause with two mutually exclusive blocks B and C.

Flow of Control

The flow of control though the graph is expressed using directed edges that connect the basic blocks. An edge from block A to block B states that the last instruction of A may precede the first instruction of B. A is therefore a *predecessor* of B, and B a *successor* of A.

Entry and Exit

In addition to the basic blocks that represent the program code, flow graphs usually contain two virtual nodes called entry and exit, which do not correspond to any instruction sequences. Instead, there is an edge from *entry* to the basic block that will be executed first. Accordingly, there is an edge to *exit* from all blocks that could be executed last.

Flow Graph Examples

Figure 1.5 illustrates three examples of common flow graph patterns. 1.5a shows an if-else clause where the execution of block B is not guaranteed. In 1.5b, blocks A and B form a while-loop, which will be traversed an indeterminate number of times, ranging from zero to infinity. 1.5c shows an if-then-else clause with two mutually exclusive blocks A and B.

In addition, the examples in Figure 1.5 have a common property that is essential to this thesis. They all contain *optional* blocks, which may or may not be executed, depending on the current program state. The fact that these optional blocks exist is something to keep in mind for a better understanding of the later chapters in this thesis.

A more detailed example that shows the interaction between source code and flow graph is illustrated using the program in Listing 1.7. The objective of this program is to read single integers from the standard input and compute their total sum. After a 0 is read, the program prints out the current total and terminates. The flow graph corresponding to the example program is

illustrated in Figure 1.6. It contains five basic blocks, including *entry* and *exit*. The directed edges depict the possible flow of control between the blocks. The program execution may only follow these edges. Blocks $B2$ and $B3$ represent the while-loop, which will be traversed by the program an unknown, potentially infinite, number of times.

Listing 1.7: This program reads single integers from the standard input and adds their total sum. When the program reads a 0, it prints out the sum and terminates

```
#include <iostream>

int main(int argc, char **argv) {
  int i{};
  int sum{};
  while(std::cin >> i) {
    sum+=i;
  }
  std::cout << sum;
}
```

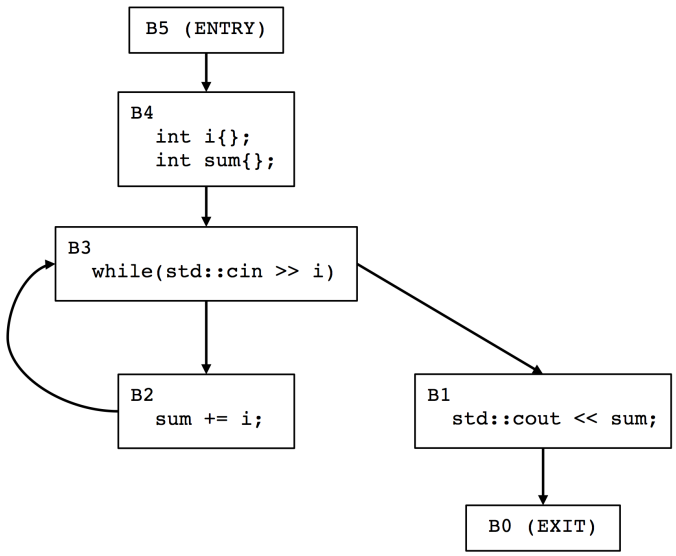


Figure 1.6.: The flow graph corresponding to the program in Listing 1.7. It contains five basic blocks, including *entry* and *exit*. Blocks $B3$ and $B2$ represent the while-loop, which will be traversed number of times, ranging from zero to infinity.

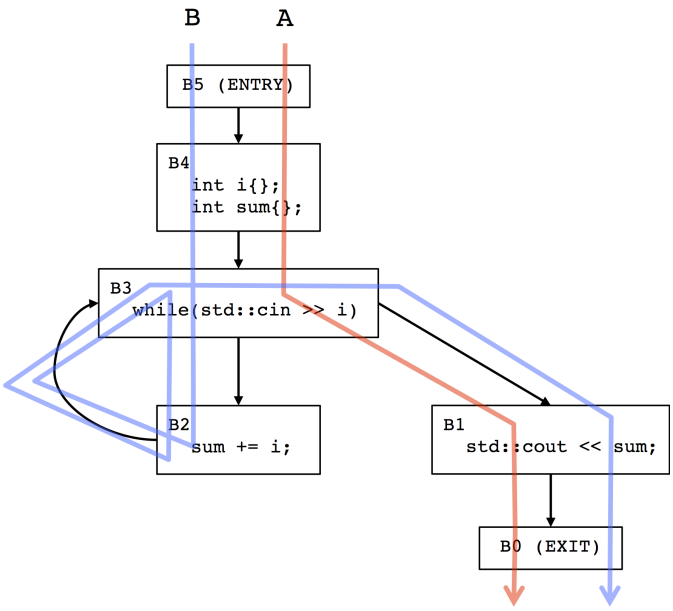


Figure 1.7.: Two possible execution paths through the program in Listing 1.8. Path A does not enter the body of the while-loop (0 is the first program input) and is very short. Path B traverses the loop body twice, resulting in a longer execution path.

1.4.2. Data-Flow Analysis

The race detection algorithm is based on a technique called *data-flow analysis*. Data-flow analysis comprises a set of techniques that derive information about the flow of data along program execution paths [ALSU06]. In order to do this, data-flow analyses view the execution of a program as a sequence of program state transformations. Starting from a default state, each executed program instruction transforms the current program state into a new intermediate state that reflects the effects of said instruction. Each intermediate program state is associated with a so-called *program point*. In addition to the program state, a program point often contains additional context information, such as pointers to its predecessor and successor, the instruction that produced the current program state, and so on. The sequence of program states and program points that results from a single program execution is called an *execution path*.

Execution Paths

Figure 1.7 shows two possible execution paths through the program in Listing 1.8, illustrated on its flow graph. Path A does not enter the body of the while-loop and is very short. Path B traverses the loop body twice, resulting in a longer execution path. The program point sequences corresponding to paths A and B are visualized in Figure 1.8, with emphasis on showing the interaction between executed instructions and program points: each executed instruction produces a new program point.

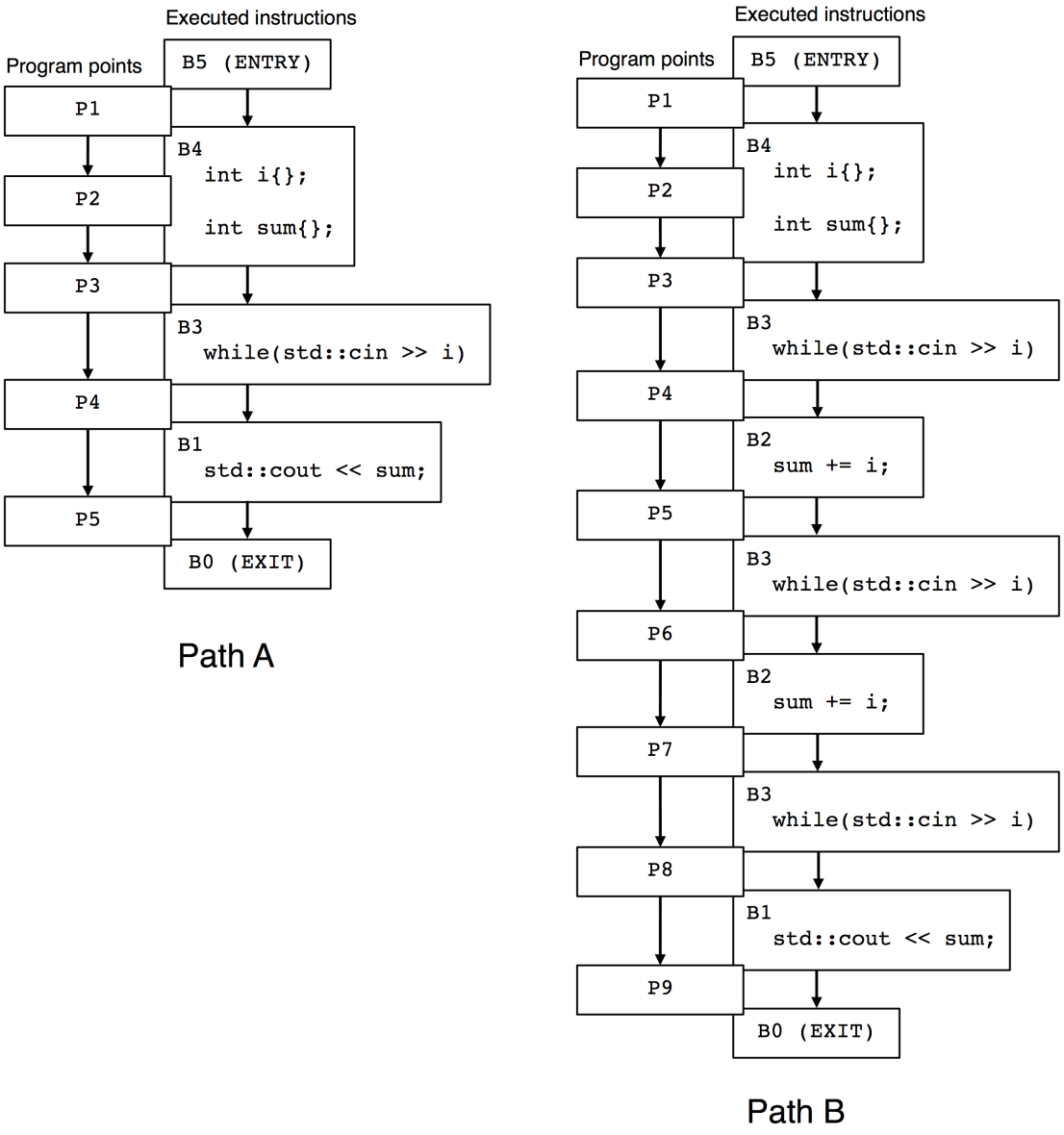


Figure 1.8.: The program point sequences corresponding to execution paths A and B in Figure 1.7. Each executed instruction produces a new program point.

To analyze the behavior of a program, data-flow analyses must consider all possible execution paths, which can be produced by this program. The information relevant for the particular data-flow analysis is then extracted from these execution paths. In general, every program has an infinite number of possible execution paths. Data-flow analyses must often summarize and simplify the program states that can be produced [ALSU06] in order to reduce the input size and make the analysis feasible.

Flow-Sensitivity

A *flow-sensitive* analysis takes into account the instruction order of the analyzed program. Data-flow analyses are inherently flow-sensitive, because they track the program state transformations in the order they occur. Their counterpart, *flow-insensitive* analyses, describe program facts that are true at all program points.

Path-Sensitivity

In addition to instruction order, a *path-sensitive* analysis also considers all possible execution paths (cf. Section 1.4.2). Path-sensitive analyses are more precise, because they rely on a more accurate representation of the program. However, because the number of paths grows exponentially with the number of conditionals in the program, path-sensitive analysis can be very costly.

Conservatism

All data-flow analyses build and operate on an abstract model of the program under analysis. This model is merely an approximation of the reality, which makes analysis mistakes inevitable. A *conservative* analysis makes sure that any potential errors it produces are *safe*. What exactly connotes a *safe error* is very loosely defined and depends on the particular analysis. In general, it means that any approximation made by the analysis must bw safe in a way, such that if the assumption happens to be wrong, the resulting error must be the least harmful alternative.

To illustrate, consider this example: The race detection algorithm used in this thesis cannot determine the exact amount of program threads running concurrently, due to its static nature. As a result, it approximates this property by always assuming the highest possible amount of concurrent threads that is allowed by the program constraints. If this approximation happens to be wrong and in reality there are fewer concurrent threads, some of the data races detected by algorithm will be false positives. However, reporting a false positive is considered less harmful than missing a real data race. Thus, the approximation made by the race detection algorithm, which ultimately lead to the erroneous reports, is considered to be conservative.

1.4.3. Interprocedural Analysis

An *interprocedural analysis* operates across an entire program. This is challenging because most programs are not written as one monolithic instruction sequence; they are structured as a set of procedures that get called repeatedly, with different arguments.

A simple, yet effective interprocedural analysis technique is to inline all procedure calls and subsequently treat the entire program as a single procedure. However, inlining can be difficult. In object-oriented languages with dynamic polymorphism, procedure calls may be referenced indirectly via pointer and resolved at runtime using a dynamic-dispatch mechanism. This makes it hard to statically determine the call target and thus to inline the procedure call. Another difficult obstacle are recursively defined procedures. In general, it is not possible to inline recursive procedures directly [ALSU06].

Call Graphs

A more exact approach to interprocedural analysis is by using a *call graph*. A call graph is a representation of a program that holds information about procedure calls. In particular, a call graph contains a node for each procedure, and one for each *call site*. A call site is a program location, where a procedure is invoked. There is an edge from a call site S to procedure P, if S calls P. The information in the call graph is used construct the execution path.

Context sensitivity

An interprocedural analysis is *context-sensitive* if it takes into account the context in which procedures are called, such as the procedure argument values, or how often the procedure has been called already. A *context-insensitive* analysis treats procedure calls and returns as *goto* instructions. This can be realized by creating a super flow graph that contains additional edges connecting the intraprocedural flow graphs of all procedures in the program [ALSU06].

1.5. Motivation for C++ Tools

C++ is almost notorious for its lack of good tooling support, compared to other languages. In the recent years, many good tooling infrastructures have emerged to build language tools on, like Clang, Eclipse CDT, and others. Despite the presence of these projects, the tooling landscape of C++ has remained relatively scarce. To help the C++ language sustain and even increase its popularity, it is important that good tools are available for the developers to use.

1.6. Thesis Goals

The main goal of this thesis is to build a tool that helps developers write correct multi-threaded programs by providing automatic detection of data races in C++ source code, based on the aforementioned data race detection algorithm developed by Prof. Dr. Luc Bläser. The core challenge of this project consists of two items:

- Adaption and extension of the algorithm to support C++. This also includes a reimplementation of the algorithm, because Prof. Dr. Bläsers prototype was built for, and written in C#. There will likely not be any fundamental changes to the algorithm's structure. However, many technical details will be more complex and challenging when applied to C++.
- Visualization of the algorithm's results. This part of the thesis goes beyond Prof. Dr. Bläsers work on this subject. Since data races are the result of a combination of erroneous parts of a program, it's usually not enough to just point the programmer to the relevant source locations without any further information pertaining to their relationships.

Because of the complexity and experimental nature of the thesis objective, a reliable prediction of the achievable functionality and quality for the project results was difficult. Consequently, the functional requirements were divided into a set of minimum (M) and extended (E) requirements, in order to have an instrument to gage the success of the project.

- Algorithm functionality
  - (M) Support for the most basic thread synchronization features of C++ (std::thread and std::mutex).
  - (M) Support for cross-translation unit analysis.
  - (E) Support for additional C++ concurrency features (std::async, std::future, ...).
- Visualization
  - (M) Visualization of data races in Eclipse CDT/Cevelop using traditional code markers.
  - (E) Advanced visualization and exploration features, whose nature of will be designed when the project reaches this step.

In addition to the requirements, it was specified that the problem of pointer and reference aliasing goes beyond the project scope, and do not need be solved.

Time Budget

This thesis has a budget of roughly 810 hours. However, this is not a fixed constraint, and investing more time is allowed.

Deliverables

The expected deliverables for this project are:

- Three hard copies of this document.
- Three copies of a CD with the following contents:
  - The source code of the created software components
  - This document, as PDF.
  - Executable versions of the created software components, where applicable.
  - A virtual machine containing a preconfigured, ready-to-run installation of the created software components.

2. Data Race Detection Algorithm

This chapter introduces the data race detection algorithm from a conceptual point of view. It outlines its objectives, properties and functionality, but refrains from describing a concrete implementation, and will not go into to much technical detail.

2.1. Algorithm Description

The goal of the algorithm is to find low-level data races in shared-state programs with lock-based synchronization mechanisms. The algorithm analyzes the target program using *static program analysis*; It reasons about the source code of a program without executing it. The employed analysis can be characterized as *flow-sensitive*, *path-sensitive*, *context-insensitive*, and *conservative*. The conservative property influences the analysis as follows:

1. The exact number of active threads a given program point is indeterminate, and must be approximated. This analysis will always assume the highest possible number of concurrently active threads allowed by the program constraints. Because this approximation will sometimes be incorrect, and in reality there will be less active threads than presumed, the analysis is susceptible to reporting false positives. it will find data races between threads that are either not existing, or that will never run concurrently. However, reporting a false positive is considered less harmful than missing a real data race.

2. The number of locks held by a thread at any program point is indeterminate. For instance, a lock acquisition might happen inside an *optional* basic block. For any subsequent basic blocks, it is impossible decide whether the lock was actually taken. To approximate, this analysis will always assume the least protection of critical sections, which implies the highest possible amount of concurrent activities inside critical sections. This again, means that no potential data races inside critical sections will be missed, but that some false positives will be reported.

How the approximation of these indeterminate properties works is covered in the appropriate sections throughout the chapter.

2.2. Limitations

The algorithm is focused purely on the detection of low-level data races, such as the ones shown in the example. It does not detect higher-level races that involve accessing multiple independent shared resources. These errors are much more difficult to locate, because they can exist in otherwise correctly synchronized concurrent code, and their detection requires knowledge about the program's intended semantics (cf. Section 1.1.5, General Races).

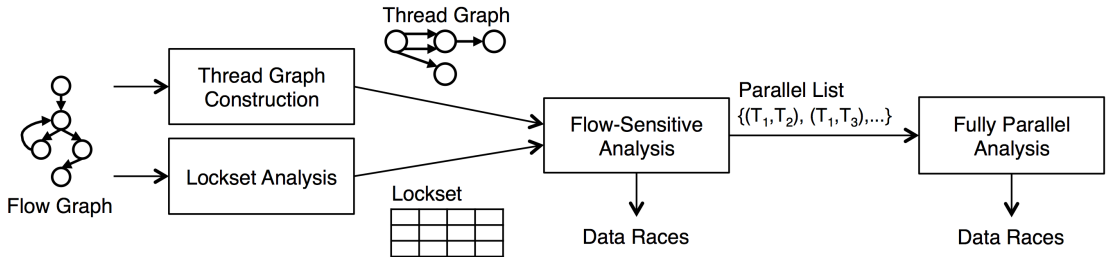


Figure 2.1.: Overview of the race detection algorithm. The arrows represent the data flow between the individual tasks.

2.3. Functionality

This section provides an overview of the algorithm functionality. The subsequent sections will explain each step of the analysis in detail, including any newly introduced terminology. The data race detection algorithm consists of the following four tasks:

1. A so-called *thread graph*, which contains potential start- and join-relationships between *thread entries*, is constructed from the program's flow graph.
2. A Lockset analysis records variable accesses and locks held.
3. A flow-sensitive analysis and locates potential data races between temporally related threads, based on the thread graph and the lockset. This step also records fully concurrent threads that have no temporal relationship.
4. A so-called *fully concurrent* analysis locates potential data races between threads without temporal ordering, based on the *fully concurrent thread list* supplied by previous task.

The first two steps constitute the preparation phase, where the data required by the actual analysis is extracted from the source code and organized in the appropriate data structures. An overview of the race detection algorithm, including the data flow between the individual tasks, is illustrated in Figure 2.1.

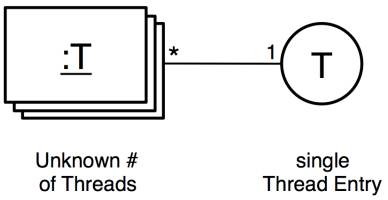
2.3.1. Thread Graph Construction

The data preparation begins with the construction of the *thread graph*. A thread graph is a regular graph, commonly defined in graph theory as an ordered pair $G = (V, E)$, where V is a set of nodes (vertices), and E a set of edges that connect the nodes. The purpose of a thread graph is to represent the temporal relationships between a program's threads. This information is crucial, as it helps later analysis steps to determine which threads could potentially run concurrently and cause data races. The following subsections will elaborate on the thread graph data structure and clarify the meaning of the nodes and edges.

Thread Entries

The thread graph nodes represent *thread entries*. The term thread entry is used by Dr. Bläser to describe a program section that runs concurrently, but unlike a traditional, uniquely identifiable thread, each thread entry may have multiple instances of itself running at the same time. The example below illustrates the relation between threads and thread entries (triple-points signify elided code):

```
...
while (notStopped()) {
  std::thread T(...);
  T.detach();
}
...
```



From a static point of view, it is impossible to determine how many times T will be started, or if the start-statement is reached at all. The exact number of threads started is indeterminate and the potential amount ranges from 0 to n. Because the analysis is conservative, it always assumes the situation that causes the highest level of concurrency: the start statement is not only reached once, but multiple times. The notion of thread entries is used to account for this property in the algorithm. The unknown number of threads can be expressed using a single thread entry.

Potential Start-Edges

The thread graph edges represent the *potential start*- and *guaranteed join*-relationships between the thread entries in the graph. A potential start from thread entry A to thread entry B (hence expressed using the symbol $A \xrightarrow{start} B$), and an according directed edge in the thread graph, exists iff there is a start-statement that starts B, and this statement is reachable by A. Figure 2.3 illustrates a potential start $A \xrightarrow{start} B$ by showing the code that expresses the relationship, as well as the resulting thread graph.

The analysis does not evaluate branching conditions: If it encounters a start-statement inside an optional branch, the analysis simply assumes the statement will be reached during execution and the thread will be started at least once. This is done in accordance with the conservative nature of the analysis, that forces it to always assume the maximum number of concurrent threads.

As previously mentioned, this assumption is a source for potential false positives, because the analysis is prone to recording relationships that will never actually happen during execution. Figure 2.4 shows two examples of thread entry relationships that can never occur at runtime. In Figure 2.4a, the relation $A \xrightarrow{start} B$ is recorded although the corresponding start-statement is unreachable because of the branch condition value. Accordingly, $A \xrightarrow{start} B$ and $A \xrightarrow{start} C$ present in Figure 2.4b will be recorded, although they are mutually exclusive.

Thread entries, unlike traditional threads, may have a potential-start relationship to itself. The code sample in Figure 2.5 shows an example of a situation where a thread entry T continuously starts itself, as long a condition is satisfied.

```
...
std::thread A([]{
  std::thread B(...);
  ...
});
...
```

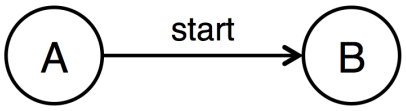


Figure 2.3.: A simple potential start-relationship from thread entry A to B. The code listing on the left shows the statements expressing the relationship in the program. The illustration on the right shows the resulting thread graph.

```
std::thread A([]{
  while (false) {
    std::thread B(...);
    ...
  });
}
```

(a) Unreachable start.

```
std::thread A([]{
  if (...) {
    std::thread B(...);
    ...
  } else {
    std::thread C(...);
    ...
});
```

(b) Mutually exclusive starts.

Figure 2.4.: False positives. The analysis does not reason about conditional branches and assumes that all existing branches can be reached at runtime. The relation $A \xrightarrow{start} B$ in Figure 2.4a will be recorded although it can never occur an runtime. Accordingly, in Figure 2.4b, both relations $A \xrightarrow{start} B$ and $A \xrightarrow{start} C$ will be recorded although they are mutually exclusive.

```
void t() {
  if (...) {
    std::thread T(t);
  }
  ...
}

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

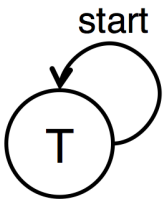


Figure 2.5.: Recursive potential-start relationship. The code listing on the left shows an situation where a thread entry T continuously starts itself, as long a condition is satisfied. The illustration on the right shows the resulting thread graph.

```
std::thread A([]{
  std::thread B(..);
  ...
  B.join();
});
```

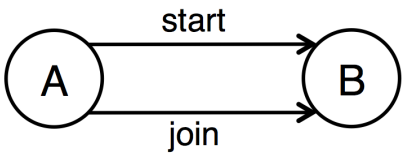


Figure 2.6.: Guaranteed join-relationship from thread entry A to B. The illustration on the right shows the resulting thread graph.

Guaranteed Join-Edges

A guaranteed join from thread entry A to thread entry B (hence expressed using the symbol $A \xrightarrow{join} B$), and an according directed edge in the thread graph, exists iff there is a join-statement that joins B, and this statement is reached on all possible execution paths of A. This condition is due to the conservative nature of the algorithm. The thread graph only records join relationships that are guaranteed to happen, in order to keep the maximum possible amount of concurrent threads running.

Figure 2.6 shows an example of the relationship $A \xrightarrow{join} B$ in a program and the resulting thread graph. The join statement is reached unconditionally.

The code listing below shows an example of a join that does not satisfy the requirements to be recorded in the thread graph. Depending on a branching condition, T is either joined or detached from the *MAIN* thread. Note that an std::thread must be joined before destruction, unless it has been detached beforehand using the designated member function $detach()$.

```
int main() {
  std::thread T(...);
  if (...) {
    T.join();
  } else {
    T.detach();
  }
  ... // the analysis assumes that T is still running here
}
```

Because the join happens inside an optional basic block that is not reached by all execution paths, it is not considered a guaranteed join. The analysis assumes that T has not been joined, because it wants to keep as many threads running as possible. Accordingly, no *MAIN* $\xrightarrow{join} T$ is recorded in the thread graph.

Constructing the Graph

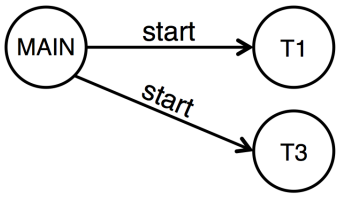
The construction of the thread graph is an iterative process. Beginning with the *MAIN* thread, the graph is successively assembled by first analyzing the current thread T, followed by any

found child threads C that are started by T. the analysis performs the following operations for each thread:

1. In order to gather all outgoing start edges, collect all unique start statements from the instructions reachable by current thread. For each start statement, add a corresponding start edge to the graph.

2. For each execution path P_1 to P_n of the current thread, collect the set of join-statements $J_i \subset P_i$. Add a join edge to the thread graph for each join-statement $j \in J_1 \cap J_2 \cap ... \cap J_n$. The intersection of all join sets contains exactly the guaranteed join-statements of the current thread.

The construction process is exemplified using the example program in Listing 2.1. Finding the outgoing potential start relationships of *MAIN* is simply a matter of collecting the unique start statements from the set of all reachable instructions. The result is two start relationships *MAIN* $\xrightarrow{start}$ $T1$ and *MAIN* $\xrightarrow{start}$ $T3$. Edges and nodes representing these relationships are added to the previously empty thread graph:



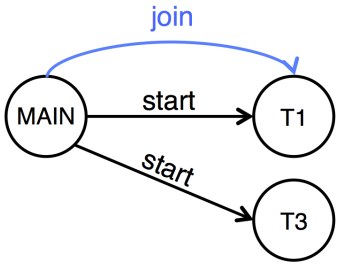
In order to find the guaranteed joins, the analysis must consider the execution paths of *MAIN*. In this example, the *MAIN* thread has two execution paths P_1 and P_2, because there is an if-then-else branch in the program:

$$P_1 = \{\text{thread t1(...)}, \text{thread t3(...)}, \text{t1.join()}, \text{t3.detach()}, \text{cout « global}\}$$
$$P_2 = \{\text{thread t1(...)}, \text{thread t3(...)}, \text{t1.join()}, \text{t3.join()}, \text{cout « global}\}$$

The guaranteed join relationships are obtained by collecting the sets of join-statements J_1 and J_2 from both execution paths, and computing their intersection:

$$J_1 = \{\text{t1.join()}\}$$
$$J_2 = \{\text{t1.join()}, \text{t3.join()}\}$$
$$J_1 \cap J_2 = \{\text{t1.join()}\}$$

Doing so yields one guaranteed join *MAIN* $\xrightarrow{join}$ $T1$. An edge representing this relationship is added to the thread graph:



This concludes the analysis of the *MAIN* thread. The illustrated process is repeated for all discovered threads until no more unanalyzed threads are found. The complete thread graph corresponding to the example program is illustrated in Figure 2.7.

Listing 2.1: Example program to illustrate the thread graph construction. Four thread entries in total (*MAIN*, $T1$, $T2$, and $T3$) are active and compete for the access to the shared variable $global$. The following relationships between the thread entries exist: *MAIN* $\xrightarrow{start}$ $T1$, *MAIN* $\xrightarrow{start}$ $T3$, *MAIN* $\xrightarrow{join}$ $T1$ and $T1$ $\xrightarrow{start}$ $T2$. The corresponding thread graph is illustrated in Figure 2.7.

```
#include <thread>
#include <iostream>
using namespace std;

int global = 0;
mutex m;

int main() {
  thread t1([] {
    lock_guard<mutex> guard(m);
    global = 1;
    thread t2([] { global = 2; } );
    t2.detach();
  });
  thread t3([] {
    lock_guard<mutex> guard(m);
    cout << global;
  });
  t1.join();
  if (...) { t3.detach(); }
  else { t3.join(); }
  cout << global;
}
```

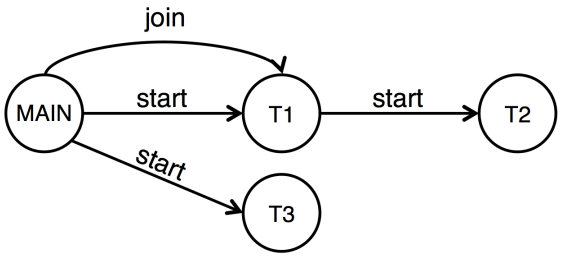
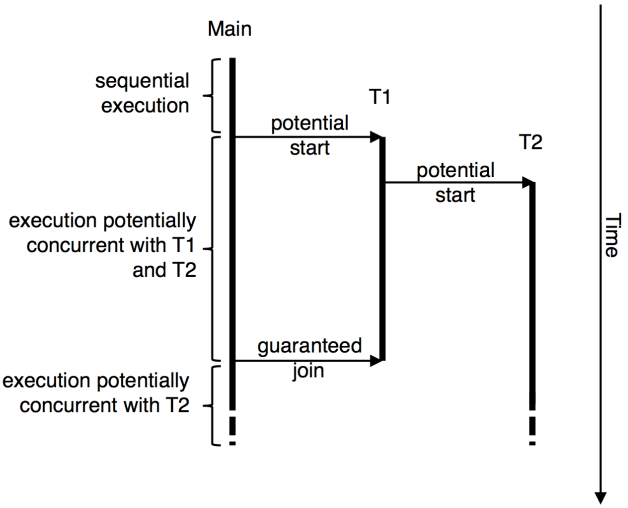


Figure 2.7.: The finished thread graph corresponding to the program in Listing 2.1: The nodes represent thread entries and the edges depict existing potential start- and guaranteed join-relationships.

Conditional Concurrency

Thread entries connected by start- or join-relationships are considered to be *conditionally concurrent*. The defining property of conditionally concurrent thread entries is the fact that it is possible to approximate their relative time of execution.



Consider the above illustration, which shows the approximated relative time of execution of three threads *main*, $T1$, $T2$, and the conditional concurrency with respect to *main*. All statements preceding *MAIN* $\xrightarrow{start}$ $T1$ are guaranteed to not be executed concurrently with any $T1$ and $T2$. Accordingly, all statements after *MAIN* $\xrightarrow{join}$ $T1$ are guaranteed not to be concurrent with respect to $T1$, but might be concurrent with $T2$. The statements between the potential start and the guaranteed join might be concurrent with respect to both $T1$ and $T2$.

Thread entries that are not connected directly or indirectly by neither start- or join-relationships are considered to be *fully concurrent*. Nothing is known regarding the relative timing of fully concurrent threads. Thus, The only safe assumption is that they run concurrent relative to each other during their entire execution.

2.3.2. Lockset Analysis

The term *lockset* is frequently used to describe a data structure that stores information concerning locks in combination with some additional data regarding the program under analysis. The process of constructing this data structure is usually called *lockset analysis*. However, the actual contents of a lockset, as well as its construction and usage is not uniform and may show significant differences depending on the working context. Examples of works that use the term lockset in dissimilar ways are [EA03] and [SBN+97].

In this thesis, the term lockset designates a data structure that stores all memory accesses, associated with the thread entries that performed them and the locks held by the thread entry at the time of access. This information is used later on, in combination with the thread graph, to find conflicting memory accesses between concurrently running thread entries.

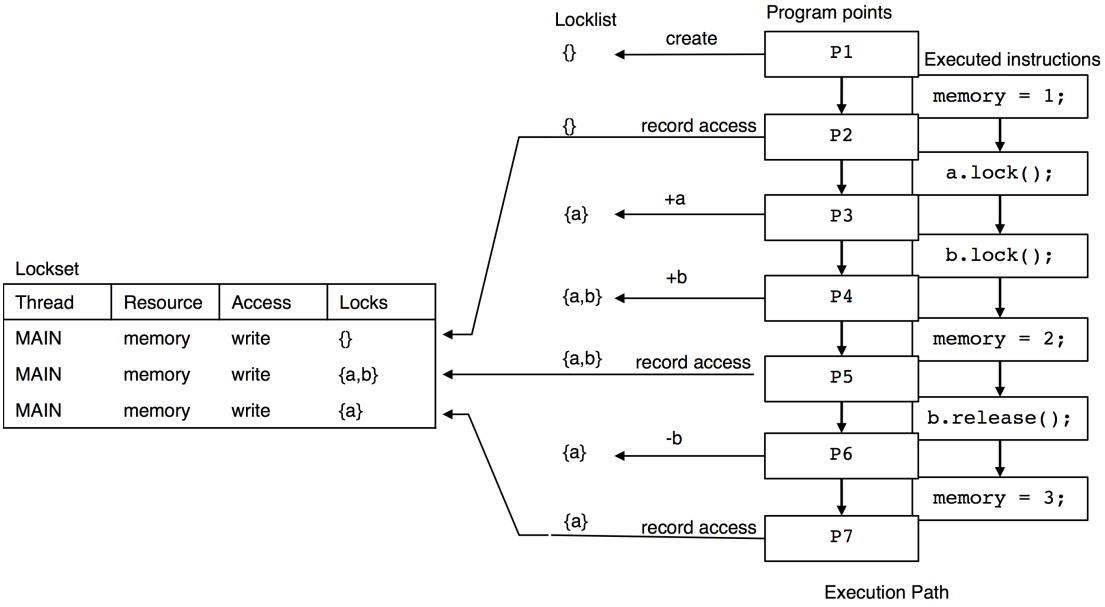


Figure 2.8.: Lockset analysis visualization. The analysis traverses all execution paths of all thread entries in the program and carries along a *locklist*, which contains the currently held locks.

Guaranteed Locks

The lockset analysis is flow- and path-sensitive. It traverses all execution paths and associates all locks with a memory access, which the accessing thread entry is holding at the time of access. A thread entry T holds lock L during memory access A, iff T has taken L prior to the memory access, and has not yet released it. However, the same memory access may be reached on multiple execution paths P_i, which may have acquired different locks. Recording multiple access instances A_i with different locks L_i for each execution path is unnecessary. Instead, the analysis will compute and store the minimal, or *guaranteed*, set of locks L held for memory access A. The reason for this behavior is the conservative nature of the algorithm.

Recording the minimum amount of locks held results in having the maximum number of threads concurrently active inside a critical section, and thus leads to the detection of the maximum number of potential data races.

The minimal set of locks held during memory access A is equal to the intersection of the locks held at each access instance $L_1 \cap L_2 \cap ... \cap L_n$.

Performing the Analysis

The lockset analysis traverses all execution paths of all thread entries in the program, and carries along a *locklist*. This locklist contains the currently held locks. The analysis performs one of four possible actions at every encountered program point:

1. If the instruction preceding the current program point acquired a lock, add the lock to the locklist.
2. If the instruction preceding the current program point released a lock, remove the lock from the locklist.
3. If the instruction preceding the current program point was a memory access, execute one of two possible actions:
   a) If the lockset does not contain previous instance of this access, store the access in the lockset, together with the thread entry and the current locklist.
   b) Else, compute the intersection of the locks associated with the existing access and the current locklist, and update the lockset accordingly.
4. Otherwise, continue to the next program point.

The lockset analysis is exemplified on the program in Listing 2.9. This program contains two execution paths P_1 and P_2, which are analyzed successively. Both paths visit the memory access on Line 15 while holding different sets of locks L_1 and L_2:

$$L_1 = \{a\}$$
$$L_2 = \{a, c\}$$

According to the previously defined behavior, the analysis computes the set of guaranteed locks L and stores it in the lockset:

$$L = L_1 \cap L_2 = \{a\}$$

The traversal of one execution path is illustrated in Figure 2.8, aiming to show the interactions of program points, locklist and lockset.

```
1  int memory{};
2  std::mutex a, b, c;
3
4  int main() {
5    memory = 1;
6    a.lock();
7    if (...) {
8      b.lock();
9      memory = 2;
10     b.release();
11   } else {
12     c.lock();
13     std::cout << memory;
14   }
15   memory = 3;
16 }
```

(a) Program with two possible execution paths.

Thread	Resource	Access	Locks	Location
MAIN	memory	write	∅	Line 5
MAIN	memory	write	a, b	Line 9
MAIN	memory	read	a, c	Line 13
MAIN	memory	write	a	Line 15

(b) Lockset corresponding to the program in Listing a.

Figure 2.9.: Example program to illustrate the lockset construction.

2.3.3. Flow-Sensitive Data Race Analysis

The preparation of the data structures is now complete. The two remaining tasks of the analysis are concerned with the actual data race detection. The first task, the *flow-sensitive data race analysis*, has two objectives:

1. Locate all potential data races between conditionally concurrent thread entries, based on the information stored in the thread graph and lockset.

2. Locate all *unconditionally concurrent* thread entries and store them in a concurrent thread list, which will constitute the basis for the subsequent analysis task, the *fully concurrent data race analysis*.

The following sections describes and illustrates in detail how theses objectives are achieved by the analysis.

Concurrent Thread List

An important data structure needed to achieve objectives of this task is the so-called *concurrent thread list*. In order to locate data races and detect unconditionally concurrent threads, the analysis traverses all execution paths of each thread in the program. While doing so, it carries along a list holding all concurrently active thread entries. The reason for maintaining this concurrent thread list is that the analysis must know at each program point, which thread entries

are concurrently active. This information is crucial both in finding data races, and detecting unconditionally concurrent threads. The analysis performs the following actions to keep the concurrent thread list updated:

1. When the analysis encounters a thread start statement s, it:
   a) Adds the newly started thread entry to the concurrent thread list.
   b) Traverses the thread-graph along the $\xrightarrow{start}$ edges, starting from of the node corresponding to s, in order find any entries that are conditionally started. The analysis then adds all conditionally started thread entries to the concurrent thread list, because they are potentially running concurrent to the main thread beyond this program point, and must be considered when looking for data races.
2. When the analysis encounters a thread join statement j, it:
   a) Removes the joined thread entry from the concurrent thread list.
   b) Traverses the thread-graph along the $\xrightarrow{join}$ edges, starting from of the node corresponding to j, in order find any threads entries that are conditionally joined. The analysis then removes all conditionally joined thread entries from the concurrent thread list, because after this program point, they cannot be alive anymore.

Figure 2.10 illustrates the interaction of start- and join-statements, the thread graph, and the concurrent thread list. It shows how the content of concurrent thread list is updated during the analysis of the program in Listing 2.1. Whenever a start or join is encountered, the thread graph is queried, and the results (shaded in blue) are added to, or removed from the concurrent thread list.

Data Race Detection

After the concurrent thread list has been computed for all program points, the data race detection is performed. Having knowledge of the potential concurrency on each program point greatly simplifies this process. The analysis may analyze the program points out of order, because the flow-sensitive information has already been precomputed. In order to find the potential data races between conditionally concurrent threads, the analysis examines all program points that pertain to a memory access. A data race exists if the following conditions are satisfied:

- At least one other concurrently active thread accesses the memory location.
- At least one memory access performs a write operation.
- There is no *common lock* held between *all* accessing threads.

The common locks for a particular memory access are defined as the intersection over the individual locksets of all concurrent accesses. Concurrent accesses on a particular memory access can be obtained by looking up all accesses on the same memory location, which are associated with any threads currently residing in the concurrent thread list.

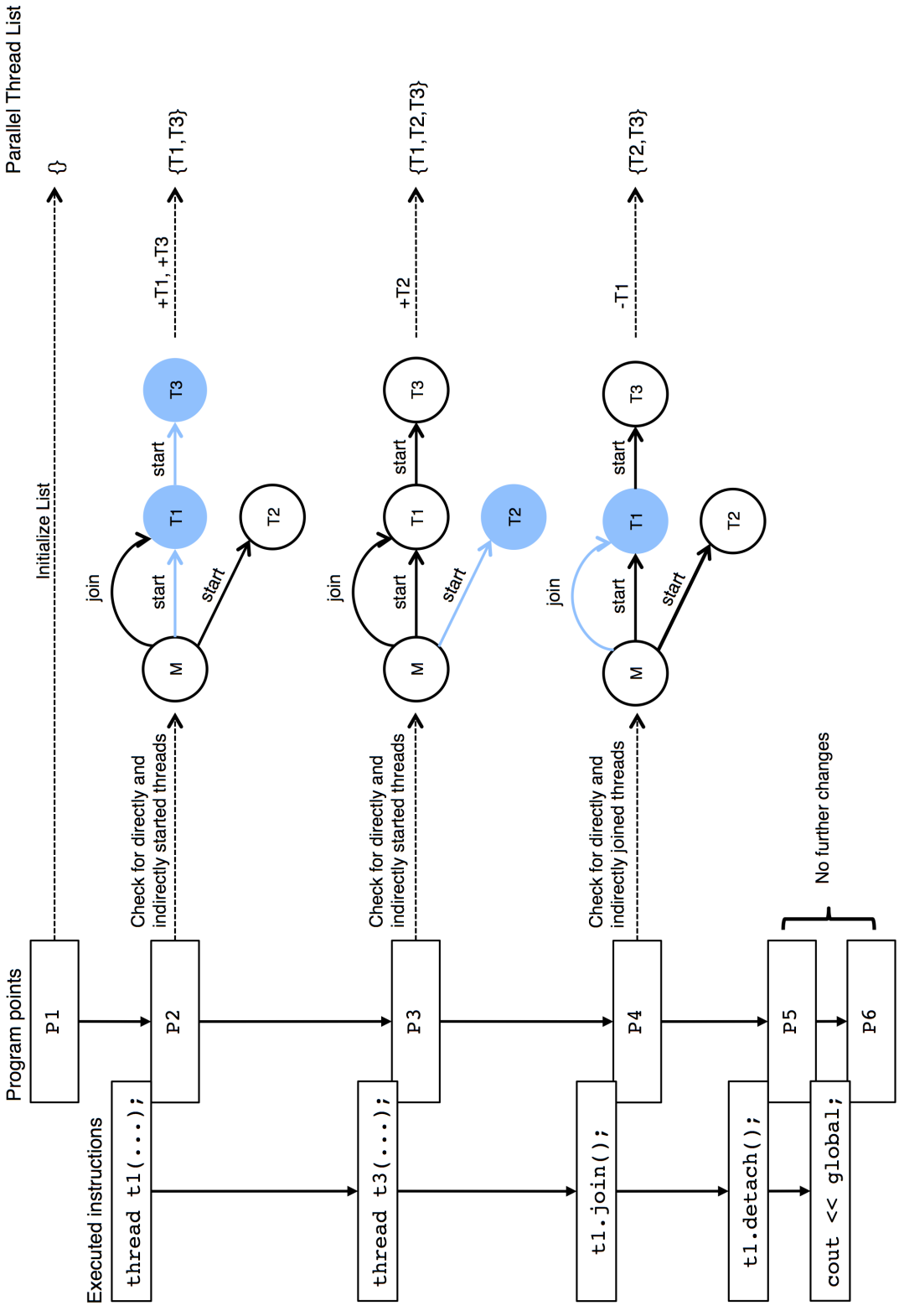
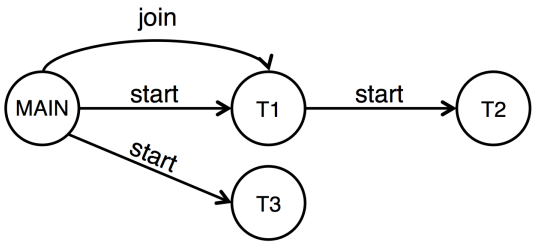


Figure 2.10.: The interaction of start- and join-statements, the thread graph, and the concurrent thread list. The illustration shows how the content of concurrent thread list changes during the analysis of one execution path of the program in Listing 2.1. Whenever a start or join is encountered, the thread graph is queried, and the results (shaded in blue) are added to, or removed from the concurrent thread list

When examining a particular program point, the analysis is unable to make assumptions about the execution timing of the concurrently active threads. This means it must always consider *all* memory accesses on the same location made by the concurrent thread entries. It may ignore memory accesses made by the thread entry, which the examined program point belongs to, if the program point belongs to thread entry *MAIN*.

```
1  #include <thread>
2  #include <iostream>
3  using namespace std;
4  int memory = 0;
5  mutex a;
6
7  int main() {
8    thread t1([]() {
9      lock_guard<mutex> guard(a);
10     memory = 1;
11     thread t2([] {
12       memory = 2;
13       lock_guard<mutex> guard(a);
14       cout « memory
15     });
16     t2.detach();
17   });
18   thread t3([] {
19     memory = 3;
20   })
21   t3.join();
22   lock_guard<mutex> guard(a);
23   cout « memory;
24   t1.join();
25 }
```

(a) Program.



(b) Thread graph.

Thread	Resource	Access	Locks	Location
MAIN	memory	read	a	Line 23
T1	memory	write	a	Line 10
T2	memory	write	∅	Line 12
T2	memory	read	a	Line 14
T3	memory	write	∅	Line 19

(c) Lockset.

Figure 2.11.: Example program to illustrate the flow-sensitive data race analysis. It contains four thread entries (*MAIN*, $T1$, and $T2$), which are concurrently accessing the shared variable $memory$. The program contains two data races due to incorrect synchronization. The statements responsible for the data race are shaded in blue and pink, respectively. The blue data race can be detected with the flow sensitive analysis, the pink one cannot.

Performing the Data Race Analysis

The prose specification of the data race analysis can be translated into the following operations, which are performed for each program point p that is associated with a memory access:

1. For each thread T_1 to T_n in the concurrent thread list of p, obtain the set of accesses to the same memory location $A_i \subset T_i$ from the lockset, including the associated locks.

2. Compute the pairwise intersection of the locksets of all concurrent accesses $a_{i,j} \in A_i$, as well as the lockset of p. If the intersection is empty, and at least one access performs a write operation, report a data race.

To illustrate, consider the example program illustrated in Figure 2.11, which contains four thread entries (*MAIN*, $T1$, and $T2$) that are concurrently accessing the shared variable $memory$. The program contains two data races due to incorrect synchronization. The statements responsible for the data race are shaded in blue and pink, respectively. The blue data race can be detected with the flow sensitive analysis, because it is caused by conditionally concurrent thread entries *MAIN* and $T2$. However, the detection of the pink data race between the thread entries $T2$ and $T3$, which have no conditional relationship of any kind, must be postponed until the final analysis step.

As previously mentioned, the data race analysis is only concerned with program points that directly follow a memory access. The analysis starts with thread entry *MAIN*, which performs exactly one memory access on the location $memory$[1]:

$$a_{MAIN} = (memory, read, a, 23)$$

In order to find potential data races on this memory access, the analysis first obtains all accesses to the same memory location associated with threads, which are concurrently active at the moment. The concurrent threads are $T1$ and $T2$, because thread $T3$ has already been joined at an earlier time and is not contained in the concurrent thread list when a_{MAIN} is executed.

$$A_{T1} = \{(memory, write, a, 10)\}$$
$$A_{T2} = \{(memory, read, \varnothing, 12), (memory, write, a, 14)\}$$

The analysis now computes the pairwise intersections between a_{MAIN} and all concurrent accesses:

$$I = a_{T1,1} \cap a_{MAIN} = \{a\} \cap \{a\} = \{a\}$$
$$J = a_{T2,1} \cap a_{MAIN} = \varnothing \cap \{a\} = \varnothing$$
$$K = a_{T2,2} \cap a_{MAIN} = \{a\} \cap \{a\} = \{a\}$$

The intersection J is empty, therefore the two accesses are not synchronized. J also satisfies the remaining condition that constitutes a data race: at least one access performs a write operation. The analysis has thus detected a data race on location $memory$, between the accesses $a_{T2,1}$ on Line 12 and a_{MAIN} on Line 23. This concludes the example. The illustrated analysis will now be applied to all execution paths of the remaining unanalyzed thread entries.

[1] All memory accesses in this example will be illustrated as 4-tuples of (*memory-loc*, *type*, *locks*, *access-loc*)

Fully Concurrent Thread Detection

The second objective of the flow-sensitive analysis is the detection of fully concurrent threads. As already mentioned in Section 2.3.1, two threads are considered to be fully concurrent if the analysis cannot determine any constraints pertaining to their relative time of execution. In other words, if there is no visible start-relationship between two threads in the thread graph, they are assumed to run fully concurrent with respect to each other. Example 1 on Page 42 elaborates more on fully concurrent threads.

Similar to the flow-sensitive data race analysis, the detection of fully concurrent threads is based on the thread graph, the flow graph, and the data stored in the precomputed concurrent thread list. In order to find fully concurrent threads, the analysis examines the program points of all thread entries, which are associated with a start statement. At each program point, the analysis looks for any thread entries that run fully concurrent with the newly started thread. Any found thread entries are added pairwise with the newly started thread to the *fully concurrent thread list*. Ather the fully concurrent thread detection is finished, the fully concurrent thread list is forwarded to the last task of the analysis, the fully concurrent data race detection.

Performing the Fully Concurrent Thread Detection

The fully concurrent thread detection requires that the concurrent thread list has been precomputed for the entire program. In order to detect fully concurrent threads, the detection examines each program point associated with associated with a thread-start statement. Similar to the flow-sensitive data race detection, the program points may be analyzed out of order. The detection performs the following operations on each thread start statement s:

1. Collect the set of all conditionally started thread entries C by traversing the thread graph along the $\xrightarrow{start}$ edges, starting from the node corresponding to s.[2]
2. Compute the relative complement of C with respect to the set of thread entries Q in the concurrent thread list.
3. For each thread entry $t \in (C \setminus Q)$, add a 2-tuple (s, t) to the fully concurrent thread list.

The fully concurrent thread detection is exemplified using the program in Listing 2.11a. This program contains three program points that are associated with a thread start command[3]:

$$P_1 = (\text{start t1(...)}, \mathit{MAIN}, \varnothing)$$
$$P_2 = (\text{start t3(...)}, \mathit{MAIN}, \{T1, T2\})$$
$$P_3 = (\text{start t2(...)}, T1, \varnothing)$$

[2] The attentive reader may have noticed that the computation of conditionally started thread entries happens twice; a first time during the computation of the concurrent thread list, and again during the fully concurrent thread detection. This redundancy only exists to keep the description of the individual analysis tasks self-contained. It is not present in the actual algorithm implementation.

[3] In this example, program points are represented as 3-tuples of (*statement*, *thread-entry*, *concurrent-thread-list*)

The detection begins with the analysis of program point P_1.. First, the set of all conditionally started thread entries C_1 is collected from the thread graph.

$$C_1 = \{T1, T2\}$$

Second, the relative complement of C_1 with respect to the concurrent thread list Q_1 is computed. Because $Q_1 = \varnothing$, there are no fully concurrent threads, an nothing is added to the fully concurrent thread list.

$$C_1 \setminus Q_1 = \{T1, T2\} \setminus \varnothing = \varnothing$$

The analysis continues by performing the same operations on program point P_2.

$$\begin{aligned} C_2 &= \{T3\} \\ Q_2 &= \{T1, T2\} \\ C_2 \setminus Q_2 &= \{T3\} \setminus \{T1, T2\} = \{T1, T2\} \end{aligned}$$

$C_2 \setminus Q_2$ contains two thread entries $T1$ and $T2$. Because they were not started conditionally with $T3$, they are independent, and fully concurrent with respect of $T3$. Thus, two tuples ($T1$, $T3$) and ($T2$, $T3$) are added to the fully concurrent thread list. The same operations are now performed again for P_3, but they are omitted. The final list has the following contents:

$$\{(T1, T3), (T2, T3)\}$$

This concludes the illustration of the fully concurrent thread detection.

2.3.4. Fully Concurrent Data Race Analysis

The final task of the analysis is the so-called *fully concurrent data race analysis*. Its primary and only objective is the detection of data races between fully concurrent threads. This analysis is flow-insensitive; it is based entirely on the fully concurrent thread list, which has been assembled by the previous task, and information from the lockset.

The principle of the analysis is very simple. For each pair of thread entries (T_1, T_2) in the fully concurrent thread, list the analysis performs the following operations:

1. Using the lockset, compute the set of shared memory locations M_{shared}, which is accessed by both T_1 and T_2.

2. Obtain the set of accesses A_1 made by T_1 on M_{shared}, and the set A_2 made by T_2 on M_{shared}, from the lockset.

3. for each access $a \in A_1$,compute the pairwise lockset-intersections with all accesses A_2 to find potential data races between T_1 and T_2. The conditions that constitute a data race are the same as during the flow-sensitive data race analysis, as described in Section 2.3.3, Data Race Detection.

The fully concurrent data race analysis is exemplified on the program in Listing 2.11a. This program has already been used to illustrate the flow-sensitive data race analysis, which was able to detect one potential data race. However, a second data race remained undiscovered, because it is caused by two fully concurrent threads, of whom the flow-sensitive analysis has no knowledge. The fully concurrent data race analysis is able to detect this data race.

The fully concurrent thread list L, corresponding to the example program is shown below. Its computation was illustrated in the previous Section.

$$L = \{(T1, T3), (T2, T3)\}$$

The analysis starts with the first pair L_1. First, it computes the set of shared memory locations M_{shared}, and the set of Accesses A_{T1} and A_{T2} on M_{shared}.

$$\begin{aligned} M_{\text{shared}} &= \{\textit{memory}\} \\ A_{T1} &= \{(\textit{memory}, \textit{write}, a, 10)\} \\ A_{T3} &= \{(\textit{memory}, \textit{write}, \varnothing, 19)\} \end{aligned}$$

The data race detection is executed analogous to its flow-sensitive counterpart. The analysis computes the pairwise intersection between all concurrent accesses:

$$I = a_{T1,1} \cap a_{T3,1} = \varnothing \cap \{a\} = \varnothing$$

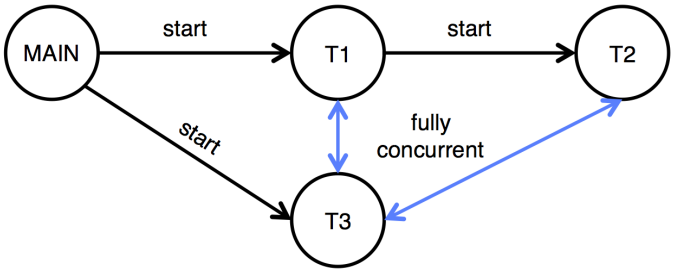
Intersection I is empty, therefore $a_{T1,1}$ and $a_{T3,1}$ are not synchronized, and potentially cause a data race. The second condition for a data race is also satisfied; both accesses perform write operations. Thus, the analysis has found a potential data race on location *memory*, between the accesses $a_{T1,1}$ on Line 10, and $a_{T3,1}$ on Line 19. The same operations are now applied to the second pair L_2, but they are omitted. This concludes the example.

2.4. Summary

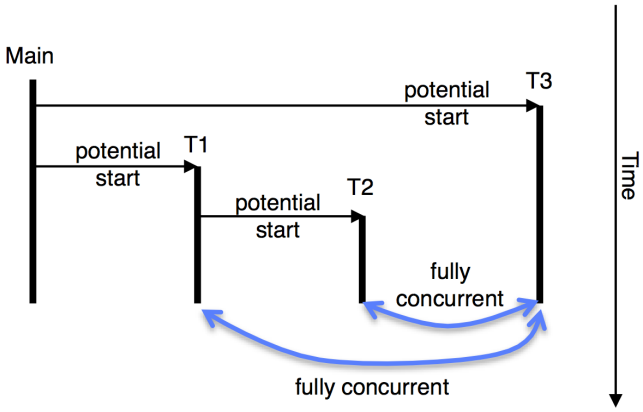
The presented algorithm is the foundation for the software that was developed during this master thesis. The presentation has been focused on the algorithm semantics, and was intentionally kept clean of references to the actual implementation. All algorithm steps have been described: the thread graph construction, lockset analysis, Flow-sensitive data race analysis, and the fully concurrent data race analysis.

The following Chapter, "Design and Architecture", will describe the implementation aspects of the described concepts.

Example 1: The thread graph below, which should be familiar to the reader by now, contains fully concurrent threads. The graph has been extended with blue edges that indicate full concurrency between the connected nodes. Note, these edges are not part of the thread graph data structure. They have been added solely for illustration purposes.



Full concurrency is difficult to see by simply looking at the thread graph. The figure below shows one possible execution of the thread entries in the graph, with the aim to illustrate fully concurrent relationships more clearly.



It can be observed that the execution of threads $T1$ and $T2$ is completely independent from $T3$. During this particular exectuion example, $T1$, $T2$ and $T3$ actually run concurrently. However, this is not guaranteed. Because there is no start- or join-relationship that constrains their relative execution in any way, $T3$ might already have terminated before $T1$, and indirectly $T2$, have even started during a different execution of the program. To account for this inherent fuzzyness of information, the analysis assumes they run concurrently for their entire lifetime. This assumption is in accordance with the conservative nature of the analysis, because it ensures maximum potential concurrency, and leads to the detection of the most potential data races.

3. Design and Architecture

The goal of this chapter is to describe the architecture of the software developed during this master thesis, as well as any important design decisions that influenced its development. The chapter is focused on giving a high-level view of the software by showing the individual components, interfaces, and their interactions. It starts with showing the developed software system and its context, and introduce the technological constraints and conventions. It then successively descents deeper into the system itself, revealing and illustrating its components and subcomponents. After the description of the static components is complete, heir dynamic interactions as runtime elements are shown, followed by a description of the environment within which the system is executed. The chapter concludes with a documentation of the important design decisions.

The previous chapters were focused on the description of the theoretical foundations of this work, with very few references to any technical aspects. This property is also reflected in the textual style and structure of these chapters, which is inspired by traditional research papers. The remainder of this report cover the engineering part of this thesis. Accordingly, the textual style, structure and tone are more oriented towards a system specification, or design document, commonly used to describe software systems.

3.1. Requirements

This section briefly covers the functional- and non-functional requirements, which must be met by the developed software at the end of the thesis. Due to the experimental nature of the project, these requirements are small in number and not very detailed.

3.1.1. Functional Requirements

The main goal of this thesis is to build a tool that helps developers write correct multi-threaded programs by providing automatic detection of data races in C++ source code. The following list contains a short summary of the functional requirements that must be met by the analysis tool:

- The tool must implement a data race analysis for C++ programs, based on the algorithm described in Chapter 2.
- The tool must provide functionality to visualize the analysis results. This feature is defined very openly. At minimum, the analysis tool should provide the source locations of the statements, which are involved in potential data races.

The functional requirements are described more extensively in Section 1.6, "Thesis Goals".

3.1.2. Non-Functional Requirements

In addition, a set of non-functional requirements was defined:

- Performance:
  - There are no requirements regarding the performance of the system. The analysis might be very slow, especially when applied to larger programs. This is not a concern for the project at this stage.
- Software Quality:
  - The system should be ready for tests by actual users. It must be able to recover from analysis errors and incorrect program code.
- Portability:
  - There are no requirements regarding the portability of the system at this stage of the project. Currently supported are Mac OS X and recent versions of Linux. Porting the system to Windows should not be too difficult, but was not attempted during the project.

As already mentioned, these requirements are not as fleshed out as they would be during a "regular" application development project.

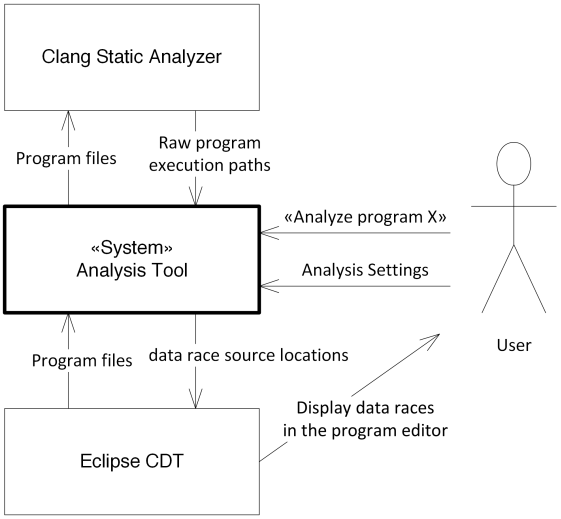


Figure 3.1.: The system context diagram shows the analysis tool, its adjacent systems, and the actors involved. The arrows connecting the systems represent the communication channels.

3.2. System Context

This section defines the context of the system under development, hence referred to as *analysis tool*. It illustrates the system boundaries of the analysis tool, and distinguishes it from the adjacent systems and actors. The system context is depicted in Figure 3.1. The arrows in this illustration represent the communication channels between the analysis tool and its neighbors, and show the logical information, which is exchanged over these channels.

The following sections describe the the analysis tool's neighbor systems and the communication channels.

3.2.1. Clang Static Analyzer

The Clang Static Analyzer is a bug finding tool for C, C++, and Objective-C programs, which is part of the Clang project and ships as part of the clang compiler. It provides a static analysis engine with support for path- and context-sensitive static program analysis. In addition to the analysis engine itself, the static analyzer provides a number of so-called *checkers*. Checkers are programs that use the analysis engine to find bugs in the program code. The static analyzer also provides an API, which can be used to develop custom checkers and program analyses [llv15c].

The Clang Static Analyzer and its analysis engine are implemented as C++ libraries. This allows them to be reused in different contexts. The analyzer can be run either as a standalone tool from the command line, or it can be included as a library, and integrated into other applications and tools.

Interaction with the Analysis Tool

The analysis tool uses the Clang Static Analyzer to extract the interprocedural execution paths from the C++ source code of the programs under analysis. This is done using a custom checker that simply traverses the execution paths supplied by the analysis engine, records them, and delivers them to the algorithm module.

3.2.2. Eclipse CDT / Cevelop

Eclipse CDT is an Integrated Development Environment for C and C++ based on the Eclipse platform, which can be extended and modified via a rich plug-in API.

Interaction with the Analysis Tool

The analysis tool uses Eclipse CDT to provide a user interface to run the analysis. Part of the system is an Eclipse plugin that extends the CDT UI with control elements to start the analysis, code markers that highlight data races found, and preference pages that allow configuration of the analysis tool. When the user runs the data race analysis on a program, the Eclipse plugin automatically locates all relevant source files and delivers them to the analysis module. After the analysis is finished, the plugin feeds the source locations of any detected data races back into Eclipse CDT, where they are displayed in the code editor.

3.2.3. User

The user is the person that starts the analysis tool, and reasons about its results. He is also in charge of the configuration.

Interaction with the Analysis Tool

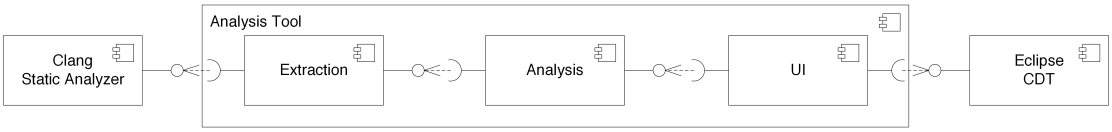
Users can interact in one of three ways with the analysis tool. They he can change the configuration settings, start program analyses, and evaluate the analysis results, which are displayed in the code editor. All interactions between the analysis tool and the users happens either via the aforementioned Eclipse Plug-in, or standard Eclipse components.

3.3. Building Block View

This section contains a static decomposition of the analysis tool into its building blocks and a documentation of the relationships between them. It starts with a white box description of the entire system, and successively descends down into the components and subcomponents, describing their contents, relationships, and interfaces.

3.3.1. The Analysis Tool

The diagram below shows the top-level components of the analysis tool, and the interfaces linking them together.



The analysis tool is comprised of three components:

- Extraction. This component is responsible for extracting the data from the source code that is required for the analysis.
- Analysis. This component performs the data race analysis. The implementation of the algorithm is located here. The component also contains a simple command line interface.
- UI. This component is the primary point of interaction between the user and the analysis tool. It is embedded into Eclipse CDT in the form of a plug-in, which extends CDT's user interface with additional buttons and menus that enable users to configure and run the analysis.

The following sections describe these components, the interfaces they use and provide, and the protocols and data structures that are used for communication over these interfaces.

3.3.2. Component: Extraction

The extraction component is responsible for the path extraction from the source code of the analyzed program. The extracted paths can be serialized to JSON, and may either be written to a file, or directly put on the standard output in oder to be processed further by another program. This component interacts directly with the Clang Static Analyzer, one of the neighbor systems.

In order to properly describe the interface facilitating this interaction, this section first introduces certain aspects of the Clang Static Analyzer itself. Afterwards, it describes the usage, application and integration of checkers for the Clang Static Analyzer, which have already been mentioned earlier. Lastly, the section documents the extraction component itself.

Symbolic Execution

The Clang Static Analyzer uses a technique called *symbolic execution* to analyze programs: It performs a path-sensitive walk over the program's flow graph. Symbolic execution is similar to regular program execution, but also has one key difference; it explores only one possible execution path, which is defined by the program's input values. Symbolic execution on the other hand explores all paths through the program. Also, instead of using concrete values for variables, it uses *symbols*. For instance, if the analyzer simulates a call to a function where it does not know the concrete return value, it conjures a new *symbol* that represents the abstract return value. This symbolic value is then associated with the variable that would store the concrete return value during regular execution.

The analyzer also accumulates constraints on these symbols during the path exploration. For instance, if the analyzer simulates an if-then-else statement that checks the condition *x != null*, it will store the constraint *x != null* on the $true$-branch, and *x == null* on the false branch. The static analyzer uses these constraints to narrow down the set of valid values on symbols.

Program State Graph

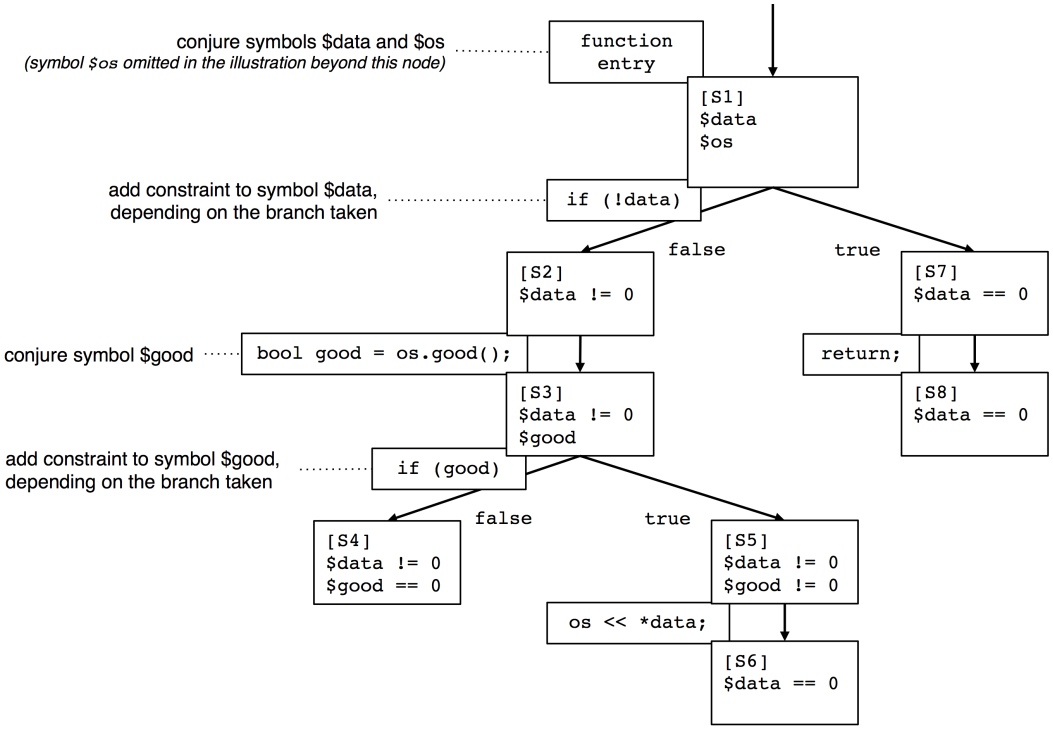
The result of a symbolic program execution is a graph that contains all reachable program states. Program state graphs are an alternative possibility to represent a program's potential execution paths, which, as opposed to traditional flow graphs, can be augmented with additional flow- and path-sensitive data.

An example of such a program state graph, including the corresponding program code, is shown in Figure 3.2. The analyzer successively constructs the graph illustrated in Figure 3.2b by traversing all execution paths of the program in Figure 3.2a. Note that the illustrated graph is heavily simplified. In reality, the graph is much larger. This is partially caused by the fact that the example program uses standard library functions that perform I/O, which introduces a lot of hidden complexity. The unsimplified program state graph corresponding to the program in 3.2a is illustrated in Appendix C.1, in order to give the reader a sense of the graph's scale [1].

[1] The graph was extracted using one of the clang debug checkers [llv15d].

```
void writeToStream(std::ostream os, std::string * data) {
  if (!data) {
    return;
  }
  bool good = os.good();
  if (good) {
    os << *data;
  }
}
```

(a) Example program



(b) Program state graph corresponding to (a)

Figure 3.2.: Program state graph example. This illustration shows a small program (3.2a) and the corresponding program state graph (3.2b). During the construction, the analyzer slowly accumulates constraints on the symbolic values.

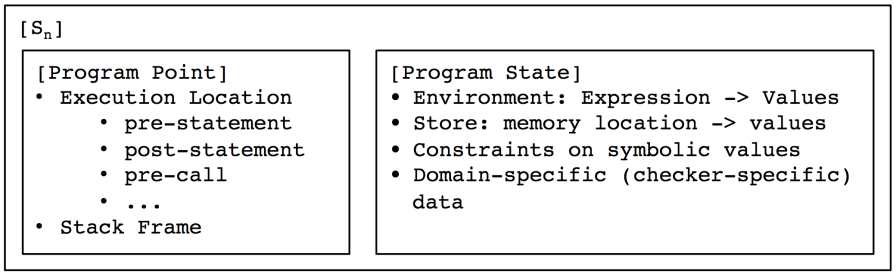


Figure 3.3.: Contents of a program state graph node.

Graph Contents

Each node in the program state graph contains two pieces of data: a program point and a program state[2] (cf. Figure 3.3).

- The program point contains information regarding the current *execution location*. The execution location property is often used by checkers to find program nodes they are interested in. Examples[3] for execution locations are:
  - *pre-statement*. the next operation will be the simulation of a particular statement.
  - *post-statement*. the last executed operation was the simulation of a particular statement.
  - *pre-call*. The next operation will be the execution of a call.

  The program point also contains a simulated *stack frame*, which is mostly used by interprocedural analyses, and the analyzer core itself.
- The program state contains the following information:
  - The *Environment* maps expressions to symbolic values. This information can be used to perform recursive function analysis, among other things.
  - The *Store* represents the current contents of the heap and the stack.
  - The current *Constraints* on symbolic values.
  - A generic data store for custom information pertaining to a particular analysis. Checkers can use this store to save arbitrary information.

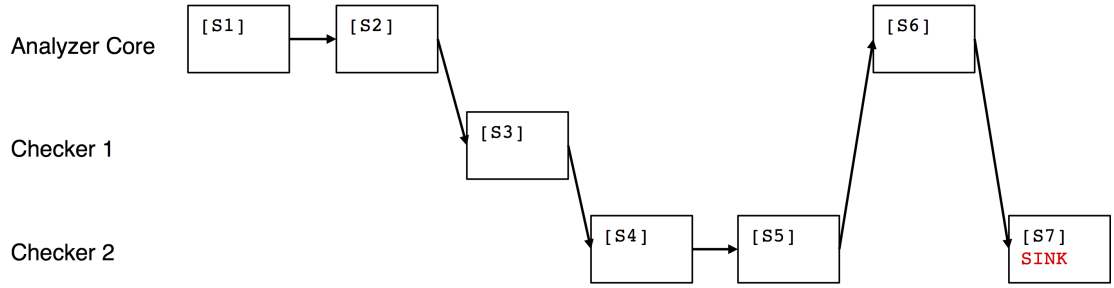
This concludes the introduction to the program state graph. The next section introduces custom checkers and how they can be integrated into the Clang Static Analyzer.

[2] In this section, the meaning of the terms *program point* and *program state* deviates slightly from their definitions in the introduction of this report. In this context, the terms were adopted from the checker documentation [llv15b], and the author chose not to change them for the sake of consistency with that document

[3] For a complete list of execution locations currently supported by the Clang Static Analyzer, refer to the Developer Manual [llv15a] and the Checker Documentation [llv15b].

Checker Interaction

All active checkers take part in the construction of the program state graph. The figure below illustrates how the construction of a particular program state graph might happen. The analyzer core generates the first couple of nodes, $S1$ and $S2$. The following nodes, $S3$, $S4$, and $S5$, are generated by two checkers, before the control is returned to the analyzer core. The construction continues in this fashion, until all execution paths are explored.



In addition, checkers can tell the analyzer core to stop the exploration of the execution path. This is useful in many situations, for instance when a checker finds a critical error that would cause the program to crash at this point in the execution. Any further analysis of subsequent program states becomes redundant, because they will never be reached by an actual program execution. In order to stop the execution, checkers can generate so-called `SINK` nodes, and add them to the program state graph. If the analyzer core encounters such a node, it will stop the exploration of this path.

Checker Integration

Checkers are integrated into the Clang Static Analyzer using the visitor pattern. All checkers must extend the `Checker` class template, which provides a set of *visit* functions. These functions correspond to various attributes of the program state graph nodes, such as the various execution locations, and other events, for instance *end-translationunit*, or *begin-analysis*.

In order to be invoked on specific attributes or events, concrete checkers must satisfy two requirements:

- They must override the visit function handling the attribute or event.
- They must be specify the attribute or event in the template arguments of the `Checker` class template.

The only place where checkers can store state information in between visit-function invocations is the generic data map of the current program state node. Any data stored in there will be available in each subsequent node of the same execution path.

Listing 3.1 shows a class declaration of a checker that has registered with two event types. The template arguments `check::PreCall` and `check::PreStatement<ReturnStmt>` specify that the checker wants to be invoked before calls are evaluated, and each time a `return` statement is

Listing 3.1: `Checker` subclass declaration example. This code shows what the class declaration of the `PreStmtAndPreClasschecker` from Figure 3.4 might look like.

```
#include "clang/StaticAnalyzer/Core/Checker.h"

using namespace clang;
using namespace ento;

class PreStmtAndPreCallChecker :
       public Checker<check::PreCall, check::PreStmt<ReturnStmt>> {
public:
  void checkPreCall(const CallEvent &Call, CheckerContext &C) const;
  void checkPreStmt(const Stmt &S, CheckerContext &C) const;
};
```

encountered. The member functions handling the specified event types are overridden to specify the visit behavior.

The class diagram in Figure 3.4 shows the `Checker` class template, which must be extended by all concrete analyzer checkers. It also shows other notable classes and packages. Three concrete checkers, which have registered to various events by defining the corresponding member functions and supplying the required arguments to the `Checker` class template, are also illustrated, including the `PreStmtAndPreCallChecker` from Listing 3.1.

Note that neither this chapter nor the class diagram contains a full recital of all events or properties for which checkers can potentially register. This report refrains from enumerating them, and many have been left out to make the illustration more concise. A complete description of all events and properties, including their function signatures and possible uses, can be found in the Checker Developer Manual [llv15a].

Checker Registration

Checkers must be registered with the Clang Static Analyzer before they can be executed. There are several different ways of accomplishing this. According to the Checker Developer Manual [llv15a], new checkers may only be registered during compilation of the analyzer, which happens as part of the compilation of Clang. For this to work properly, the checker implementation file must be located in a specific folder inside the Clang source tree, and in addition, the checker name must be added to a so-called *checker index table*. This approach is described in detail in the developer manual. Since it was not used during this project, it is omitted here.

Fortunately, there exists a second, lesser known approach. The Clang Static Analyzer allows for checkers to be written as plug-ins and loaded by the user at runtime. However, this functionality lacks official documentation. The author discovered this approach by asking on the clang mailing list[4]. The following subsection describes the structure of an analyzer plug-in and how it can be loaded an run as part of the Clang Static Analyzer.

[4]The corresponding mailing list conversation is can be found in Appendix C.2

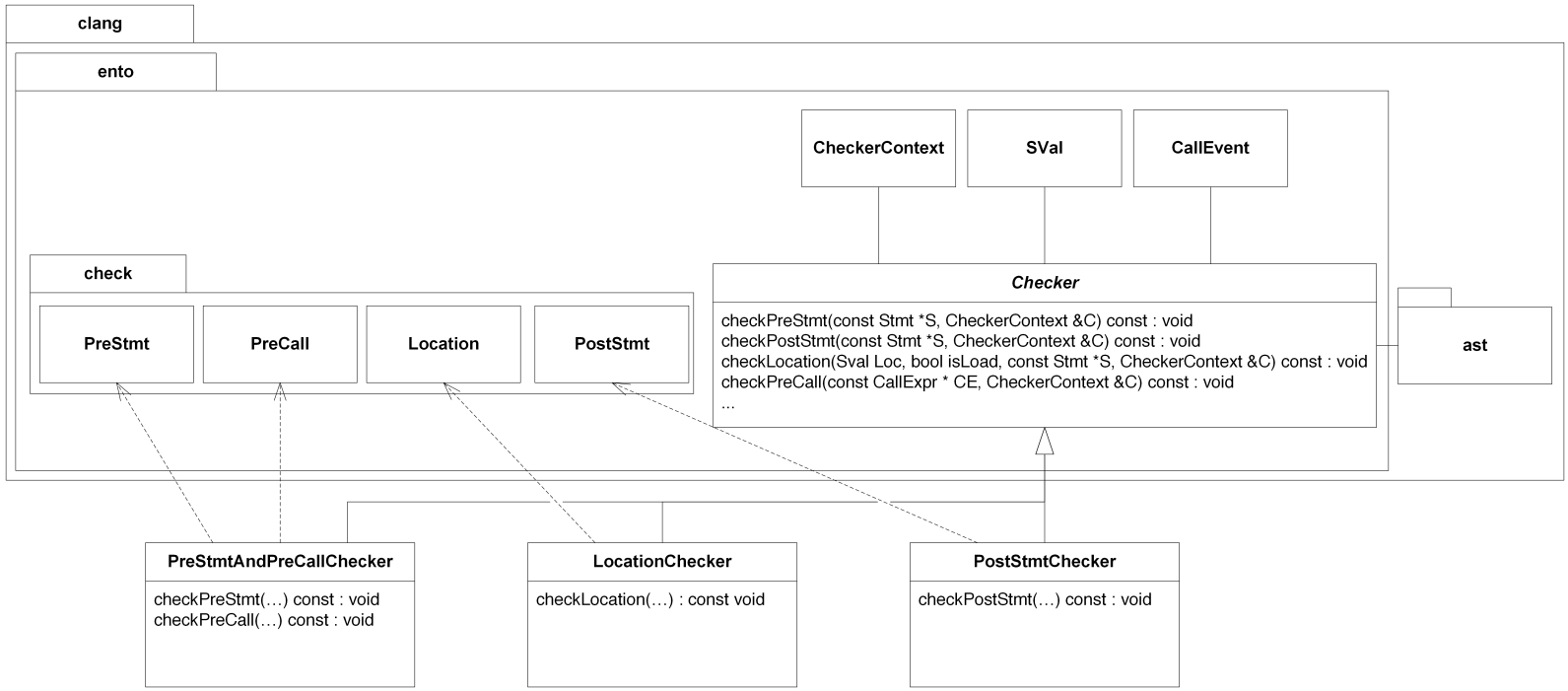


Figure 3.4.: `Checker` class template, implementations, and environment. This class diagram shows the abstract class template, which must be extended by all checkers, as well as other notable classes and packages. It also contains three concrete checkers (`PreStmtAndPreCallChecker`, `LocationChecker`, `PostStmtChecker`) that have registered to various events, by defining the corresponding member functions and supplying the required arguments to the `Checker` class template.

Static Analyzer Plugins

Plugins for the Clang Static Analyzer are dynamic libraries that expose a specific set of symbols. These symbols define the plugin contents. They are read by the analyzer when loading the plug-in.

- The first symbol is `clang_analyzerAPIVersionString`, which specifies the version of the Clang Static Analyzer API used by the plug-in. This information is used to detect compatibility issues when running a plug-in with versions of clang other than the one the plug-in was compiled against. The API version is defined by the preprocessor variable `CLANG_ANALYZER_API_VERSION_STRING` in one of the libraries required to compile the plug-in. Therefore, the symbol is usually defined using the following line:

  ```
  extern "C" const char clang_analyzerAPIVersionString[] =
         CLANG_ANALYZER_API_VERSION_STRING;
  ```

- The second symbol that a plug-in must export is the function `clang_registerCheckers`. This function will be called when the plug-in is loaded, and its purpose is to register the checkers with the Analyzer, which are contained within the plug-in.

  ```
  extern "C" void clang_registerCheckers(CheckerRegistry &reg) {...}
  ```

  The `CheckerRegistry`, which is passed to this function as an argument, manages the checkers available to the static analyzer. The member function `addChecker` must be used to register new checkers with the analyzer. This function takes three arguments:

  1. The checker class. This is a template argument.
  2. The name of the checker. The name is used to select the checker when running the static analyzer. Checkers are usually grouped into packages, with the name of the checker being "package.checker" or "package.subpackage.checker".
  3. A description of the checker. This will appear when the list of checkers is shown on the command line.

  For example, if a plugin wants to register a checker that is implemented in a class called `ExampleChecker`, the implementation of the `clang_registerCheckers` function would look something like this:

  ```
  extern "C" void clang_registerCheckers(CheckerRegistry &registry) {
    registry.addChecker<ExampleChecker>(
      "example.ExampleChecker",
      "Checker to exemplify checker registrateion"
    );
  }
  ```

In addition, it is considered standard practice to place all code pieces of the plugin into one or more anonymous namespaces, except for the aforementioned symbols.

Loading and Executing Checkers

The Clang Static Analyzer is built into the clang compiler. It must be invoked from the command line, and can be configured using a series of arguments. For instance, the following command analyses a file `main.cpp` using the aforementioned `ExampleChecker`.

```
$ clang -cc1 -analyze -analyzer-checker=example.ExampleChecker main.cpp
```

If the checker is located in an analyzer plugin that must be dynamically loaded, the following argument must be included.

```
$ clang -cc1 -analyze -analyzer-checker=example.ExampleChecker -load plugin.so
 main.cpp
```

In addition, there is a number of arguments that were useful during this project. Most of these are also used frequently when compiling C++ programs.

- `-std=c++11`. Sets the language level of the compiler to C++11.
- `-stdlib=libc++` or `stdlibc++` Specifies which implementation of the C++ library to use
- `-I/path/to/library`. Specifies additional library paths to consider when parsing and analyzing a program. Can be used repeatedly.
- `-function` *`function-name`*. Only analyze the specified function, instead of the entire program.

Both the Clang compiler and the Clang Static Analyzer support many more command line arguments and options than the small selection presented in this section. A synopsis thereof can be viewed using the commands `clang -help` and `clang -cc1 -help`, respectively. The output is omitted here.

This concludes the introduction and description of the Clang Static Analyzer functionality. The following sections cover the implementation of the extraction component.

Component Overview

A white-box view of the extraction component is illustrated in Figure 3.5. The entire extraction component is centered around a custom checker for the static analyzer, which is responsible for the extraction of the data structure needed by the algorithm component. The individual pieces of the graph data structure are constructed using a set of helper classes, which are located in the `graph` package. This checker itself is of course dependent on the Clang Static Analyzer infrastructure, as described in the preceding sections. The illustration simplifies this complex relationship by just showing a dependency to a *clang* package.

The extraction results can be serialized to JSON, and may either be written to a file, or directly to the standard output to be parsed and processed further by another program. The serialization was implemented using *rapidjson*, a modern, open source JSON library for C++.

The subcomponents illustrated in the package diagram in Figure 3.5 are described over the course of the following sections.

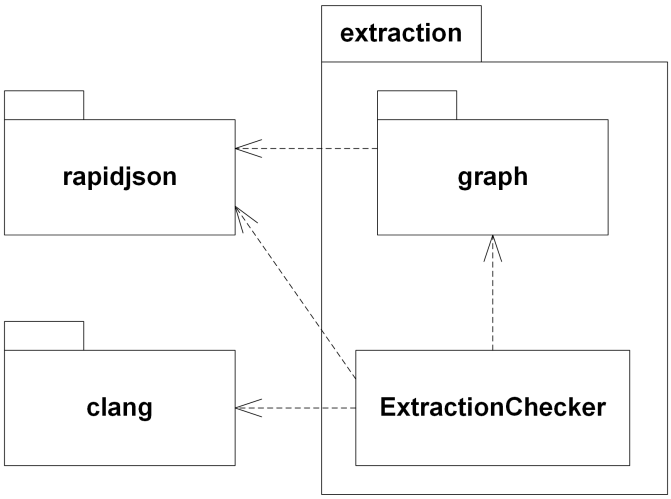


Figure 3.5.: White-box view of the extraction component.

The Extraction Checker

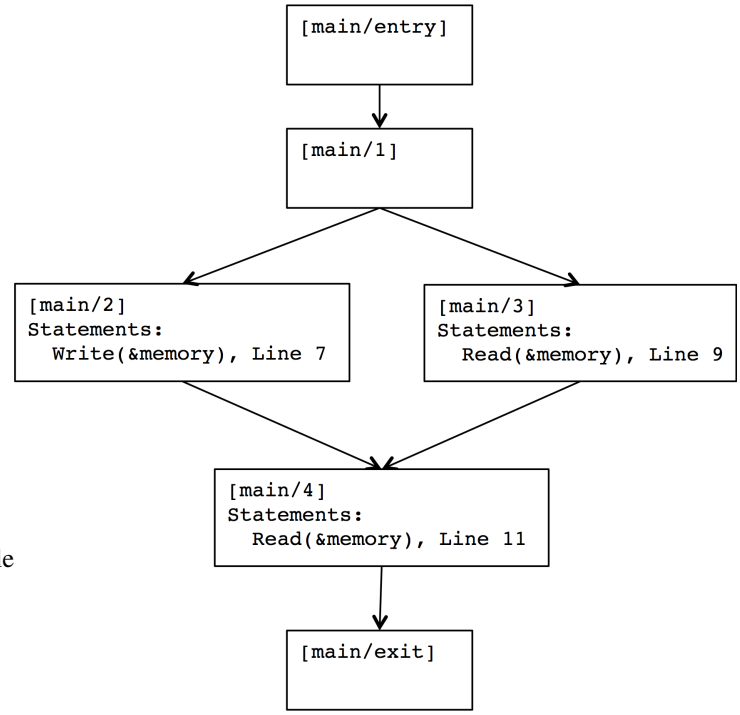
In order to extract the necessary data for the analysis, the `ExtractionChecker` traverses the program state graph, as described in the preceding sections, and constructs a graph containing all possible execution paths.

The resulting graph is reminiscent of a traditional flow graph. It is comprised of basic blocks, which contain statement sequences and are connected by edges that define the flow of control between individual blocks. It also features special entry and exit nodes, which are common in flow graphs. However, because it is constructed from traversing the state graph of a program, its concrete layout is usually dissimilar from that of a traditional flow graph of the same program. The reason for this condition is that program state graphs, which explicitly express notion of program state in the form of graph nodes, may contain the same statement multiple times, as part of different program states, whereas traditional flow graphs contain each statement exactly once. Because the extraction checker traverses the program state graph and not the flow graph, it may visit and record the same statements multiple times as well. Therefore, the resulting graph is more accurately described as an *execution path graph*, despite looking like a flow graph.

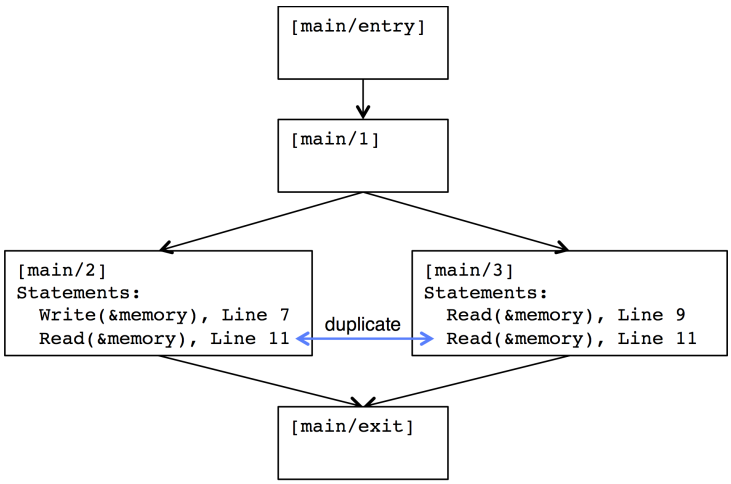
The difference between flow graphs and execution path graphs is exemplified in Figure 3.6. Illustration 3.6b shows the flow graph of the example program; the statements are distributed over the basic blocks, and each statement only exists once. Illustration 3.6c shows the execution paths. The read on line 11 is contained twice in this graph, because it can be reached on more than one execution path.

```
1  #include <iostream>
2  #include <cstdlib>
3  int memory{};
4
5  int main() {
6    if (rand() % 2) {
7      memory = 2;
8    } else {
9      std::cout << memory;
10   }
11   std::cout << memory;
12 }
```

(a) Example program with multiple execution paths.



(b) Traditional flow graph.



(c) Extracted execution paths.

Figure 3.6.: Difference between the traditional program flow graph (b) and the representation used during the extraction of the execution paths (c). The read on line 11 is contained twice in the execution path graph because it can be reached from multiple program contexts.

Contradiction with the Algorithm Description

In Chapter 2, "Data Race Detection Algorithm" it was stated that the algorithm requires an interprocedural flow graph as input for the analysis. The graphs presented in this section seem to contradict the algorithm description. They are clearly not flow graphs, yet they are used as input for the algorithm component.

This contradiction is caused by the fact that the extracted graph data structure is already one step ahead. They execution paths, which are only implicitly visible in the flow graph, are expressed explicitly by the extracted graphs in the form of concrete nodes. The computation of these paths has been delegated to the Clang Static Analyzer. This makes the analysis steps significantly easier. Gathering all execution paths requires simply a traversal of the execution path graph. If the analysis only had access to the flow graph, a process similar to the symbolic execution of the static analyzer would be required in order to first extract the execution paths, before the actual data race detection could begin.

The Extraction Data Structure

The extracted graph only contains information pertaining directly to the data race analysis. In order to simplify and speed up the processing and analysis of this graph, as much detail as possible is omitted during the extraction. The analysis is only interested the following types of statements:

- Read and write operations. Only the memory location is important. The operators and concrete values involved in the operations are not relevant. Examples:

```
std::cout << x;
int temp = x;
passByValue(x);
x = function();
```

- Mutex acquisitions:

```
mutex.lock()
std::lock_guard<std::mutex> guard(mutex);// deadlock-free
```

- Mutex releases:

```
mutex.unlock()
guard.~lock_guard() // destructor call / guard goes out of scope
```

- Thread starts:

```
std::thread T(...);
```

- Thread joins:

```
T.join();
```

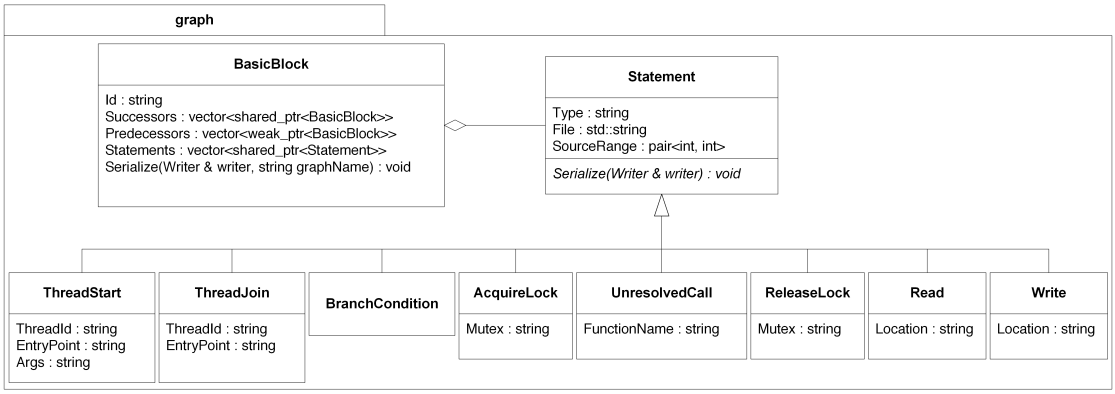


Figure 3.7.: Contents of the `graph` package. The classes representing the graph data structure do not contain much functionality except the `Serialize` function, which creates a JSON representation of the currently represented graph.

- Unresolved Function calls. These do not directly pertain to the data race detection. However, they are needed to support analysis across multiple C++ translation-units, which is not yet supported by the Clang Static Analyzer. More this subject later.

These statements are represented in the extraction component using a simple class hierarchy (cf. Figure 3.7). When the checker encounters any of the statements mentioned above, it adds an instance of the corresponding class to the statement sequence of the current basic block. All other statements are omitted and not added to the extracted graph.

Basic Blocks and Branching

Each basic block is comprised of two lists that represent its predecessors and successors, and a list holding instances of the classes described above, which represents its execution sequence contained in the block. The `BasicBlock` class is illustrated in Figure 3.7.

In order to correctly represent the execution paths, the branching present in the raw program state graph must be preserved in the extracted data structure. During the extraction, the checker looks for certain events that cause execution path branching, and updates the graph accordingly, inserting a new basic block for each new branch. Currently, there are two events that cause the checker to insert new basic blocks.

- Statements of `check::BranchCondition` type, which represent any kind of branch condition, such as *if*, *else-if*, *switch*, and so on. They are the common denominator of the different control flow constructs in C++ and indicate that branching is imminent. Statements representing branch conditions are also stored temporarily in the graph during the extraction process. However, they are not serialized to JSON, because they serve no further purpose after the graph has been extracted successfully.

- Unresolved functions calls. These do not necessarily cause branching. However, the resolution of these function calls will cause the flow of control to be redirected to other

basic blocks. Anticipating this by breaking up the graph in advance at these redirection points greatly simplifies function call resolution. The general problem of cross translation unit analysis and function call resolution will be addressed later, in Section 3.3.3.

In addition, each basic block has an unique id, which is a composition of a sequence number and the name of the graph, which the block belongs to. Examples of these block ids can be seen in Figures 3.6b and 3.6c.

Extraction Process

Multi-threaded programs are almost always comprised of more than one execution path graph. Usually, each thread entry present in the program has its own execution path graph. Also, unresolved function calls, whose definitions reside in other translation units, have their own graphs, which are later incorporated into the graph from where the function call originated.

The extraction process is iterative. It starts by extracting a graph for the main thread, meanwhile recording all threads and unresolved functions that are discovered. As soon as the extraction of the main thread is completed, the extraction continues by subsequently extracting the graphs for all functions and threads, until no new extraction candidates are found. Each extracted graph gets a unique name, which is equal to the mangled name of the function at the origin of the graph.

```
void f() { ... }
main() {
  std::thread thread(f);
}
```

For instance, the name of the graph for the thread `thread` in the listing above would be called something like `_Z4funcv`, because its origin is the function `func`.

Extraction Output

The output of the extraction is a JSON string representing the execution path graphs for all threads and unresolved functions that are present in the program under analysis. This string will be parsed and processed further by the other components of the analysis tool. The JSON object structure is an almost identical mapping from the previously described graph data structure. There are objects that represent basic blocks and statements, and arrays representing predecessor and succressors, as well as statement sequences. The JSON representation deviates only in one aspect from the in-memory C++ representation: It uses an attribute called `|type` to specify the kind of statement that an object represents. This is necessary because unlike their C++ counterparts, JSON objects cannot be distinguished using static or dynamic type information.

The untouched JSON string generated by the extraction component when applied to the program in Listing 3.6a is shown in Appendix C.3. The graph represented by the JSON string is illustrated in Figure 3.6c.

3.3.3. Component: Analysis

This component contains the implementation of the data race algorithm. It is responsible for the preprocessing and analysis of the extracted execution paths. The entire analysis component, including the data race detection algorithm, is written in Scala.

Component Overview

Figure 3.8 shows a white-box view of the analysis component and its relevant dependencies. Evidently, it is comprised of three packages:

- graph. This package contains the classes representing the extracted graph. Analogously to their C++ counterparts in the extraction component, these classes support serialization and deserialization to and from JSON. The serialization was implemented using the *spray-json* library, which is part of the well-known *spray* framework.
- analysis. This package contains the implementation of the data race algorithm.
- driver. This package contains classes that are responsible for the operation of this component. The driver interacts with the Clang Static Analyzer to extract and parse the execution path graphs, loads configurations, runs the algorithm, and prints out its results. It also provides a public API that serves as the interface for the UI component.

The following sections will describe the contents of these packages. Keep in mind that the algorithm functionality was already covered in Chapter 2 and will not be explained again. Where necessary, the text will provide references to the corresponding sections in the algorithm chapter.

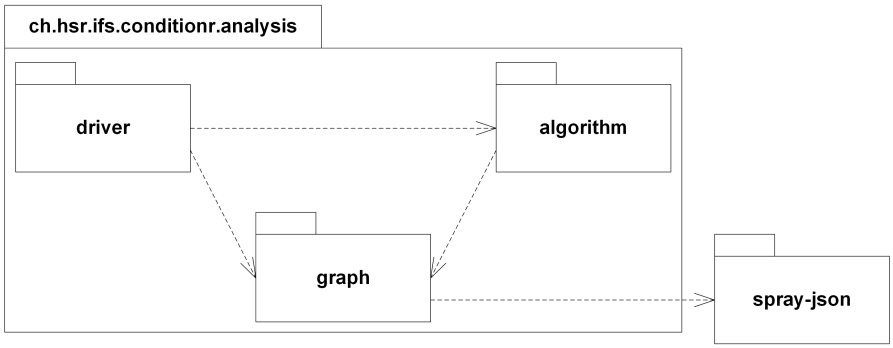


Figure 3.8.: White-box view of the analysis component and its relevant dependencies.

Graph

The graph package contains the types that represent the graph data structure. These types are used across the entire analysis component. In addition, the graph package also contains the JSON serialization logic. A class diagram of the package contents is illustrated in Figure 3.9. The following list summarizes the contents of the class diagram and elaborates more on some noteworthy properties.

- `GraphJsonProtocol`. This class is contains the logic to deserialize JSON strings obtained from the extraction component. During the deserialization, the `GraphJsonProtocol` will construct an in-memory representation of the extracted execution path graphs using the other classes that are shown in the diagram.
- `Graph`. This is the top-level container for all execution path graphs of an entire program. It is an aggregation of a number of `PartialGraph` instances.
- `PartialGraph`. Each instance of this class contains an actual execution path, and is associated with either a thread or an unresolved function. A partial graph is an aggregation of `BasicBlock` instances, which are the actual contents of the execution path graph.
- `Block` and subclasses. These classes represent the different types of nodes, which are present in the `PartialGraph`. The subclasses `Entry` and `Exit` represent the entry and exit nodes, which are unique in each graph.
- `Statement` and subclasses. These classes represent the statements needed by the algorithm to perform the data race detection analysis, and have already been discussed in Section 3.3.2, under "The Extraction Data Structure". They are mapped over directly from their C++ counterparts.
- `AnyVal` subclasses. These classes exist to improve the code readability. Scala allows the definition of custom value types by simply extending *scala.AnyVal*. Representing data using these value types rather than primitives and strings is considered good programming practice.

The classes described and illustrated in this section will be used in subsequent diagrams of this chapter without explicit referencing, to preserve space both in the diagrams and in the accompanying descriptions.

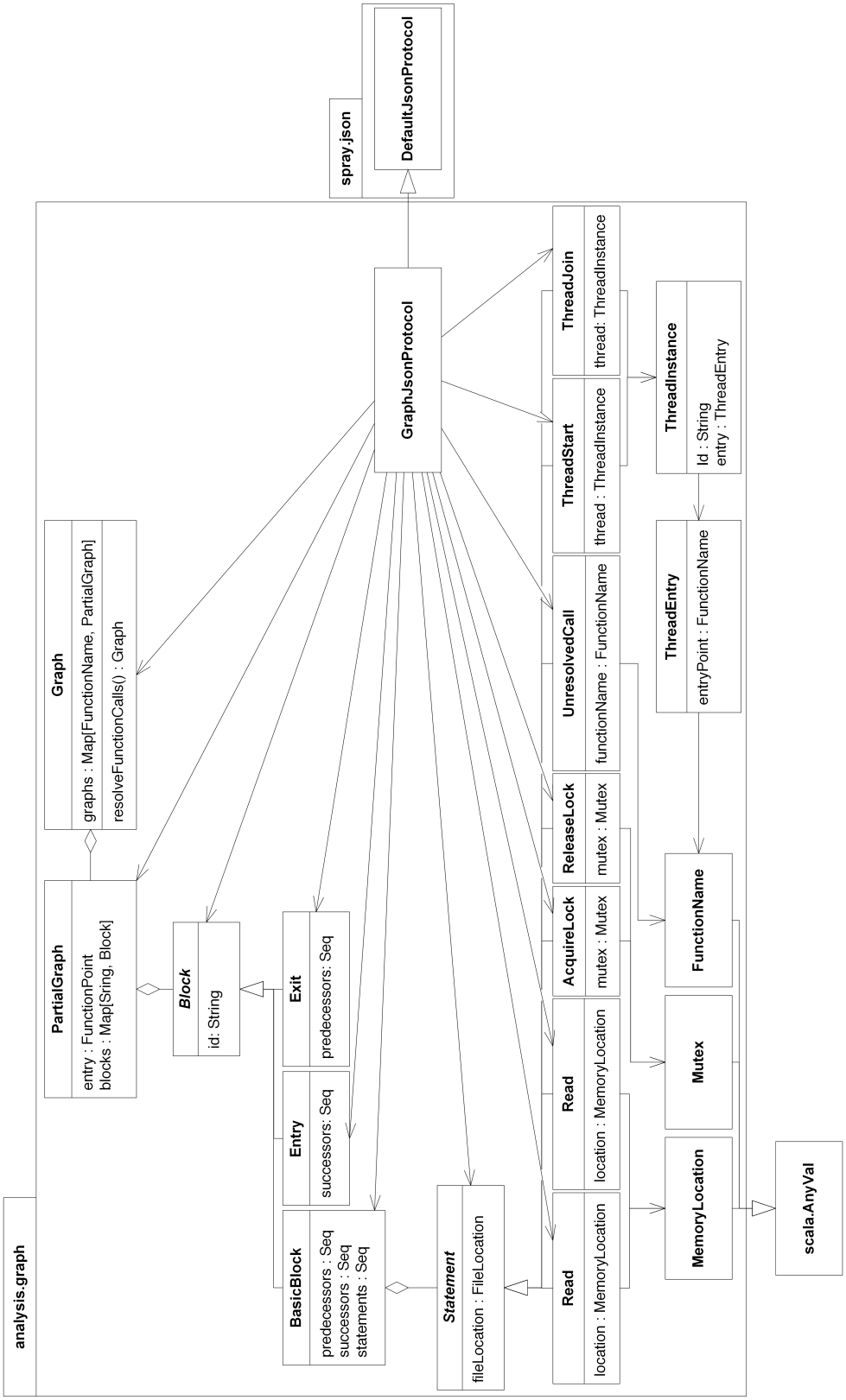


Figure 3.9.: Class diagram showing the contents of the graph package.

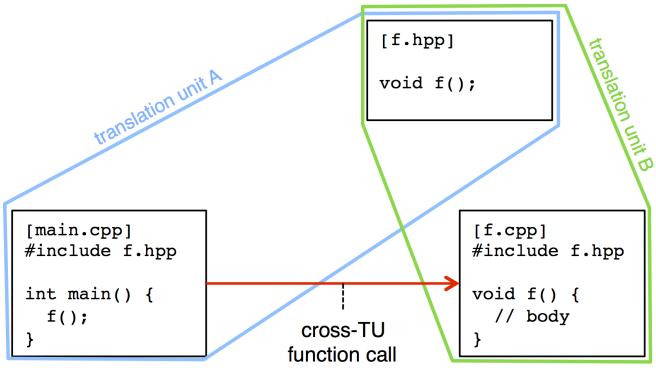


Figure 3.10.: Class diagram showing the contents of the graph package.

Support for Cross Translation Unit Analysis

An important feature that was not yet been addressed is the support for cross translation unit analysis. Being a cross-cutting issue, which affects both the extraction and the analysis component, its description was delayed purposefully up until this point, in order to prevent extensive references to later sections in the document. Nevertheless, aspects relating to this topic were sometimes briefly touched on, with mentions of unresolved functions, function call resolution, and such. In this section, the hinting will stop, and the cross translation unit support will be covered in full.

A translation unit is a *preprocessed* source file. This means essentially that all preprocessor directives have been expanded (`#include`, `#define`, etc.). Translation units serve as the input to the compiler. During compilation, the compiler generates a so-called object file for each supplied translation unit. Usually, these object files are then linked together in a subsequent step, to form an executable program or a library.

The issue here is that the compiler processes each translation unit independently from the others. Calls to functions defined in other translation units are resolved later by the linker. The compiler merely generates some information for the linker, so that it can correctly wired together the function call and the function body.

Figure 3.10 illustrates this situation. It shows two translation units, A, and B. Unit A represents the source file `main.cpp`, and unit B represents the source file `f.cpp`. Note that the contents of the header `f.hpp` are present in both units, because they were copied into the source files during preprocessing. Unit A contains a call to function `f()`, which is defined in unit B. Because both translation units are compiled independently, the compiler does not follow the function call into unit B during the compilation of A.

The Clang Static Analyzer has the same behavior as the compiler. It operates on translation units, and it analyzes each translation unit individually.

Function Call Resolution

It has already been mentioned that the extraction component adds `UnresolvedCall` nodes to the extracted execution paths, when it encounters function calls, which cannot be resolved on the spot. Also, it extracts the execution path graphs corresponding to the bodies of all unresolved functions, just like the graphs that represent thread entries.

A function call resolution has been implemented, which substitutes the unresolved calls with the execution path graph of the function body, using the information stored in the `UnresolvedCall` nodes. A substitution limit was incorporated to prevent infinite nesting of recursive cross translation unit function calls. The implementation is located in the `resolveFunctionCalls()` method of the class `Graph` (cf. Figure 3.9). When invoked, this method executes the algorithm and returns a new `Graph` instance, were all `UnresolvedCall` nodes have been replaced with the partial graphs of the corresponding function bodies.

The function call resolution is currently only implemented for regular C++ functions. It serves as a proof-of-concept for the substitution algorithm. Additional cases which must be implemented and tested are:

- Class member functions. During experiments, this was actually working fairly consistently. However, at the moment, there exist no test cases to verify the behavior.
- Template functions. Similar to the class member functions, these are working to some extent, but untested.
- Class Templates. No tests or experiments have been conducted.
- Function Pointers. Not working.
- Operators. No tests or experiments have been conducted.

The implementation of these cases was not even attempted during the project, due to a lack of time. This concludes the description of the cross translation unit support and the function call resolution.

Algorithm

As the name suggests, the algorithm package contains the implementation of the data race detection algorithm. For the most part, the algorithm implementation follows the description in Chapter 2. In some aspects however, the implementation diverges from the theory, mainly in favor of code simplicity and to alleviate unnecessary overhead. This section describes the algorithm implementation and highlight any differences with respect to the conceptual description of the algorithm. It follows the same order as Chapter 2, first introducing the implementation of thread graph, followed by the lockset, and finally the analyses themselves.

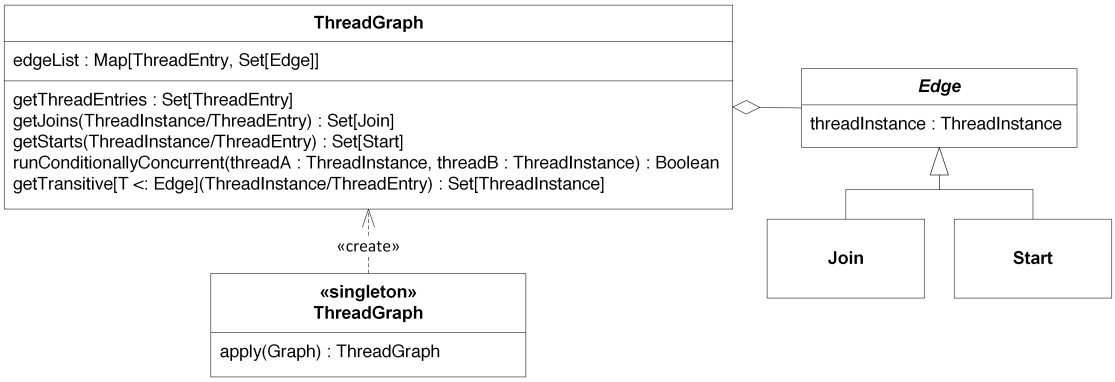


Figure 3.11.: Diagram showing the classes involved in the thread graph construction .

Thread Graph Construction

The implementation of the thread graph construction process is very similar to its description in Chapter 2. The graph itself is represented using a map that associates thread entries with their immediate outgoing $\xrightarrow{start}$ and $\xrightarrow{join}$ relationships. Transitive $\xrightarrow{start}$ and $\xrightarrow{join}$ sets for a particular thread can be computed by recursively looking up the direct relationships of the same type. The classes involved in the construction process are illustrated in Figure 3.11:

- `ThreadGraph`. The central class of the construction process. It holds the map representing the graph nodes and edges, and provides query functions for various graph properties.
- `ThreadGraph` companion object. The `apply` method of this object serves as a factory method for `ThreadGraph` objects. It computes the contents of the thread graph based on the data from the extracted execution path graph and creates a `ThreadGraph` afterwards. The computation is implemented exactly as described in Section 2.3.1 of the algorithm chapter.
- `Edge` and subclasses. These classes represent the directed edges, and thus the $\xrightarrow{start}$ and $\xrightarrow{join}$ relationships between the thread entries in the graph. the `ThreadInstance` member describes the edge destination.

There is one important difference between the thread graph described in Chapter 2 and the version that was implemented. Note that the diagram illustrating the thread graph implementation contains references to the classes `ThreadEntry` and `ThreadInstance`. In particular, the nodes in the graph are represented as `ThreadEntry` objects, whereas the `Edges` point to `ThreadInstances`. The differences between thread entries and thread instances have been discussed and exemplified in Section 2.3.1. To recapitulate:

- A Thread instance represents a concrete thread, running on the machine.
- A Thread entry is a more abstract concept. It incorporates the fact, that from a static point of view, the exact amount of threads running at the same time is indeterminate in certain circumstances.

However, the analysis engine of the Clang Static Analyzer, which performs an extremely accurate simulation of the program execution, allows a more precise approximation of the number of running threads. It removes the necessity to assume that a thread-start located inside a loop must be started between 0 and n times. For instance, given a loop, the analysis engine simulates an estimated number of loop executions. Depending on the simplicity of the loop condition, the analysis is even able to predict the exact number of iterations that will be performed. This in turn allows the analysis to narrow down the number of possible thread instances inside a loop. There are many other situations where the same principle applies, for instance in recursively defined functions. The implementation of the thread graph was adapted to account for this increased precision. The edges are still outgoing from thread entries, mainly because recording the same edges for multiple instances of the same thread would be redundant. However, the edges point to thread instances. If a thread entry starts multiple instances of the same thread, there will be an according start edge for each instance.

This change generally causes thread graphs to grow larger. Nevertheless, having a more precise model of the running threads should result in a better precision of the algorithm results, which warrants the change, especially given the experimental context of this project.

Program Point Computation

After the thread graph construction follows the program point computation, a task that was not explicitly described in Chapter 2. However, since all subsequent algorithm tasks are flow- and path-sensitive and thus require the execution paths in the form of program point sequences (cf. Section 1.4.2, "Data-Flow Analysis"), it makes sense to precompute and reuse this information.

The program point sequences are computed by performing a full walk of the extracted execution path graph. While doing so, the program state information that is relevant to the analysis is carried along:

- The locklist, containing all currently held locks. This information is later used by the lockset analysis and the data race detection.
- The concurrent thread list, which is used during the flow-sensitive data race analysis.

For each visited `graph.Statement`, an `algorithm.ProgramPoint` instance is created where the current values of both the locklist and the concurrent thread list are stored[5].

The result of this operation is a representation of the program as a sequence of program points, which can be used as basis for the remaining algorithm tasks. Moreover, the order in which these program points are processed is insignificant, because the flow- and path sensitive information has already been precomputed. This greatly simplifies the implementation of the remaining tasks. They can just iterate over the program point sequences and analyze each program point individually, instead of having to perform a correct traversal of all execution paths.

To illustrate this, consider the flow-sensitive data race analysis. All information required to perform this analysis, namely the locks held during memory operations and the conditionally

[5] An in-depth description of the locklist and concurrent thread list properties and behavior, including the conditions under which they are updated during the execution path traversal, is contained in Sections 2.3.2 and 2.3.3 of the algorithm chapter.

concurrent threads, is stored in each individual program point. Therefore, all program points can be examined independently of each other. In fact, most program points can be skipped, except the ones associated with statements relevant for the lockset analysis.

The classes involved in the program point computation are illustrated in Figure 3.12:

- `ProgramPoints`. This class holds the precomputed program point sequences for all thread entries.
- `ProgramPoints` companion object. The `apply` method of this object serves as a factory method for `ProgramPoints` objects. It performs the program point computation, given a thread graph the extracted execution paths. After the computation is finished, it creates a `ProgramPoints` instance holding the resulting program point sequences.
- `ProgramPoint`. This class represents a single program points. Its members store program state information.

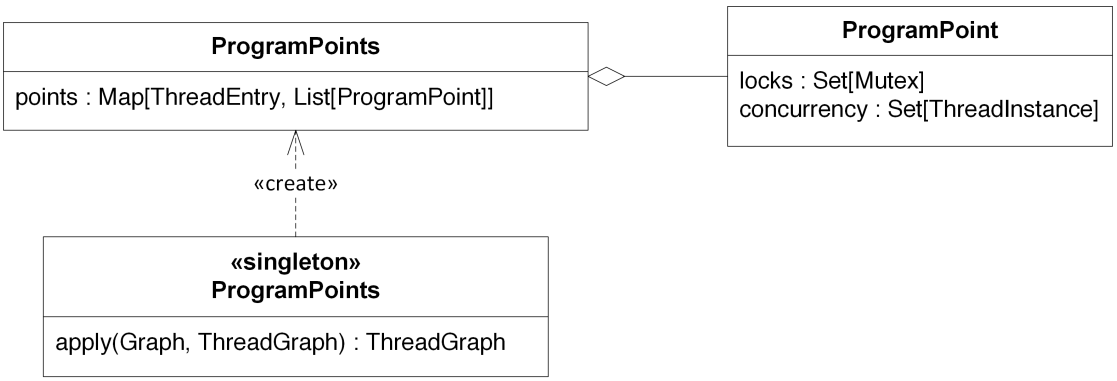


Figure 3.12.: Diagram showing the classes involved in the program point computation.

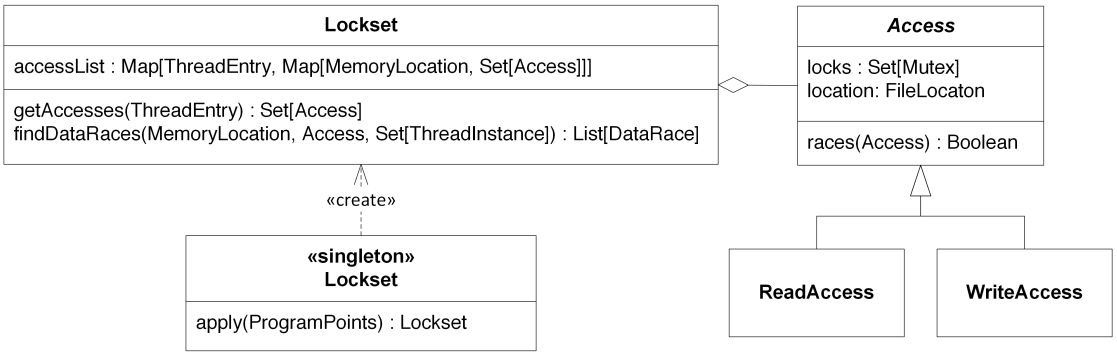


Figure 3.13.: Diagram showing the classes involved in the lockset analysis.

Lockset Analysis

The lockset analysis process is implemented as described in Section 2.3.2, with no significant changes. The lockset itself is represented as a map that associates thread entries and memory accesses. The classes involved are illustrated in Figure 3.13:

- `Lockset`. This class holds the lockset in the form of a map that associates thread entries to memory accesses. In addition, this class provides a method `findDataRaces`, which can be used to find potential conflicting accesses made from a set of threads on a particular memory location.
- `Lockset` companion object. The `apply` method of this object serves as a factory method for `Lockset` objects. Given a set of program point sequence, this method performs the lockset analysis as described in Chapter 2, and subsequently creates a `Lockset` object.
- `Access` and subclasses. These classes represent memory accesses. In addition to the locks held, they provide a method `races()`, which can be used to determine whether there is a potential conflict with another access. The file location of the access is stored to display useful information to the user.

Flow-Sensitive Data Race Analysis

The flow-sensitive data race analysis is implemented as described in Section 2.3.3, with no significant changes. The classes involved are illustrated in Figure 3.14.

- `Analysis Context`. This class is an aggregation of the data needed by the analysis, such as the thread graph, the program points, and the lockset.
- `FlowAnalysis`. This class implements the flow-sensitive data race analysis. It provides two methods, each corresponding to one objective of the analysis. The method `conflicts()` yields list of potentials data races between conditionally concurrent threads, and `unconditionallyConcurrentThreads()` yields a set of pairs of unconditionally concurrent thread instances.
- `DataRace`. This class represents a potential data race. The members store information used to describe the location of the data race and to provide useful information to the user.

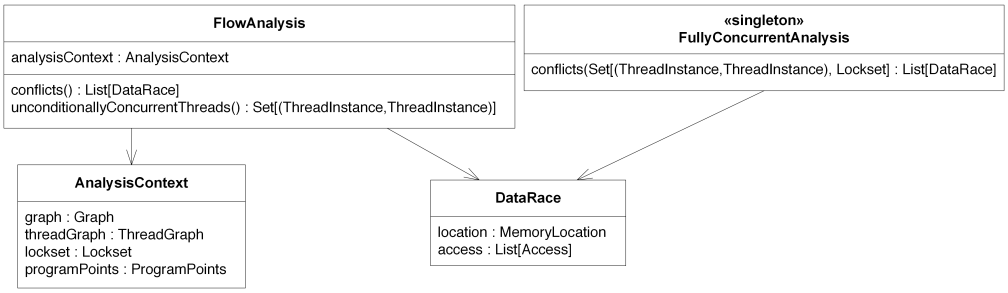


Figure 3.14.: Diagram showing the classes involved in the flow-sensitive data race analysis.

Fully Concurrent Data Race Analysis

The implementation of the fully concurrent data race analysis shows no significant differences to its description in Section 2.3.4. The analysis implementation is located in the `FullyConcurrentAnalysis` class, illustrated in Figure 3.14. This class provides one method `conflicts()` that performs the analysis. This method requires a `Lockset` instance and a set of fully concurrent thread pairs that, as precomputed by the Flow-sensitve data analysis class. The result of the analysis is a list of potential data races between the supplied thread pairs.

Driver

The driver package contains classes that are responsible for the operation of this component, and glues It uses the Clang Static Analyzer to extract and parse the execution path graphs, runs the algorithm, and prints out its results. In addition, the driver package provides a public API that serves as the interface for the interaction with the UI component.

All classes that are part of the driver are illustrated in the diagram in Figure 3.15. The structure of the driver package is fairly simple. The class `Extractor` is in charge of the graph extraction. It requires an instance of `Clang` that has been parameterized with a valid `ClangConfig` in order to work properly. The `ClangConfig` contains the information necessary to invoke the Static Analyzer; the location of the clang executable and of the extraction plugin, the checker name, and any additional flags that should be passed to clang. Once the `Extractor` has completed the graph extraction, the `DataRaceDetector`, a stateless scala object, instantiates and executes the data race analysis. The `Driver` class, or more precisely its `run()` method, coordinates the extraction and analysis.

In order to run the analysis, clients must instantiate the `Driver`, pass it a `ClangConfig` and invoke the `run()` method with list of `java.io.Files` that represent the source files to analyze. The `Driver` then instantiates the necessary components, runs the extraction and analysis, and returns a list of potential data races that were found by the algorithm.

The analysis component is meant to be used as a libary. However, it is also possible to run it as a stand-alone program with a rudimentary command line interface. This interface is implemented in the `CLI` class. When the analysis component is started using the `CLI`'s main method, the command line interface will appear. Note that this interface is not meant for actual users of the analysis tool. Its purpose is to provide developers with a quick way to instantiate and run the data race analysis, without having to load up the entire Eclipse CDT infrastructure.

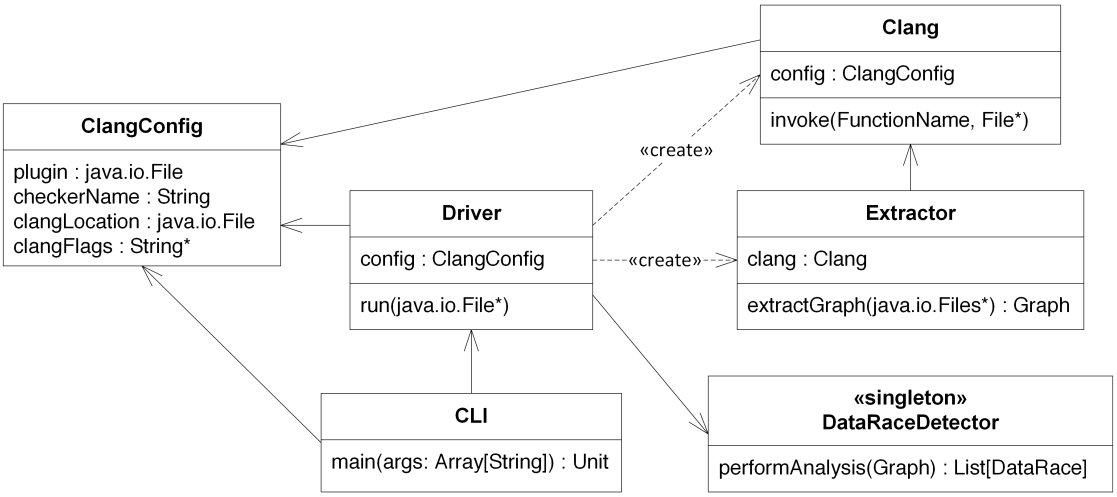


Figure 3.15.: Class diagram showing the driver package contents.

3.3.4. Component: User Interface

The responsibility of the user interface component (UI component) is to manage the entire interaction between the user and the analysis tool. This section describes the aspects relevant to the implementation of UI component. For a documentation of its functionality from the user's perspective, confer Chapter 4, "User Interface".

As previously shown in the system context diagram (cf. Figure3.1), the analysis tool provides three distinct user actions:

- "Start data race analyses": The component provides a UI element that enables users to run data race analyses on specific C++ programs.
- "Change the configuration of the analysis tool": The UI component provides a dedicated menu where users are able to change the following settings:
  - Clang location: Specifies the location of the clang executable to be used for the graph extraction. This allows users a specific version of clang to use.
  - Extraction plug-in location: Specifies the location of the static analyzer plugin that performs the extraction. Necessary, because the plug-in must be compiled separately for each system at the moment.
  - Extraction checker name. The name of the extraction checker that will be registered when the plugin is loaded. The value of this setting will most likely not change in the foreseeable future, but it having the option to change it is better than hard-coding the value.
  - Additional clang flags. Allows the user do pass compiler flags to the Static Analyzer, such as which version of the standard library to use.
- "View the results of the data race analysis": After a data race analysis has been run, the user interface automatically highlights potential data races in the source code using eclipse code markers. In addition to these traditional markers, the UI component provides a more advanced navigation and visualization, which groups together memory locations and conflicting accesses. The aim of this visualization is to show the user potentially conflicting accesses in a more coherent way, as they might be spread out over multiple files, functions, and classes.

The UI component is implemented as a plug-in for Eclipse CDT. As such, it is written entirely in Java. Using the Eclipse infrastructure was advantageous because it provides a sophisticated framework to create new UI elements, menus, and user interactions.

Crash Course: Eclipse Plug-ins, Extension Points and Extensions

This section introduces a few fundamental aspects regarding the Eclipse architecture, which are necessary to better understand the documentation of the UI component.

Eclipse is a highly modular application platform with focus on loose coupling between the individual components. These components, or *plug-ins*, are the building blocks of every Eclipse-based application. The Eclipse Wiki concisely describes the concrete structure that all Eclipse plug-ins must follow:

> A plug-in minimally consists of a bundle manifest file, `MANIFEST.MF`. This manifest provides important details about the plug-in, such as its name, ID, and version number. The manifest may also tell the platform what Java code it supplies and what other plug-ins it requires, if any. Note that everything except the basic plug-in description is optional. A plug-in may provide code, or it may provide only documentation, resource bundles, or other data to be used by other plug-ins. A plug-in also typically provides a plug-in manifest file, `plugin.xml`, that describes how it extends other plug-ins, or what capabilities it exposes to be extended by others (extensions and extension points).
>
> A plug-in that provides Java code may specify in the manifest a concrete subclass of `org.eclipse.core.runtime.Plugin`. This class consists mostly of convenience methods for accessing various platform utilities [...]. [ecl15c]

Every Eclipse component is implemented as a plug-in and follows these principles, including major ones, such as the code editor or the menu bars. The above quote mentions that Eclipse plug-ins can extend each other using a feature called extensions and extension points. Taken again from the Eclipse Wiki:

> When a plug-in wants to allow other plug-ins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract, typically a combination of XML markup and Java interfaces, that extensions must conform to. Plug-ins that want to connect to that extension point must implement that contract in their extension. [...]
>
> Some extensions are entirely declarative; that is, they contribute no code at all. For example, one extension point provides customized key bindings, and another defines custom file annotations, called markers; neither of these extension points requires any code on behalf of the extension.
>
> Another category of extension points is for overriding the default behavior of a component. For example, the Java development tools include a code formatter but also supply an extension point for third-party code formatters to be plugged in. The resources plug-in has an extension point that allows certain plug-ins to replace the implementation of basic file operations, such as moving and deletion.
>
> Yet another category of extension points is used to group related elements in the user interface. For example, extension points for providing views, editors, and wizards to the UI allow the base UI plug-in to group common features, such as putting all import wizards into a single dialog, and to define a consistent way of presenting UI contributions from a wide variety of other plug-ins. [ecl15b]

To give a concrete example, consider a plug-in that wants to add a new page to the Eclipse Preferences menu. Eclipse provides the extension point `org.eclipse.ui.preferencePages` exactly for this purpose. This particular extension point requires extending plug-ins to specify the following information in their plug-in manifest, `plugin.xml`:

- A name for the newly added page.
- A unique id for the plage.
- The name of a class that implements the layout and behavior of the page. The specified class must implement the interface `org.eclipse.ui.IWorkbenchPreferencePage` and must be present in the plug-in classpath.

Thus, a plug-in that wants to add a new preference page with the name *ConditionR*, which is implemented using a class called `ConditionRPreferencePage`, must define the following extension in its `plugin.xml`.

```
<extension point="org.eclipse.ui.preferencePages">
  <page name="ConditionR"
    class="ch.hsr.ifs.conditionr.ui.preferences.ConditionRPreferencePage"
    id="ch.hsr.ifs.extractor.refactoring.ui.preference.page"/>
</extension>
```

This concludes the crash course. Two great sources for additional information regarding Eclipse plug-ins, extensions, and extension points are the Eclipse Plug-in Development FAQ [ecl15a] and the tutorials by Lars Vogel [Vog15].

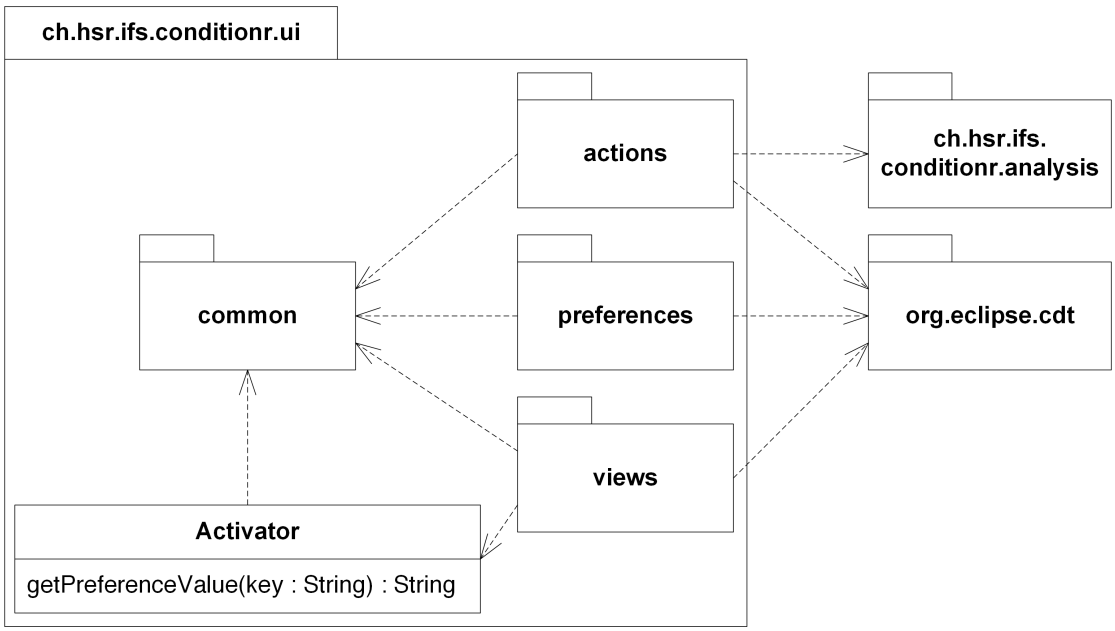


Figure 3.16.: White-box view of the UI components.

Component Overview

A white-box view of the UI component is illustrated in Figure 3.16. The source code is organized in four packages. The following list gives an overview of the package contents.

- commands. This package contains all commands that are added to the Eclipse user interface. Currently, only a single command is implemented.
- preferences. This package contains a preference page where the user can view and modify the previously described settings.
- views. This package contains the classes that implement an experimental and interactive visualization of the algorithm results.
- common. This package contains miscellaneous classes, including the message bundle classes containing the user interface strings, that are used by more than one of the other packages.
- `Activator`. This class which extends the abstract `org.eclipse.core.runtime.Plugin` class. It provides access to the plug-in preference store, a simple key/value map where the plug-in preferences are persistently stored.

The following sections illustrate the package contents. The interactions with Eclipse CDT will only be described superficially, because explaining them in detail would require the documentation to digress extensively. The text will provide references to further information were necessary.

ui.commands

The classes in the `ui.commands` package implement the all user actions that were implemented using the `org.eclipse.ui.commands` extension point. "Start data race analyses" is currently the only action realized this way. If the functionality of the analysis tool is extended in the future, more commands may be added to this package. The `ui.commands` package contents, including any important dependencies, are illustrated in Figure 3.17. The diagram may look confusing at first, due to the many different namespaces present in the illustration, but in reality, the contents are very simple:

- `RunAnalysisCommand` is the point of origin for the analysis command. All this class does, is to instantiate and schedule an `AnalysisJob`.
- `AnalysisJob`. This class is in charge of performing the data race analysis. As such, it is the point of interaction with the analysis component. The `AnalysisJob` runs asynchronously in a separate thread to prevent it from blocking the UI and making Eclipse feel unresponsive.
  After it has been started, the `AnalysisJob` first retrieves the source files of the currently active Eclipse project from the `CoreModel` and loads the current tool configuration from the plug-in store using via the `Activator`. Then, it instantiates the `analysis.Driver`

and executes the analysis. When the analysis is complete, the `AnalysisJob` instantiates and schedules `UIUpdateJob` with the analysis results.

- `UIUpdateJob`. This class displays the analysis results in Eclipse. It creates code markers that highlight source locations of the potential data races the code editor. In addition, it publishes the analysis results on the `common.EventBus`.
- `common.EventBus`. The event bus serves as communication channel between the analysis command, and other parts of the plug-in. In particular, the interactive data race view, which was mentioned earlier, is subscribed to the event bus and updates itself automatically each time new analysis results are published. The data race view is documented later in this section.

The analysis command can be started via the user interface. A dedicated push-button is defined in the plugin.xml using the extension points `org.eclipse.menu` and `org.eclipse.ui.commands`. The corresponding extension definitions are illustrated in Appendix B.2.1 and B.2.2.

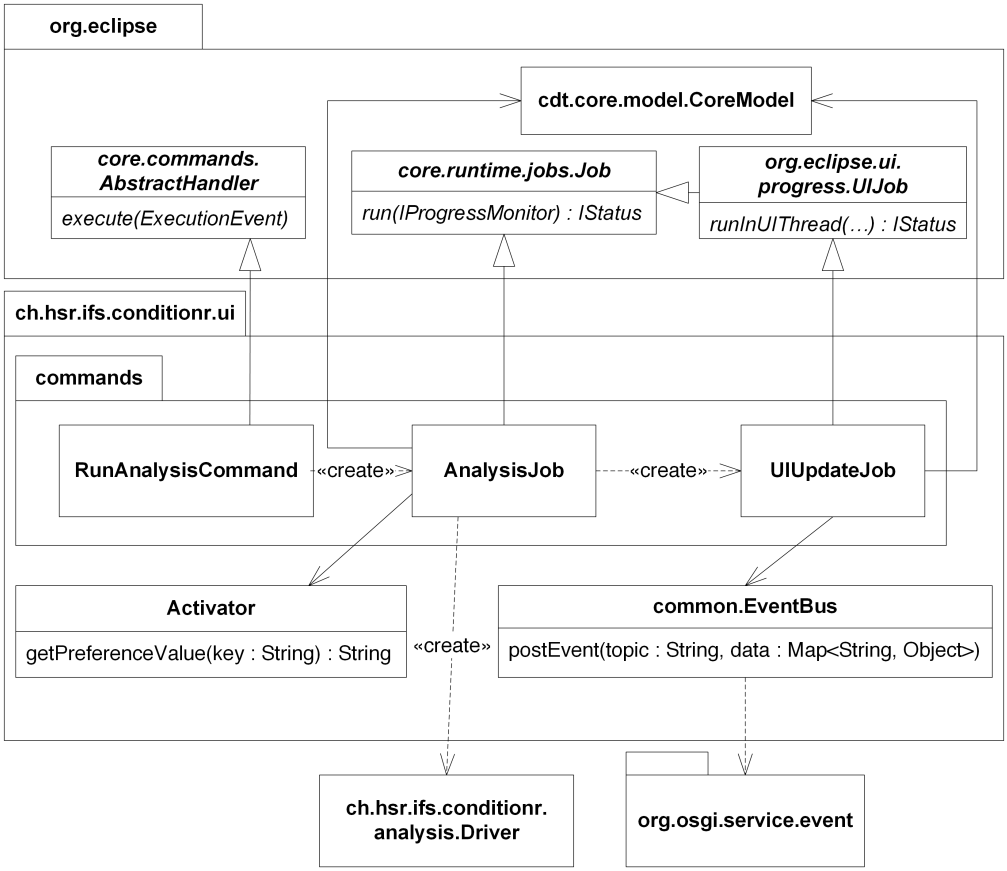


Figure 3.17.: Diagram showing the classes in the `ui.commands` package and their dependencies.

ui.preferences

The `ui.preferences` package contains the classes that implement the preference page where the user can change the configuration of the analysis tool. The package contents, including important associations and dependencies, are illustrated in Figure 3.18:

- `ConditionRPreferencePage` This class defines the layout and the behavior of the preference page. It uses control elements from the `org.eclipse.jface.preference` package, such as `FileFieldEditor` and `StringFieldEditor`, to create the elements of the input mask . These classes are not displayed in the diagram.
  Configuration settings are loaded and stored in the plug-in preference store via the `PluginActivator`.

- `StringListEditor`. A custom `ListEditor` implementation. `ListEditors` are control elements that manage a list of input values. These elements feature a list containing the values, buttons for adding and removing values, and Up and Down buttons to adjust the order of elements in the list.
  Surprisingly, there exists no `ListEditor` implementation that is able to handle arbitrary strings. Thus, a custom implementation was added.

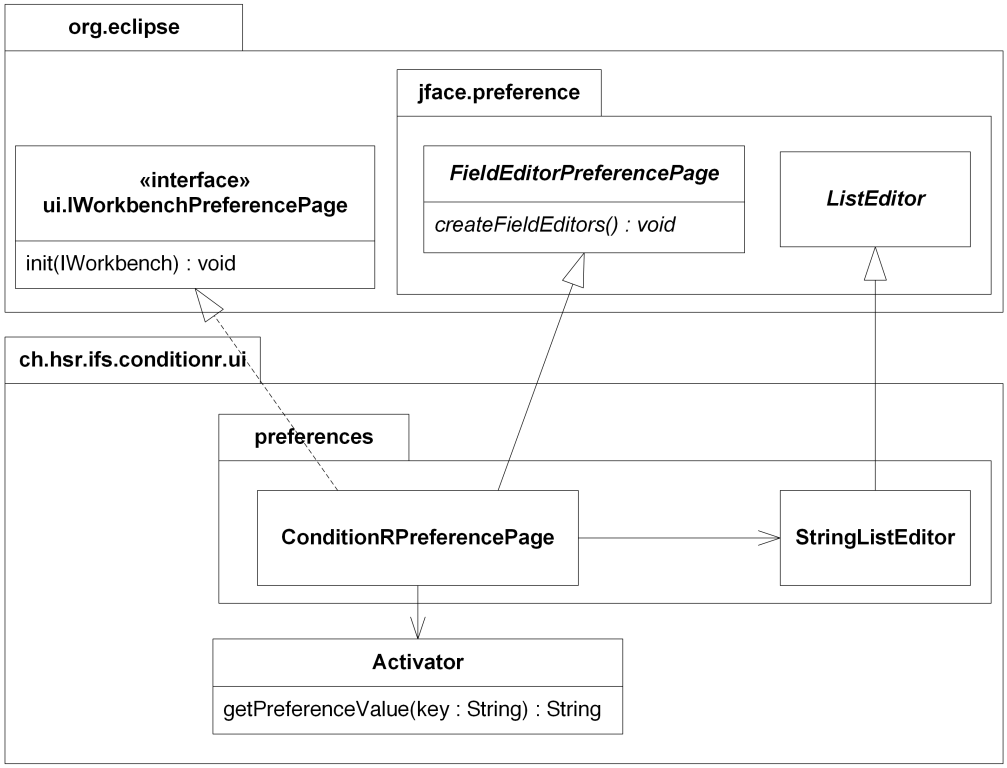


Figure 3.18.: Diagram showing the classes in the `ui.preferences` package and their dependencies.

The preference page is added to the Eclipse UI using the `org.eclipse.ui.preferencePages` extension. The corresponding extension definition `plugin.xml` was used as example in the Section "Crash Course: Eclipse Plug-ins, Extension Points and Extensions"., and it is omitted here.

ui.views

The `ui.views` package contains the classes pertaining to the interactive visualization of the data race view. As already mentioned, the functionality of this visualization from the user's perspective will be described more extensively in Chapter 4, "User Interface". This section is focused on the technical aspects and thus features only a short summary.

The goal of this visualization is to display the potential data races in a context that helps developers to find the sources of the potential conflict. The traditional way of just highlighting the conflicting statements in the code editor is not very well suited to convey this information, because related statements may be spread over several files.

Essentially, the newly developed view groups together data races, memory locations, and conflicting accesses in a node-based layout. The implementation of the visualization itself is based on `org.eclipse.ui.part.ViewPart`, which is the abstract base implementation of all workbench views. As a result, the view can be freely arranged and moved around in the Eclipse UI, just like all other Eclipse views, such as the code editor, the error log, and so on. In addition to the visual representation of the data races, the data race visualization also adds marker resolutions, which enable the user to conveniently jump from the code marker in the source editor directly to the visualization of a certain data race. The communication between the marker resolution and the visualization view happens via the `common.EventBus`.

The contents of the `ui.views` package are illustrated in the class diagram in Figure 3.19:

- `DataRaceView`. The abstract base class for all data race views. This class was factored out during the development, in order to minimize code repetitions between different iterations of the visualization. It's main purpose is to handle the `common.EventBus` communication.
- `SimpleDataRaceView`. The first iteration of the data race view. It was intended as first draft for the visual style of the view. This version was kept in the source tree in order do document the evolution of this component.
- `LocationCenteredDataRaceView`. Improved version of the data race view. This view supports communication with the marker resolutions.
- `DataRaceResolution`. This class gets invoked when the user run the marker resolution on a data race marker. It figures out, to which data race active marker belongs, and publishes this information on the event bus. Listening views then display the potential data race.
- `DataRaceResolutionGenerator`. This class is a factory for new `DataRaceResolution` instances. An implementation of `IMarkerResolutionGenerator` must be provided when extending the `org.eclipse.ui.ide.markerResolution` extension point.

- `DataRaceResolution`. When the data race resolution is invoked by the user, it sends a message over the `EventBus`, telling the data race `LocationCenteredDataRaceView`, which data race to display.

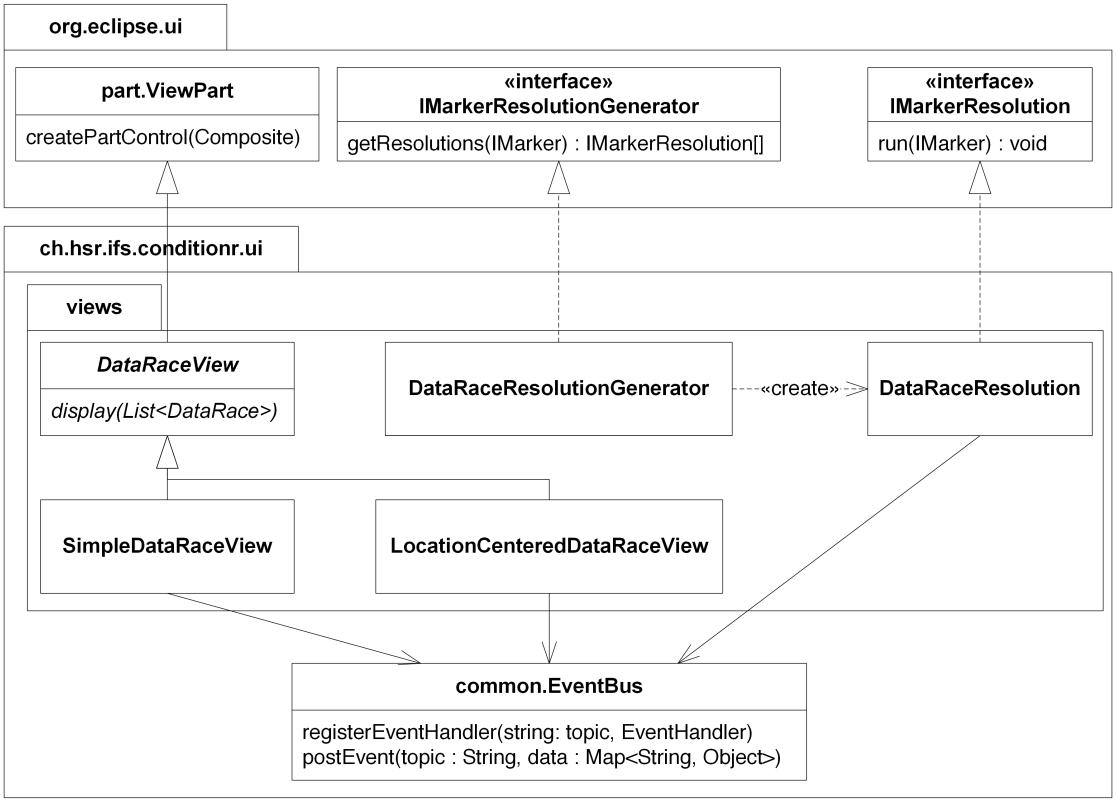


Figure 3.19.: Class diagram showing the contents of the `ui.views` package, including important dependencies.

The visualization is added to Eclipse by extending the `org.eclipse.ui.views` extension point. The complete extension definition for the most recent iteration of the view can be found in Appendix B.2.3. The marker resolution is added to Eclipse by extending the `org.eclipse.ui.ide.markerResolution` extension point. The corresponding extension definition can be found in Appendix B.2.4.

ui.common

The `ui.common` package contains miscellaneous classes that are used by more than one of the other packages. Its most important contents are the message bundle class which contains the user interface strings, and the `EventBus`, which serves as communication channel between many UI subcomponents. The package contents are illustrated in Figure 3.20:

- `Messages`. This class provides all UI strings. The strings are stored in a file called `messages.properties` and are automatically loaded when the plug-in is instantiated.
- `PluginConfig`. This static class exposes various information which is used throughout the UI component, such as labels, config-ids and default values for the analysis tool configuration. This information comes mostly from `Messages`, but some values are hard-coded into the class.
- `EventBus`. The communication channel between UI subcomponents. Components can subscribe for a topic by registering an providing an `EventHandler` instance. Whenever a message under this topic is posted, all registered handlers are notified. The `EventBus` is implemented using the Eclipse event system.
- `ConditionrException`. This class is used to throw exceptions from within the UI component.

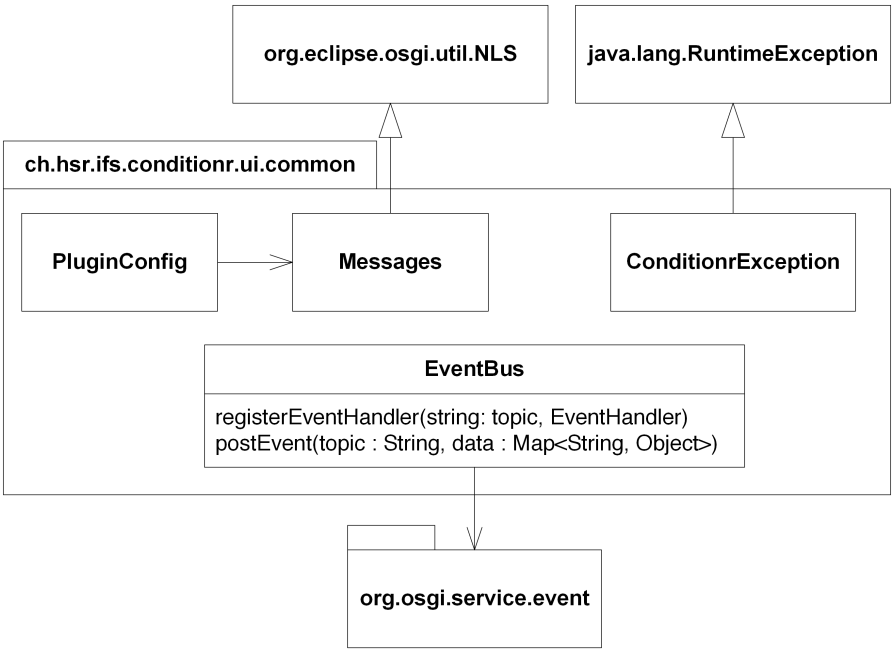


Figure 3.20.: Class diagram showing the contents of the `ui.common` package, including important dependencies.

The description of the UI component's implementation is complete. As mentioned before, any aspects pertaining to the operation concept and usage of the user interface from the user's perspective will be presented in Chapter 4, "User Interface".

3.3.5. Summary

This section has documented the static structure the components and subcomponents of the analysis tool. Where necessary, the interfaces and functionality of adjacent systems, such as the Clang Static Analyzer, and Eclipse CDT, have been explained as well. For more in-depth information regarding the static structure of the analysis tool, please consult the source code.

The focus of the next section is the description of the behavior and interaction of the system's building blocks as runtime elements, which should answer many questions that may have come up during the lecture of the current section.

3.4. Runtime View

The runtime view describes the behavior and interaction of the system components as runtime elements by the example of so-called runtime scenarios. These scenarios are specific use cases that bear architectural relevance and thus lend themselves to show the dynamic interaction of components and interfaces.

3.4.1. Scenario: Running the Analysis from Eclipse CDT

The most important feature of the analysis tool is the eponymous data race analysis, which can be initiated from the Eclipse UI, and whose results can be viewed in the C++ source code editor. The entire analysis tool was built around this feature. Thus, it will be used as primary runtime scenario to illustrate the dynamic interaction of the system components.

The scenario will be illustrated from the perspective of each component: user interface, analysis, and extraction. Each perspective will show the component's internal behavior, as well interactions with the interfaces to the adjacent components.

Overview

This section gives an overview of the entire scenario, The sequence diagram in Figure 3.21 illustrates the analysis process with focus on the interaction between the tools' individual components and neighbor systems. The diagram is purposefully informal. Its intention is to provide a basic understanding of the process without focusing to much on technical details.

Many parts of the diagram are straightforward and do not need much further explanation; after the user has initiated the analysis process, the components perform the actions described in the previous chapters in a straightforward fashion, until the results are displayed as code markers and in the advanced data race view. The most interesting aspect is the interaction between the analysis component, the Clang Static Analyzer, and the extraction component. The diagram shows, that the Clang Static Analyzer is actually the intermediary between the communication of these two system components. This is due to the fact that the analysis component invokes and parameterizes the analyzer, but the analyzer loads, instantiates, and executes the extraction plug-in. In fact, the analyzer and the extraction plug-in, being C++ programs, are running in an entirely separate execution environment, and they communicate with the analysis component using the standard input and output streams.

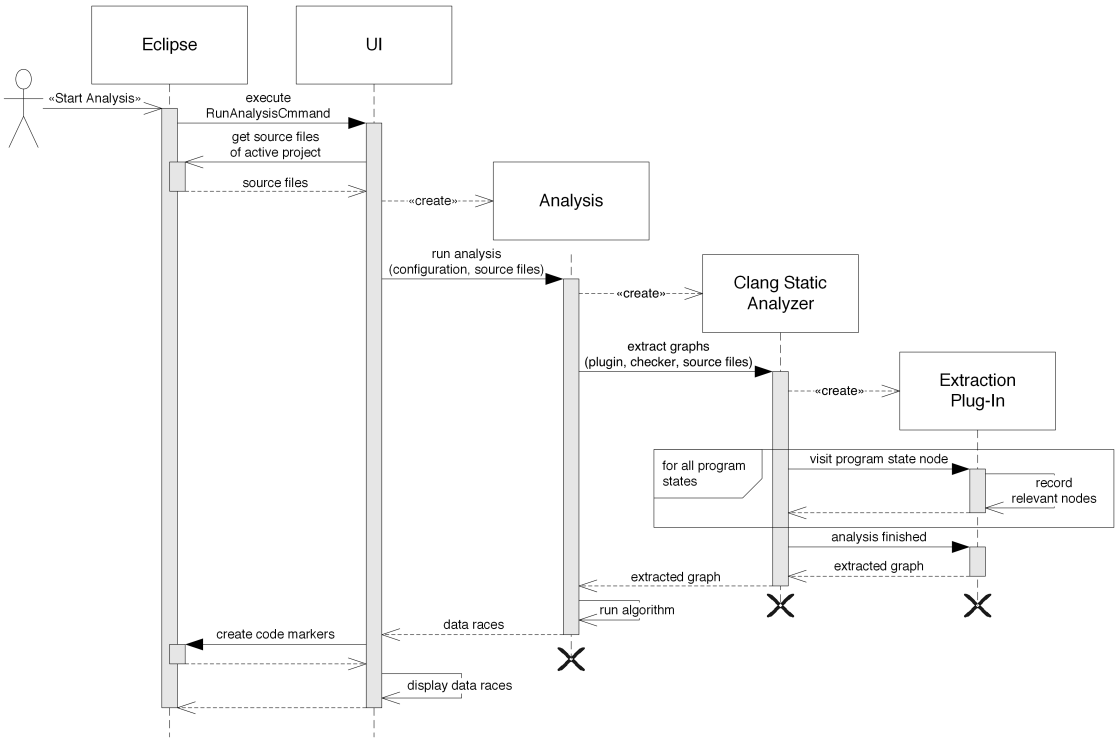


Figure 3.21.: Sequence diagram giving a high-level overview of the runtime scenario. It is purposefully informal, in order to provide a basic understanding of the process without focusing to much on technical details.

UI Component Perspective

The sequence diagram illustrated in Figure 3.22 shows the scenario from the perspective of the UI component. Its goal is to expose where and in what order the UI subcomponents are active. The diagram also shows the interaction with the analysis component: The `AnalysisJob` loads the configuration preferences from the preference store via the `Activator`, and uses them to create a `Driver` instance [6]. After that, it runs the `Driver` with the files of the currently active project, and waits for it to return the results. Once the driver has completed the analysis, the `AnalysisJob` creates an schedules a `UIUpdateJob` which is in charge of displaying the analysis results.

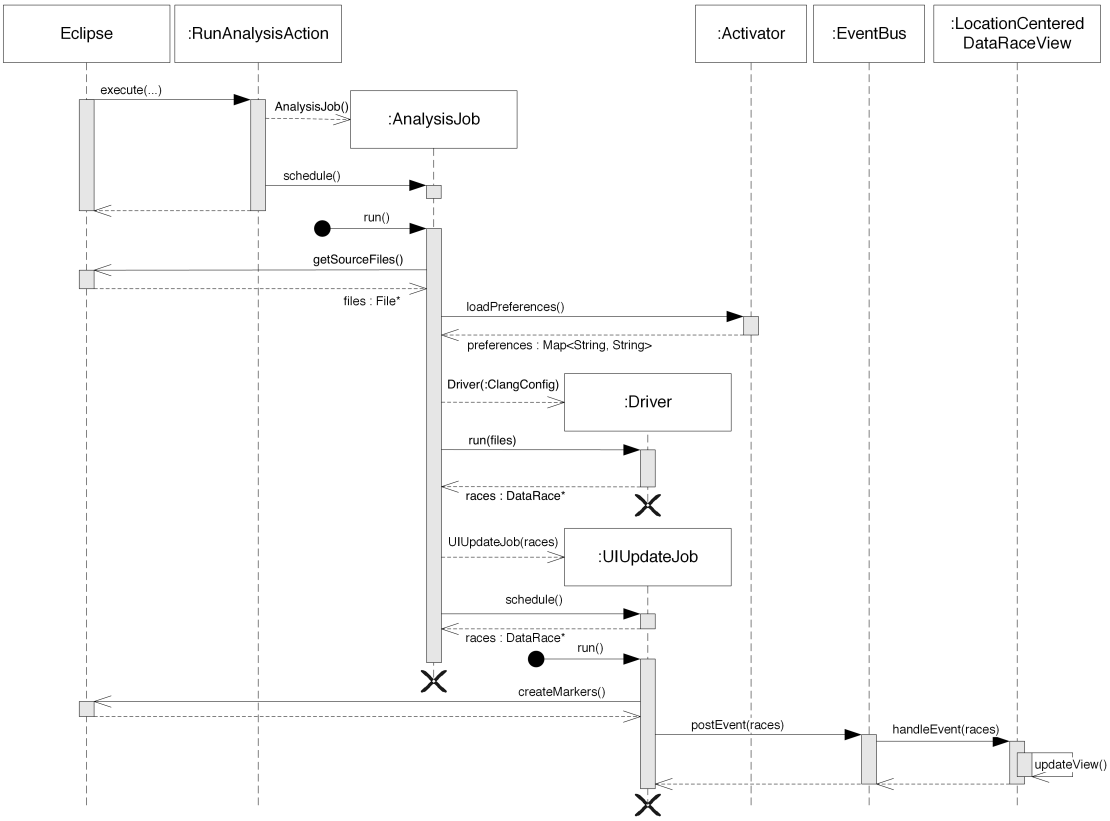


Figure 3.22.: Sequence diagram showing the scenario from the perspective of the UI component.

Note that some operations in the illustrations are simplified versions of the real implementation: the methods `getSourceFiles()` and `createMarkers()`, which interact with the Eclipse framework to retrieve the source files of the currently active project and to create code markers, stand representative of very complex call sequences. Accordingly, the method `loadPreferences()`, which loads the configuration settings from the Activator, condenses together a whole array of function calls. These simplifications are appropriate, because they bear no architectural relevance, and help making the illustration as a whole more concise.

[6] The creation of the `ClangConfig` from the plug-in preferences is omitted in the illustration due to lack of space.

Analysis Component Perspective

The runtime scenario from the perspective of the analysis component is illustrated in Figure 3.23. Keep in mind that the aim of this illustration is to show the interaction between the different parts of the analysis component, an not to explain how the data race algorithm works. Details pertaining to the algorithm execution are omitted to keep the focus of the illustration on the collaboration of the subcomponents.

This sequence diagram exposes a property that was not previously mentioned in the report; the Clang Static Analyzer and the extraction plug-in are actually invoked multiple times during the graph extraction. This works as follows: First, the execution paths of the program's main function is extracted. If any thread entries or unresolved function calls are detected, their execution paths are successively extracted as well, until no more unextracted functions are detected. Once the extraction of these thread entries and unresolved functions is complete, their execution paths are used to create a `Graph` instance and return it to the `Driver`.

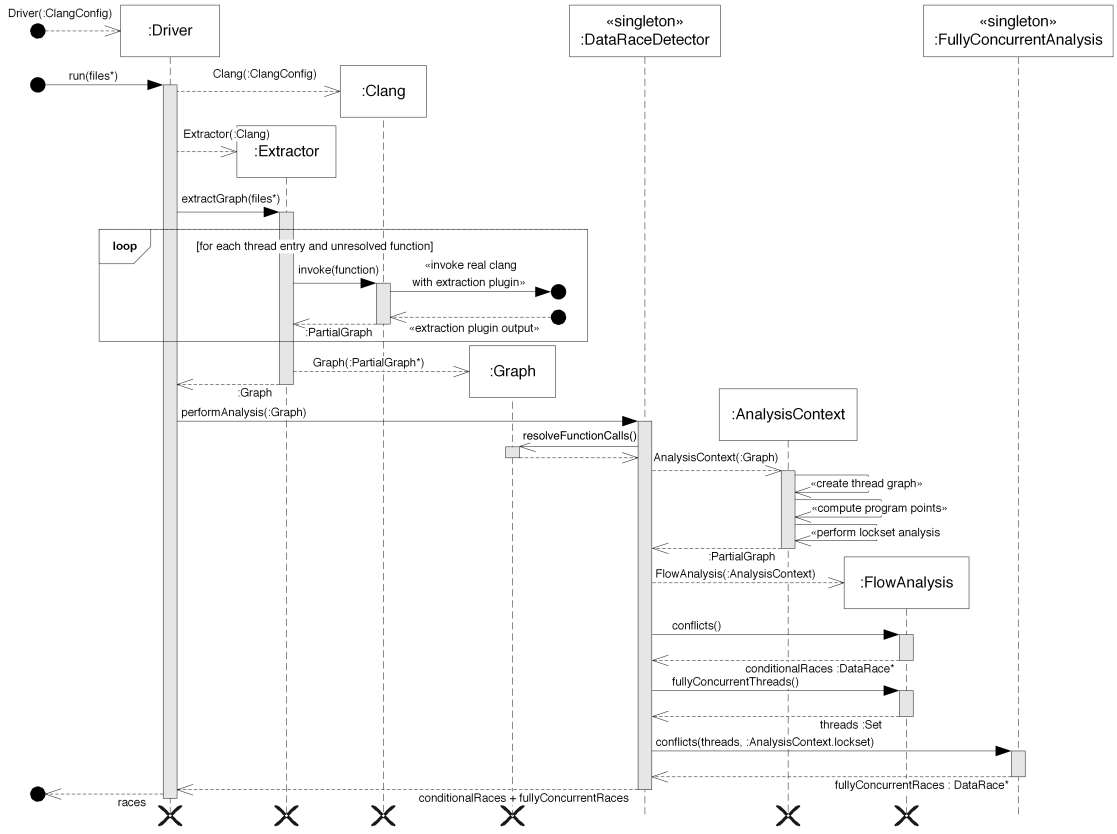


Figure 3.23.: Sequence diagram showing the screnario from the perspective of the analysis component.

Extraction Component Perspective

The important dynamic aspects of the extraction component have already been documented in Section 3.3.2 of the building block view and in the overview of the runtime scenario. Because Clang Static Analyzer plug-ins are implemented as traditional visitors to a tree-like data structure, there is not much information with architectural relevance, which has not been previously described and would be worth illustrating here.

Summary

This concludes the documentation of the system's behavior as dynamic runtime elements. This section has described the important dynamic aspects of the system components, and shown their internal and external interaction, at the example of a specific runtime scenario. Additional information pertaining to the dynamic structure can be obtained by consulting the source code and test cases.

3.5. Deployment View

The deployment view illustrates the distribution of the system into artifacts and execution environments. It also documents communication channels, environment constraints, and other elements of the physical system environment.

The diagram in Figure 3.24 shows the deployment of the analysis tool. Although all components run on the same machine, there is a number of interesting aspects to be pointed out.

- The analysis tool can be run on computers running on recent versions of Linux or OS X.
- The analysis component and the user interface run inside an Eclipse CDT execution environment. Both components are manifested as JAR-files.
- The extraction component runs in a different execution environment, as part of a Clang Static Analyzer instance. It is manifested as different versions of a dynamic library, depending on the device operating system.
- The components interface across the execution environments through standard input and output streams.

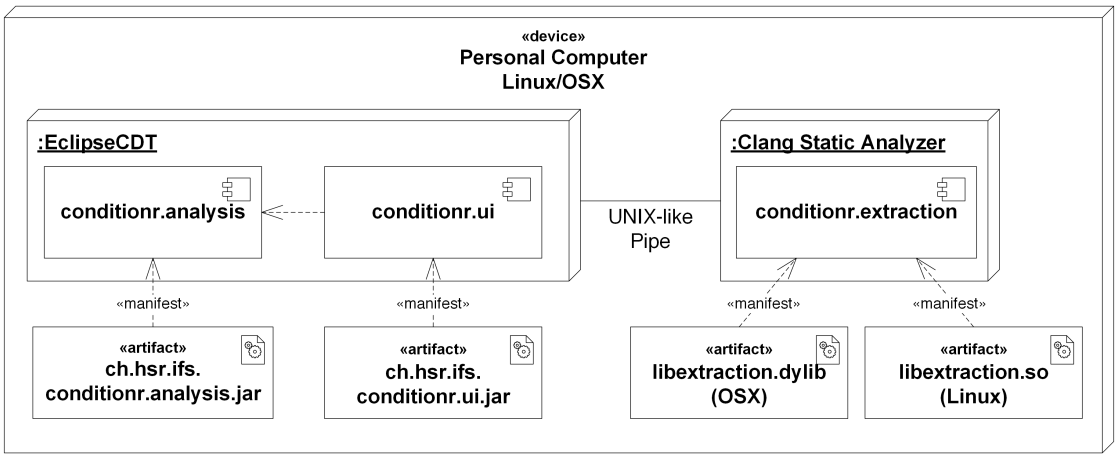


Figure 3.24.: Deployment diagram showing distribution system artifacts, communication channels, and other elements of the physical system environment.

Currently, machines running windows are not supported as an execution device. This constraint is imposed by the extraction component, which is written in C++ and must be compiled individually for each execution platform. The development of the analysis tool happened exclusively on Linux and Mac OS. No time was invested into attempting to port its the C++ portion to other platforms. However, getting the tool to run on Windows should not be tremendously challenging, because Clang generally is compatible with that platform. This concludes the documentation of the physical system environment.

3.6. Design Decisions

This section covers important decisions made during the project that influenced the design and architecture of the resulting program.

3.6.1. Choosing the Clang Static Analyzer

Extracting the information required for the data race analysis from the source code of a program requires a lot of infrastructure:

- A lexer and parser are needed to generate the abstract syntax tree of the program, which is necessary to understand and reason about the program's syntactic structure.
- More advanced flow- and context sensitive analyses usually rely on flow-graphs. Delegating the construction of these graphs to a dedicated tool is generally preferred to constructing them by hand, as it is very challenging and error-prone.
- To make the analysis more precise, many other analysis features are required, such as pointer- and name aliasing, symbolic execution, and so on.

Building the entire infrastructure by hand would take too much time, and would add huge risks to the project [7]. The only feasible option was to build the data race analysis on top of a preexisting static analysis framework.

At the beginning of the project, a decision had to be made regarding what analysis framework to use. This decision has a significant impact on the project result, because it influences the quality of the data race analysis, the development speed, and the system architecture.

Considered Alternatives

Two analysis frameworks were eligible for consideration:

1. CODAN. A static analysis framework based on Eclipse CDT's AST, written in Java. CODAN is the go-to tool for most static analysis projects conducted by the IFS. The author has practical experience using this analysis tool.
2. Clang Static Analyzer. The static analysis tool that ships with the Clang compiler infrastructure. For more information on the static analyzer, confer Sections 3.2.1 and 3.3.2.

The alternatives were judged by the following Factors: Their accuracy in parsing C++, the analysis infrastructure they provide on top of the parser, and the authors preexisting experience with the tools.

[7] In fact, a previous master thesis conducted at IFS dedicated its entire time to implementing a lexer and parser for C++.

Decision

When the evaluation started, there was a positive bias towards CODAN because of the experience with this tool at the Institute for Software. One of CODAN's defining features is its integration with Eclipse CDT, which allows it to be used together with the CDT refactoring infrastructure. This is a powerful combination, as it allows the creation of sophisticated automatic and semi-automatic bug-finding and refactoring tools. Using the Eclipse UI features, these tools can also be made very user-friendly.

However, this project does not require any code restructuring, since it is focused purely on analysis. The strongest argument for CODAN is therefore void. After the evaluation, the choice was made in favor of the Clang Static Analyzer for the following reasons:

1. Accuracy. The Clang Static Analyzer uses the Clang compiler front-end to parse source code, which yields a more accurate AST than CODANs Eclipse-based parser.

2. Analysis Infrastructure. On top of the more accurate AST, the Clang Static Analyzer provide a vastly superior static analysis framework, that includes context- and path-sensitive analysis facilities. CODAN on the other hand, provides mainly syntactic analysis infrastructure. The existing flow-sensitive capabilities would have to be significantly extended by hand, which was rated as to risky and time consuming.

3. Debugging Tools. The Clang Static Analyzer also provides debugging tools that allow developers to relatively comfortably inspect flow-graphs, execution paths, and the generated AST.

The major consequence of this decision was that the author had to commit time to learning how to work with the Clang infrastructure, which negatively influenced the time spent working on the actual project goals.

3.6.2. Separation of Extraction and Analysis

After the decision was made to build the analysis tool on top of the Clang Static Analyzer, the question arose as to whether the algorithm should also be implemented in C++, or using a different language. This question came up for two reasons:

- Programming Language Skills. The author is not as proficient in C++ as he is in other programming languages, namely Java, and to a lesser extent, Scala. Committing to C++ in this situation bears the risk of losing a lot of time to trivial problems that could otherwise be avoided.

- Eclipse Integration. One of the project goals is to provide a visualization of the analysis results in Eclipse CDT. Therefore, parts of the application would have to be written in Java or a compatible language regardless.

A decision was due pertaining to what language to use for the implementation of the algorithm, and potentially the user interface. This decision has a huge impact on the system architecture, as it essentially dictates the structuring of the subcomponents.

Considered Alternatives

The following alternatives where considered:

1. Implement the entire analysis tool in C++ as a standalone Clang application. In addition, provide an Eclipse Error parser that is able to parse the tool output and display it in the Code editor.

2. Extract the information relevant for the analysis from the source code using the Clang Static Analyzer, but implement the analysis using either Java or Scala. Using a JVM language facilitates the Eclipse integration.

These two alternatives were judged by the following factors: the risk they bare, the authors proficiency in the respective technology, their utility in solving the problem at hand, and their overall impact on the architecture.

Decision

The decision was made in favor of the second alternative: the algorithm will be implemented in Scala. The following reasons led to this decision:

1. Scala is well-suited for the implementation of the algorithm. It is a flexible, modular and modern programming language that combines functional and object-oriented semantics, provides a great collection library and a sophisticated type system. It is also Java-compatible. Scala programs can be integrated easily into Eclipse.

2. Using a second language encourages the decoupling of the extraction and analysis components. Naturally, it is not necessary to use different languages to decouple two components, but doing so enforces a strict separation.

3. Having the extraction separated from the analysis mitigates the risk of committing to the Clang infrastructure. If necessary, the entire extraction component can be swapped out, or even be synthesized and implemented later.

The major consequence of this decision is an increase of the overall system complexity, and its development process. If the analysis tool would have been developed as a monolithic C++ application, its testing, building process, deployment, and documentation would have been much simpler.

3.7. Testing Concept

This section describes the methods that were used during this project to test the behavior of the created software components.

Unit Testing

The software components are tested using automated unit tests. Alongside the development of the production code, a test suite for each component was authored and continuously expanded. Whenever the functionality of the software was increased or a bug was fixed, a set of tests covering the new behavior was added to the suites. Most of the time, the tests were written prior to the corresponding production code, in a manner similar to Test First (XP) or TDD.

Testing the Extraction Component

The extraction component is tested using CUTE, a unit testing framework for C++, which is developed and maintained by IFS. All unit tests for the extraction component are black-box tests. They simply run the extraction plug-in over a set of input files and verify the output. They do not examine the internals of the component. Moreover, The test suite for the extraction component solely uses real C++ source files as test input. The data for each test case consists of one or more C++ source files, which are used as input, and a corresponding JSON file representing the expected graph in JSON format. The test suite contains facilities to run the Clang Static Analyzer over a set of files and to parse and verify the JSON formatted output.

The internals of the extraction component are not tested more rigorously, because the production code had to stay agile and open for quick changes. Having a set of behavior specifications independent of the component internals was ideal, and allowed for quick iterations on the source code, sometimes with drastic changes, without having to constantly maintain and reconcile the test cases.

Testing the Analysis Component

The analysis component is tested using the ScalaTest framework in FlatSpec style. The tests follow a more traditional pattern. Individual subcomponents, such as the implementations of the different algorithm steps, or the JSON parser, have their own test suites, which test their externally visible behavior.

A small mocking infrastructure was built to test the interface to the Clang Static Analyzer. This mock intercepts the system calls to Clang made by the driver of the analysis component, verifies their correctness, and returns precomputed JSON strings back to the component.

Testing the UI component

The UI component is tested using the JUnit framework. However, because the UI layer is very thin, and most of the code is heavily intertwined with the eclipse infrastructure, this component is not tested as extensively as the others. The effort of writing good tests for the UI behavior huge, and its benefit is considered uncertain at best.

3.8. Summary

This chapter has descried the architecture of the developed software components, and illustrated their the static and a dynamic aspects. It has also highlighted major design decisions made during the development, and documented the testing concept.

4. User Interface

This chapter documents the functionality of the analysis tool from the user's perspective. It describes how to initiate data race analyses, how to evaluate the results, and to change the configuration of the tool. All described functionality requires all components and dependencies of the analysis tool to be properly installed and configured. This chapter also serves as a user guide.

4.1. Running Analyses

The UI component adds a new button to the Eclipse user interface (cf. Figure 4.1), which can be used to initiate a data race analyses. Upon clicking this button, the analysis tool starts an analysis over the project, to which the currently active source file belongs to. If there is no open file when the button is pressed, nothing happens.

The data race analysis runs in an asynchronous background task, in order to keep the Eclipse UI responsive. While it is running, the current progress of the analysis task can be observed in the Eclipse Progress Monitor (cf. Figure 4.2).

4.2. Evaluating Analysis Results

Once the analysis is complete, the UI generates code markers to highlight the source locations of statements that potentially cause data races. These markers are visible in the code editor and in the problem view.

Figure 4.3 shows the Eclipse UI after the data race analysis has been completed. For the program illustrated in this particular example, a total of three potential data races has been detected. Each pair of code markers that constitutes a data race has been associated by color. In addition, the conflicting statements, which potentially cause potential data races have been connected using edges of the same color as the corresponding markers.

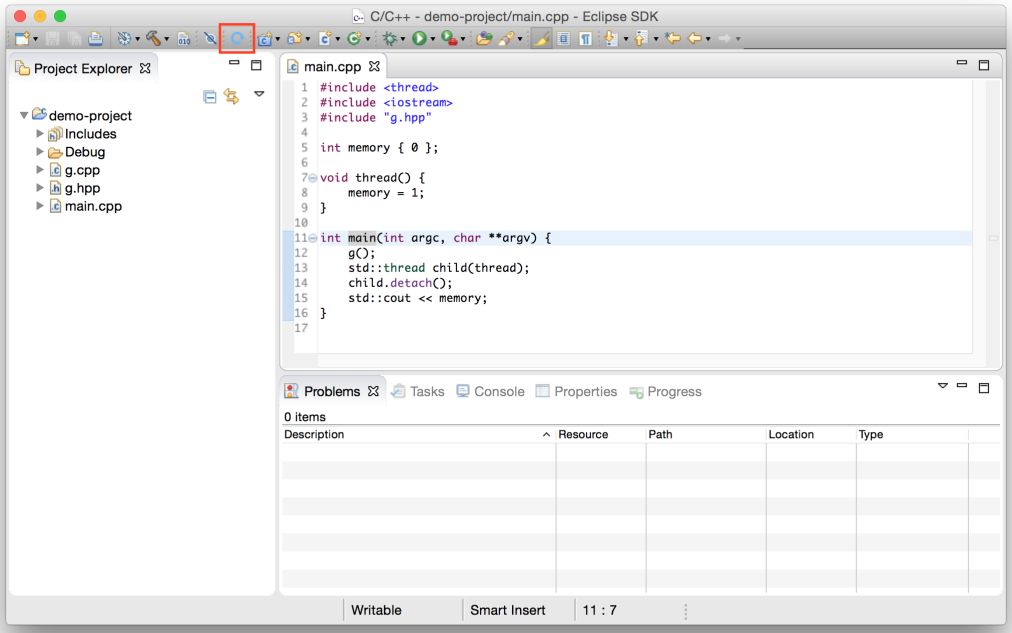


Figure 4.1.: Data race analyses can be started using the highlighted button.

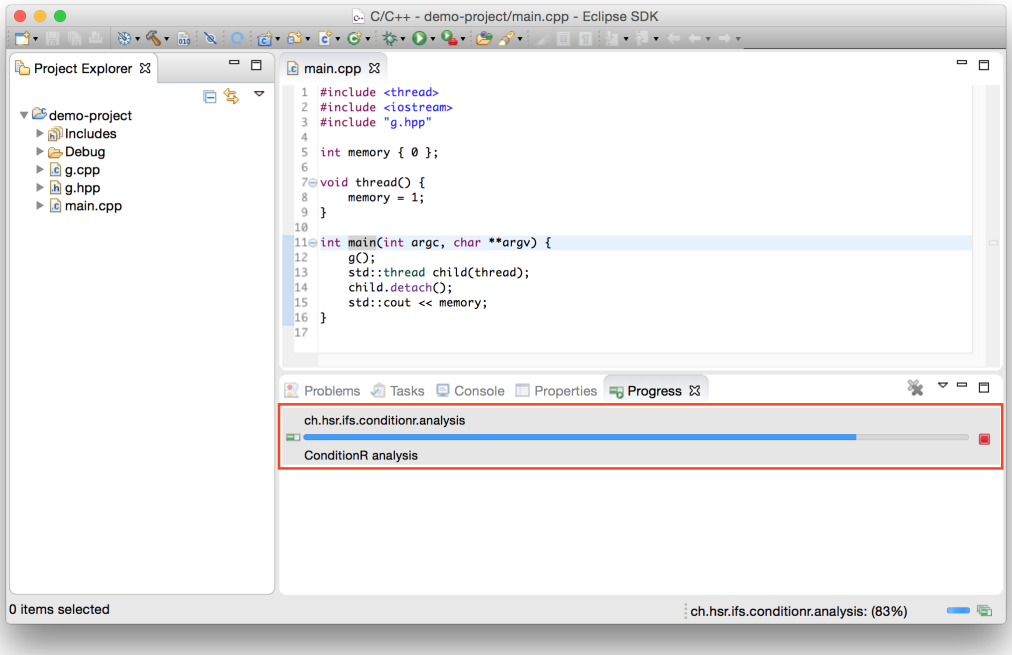


Figure 4.2.: The current progress of the analysis can be observed in the Progress Monitor.

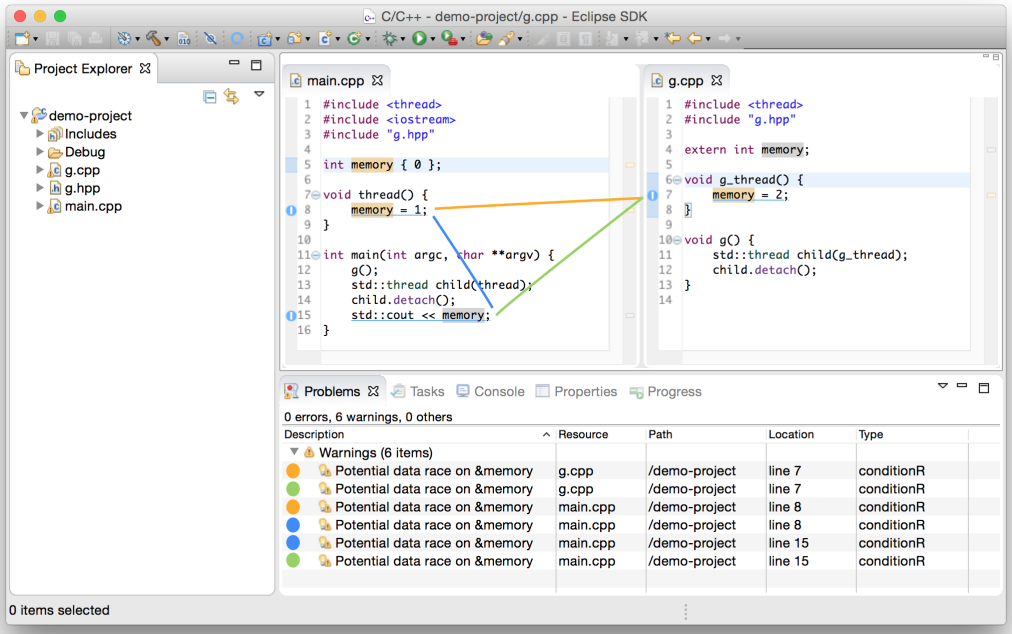


Figure 4.3.: Potential data races are highlighted using code markers, which are visible in the source editor and the problems view. Visual aids have been added to better understand the presented information.

Advanced Data Race Visualization

Without the aforementioned additional visual aids, extracting any meaningful information from these code markers would be difficult even in tiny programs, such as the one used in the illustration. Data races are a product of multiple unsynchronized memory operations, which are usually situated in distinct source locations, possibly even files. Simply putting markers on these locations is not enough to effectively visualize data races. The user must be able to see the connections between individual memory operations in order to understand why they potentially cause data races.

For this reason, the analysis tool provides an dedicated view that shows data races and their conflicting memory operations from a different perspective. In this view, the analysis results are organized in a tree-like structure with multiple tiers. While reading the following list of the descriptions of these tiers, it is recommended to coincidently look at Figure 4.4, which features illustrations corresponding to the written descriptions[1].

- Tier 1: Problem Category. The top tier of the visualization allows the user to select the category of problems to descent into. Currently, there exists only one category called "Races", but this might be extended in the future, for instance with "Deadlocks".

- Tier 2: Memory Locations. Once the problem category has been selected, the next tier becomes visible. This tier shows all memory locations that exhibit problems of the selected category, depicted as green dots. The user needs to choose which memory location to inspect.

- Tier 3: Data races. After selecting a memory operation, the potential data races associated with this location appear as orange dots, in a circular arrangement around the memory location. All other memory locations fade into the background, but can still be clicked needed. Again, the user needs to choose which data race he wants to inspect.

- Tier 4: Memory operations. Once the data race to inspect has been selected, the memory operations associated with this race become visible, while all other, not selected data races, fade into the background. Read operations are depicted as blue dots with an "R" inside, write operations as red dots with a "W" inside.

  Hovering over a memory operation opens the corresponding source location in the Eclipse code editor.

  This hover functionality, in combination with the grouping of the memory locations by data races, is the major advantage of this visualization over traditional code markers. It allows developers to quickly find conflicting memory operations and their source locations.

While the data race view on its own is well suited to browse through the analysis results, finding a specific data race and its associated memory operations can be difficult, because there is no search function, or any similar feature.

[1] The data race view and the code editor have been put in free-floating mode to preserve space in the illustration. Naturally, the functionality stays identical when they are embedded in the main Eclipse window.

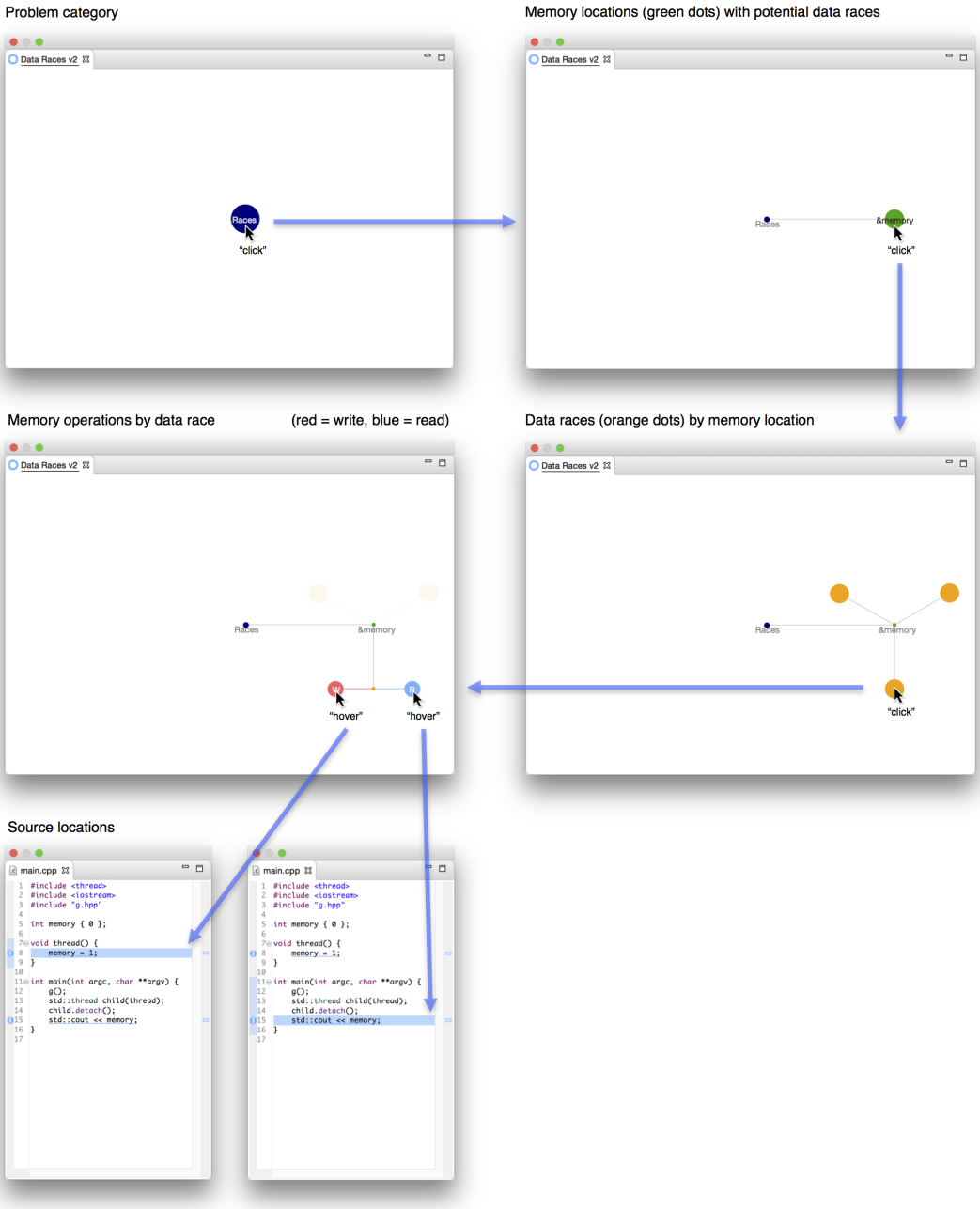


Figure 4.4.: Advanced data race view. The analysis results are organized in a tree-like structure with multiple tiers. The input program used for the screen shots is the same as the one in the previous examples.

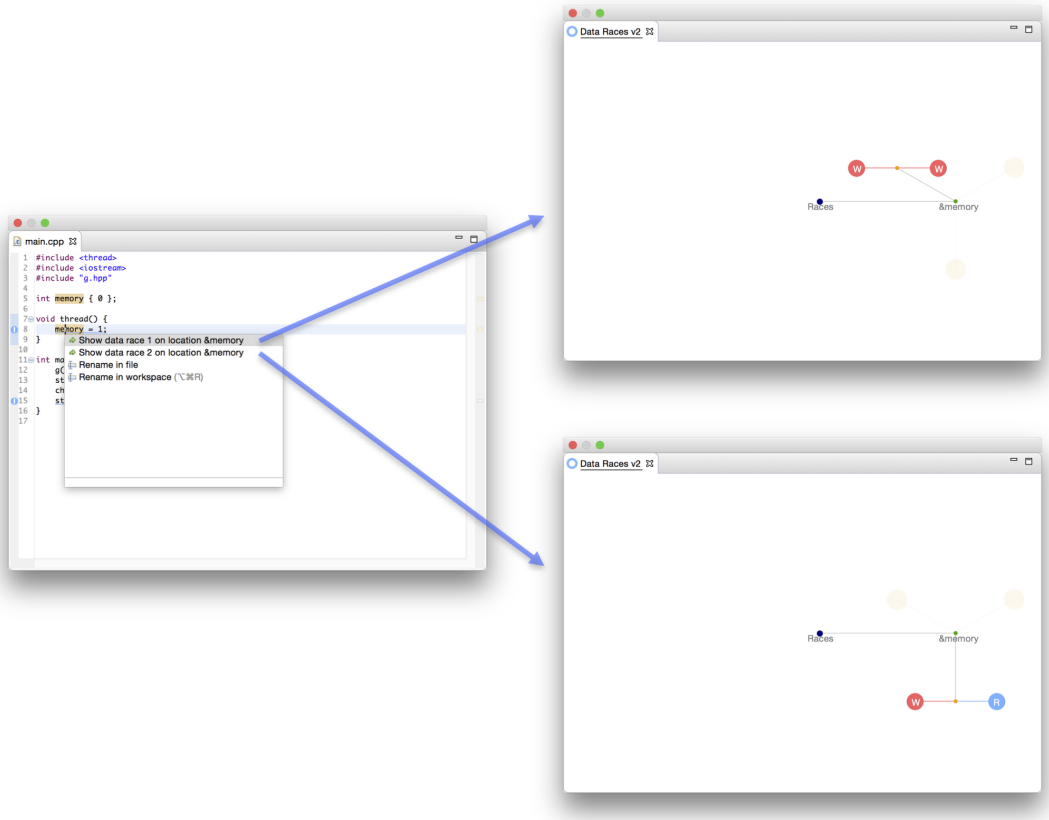


Figure 4.5.: Data race marker resolutions allow the user to jump directly from a marked source location to the corresponding data race in the advanced view.

Data Race Marker Resolution

To ameliorate the issue of finding specific data races for inspection, a marker resolution was implemented, which allows the user to jump directly from a marked source location to the corresponding data race in the advanced view. The user can activate the resolution by moving the cursor over a marked statement and pressing [CTRL] + [1]. In the subsequent drop-down menu, the user must select which data race to inspect, and confirm by pressing [Enter].

Once opened, the view can be used to analyze the data race exactly described in the previous section. Figure 4.5 shows how these marker resolutions look, and what happens when they are executed.

4.3. Configuring the Analysis Tool

For the analysis tool to be able to run, a set of preferences must be configured. The values of these preferences is heavily depended on the particular system environment, where the analysis tool is executed. The default values will most likely have to be changed on a per-installation basis. The technical aspects and implications of each setting have already been discussed in Section 3.3.4, "Component: User Interface".

To manage these configuration preferences, the analysis tool provides an Eclipse preference page with the name "ConditionR", as illustrated in Figure 4.6.

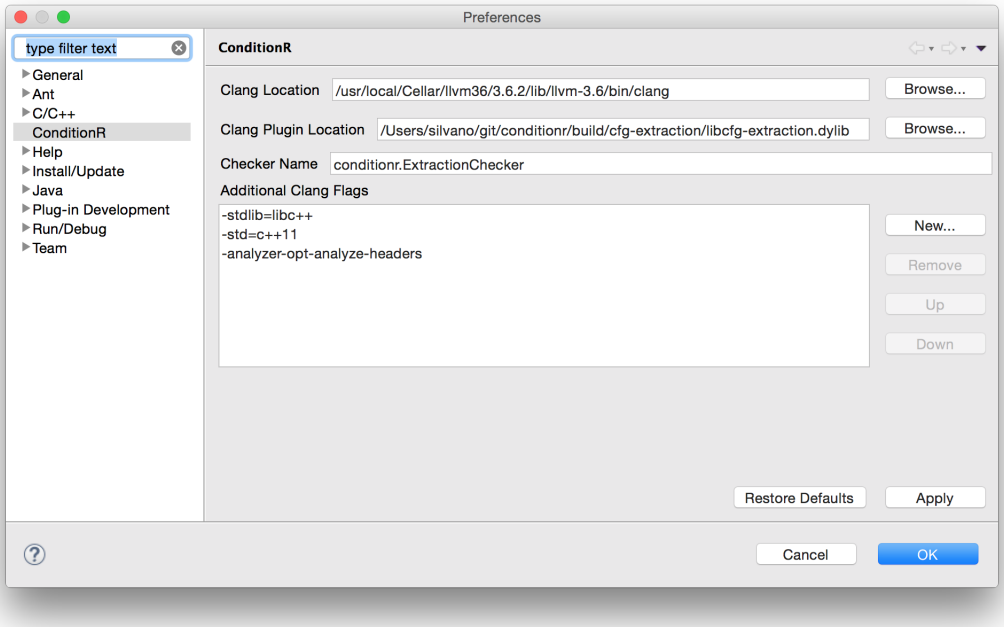


Figure 4.6.: Preference page where the analysis tool settings can be changed.

5. Analysis Features

The previous chapters have described the workings of detection algorithm and discussed the implementation details of the created software components. This chapter takes a look at the analysis tool in terms of the provided functionality. First, it documents the currently supported concurrency constructs. After that, it enumerates other C++ specific features that were added on top of the base version of the algorithm. The chapter concludes with a recital of analysis tool's current limitations.

5.1. Supported concurrency constructs

The analysis tool currently only supports the basic concurrency and synchronization features of C++11, `std::thread` and `std::mutex`, including its helpers and wrappers, such as `std::lock_guard`. Examples of programs using these constructs are scattered throughout this document.

Unfortunately, more advanced thread synchronization constructs, for instance `std::async`, `std::packaged_task`, or `std::future` are not supported. However, providing support for most of the currently unsupported constructs does not require fundamental changes to the system. Given the fact that the algorithm has no notion of any other concurrency concepts, adding support for more advanced constructs becomes a matter of modeling them in terms of threads, locks, and memory operations. For instance, adding support for `std::async` or `std::packaged_task` would only require to extend of the extraction component, so that it can correctly translate calls to these constructs into logical "thread start" and "thread join" statements, which can be understood by the algorithm. The other components would not require any changes.

5.2. C++ specific features

The most important C++ specific feature that was added is the support for cross-translation unit analysis. The functionality and limitations of this feature have already been discussed in Section 3.3.3.

5.3. Limitations

This section summarizes the current limitations of the analysis tool:

- Lambdas. The analysis tool is unable to correctly analyze thread entires that were defined using lambda functions. This limitation is caused by the Clang Static Analyzer, which lacks support for lambda functions.
- Function Pointers. Clang Static Analyzer, and in turn the analysis tool, do currently not support function pointer aliasing, and thus are unable to analyze thread entries that were defined with function pointers. For instance, the analysis tool will not find data races in the program in Listing 5.1, because it is not able to determine which function is executed by thread `t`.

  However, rudimentary pointer aliasing is supported. For example, the analysis is tool is able to correctly analyze and report potential data races for the program in Listing 5.2.
- Aliasing across translation units: Even though pointer and reference aliasing works inside translation units, it does currently not for manually resolved, cross-translation unit function calls. Keep in mind that aliasing problems were defined as being outside the scope of this project.
- Thread-local memory locations. Given two concurrent thread entries that execute the same function, the analysis tool will falsely report potential data races on thread-local variables, if they are not declared const. This is currently the most annoying limitation, because it causes the analysis tool to report many false positives.

To summarize, most of the current limitations are imposed by the Clang Static Analyzer. In case the analyzer functionality is improved in the future, the analysis tool will automatically benefit from these improvements as well.

Listing 5.1: The analysis tool does currently not support function pointer aliasing, and thus is unable to analyze thread entries that were defined with function pointers.

```
#include <thread>
#include <iostream>

int global{0};

void thread_function() {
  std::cout << global;
}

int main() {
  void (*fptr)() = thread_function;
  std::thread t(fptr);
  global++;
  t.join();
}
```

Listing 5.2: The analysis supports rudimentary pointer aliasing. Data races in this program will be correctly detected an reported.

```
#include <thread>
#include <iostream>

int global{0};

void thread_function() {
  std::cout << global;
}

int main() {
  std::thread t(thread_function);
  int * global_ptr = &global;
  *global_ptr=1;
  t.join();
}
```

6. Conclusion

This chapter contains review of the project results. It first summarizes the features that were implemented over the course of the project, followed up by an assessment of the effectiveness of the analysis. After that, the current limitations are listed. The chapter concludes with a look towards the future of the project.

6.1. Results

During this master thesis, an analysis tool was created, which finds data races in concurrent C++11 programs. The employed data races analysis is based on Dr. Luc Bläsers *Efficient Static Thread-Start-Join-Sensitive Detection of Low-Level Data Races and Deadlocks* algorithm. The created software contains the following noteworthy features:

- The information used for the analysis is extracted from the C++ source code using Clang and its static analysis engine. The other components of the tool are completely decoupled from the extraction, and are not dependent on Clang. If needed, the extraction backend can be changed without touching the implementation of the algorithm or the user interface..
- Currently, the analysis tool supports programs that were built using the most common C++11 concurrency constructs (`std::thread` and `std::mutex`).
- The tool supports cross-translation unit analysis for regular functions. This feature serves as a proof-of-concept to show that this type of analysis is possible.
- The user interface is implemented as an Eclipse CDT plug-in. As such, it employs common Eclipse UI elements, such as quick assists/marker resolutions, making it easy to learn and use for developers familiar with Eclipse.
- The analysis results are displayed using Eclipse code markers. Because the interpretation of the information contained in these markers is not straightforward, the tool also provides an interactive graphical visualization help developers understand and connect the analysis result data.

The analysis tool is currently not ready to be applied on real-world C++ programs, as it does not yet support all C++ concurrency constructs. However, it is a first step towards a world with better C++ tooling.

6.2. Outlook

The analysis tool is far from finished. A lot of work is still left to do, which has become possible because of the groundwork delivered during this thesis:

- The most obvious way to continue this project is to add support for more C++ concurrency constructs. Doing this would greatly increase the tool's usefulness and is not tremendously difficult to achieve.
- A more challenging continuation could be the addition of support for aliasing across translation units. This could possibly be realized by carrying along an aliasing list for all known memory locations, including the parameter names of unresolved functions.
- Probably on the same level of difficulty would be the extension of the cross translation unit analysis support. This continuation would encompass implementing support for calls to template functions, function pointers, and operators.
- A completely different thing to do could the implementation of the "Deadlocks" part of Dr. Bläser's algorithm. Essentially, deadlocks occur when concurrent threads do not acquire a set of common locks the same order, thereby blocking each other continuing their execution. The algorithm would have to extended to not only analyze what locks each thread holds, but also in what order they were acquired.

These potential continuations presume that the current version of the analysis tool is completely bug-free and working as intended. However, this is highly improbable. All of these suggestions also include continuous iteration on the existing features to get rid of bugs, and to generally improve the design and implementation.

6.3. Acknowledgments

First, I want to thank my advisor, Prof. Peter Sommerlad, for his guidance during this master thesis, and over my entire course of studies. He always provided honest and valuable feedback and ensured that I remain focused on my goals.

I also want to thank Thomas Corbat for the helpful technical discussions during the project meetings, and for giving me insightful feedback on the documentation.

I also want to thank Mirko Stocker, who provided feedback and suggestions regarding my Scala code, and Lukas Wegmann, who helped answered my Scala-related questions.

I also want to thank Fanny Schweizer, who provided insightful comments on the visualization and interaction aspects of the analysis tool, and Julia Schäfer, who helped me with the layout, the cover, and other graphic design related issues of this document.

Last but not least, I want to thank Dr. Luc Bläser, for allowing me to use his algorithm in my thesis, helping me understand it, and giving me very insightful feedback on its documentation.

7. Personal Commentary

I had a lot of fun working on this thesis, manly due to the diverse tasks that it encompassed. Especially working on the documentation and implementation of the algorithm, meanwhile reading more or less related the research papers was a very creative and educational experience.

I also felt that the goals were reasonably challenging. At the beginning of the project, I was very skeptical that it was doable given the infrastructure, target language, and my experience.

I'm very happy with the written report that I delivered, and that I was able to implement the data race detection algorithm without running into to much problems. Especially considering that I only spent roughly two hours in total with Dr. Bläser talking about the algorithm, and had to deal with many traps and pitfalls by myself.

Towards the end of the project, I felt that I had to rush it to completion just when things started to get interesting, forcing me to leave out a few features which I would have liked to at least try implementing. In particular, I would have liked to add support for more C++ specific concurrency constructs to the analysis. However, writing the report, especially the chapters covering the data race algorithm, the software architecture and the introduction, took a lot of time, and several iterations. Combined with the fact that I had to familiarize myself with the algorithm, the Clang API and the Clang Static Analyzer left not enough time to really start working on the more advanced C++ stuff.

Overall, working on the thesis was a positive experience. I felt that I was able use my prior knowledge and combine it with new insights to solve the problems at hand.

Appendices

A. Project Environment

This chapter contains a description of the tools that were used during this thesis, and a report providing information regarding the time spent working on it.

A.1. Tools

The following tools were used to create the analysis tool and this report:

Cevelop C++ IDE to write all C++ code.

Scala IDE for Eclipse to write all Scala code.

Eclipse Plug-in Development Environment to write all Java code.

Git to manage the source code of the software components and the LaTeX documents, from which this report was generated.

CMake to create the makefiles for the C++ source code.

SBT to build and test the Scala code.

Maven & Tycho to build and test the Eclipse plug-in.

Visio 2013 / PowerPoint 2011 to created the illustrations for this report

Inkscape To create all icons and graphics for the Eclipse plug-in.

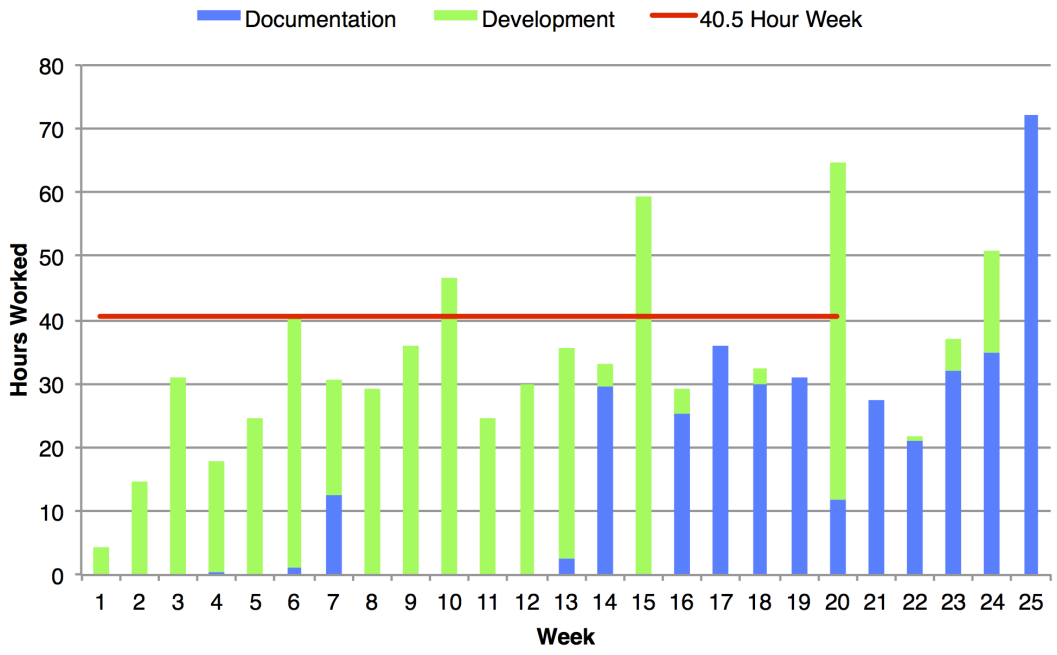
Jenkins to continuously integrate and test any new code for all components and languages: C++, Scala, and Java, and to compile the latex files of this document.

Redmine to manage the project tasks, prepare the weekly meeting notes, and track the time spent on the project.

A.2. Time Report

The mandatory work load for this master thesis project was 810 hours. The actual time worked was with a total of 870 hours above this requirement. Around 494 hours were spent on the software development and meetings, and 366 hours were spent on the documentation.

When the work on the thesis started, there was no clear deadline defined. The project originally aimed for a completion around the end of July. However, since the work load was not satisfied by this date, and the work itself was also not finished, it was decided to bump the deadline to the end of August.



The diagram above shows the work delivered over the course of the thesis, divided up in documentation and development. The horizontal red line indicates the required 40.5 hours per week, that would have had to be worked in theory, in order to meet the original July deadline.

B. Source Code

B.1. parallel_for Implementation

Listing B.1: A simple parallel_for implementation in C++. It is used to illustrate the notion of *general races* in Section 1.1.5

```
#include <vector>
#include <iterator>
#include <thread>
using namespace std;

template<class InputIt, class UnaryFunction>
UnaryFunction parallel_for(InputIt first, InputIt last, UnaryFunction f) {
  vector<thread> workers{};
  while (first != last) {
    workers.push_back(thread(
      [&f](typename iterator_traits<InputIt>::value_type current) {
        f(current);
      },
      *first++)
    );
  }
  for (auto& worker : workers) { worker.join(); }
  return f;
}
```

B.2. plugin.xml

B.2.1. "Run Analysis" Button Extension

```
<extension point="org.eclipse.ui.menus">
  <menuContribution
      allPopups="false"
      locationURI="toolbar:org.eclipse.ui.main.toolbar">
    <toolbar
        id="ch.hsr.ifs.conditionr.toolbar"
        label="ConditionR Toolbar">
      <command
          commandId="ch.hsr.ifs.conditionr.analysis"
          icon="icon/start-analysis_16.png"
          label="Run Analysis"
          style="push">
      </command>
    </toolbar>
  </menuContribution>
</extension>
```

B.2.2. "Run Analysis" Command Extension

```
<extension point="org.eclipse.ui.commands">
  <command
      defaultHandler="ch.hsr.ifs.conditionr.ui.commands.RunAnalysisCommand"
      id="ch.hsr.ifs.conditionr.analysis"
      name="Run Analysis">
  </command>
</extension>
```

B.2.3. "Data Race Visualization" Extension

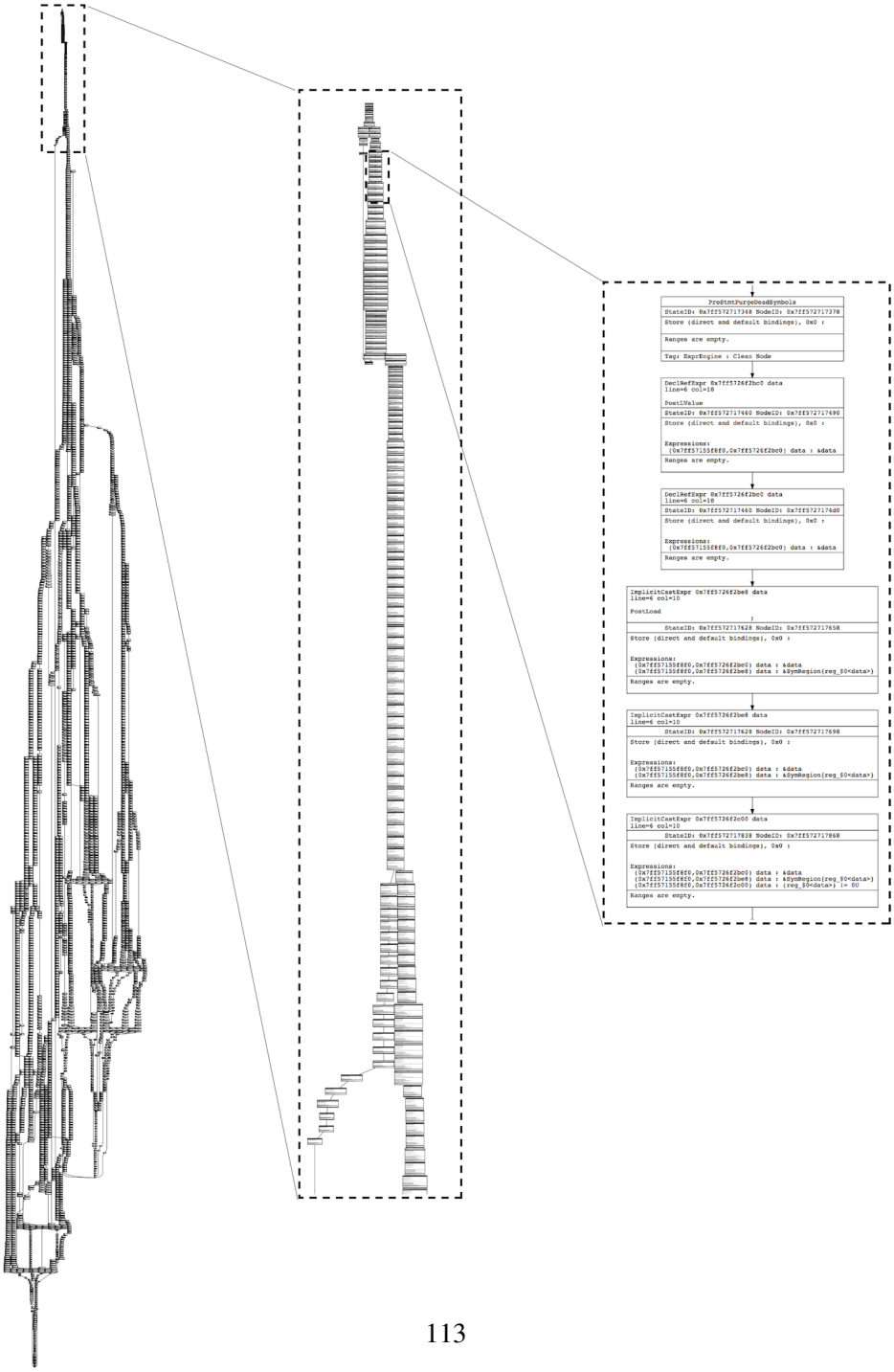
```
<extension point="org.eclipse.ui.views">
  <category id="ch.hsr.ifs.conditionr" name="conditionR" />
  <view
    category="ch.hsr.ifs.conditionr"
    class="ch.hsr.ifs.conditionr.ui.views.SimpleDataRaceView"
    icon="icon/plugin-icon_16.png"
    id="ch.hsr.ifs.conditionr.ui.dataraceview"
    name="Data Races"
    restorable="true">
  </view>
  <view
    category="ch.hsr.ifs.conditionr"
    class="ch.hsr.ifs.conditionr.ui.views.LocationCenteredDataRaceView"
    icon="icon/plugin-icon_16.png"
    id="ch.hsr.ifs.conditionr.view1"
    name="Data Races v2"
    restorable="true">
  </view>
</extension>
```

B.2.4. "Data Race Marker Resolution" Extension

```
<extension point="org.eclipse.ui.ide.markerResolution">
  <markerResolutionGenerator
      class="ch.hsr.ifs.conditionr.ui.views.DataRaceResolutionGenerator"
      markerType="ch.hsr.ifs.conditionr.dataracemarker">
  </markerResolutionGenerator>
</extension>
```

C. Illustrations

C.1. Full Program State Graph



C.2. Clang Mailing List Conversation

C.2.1. Initial Question

From: **Brugnoni Silvano** sbrugnon@hsr.ch
Subject: [cfe-dev] Static Analyzer Question
Date: 31 Mar 2015 18:23
To: Clang cfe-dev@cs.uiuc.edu

Hi all,

I'm working on a static analysis tool to detect low-level data races in C++ code. The analysis I'm trying to implement requires a control flow graph (CFG) of the entire program.

I'm currently working the construction of this whole-program CFG using the Clang Static Analyzer. I'm aware that cross-translation unit IPA is not supported by the static analyzer by default. But I think it should be possible to extract the CFG of each translation unit independently using a custom checker, and then merge them together into a whole-program CFG afterwards. I plan to perform the merging and the subsequent analysis from a standalone tool based on LibTooling. This brings me to my question:

Is it possible to run a checker from a clang standalone tool without having to compile it into clang? From what I can tell, the AnalysisConsumer, which is in charge of running the path-sensitive analyses, is located in an anonymous namespace and there is no way of passing it an additional checker to execute. I would appreciate it if you could clarify this.

Since I have not been able to run a checker from a clang standalone tool myself, I explored a different route: I tried to instantiate the clang::into::ExprEngine and its dependencies from within a custom FrontendAction in order to obtain the translation-unit CFG this way. However, I keep getting Segmentation Faults, and sometimes even Bus Errors when I run this program, depending on the exact code and compiler optimization settings. These Errors are probably caused by my lack of understanding of the clang code base, and I'm still investigating what is missing/wrong with my code. Still, I attached a simplified version of the program. Any advice is appreciated as well.

Regards,
Silvano

main.cpp _____
cfe-dev mailing list
cfe-dev@cs.uiuc.edu
http://lists.cs.uiuc.edu/mailman/listinfo/cfe-dev

C.2.2. Most Helpful Response

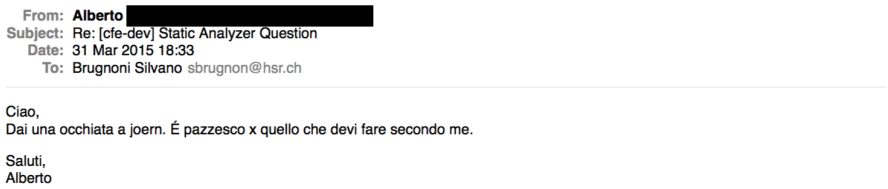
From: **Anna Zaks** ganna@apple.com
Subject: Re: [cfe-dev] Static Analyzer Question
Date: 1 Apr 2015 20:07
To: Brugnoni Silvano sbrugnon@hsr.ch
Cc: Clang cfe-dev@cs.uiuc.edu

> On Mar 31, 2015, at 9:09 AM, Brugnoni Silvano <sbrugnon@hsr.ch> wrote:
>
> Hi all,
>
> I'm working on a static analysis tool to detect low-level data races in C++ code. The analysis I'm trying to implement requires a control flow graph (CFG) of the entire program.
>
> I'm currently working the construction of this whole-program CFG using the Clang Static Analyzer. I'm aware that cross-translation unit IPA is not supported by the static analyzer by default. But I think it should be possible to extract the CFG of each translation unit independently using a custom checker, and then merge them together into a whole-program CFG afterwards. I plan to perform the merging and the subsequent analysis from a standalone tool based on LibTooling. This brings me to my question:
>
> Is it possible to run a checker from a clang standalone tool without having to compile it into clang? From what I can tell, the AnalysisConsumer, which is in charge of running the path-sensitive analyses, is located in an anonymous namespace and there is no way of passing it an additional checker to execute. I would appreciate it if you could clarify this.

We have **unsupported** plugins feature. You can find a very useful manual by searching cfe-dev for "Tutorial for Clang Analyzer Plugins".

> Since I have not been able to run a checker from a clang standalone tool myself, I explored a different route: I tried to instantiate the clang::into::ExprEngine and its dependencies from within a custom FrontendAction in order to obtain the translation-unit CFG this way. However, I keep getting Segmentation Faults, and sometimes even Bus Errors when I run this program, depending on the exact code and compiler optimization settings. These Errors are probably caused by my lack of understanding of the clang code base, and I'm still investigating what is missing/wrong with my code. Still, I attached a simplified version of the program. Any advice is appreciated as well.
>
> Regards,
> Silvano
>
> <main.cpp>_____
> cfe-dev mailing list
> cfe-dev@cs.uiuc.edu
> http://lists.cs.uiuc.edu/mailman/listinfo/cfe-dev

C.2.3. Other (not so helpful) Responses



From: **Alberto** [redacted]
Subject: Re: [cfe-dev] Static Analyzer Question
Date: 31 Mar 2015 18:33
To: Brugnoni Silvano sbrugnon@hsr.ch

Ciao,
Dai una occhiata a joern. É pazzesco x quello che devi fare secondo me.

Saluti,
Alberto

C.3. Extraction Component JSON Output

The following listing shows the untouched JSON string that is generated by the extraction component when applied to the program in Listing 3.6a. The corresponding graph is illustrated in Listing 3.6c.

```
{
        "graph": "main",
        "nodes": [{
                "id": "main/2",
                "pred": ["main/1"],
                "succ": ["main/exit"],
                "stmt": [{
                        "type": "Read",
                        "memoryLocation": "&memory",
                        "fileLocation": {
                                "offset": 9,
                                "length": 0,
                                "file": "main.cpp"
                        }
                },{
                        "type": "Read",
                        "memoryLocation": "&memory",
                        "fileLocation": {
                                "offset": 11,
                                "length": 0,
                                "file": "main.cpp"
                        }
                }]
        },{
                "id": "main/entry",
                "pred": [],
                "succ": ["main/1"],
                "stmt": []
        },{
```

```
                "id": "main/3",
                "pred": ["main/1"],
                "succ": ["main/exit"],
                "stmt": [{
                        "type": "Write",
                        "memoryLocation": "&memory",
                        "fileLocation": {
                                "offset": 7,
                                "length": 0,
                                "file": "main.cpp"
                        }
                },{
                        "type": "Read",
                        "memoryLocation": "&memory",
                        "fileLocation": {
                                "offset": 11,
                                "length": 0,
                                "file": "main.cpp"
                        }
                }]
        },{
                "id": "main/exit",
                "pred": ["main/2", "main/3"],
                "succ": [],
                "stmt": []
        },{
                "id": "main/1",
                "pred": ["main/entry"],
                "succ": ["main/2", "main/3"],
                "stmt": [{
                        "type": "BranchCondition",
                        "fileLocation": {
                                "offset": 6,
                                "length": 9,
                                "file": "main.cpp"
                        }
                }]
        }]
}
```

Bibliography

[65013] Standard for Information Technology: Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7. In: *IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1-2013)* (2013), April, S. 512–521. http://dx.doi.org/10.1109/IEEESTD.2013.6506091. – DOI 10.1109/IEEESTD.2013.6506091

[AG96] ADVE, Sarita V. ; GHARACHORLOO, Kourosh: Shared Memory Consistency Models: A Tutorial. In: *Computer* 29 (1996), Dezember, Nr. 12, 66–76. http://dx.doi.org/10.1109/2.546611. – DOI 10.1109/2.546611. – ISSN 0018–9162

[ALSU06] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[AS83] ANDREWS, Gregory R. ; SCHNEIDER, Fred B.: Concepts and Notations for Concurrent Programming. In: *ACM Comput. Surv.* 15 (1983), März, Nr. 1, 3–43. http://dx.doi.org/10.1145/356901.356903. – DOI 10.1145/356901.356903. – ISSN 0360–0300

[BA08] BOEHM, Hans-J. ; ADVE, Sarita V.: Foundations of the C++ Concurrency Memory Model. In: *SIGPLAN Not.* 43 (2008), Juni, Nr. 6, 68–78. http://dx.doi.org/10.1145/1379022.1375591. – DOI 10.1145/1379022.1375591. – ISSN 0362–1340

[Ber66] BERNSTEIN, A.J.: Analysis of Programs for Parallel Processing. In: *Electronic Computers, IEEE Transactions on* EC-15 (1966), Oct, Nr. 5, S. 757–763. http://dx.doi.org/10.1109/PGEC.1966.264565. – DOI 10.1109/PGEC.1966.264565. – ISSN 0367–7508

[Boe05] BOEHM, Hans-J.: Threads Cannot Be Implemented As a Library. In: *SIGPLAN Not.* 40 (2005), Juni, Nr. 6, 261–268. http://dx.doi.org/10.1145/1064978.1065042. – DOI 10.1145/1064978.1065042. – ISSN 0362–1340

[BOS+11] BATTY, Mark ; OWENS, Scott ; SARKAR, Susmit ; SEWELL, Peter ; WEBER, Tjark: Mathematizing C++ Concurrency. In: *SIGPLAN Not.* 46 (2011), Januar, Nr. 1, 55–66. http://dx.doi.org/10.1145/1925844.1926394. – DOI 10.1145/1925844.1926394. – ISSN 0362–1340

[CDH+00] CORBETT, James C. ; DWYER, Matthew B. ; HATCLIFF, John ; LAUBACH, Shawn ; PĂSĂREANU, Corina S. ; ROBBY ; ZHENG, Hongjun: Bandera: Extracting Finite-state Models from Java Source Code. In: *Proceedings of the 22Nd International*

Conference on Software Engineering. New York, NY, USA : ACM, 2000 (ICSE '00). – ISBN 1–58113–206–9, 439–448

[CGP99] CLARKE, Edmund M. Jr. ; GRUMBERG, Orna ; PELED, Doron A.: *Model Checking*. Cambridge, MA, USA : MIT Press, 1999. – ISBN 0–262–03270–8

[EA03] ENGLER, Dawson ; ASHCRAFT, Ken: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: *SIGOPS Oper. Syst. Rev.* 37 (2003), Oktober, Nr. 5, 237–252. http://dx.doi.org/10.1145/1165389.945468. – DOI 10.1145/1165389.945468. – ISSN 0163–5980

[ecl15a] ECLIPSE.ORG: *Eclipse Plug-in Development FAQ*. https://wiki.eclipse.org/Eclipse_Plug-in_Development_FAQ. Version: 2015. – Retrieved 23. August, 2015

[ecl15b] ECLIPSE.ORG: *What are extensions and extension points?* https://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F. Version: 2015. – Retrieved 23. August, 2015

[ecl15c] ECLIPSE.ORG: *What is a plug-in?* https://wiki.eclipse.org/FAQ_What_is_a_plug-in%3F. Version: 2015. – Retrieved 23. August, 2015

[EP88] EMRATH, Perry A. ; PADUA, David A.: Automatic Detection of Nondeterminacy in Parallel Programs. In: *SIGPLAN Not.* 24 (1988), November, Nr. 1, 89–99. http://dx.doi.org/10.1145/69215.69224. – DOI 10.1145/69215.69224. – ISSN 0362–1340

[FF00] FLANAGAN, Cormac ; FREUND, Stephen N.: Type-based Race Detection for Java. In: *SIGPLAN Not.* 35 (2000), Mai, Nr. 5, 219–232. http://dx.doi.org/10.1145/358438.349328. – DOI 10.1145/358438.349328. – ISSN 0362–1340

[HM94] HELMBOLD, D. P. ; MCDOWELL, C. E.: A TAXONOMY OF RACE DETECTION ALGORITHMS. Santa Cruz, CA, USA : University of California at Santa Cruz, 1994. – Forschungsbericht

[Hoa74] HOARE, C. A. R.: Monitors: An Operating System Structuring Concept. In: *Commun. ACM* 17 (1974), Oktober, Nr. 10, 549–557. http://dx.doi.org/10.1145/355620.361161. – DOI 10.1145/355620.361161. – ISSN 0001–0782

[KSY05] KAMIL, Amir ; SU, Jimmy ; YELICK, Katherine: Making Sequential Consistency Practical in Titanium. In: *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. Washington, DC, USA : IEEE Computer Society, 2005 (SC '05). – ISBN 1–59593–061–2, 15

[Lam79] LAMPORT, L.: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. In: *IEEE Trans. Comput.* 28 (1979), September, Nr. 9, 690–691. http://dx.doi.org/10.1109/TC.1979.1675439. – DOI 10.1109/TC.1979.1675439. – ISSN 0018–9340

[llv15a] LLVM.ORG: *Checker Developer Manual*. `http://clang-analyzer.llvm.org/checker_dev_manual.html`. Version: 2015. – Retrieved 30. July, 2015

[llv15b] LLVM.ORG: *Checker Documentation*. `http://clang.llvm.org/doxygen/CheckerDocumentation_8cpp_source.html`. Version: 2015. – Retrieved 30. July, 2015

[llv15c] LLVM.ORG: *Clang Static Analyzer*. `http://clang-analyzer.llvm.org/`. Version: 2015. – Retrieved 28. July, 2015

[llv15d] LLVM.ORG: *Static Analyzer Debug Checkers*. `http://clang-analyzer.llvm.org/docs/DebugChecks.html`. Version: 2015. – Retrieved 4. August, 2015

[MC91] MELLOR-CRUMMEY, John: On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism. In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. New York, NY, USA : ACM, 1991 (Supercomputing '91). – ISBN 0–89791–459–7, 24–33

[NM92] NETZER, Robert H. B. ; MILLER, Barton P.: What Are Race Conditions?: Some Issues and Formalizations. In: *ACM Lett. Program. Lang. Syst.* 1 (1992), März, Nr. 1, 74–88. `http://dx.doi.org/10.1145/130616.130623`. – DOI 10.1145/130616.130623. – ISSN 1057–4514

[PG01] PRAUN, Christoph von ; GROSS, Thomas R.: Object Race Detection. In: *SIGPLAN Not.* 36 (2001), Oktober, Nr. 11, 70–82. `http://dx.doi.org/10.1145/504311.504288`. – DOI 10.1145/504311.504288. – ISSN 0362–1340

[RB00] RONSSE, Michiel ; BOSSCHERE, Koenraad D.: Non-intrusive on-the-fly data race detection using execution replay. In: *AADEBUG*, 2000

[RH04] ROY, Peter V. ; HARIDI, Seif: *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA, USA : MIT Press, 2004. – ISBN 0262220695

[SBN+97] SAVAGE, Stefan ; BURROWS, Michael ; NELSON, Greg ; SOBALVARRO, Patrick ; ANDERSON, Thomas: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In: *ACM Trans. Comput. Syst.* 15 (1997), November, Nr. 4, 391–411. `http://dx.doi.org/10.1145/265924.265927`. – DOI 10.1145/265924.265927. – ISSN 0734–2071

[SS88] SHASHA, Dennis ; SNIR, Marc: Efficient and Correct Execution of Parallel Programs That Share Memory. In: *ACM Trans. Program. Lang. Syst.* 10 (1988), April, Nr. 2, 282–312. `http://dx.doi.org/10.1145/42190.42277`. – DOI 10.1145/42190.42277. – ISSN 0164–0925

[VHB+03] VISSER, Willem ; HAVELUND, Klaus ; BRAT, Guillaume ; PARK, Seungjoon ; LERDA, Flavio: Model Checking Programs. In: *Automated Software Engg.* 10 (2003), April, Nr. 2, 203–232. `http://dx.doi.org/10.1023/A:1022920129859`. – DOI 10.1023/A:1022920129859. – ISSN 0928–8910

[Vog15] VOGEL, Lars: *Eclipse Plug-in Tutorials*. `http://www.vogella.com/tutorials/eclipse.html`. Version: 2015. – Retrieved 23. August, 2015

[Wil12] WILLIAMS, Anthony: *C++ concurrency in action: practical multithreading*. Shelter Island, NY : Manning Publ., 2012 `https://cds.cern.ch/record/1483005`